Scalable Hierarchical Locking for Distributed Systems *

Nirmit Desai and Frank Mueller

Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695

e-mail: mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7925

ABSTRACT

Middleware components are becoming increasingly important as applications share computational resources in distributed environments, such as high-end clusters with ever larger number of processors, computational grids and increasingly large server farms. One of the main challenges in such environments is to achieve scalability of synchronization. In general, concurrency services arbitrate resource requests in distributed systems. But concurrency protocols currently lack scalability. Adding such guarantees enables resource sharing and computing with distributed objects in systems with a large number of nodes.

The objective of our work is to enhance middleware services to provide scalability of synchronization and to support state replication in distributed systems. We have designed and implemented a middleware protocol in support of these objectives. Its essence is a peer-to-peer protocol for multi-mode hierarchical locking, which is applicable to transaction-style processing and distributed agreement. We demonstrate high scalability combined with low response times in high-performance cluster environments. Our technical contribution is a novel, fully decentralized, hierarchical locking protocol to enhance concurrency in distributed resource allocation following the specification of general concurrency services for large-scale data and object repositories. Our experiments on an IBM SP show that the number of messages approaches an asymptote at 15 nodes, from which point on the message overhead is in the order of 3-9 messages per request, depending on system parameters. At the same time, response times increase linearly with a proportional increase in requests and, consequently, higher concurrency levels. Specifically, in the range of up to 80 nodes, response times under 10 msec are observed for critical sections that are one 25th the size of non-critical code. The high degree of scalability and responsiveness of our protocol is due in large to a high level of concurrency upon resolving requests combined with dynamic path compression for request propagation paths. Our approach is not only applicable to CORBA, its principles are shown to provide

*This work was supported in part by NSF CAREER grant CCR-0237570

benefits to general distributed concurrency services and transaction models. Besides its technical strengths, our approach is intriguing due to its simplicity and its wide applicability, ranging from large-scale clusters to server-style computing.

Keywords: Distributed mutual exclusion, middleware services, distributed resource allocation, concurrency services, hierarchical locking, peer-to-peer protocols, scalability, large-scale distributed computing, distributed agreement, distributed transactions

1. INTRODUCTION

Distributed computing is rapidly becoming a commodity to share resources, such as objects, on a larger and larger scale. In the past, applications relied on message passing, shared memory, remote procedure calls and their object counterparts, such as remote method invocations, to exploit parallelism in distributed environments or invoke remote services in a client-server paradigm. The problem with these approaches is its reliance on access to a centralized facility and its limitations in scalability.

In contrast, recent trends aim at peer-to-peer computing with distributed objects. This paradigm is generally supported by middleware to provide distributed services. This middleware provides a software layer between the operating system and the applications that supports cooperative problem solving and provides user transparency. This middleware constitutes the enabling technology for distributed object services, such as resource arbitration in distributed systems.

Another relevant trend regarding our work relates to clusters and the Grid. High-end clusters and the Grid have increased considerably in size over the past years. These clusters profit not only from advances in processor design and interconnects but their main advantage is its mere size, currently ranging up to 8,000 processors with future projections over 10,000, *e.g.*, for IBM's Blue Gene Light and potentially even larger systems in the Grid [1, 12]. We see a similar trend in commercial computing areas, such as server computing. Servers are increasingly organized in ever larger server farms. This trend is in response to requirements for high availability and faster response times. Multiple server farms may exist in geographically distant locations so that accesses can be quickly delegated to a server in the requester's vicinity.

One of the main challenges in such environments is to achieve scalability of synchronization. We address the issues of scalability through middleware protocols. Though our protocol is compatible with existing standards, such as CORBA, its model is applicable to any distributed resource allocation scheme. For example, distributed agreement, originally designed for distributed database systems, has recently been adopted for cluster computing [10, 11]. The use of transactions in such environments requires support for

hierarchical locking services to arbitrate between requests of different modes at multiple levels within the shared / replicated data. Hierarchical locks have been studied in the context of database system with a limited number of nodes [16, 22, 21, 19, 3].

The sheer size of clusters and server farms requires us to consider hierarchical locking again, but this time under the aspect of scalability. The challenge in environments with a large number of nodes is to provide short response times for lock requests through a protocol that scales well with the number of nodes.

The work presented in this paper provides a solution to this problem and extends our previous results [8, 9]. We present a hierarchical locking protocol that supports a high degree of concurrency for a number of access modes. The protocol is aimed at global state replication in distributed systems. The underlying protocol follows a peer-to-peer paradigm, which is applicable to transaction-style processing and distributed agreement. The peer-to-peer paradigm ensures scalability by relying only on fully decentralized data structures and symmetric algorithms. As a result, our protocol is highly scalable due to its O(log n) message complexity for n nodes, it accommodates a large number of concurrent requests, provides progress guarantees to prevent starvation and delivers short response times, even in large networks. Experimental results on an IBM SP show overheads of 3-9 messages and response times lower than 10 msec up to 80 nodes depending on the ratio of non-critical code and critical sections.

In this paper, we first review a non-hierarchical locking protocol, which we compare against during later experimentations. We then introduce our peer-to-peer hierarchical locking protocol, define its operations through a set of rules and tables, and we provide several examples together with pseudo-code. Our work is based on the semantics of concurrency services widely used in database systems and also defined in the CORBA concurrency services. We modify the specification of CORBA concurrency services, which results in a specialization of the services that does not affect its original properties (since it is a specialization). New properties, such as fairness, are due to FIFO queuing within token owners. The primary objective of this work is to demonstrate strengths of our approach for providing a scalable middleware protocol. This is demonstrated by the rules, tables and examples describing the dynamics of the protocol. The second emphasis is on extensive experiments on actual systems confirming our claims. Our experiments are comprised of the aforementioned comparison with non-hierarchical locking on a Linux cluster on one hand and results for scalability and latency evaluations on an IBM SP cluster on the other hand. We then discuss related work and summarize our contributions.

2. MUTUAL EXCLUSION FOR DISTRIBUTED SYSTEMS

Concurrent requests to access shared resources in a distributed environment have to be arbitrated by means of mutual exclusion, *e.g.*, to provide hierarchical locking and support transaction processing. In the absence of shared memory, mutual exclusion is realized via a series of messages passed between nodes that share a certain resource. Several algorithms have been developed to provide mutual exclusion for distributed systems [6]. They can be distinguished by their approaches as token-based and non-token-based. The former may rely on broadcast protocols or may use logical structures with point-to-point communication. Broadcast and non-token-based protocols generally suffer from limited scalability due to centralized control, due to their message overhead or because of topological constraints. In contrast, token-based protocols exploiting point-to-point connectivity may result in logarithmic message complexity with regard to the number of nodes. In the following, a fully decentralized token-based protocol is introduced.

Token-based algorithms for mutual exclusion employ a single token representing the lock object, which is passed between nodes within the system [28]. Possession of the token represents the right to enter the critical region of the lock object. Requests that cannot be served right away are registered in a distributed linked list originating at the token owner. Once a token becomes available, the owner passes it on to the *next* requester within the distributed list. In addition, nodes form a logical tree pointing via probable *owner* links toward the root. Initially, the root is the token owner. When requests are issued, they are guided by a chain of probable owners to the current root. Each node on the propagation path sets its probable owner to the requester, *i.e.*, the tree is modified dynamically.

In Figure 1, the root T initially holds the token for mutual exclusion. A request by A is sent to B following the probable owner (solid arcs). Node B forwards the request along the chain of probable owners to T and sets its probable owner to A. When the request arrives at the root T, the probable owner and a *next* pointer (dotted arc) are set to the requester A. The next request from C is sent to B. 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 owner T returns from its critical section, the token is sent to the target node that *next* points to. Hence, T passes the token to A and deletes the *next* pointer.

This algorithm has an average message overhead of $O(log\ n)$ since requests are relayed through a dynamically adjusted tree, which results in path compression with regard to future request propagation. It is fully decentralized, which ensures scalability for large numbers of nodes. The model assures that requests are ordered FIFO. Our contribution in prior work was to alleviate shortcomings in priority support [24, 25]. In this work, we develop a novel protocol for hierarchical locking building on our past results and demonstrate

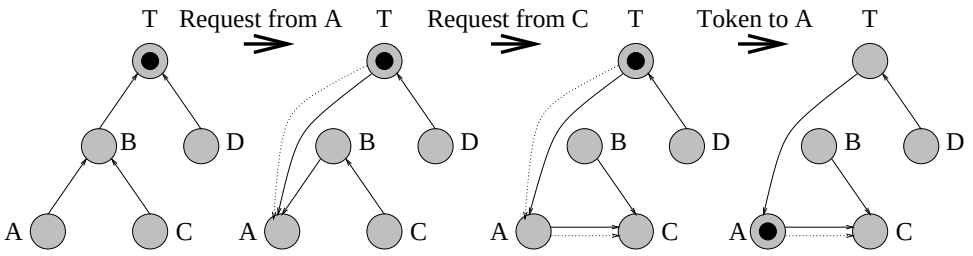


Figure 1: Non-hierarchical Example

its suitability to deliver short latencies and low message overhead in cluster environments.

3. A PEER-TO-PEER HIERARCHICAL LOCKING PROTOCOL

This section introduces a novel locking protocol. This protocol strictly follows a peer-to-peer paradigm in that all data structures are fully decentralized and each node runs a symmetric instance of the protocol. These operational characteristics combined with a O(log n) message complexity ensure scalability and are demonstrated to also yield low response times. The protocol distinguishes a number of access modes in support of concurrency services for distributed computing. In the following, we refer to the Concurrency Services of CORBA, which follows the *de facto* standard hierarchical locking model widely used in database systems, as the underlying model without restricting the generality of our proposed protocol [17].

3.1 Compatibility between Lock Modes

The main objective of our approach is to ensure a high degree of concurrency for the distributed mutual exclusion protocol. We allow multiple nodes to share access to a resource if possible by supporting a set of five access modes compatible with common access requirements of database systems and distributed object systems.

As in Naimi's protocol, nodes form a logical tree structure by maintaining their local parent pointers. But our protocol does not require next pointers. The root node of the tree holds the token and is referred to as *the token* node. All other nodes are *non-token* nodes. We support the following access modes. First, we distinguish read (R) locks and write (W) locks with shared and exclusive access, respectively. Second, we support upgrade (U) locks, which represent an exclusive read lock that is followed by an upgrade request for a write lock. Upgrade locks ensure data consistency between a read followed by an update value that was derived from the read value. Third, we provide intent locks for reading (IR) and writing (IW).

Intent locks are motivated by hierarchical locking paradigms, which allow the distinction between lock modes on the structural data representation, *e.g.*, when a database, multiple tables within the database and

entries within tables are associated with distinct locks [16, 22]. For example, an entity may first acquire an intent write lock on a database and then disjoint write (or upgrade) locks on the next lower granularity. Since the low-level locks are assumed to be disjoint, hierarchical locks greatly enhance parallelism by allowing simultaneous access for such threads. In general, lock requests may proceed in parallel if modes for a lock are compatible.

The compatibility between these basic lock modes defines which modes may be used in parallel by different requesters. Conversely, incompatibility of locks modes indicates a need for serialization of two requests. Let R be a resource and L_R be the lock associated with it. Table 1 shows the rules for granting L_R in different modes according to the specification of concurrency services [17]. Column one specifies the presently held lock modes for L_R and the remaining columns represent the type of mode requests received for L_R.

To define our protocol, we derive several rules for locking and specify if concurrent access modes are permissible through a set of tables. These tables not only demonstrate the elegance of the protocol, but they also facilitate its implementation.

Rule 1: Modes M_1 and M_2 are said to be *compatible* with each other if and only if they are not in conflict according to Table 1.

Mode M_1	Mode M_2				
	IR	R	U	IW	W
No lock – Φ					
Intent Read – IR					X
Read – R				X	X
Upgrade – U			X	X	X
Intent Write – IW		X	X		X
Write – W	X	X	X	X	X

Table 1: Incompatibility of Lock Modes (Conflicts Indicated as X)

Definition 1: Lock A is said to be *stronger* than lock B if the former constrains the degree of concurrency over the latter. In other words, A is compatible with fewer other modes than B is. The order of lock strengths is defined by the following inequations:

$$\Phi < IR < R < U = IW < W \tag{1}$$

A higher degree of strength implies a potentially lower level of concurrency between multiple requests. For example, a write lock allows less concurrency than a read lock, so W is stronger than R. Table 1 depicts lock modes in increasing order of lock strength – with the exception of upgrade and intent writes that,

conceptually, share the same degree of concurrency. In the following, we distinguish cases when a node *holds* a lock *vs.* when a node *owns* a lock.

Definition 2: Node A is said to *hold* the lock L_R in mode M_H if A is inside a critical section protected by the lock, *i.e.*, after A has acquired the lock and before it releases it.

Definition 3: Node A is said to *own* the lock L_R in mode M_O if M_O is the strongest mode being held by any node in the tree rooted in node A.

3.2 Local Queues, Intent Locks and Copysets

In our protocol for hierarchical locking, we employ a token-based approach. A novel aspect of our protocols is the handling of requests. While Naimi's protocol constructs a single, distributed FIFO queue, our protocol combines multiple local queues for logging incompatible requests. These local queues are logically equivalent to a single distributed FIFO, as will be seen later.

Another novelty is our handling of intent locks for token-based protocols. To distinguish different levels of lock granularities (hierarchical locks) and, at the same time, to maximize the degree of concurrency, we support intent lock modes. For example, a node wishing to read an attribute of an object will request an intent read (IR) lock on the object itself and, once acquired, it will request a read (R) lock on the attribute it wants to read without releasing IR. Note that the resources being requested in the above requests are at different levels of granularities – the object contains the attribute. Each of these resource requests can only be granted in accordance with lock compatibility requirements.

Compatible requests can be served concurrently by the first receiver of the request with a sufficient access mode. Concurrent locks are recorded, together with their access level, as so-called *copysets* of child nodes whose requests have been granted. This is a generalization of Li/Hudak's more restrictive copysets [23].

Definition 4: *Copyset* of a node is a set of nodes holding a common lock at the same time with any parent node owning the lock in a mode stronger than the mode granted to its children.

In the following, we refer to a token owner and its children to establish the relation of a copyset.

3.3 Request Granting

The next rule governs the dispatching of the lock requests.

Rule 2: A node sends a request for the lock in mode M_R to its parent if and only if the node owns the lock in mode M_O where $M_O < M_R$ (and M_O may be Φ), or M_O and M_R are incompatible. In all other cases,

only the local copyset is updated and critical section is entered without sending any messages.

Furthermore, lock requests are granted under the following conditions.

Rule 3:

1. A non-token node holding L_R in mode M_O can grant a request for L_R in mode M_R if M_O and M_R are compatible and $M_O \geq M_R$.

2. The token node owning L_R in mode M_O can grant a request for L_R in mode M_R if M_O and M_R are compatible.

The operational specification for our protocol further requires that

in case 1: the requester becomes a child of the node;

in case 2: if modes are compatible and if $M_O < M_R$, the token is transferred to the requester. Thus, the requester becomes the new token node and parent of the original token node. If $M_O \geq M_R$, the requester receives a granted copy from the token node and becomes a child of the token node. (See Rule 4 for the case when M_O and M_R are incompatible.)

Table 2 depicts legal modes for granting another mode according to this rule, indicated by the absence of an X. For the token node, compatibility represents a necessary and sufficient condition. Hence, access is subject to Rule 1 in conjunction with Table 1.

Non-token Owned Mode M_O	Requested Mode M_R				
	IR	R	U	IW	W
No lock – Φ	X	X	X	X	X
Intention Read – IR		X	X	X	X
Read – R			X	X	X
Upgrade – U			X	X	X
Intention Write – IW		X	X		X
Write – W	X	X	X	X	X

Table 2: Granting New Lock Requests by Children

In the following, we denote the tuple (M_O, M_H, M_P) corresponding to the owned, held and pending mode for each node, respectively. Shaded nodes are holding a lock and constitute the copyset, which indicates the degree of concurrency achieved at that state. A dotted, directed arc from A to B indicates that the

parent/child relation is only known to the source A but not yet to the sink B. Solid arcs indicate mutual awareness. The token is depicted as a solid circle inside a node.

Example: Consider the initial state as shown in Figure 2(a). When E requires the lock in IR, it checks Rule 2 to decide if sending a request is necessary. As $M_O = \Phi < IR = M_R$, it sends the request to its parent, node A. A receives the request, checks Rule 3.2 and responds to the request by sending a grant message. E becomes a child of A according to the operational specification. In (b) when B requires the lock in R, it checks Rule 2 and sends the request to its parent, node A. A receives the request, checks Rule 3.2 and grants the request by transferring the token to B (for A, $M_O < M_R$). B becomes the new token node and A becomes a child of B. (c) shows the final state of the nodes.

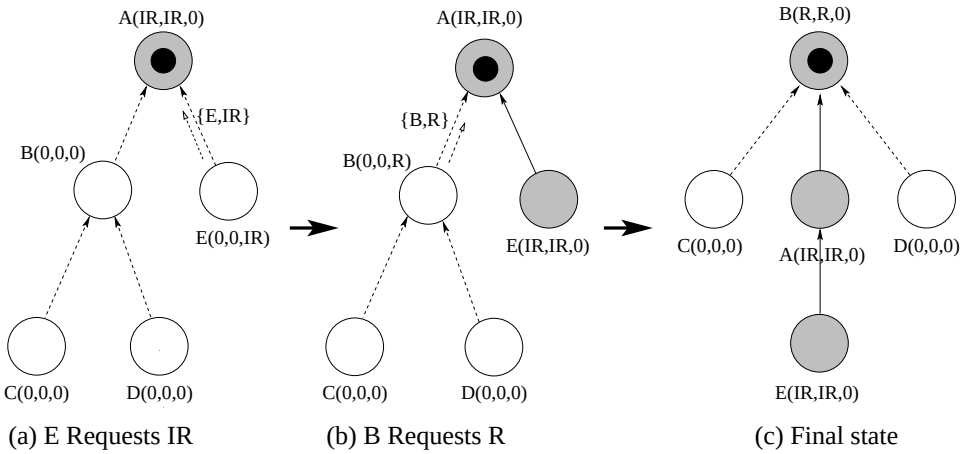


Figure 2: Request Granting Example

3.4 Request Queuing/Forwarding

When a node issues a request that cannot be granted right away due to mode incompatibility, the following rule applies.

Rule 4:

1. If a non-token node cannot grant a request, it will either forward the request to its parent or queue the request locally based on the present state of a pending request of the node according to Table 3.

2. If the token node cannot grant a request, it will queue the request locally regardless of the state of its pending request.

Rule 4 is supplemented by the following operational specification: Locally queued requests are considered for granting when the pending request comes through or a release message is received.

In Table 3, local queuing and forwarding are indicated as *Q* and *F*, respectively. The aim here is to queue

Non-token Pending Mode M_P	Requested Mode M_R				
	IR	R	U	IW	W
No pending – Φ	F	F	F	F	F
Intention Read – IR	Q	F	F	F	F
Read – R	F	Q	F	F	F
Upgrade – U	F	F	Q	Q	Q
Intention Write – IW	F	F	F	Q	F
Write – W	Q	Q	Q	Q	Q

Table 3: Queue / Forward Decision

as many requests as possible to suppress message passing overhead without compromising FIFO ordering.

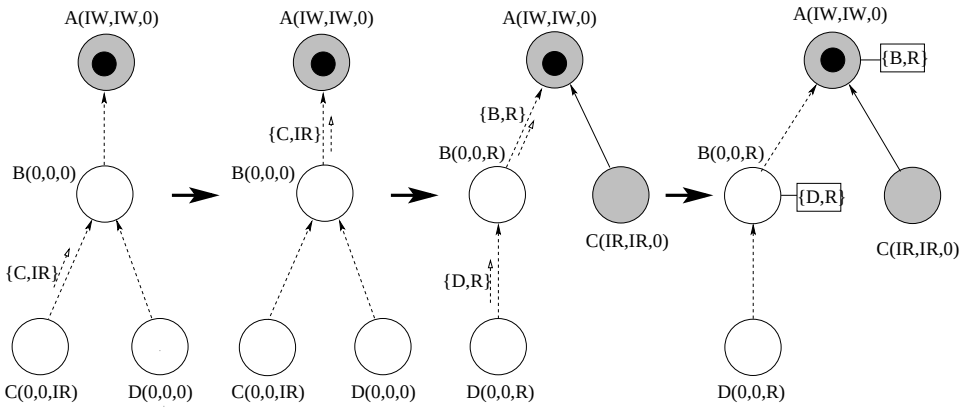
Example: In Figure 3(a), C sends a request for IR to its parent B (after checking Rule 2). When B receives the request (as it cannot grant it due to Rule 3.1), it derives from Table 3 that it can queue the request locally or not (according to Rule 4.1). As B does not have any pending requests, $M_P = \Phi$, so B has to forward the request to its parent node A, as shown in (b). A receives the requests and sends a grant as discussed above. In (c), B and D concurrently make requests sent to their respective parents, nodes A and B. When B receives D's request (as it cannot grant it due to Rule 3.1), it derives from Table 3 that it can queue the D's request locally since B has a pending request (Rule 4.1). On the other hand, when A receives B's request (as it cannot grant it due to Rule 3.2), it locally queues the request (Rule 4.2), as shown in (d). These queued requests are eventually granted when A releases IW, as specified by Rule 5 (see below).

Figure 4 depicts the pseudo-code of the above described rules for lock request handling. *RequestLock()* represents the user API whereas other operations are handlers invoked in response to message reception. (Ignore the details regarding frozen modes for now). Some macros like *compatible(), grantable()* and *tokenable(),* refer to the corresponding tables and return TRUE or FALSE, depending on the corresponding table entries. A pseudo-procedure *Check_requests_on_queue()* takes care of handling the locally queued requests as described in Rules 4 and 5.

3.5 Lock Release

The following rule governs the handling of lock releases. For this rule, let us consider parent nodes that have knowledge of only the owned modes of their immediate children.

Rule 5:



(a) C Requests IR (b) B Forwards {C,IR} (c) B Requests R, D Requests R (d) Final state

Figure 3: Request Queuing/Forwarding Example

1. When the token node releases a lock or receives release from one of its children, it considers the locally queued requests for granting under constraints of Rule 3.
2. When a non-token node A releases a lock or receives a release in some mode M_R, it will send a release message to its parent only if the owned mode of A is changed (weakened) due to this release.

The first part of this rule is similar to Naimi's protocol in that queued locks are served upon a release. If a root node has received release notifications from its children and if the root node is no longer engaged in a critical section, it will send the token to the first requester in its local queue. In addition, the local queue is piggybacked to ensure that other requests will be considered by the token recipient.

The second part of the above rule ensures that release messages are only sent towards the root of the copyset if necessary, *i.e.*, when modes change. Upon receipt of a release message, the parent may safely assume that neither the child nor its children (or grandchildren) are holding the lock in the released mode. Nonetheless, the child may still own the lock in some weaker mode, which is included in the message to allow the parent to update the log of modes for its children. Overall, this protocol reduces the number of messages compared to a more eager variant with immediate notification upon lock releases. In our approach, one message suffices, irrespective of the number of grandchildren.

Example: Consider Figure 5(a) as the initial configuration. Here, C is waiting for the IW request to be granted, which is queued locally by A. Suppose B releases the lock in R. According to Rule 5.2, it will not notify its parent about the release as the *owned* mode of B is still R due to D (one of its children)

```
RequestLock(M_R)
    if Self ≠ Token_Node then
        if M_O ≥ M_R ∧ compatible(M_O , M_R) ∧
            Frozen_Modes then [Rule 2]
            Acquire Lock
            Copyset ← Copyset + M_R
        else [Rule 2]
            M_P = M_R
            Send Request to Parent
    else
        if compatible(M_O , M_R) ∧
            Frozen_Modes then [Rule 3(2)]
            Acquire Lock
            Copyset ← Copyset + M_R
        else [Rule 4(2)]
            Queue ← Queue + M_R
            M_P ← M_R
            Update Frozen_Modes [Tab 4]
            Send Freeze to Children if required ([a])

ReceiveToken()
    Parent ← NULL
    Copyset ← Copyset + M_P
    M_H ← M_P , M_P ← φ
    Children ← Children + Sender if required [Rule 2] ([b])
    Merge Queues ([c])
    Check_requests_on_queue [Rule 4]

ReceiveGrant()
    Parent ← Sender [Rule 3.1]
    Copyset ← Copyset + M_P
    M_H ← M_P , MP ← φ
    Frozen_Modes ← Frozen_Modes + Parent_Frozen
    Check_requests_on_queue [Rule 4]
```

[a] Freeze is sent to the child only if the child is potential granter of the mode to be frozen and the mode is not already frozen

[b] The sender of the token might still be owning some mode; If so, the sender is added to the children set of the new token node. Otherwise not.

[c] The queue at the old token node is passed to the new token node along with the token. The new token node itself may have a local queue, too. These queues are merged preserving FIFO ordering as discussed in [24].

```
HandleRequest(M_R)
    if Self ≠ Token_node then
        if grantable(M_O , M_R) then [Rule 3.1, Tab 2]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send grant to Requester
        else if can_be_queued(M_P , M_R) then [Rule 4.1, Tab 3]
            Queue ← Queue + M_R
        else [Rule 4.1, Tab 3]
            Send Request to Parent
    else
        if tokenable(M_O , M_R) then [Rule 3.2, Tab 1]
            if Requester ∈ Children then
                Children ← Children - Requester
            Parent ← Requester
            Send Token to Requester
        else if grantable(M_O , M_R) then [Rule 3.2, Tab 2]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send Grant to Requester
        else [Rule 4.2]
            Queue ← Queue + M_R
            Update Frozen_Modes [Tab 4]
            Send Freeze to Children if required ([a])

Check_requests_on_queue()
    while Queue ≠ EMPTY
        M_R ← Queue.head
        if tokenable(M_O , M_R) then [Rule 3.2, Tab 1]
            if Requester ∈ Children then
                Children ← Children - Requester
            Parent ← Requester
            Send Token to Requester
        else if grantable(M_O , M_R) then [Rule 3.2, Table 2]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send Grant to Requester
        else
            Update Frozen_Modes [Tab 4]
            exit_loop
```

Figure 4: Pseudocode for Request Handling

still *owning* the lock in mode R. However, the *held* mode of B is changed to Φ. As shown in (b), when D releases R, it sends a release message to its parent (B here) because the *owned* mode of D is changed to Φ from R (which is weakened). When B receives this release, none of B's children now *own* a mode stronger than Φ. Hence, the *owned* mode of B is changed to Φ. B sends the release to its parent (A here) due to Rule 5.2. In (c), A also releases R and, because it is not aware of the change in *owned* mode of B, its *owned* mode is still R. Only when the release arrives at A does A know about the changed mode of B, which triggers a change in A's *owned* mode from R to Φ. As shown in (d), this in turn triggers the transfer of token to C (according to Rule 5.1).

Figure 6 depicts the pseudo-code of the lock release handling. *RequestUnlock()* is a user API.

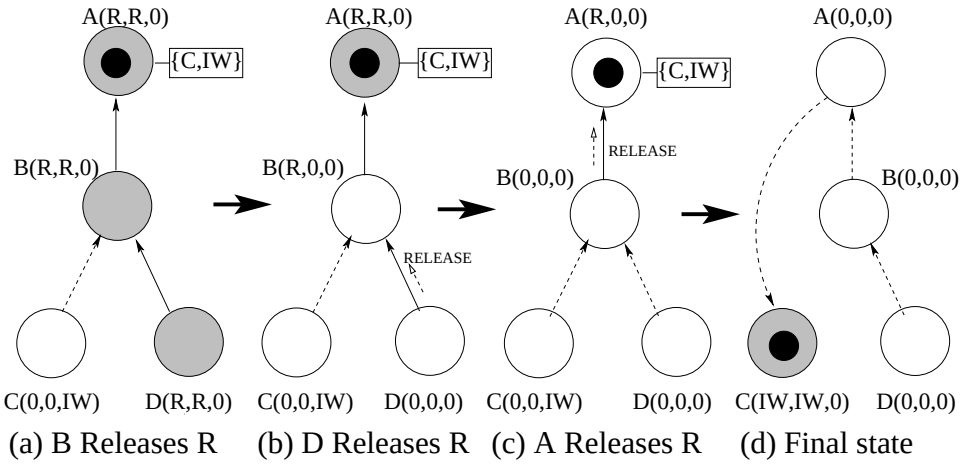


Figure 5: Lock Release Example

RequestUnlock()
Copyset ← Copyset - M_H
$M_H \leftarrow \phi$
if Changed Mode of Self **then** [Rule 5]
Send Release to Parent
Check_request_on_queue [Rule 5]

HandleRelease(M_R, M_O) [Rule 5]
Update Children for change in M_O of Child
Update Copyset for change in M_O of Child
if Changed Mode of Self **then**
Send Release to Parent
Check_requests_on_queue

Figure 6: Pseudocode for Lock Release Handling

3.6 Fairness and Absence of Starvation

The objective of providing a high degree of concurrency conflicts with common concurrency guarantees. For example, the reader-writing problem for databases is subject to starvation if new readers are accepted as long as at least one reader is active. Queued write requests, which are incompatible, would never be served if readers arrived fast enough. Consider the queuing of our protocol. Grants in response to a request may be unfair in the sense that the FIFO policy of serving requests could be violated. In essence, a newly issued request compatible with an existing lock mode could bypass already queued requests, which were deemed incompatible. Such a behavior is not only undesirable, it may lead to starvation due to the described race.

Consider a request by D for mode W in the state of Figure 7(a). (Ignore freeze messages and frozen modes for now). D, W will reach A according to the rules described above, and A will queue it according to Rule 4, as shown in (b). Once A and C release their locks for R and IR, respectively, the token will be forwarded to D due to Rule 3, as depicted in Figure 7(c). While D waits for (D, W) to advance, A may grant other

IR/R requests from other nodes according to Rule 3. As mentioned before, accepting IR/R requests potentially violates the FIFO policy since (D, W) arrived first. After W is received, we should be waiting for release of IR/R modes since they are not compatible with W. This prevents A from granting the pending (D, W) request. If, however, subsequent IR/R requests are granted, the W request may starve.

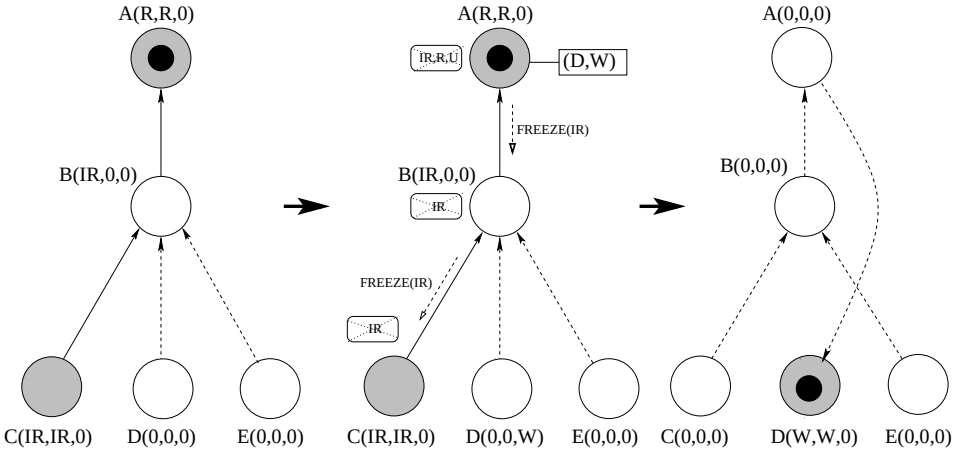


Figure 7: Frozen Modes Example

We ensure fairness in terms of FIFO queuing and, thereby, avoid starvation by freezing certain protocol states upon reception of incompatible requests. For example, the token node A, after receiving W, will not grant any other requests compatible with the waiting request (W in this case). Other modes (IR, R and U in this case) are said to be frozen when certain requests (W in this case) are received, depending on the mode owned by the token node (R in this case).

Rule 6: A node may only grant a request if the requested mode is not frozen.

In order to extend fairness beyond the token holder, mode freezing is transitively extended to the copyset where required by modes. This ensures that potential granters of any mode incompatible with the requested mode will no longer grant such requests. We ensure transitive freezing by the operational specification for this rule, which states that the token node notify children about the frozen modes. The pseudo-code of this freezing mechanism is augmented to almost all the other routines where requests are granted (Routines in Figure 4 and Figure 10). Figure 8 depicts the pseudo-code for the freeze message handler.

This rule supplements Rules 2 and 3. Table 4 depicts an enumeration of frozen modes for all combinations.

$\mathbf{HandleFreeze(Modes)}[Rule6]$
Frozen_Modes ← Frozen_Modes + Modes
Send Freeze to Children if required (^{a})

Figure 8: Pseudocode for Freezing Mechanism

For example, if the token node is owning a lock in R and a W request is received and queued locally (as depicted in Figure 7(b)) then IR, R and U are the modes to be frozen at the token node.

Token Node owning Mode M_G	Requested Mode				
	IR	R	U	IW	W
No lock – Φ					
Intention Read – IR					IR, R, U, IW
Read – R				R, U	IR, R, U
Upgrade – U				R	IR, R
Intention Write – IW		IW	IW		IR, IW
Write – W					

Table 4: Rules for Freezing Lock Modes at the Token Node

The rationale behind the construction of the Table 4 can be formalized by the following invariants:

Invariant 1: Let M_{FT} be the set of modes to be frozen at token node when the token node owning lock L_i in mode M_{OT} receives a request for L_i in mode M_R. Then,

$$\forall\ M_i \epsilon\ M_{FT} : [grantable(M_{OT}, M_i)\ \vee\ tokenable(M_{OT}, M_i)] \wedge\ conflicts(M_i, M_R) \qquad (2)$$

Invariant 2: Let M_{FC} be the set of modes to be sent with the freeze message to a child owning L_i in M_{OC}. Then,

$$\forall\ M_i \epsilon\ M_{FC} : M_i \epsilon\ M_{FT}\ \wedge\ grantable(M_{OC}, M_i) \qquad (3)$$

The predicate *grantable(M_O, M_R)* indicates if M_O and M_R satisfy Rule 3.1. *tokenable(M_O, M_R)* indicates if M_O and M_R satisfy Rule 3.2 and the token needs to be transferred. Similarly, *conflicts(M_1, M_2)* returns true if the compatibility matrix of Table 1 indicates a conflict between M_1 and M_2. Sets M_{FT} and M_{FC} calculated by invariants 1 and 2 are unified with a set $Frozen\_Modes$ recording the state for each of the six modes mentioned in Inequation 1 at each node. In the basic protocol, the only values each of the elements in $Frozen\_Modes$ can take are $FROZEN$ and NOT_FROZEN. Rule 6 uses this information and approves a grant/token for requested mode M_R if the set $Frozen\_Modes$ has NOT_FROZEN as the state value of the element corresponding to M_R.

Through this freezing mechanism, ultimately, all children and grandchildren will release the modes, and a

release message will arrive at the token node for each of its immediate children. Thus, the FIFO policy is preserved.

3.7 Upgrade Request

Upgrade locks facilitate the prevention of potential deadlocks by very simple means if the resulting reduction in the concurrency level is acceptable [17]. Upgrade locks conflict with each other (and lower strength modes), which ensures exclusive access rights for reading. This policy supports data consistency between a read in update (U) mode and a consecutive upgrade request for write (W) mode, which is commonly used when the written data depends on the prior read.This is reflected in the following rule.

Rule 7: Upon an attempt to upgrade to W, the token owner atomically changes its mode from U to W (without releasing the lock in U).

Example: As depicted in Figure 9(a), A *owns* U and requests an upgrade to W. Note the pending mode of A reflecting this situation. As this request (A, W) is waiting, freeze messages are sent to the children according to Rule 6. During this period, A does not release U (atomic upgrade) but waits for a release message to arrive from B. Ultimately when C releases IR, release messages are triggered. A, according to Rule 5, changes its *owned* mode from U to W and can now perform writes on the same data. Figure 10 is the pseudo-code of the *RequestUpgrade()* user API.

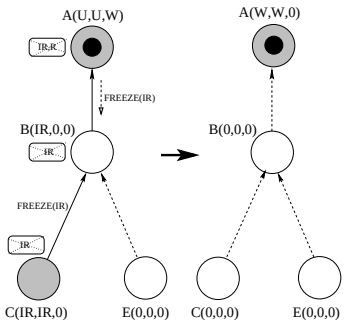


Figure 9: Request Upgrade Example

In summary, lock, unlock and upgrade operations provide the user API. The remaining operations are triggered by the protocol in response to messages, *i.e.*, for receiving request, grant, token, release, freeze and update messages. In each case, the protocol actions are directly derived from corresponding rules and tables, as indicated in the algorithm. This greatly facilitates the implementation since it reduces a seemingly complex protocol to a small set of rules defined over lookup tables in practice.

```
RequestUpgrade() [Rule 7]
    if Copyset ← φ then
        Release U
        Acquire W
    else
        M_P ← W
        Update Frozen_Modes [Tab 4]
        Send Freeze to Children if required (^α)
```

Figure 10: Pseudocode for Request Upgrade

Rules 2, 3 and 4 (the only rules governing the granting of requests) coupled with Rule 1 ensure correct mutual exclusion by enforcing compatibility. Rules 4 and 5 together ensure that each request is eventually served in the FIFO order, thus avoiding deadlocks and starvation.

The rules above are designed to solve a fundamental problem in distributed systems, *viz.* lack of global knowledge. A node has no knowledge about the modes in which other nodes are holding the lock. By virtue of our protocol, any parent node *owns* the strongest of all the modes *held/owned* in the tree rooted at that node (inequality test in Rule 3). The token node *owns* the strongest lock mode of all other *held/owned* modes. As stronger modes have lesser compatibility, while granting a request at node A, it is safe to test the compatibility with the *owned* mode of A only (meaning A does not have to check with any of its children or parents about their owned modes) , *i.e.*, local knowledge is sufficient to ensure correctness.

4. EXPERIMENTS AND ANALYSIS

Our experiments are designed to assess the capabilities of the protocol in multiple respects.

First, we evaluate the performance of our protocol relative to the protocol by Naimi et al. [28], which has the best known average case message complexity of $O(log\ n)$.[1] Second, we analyze the effect of protocol overhead on response times, *i.e.*, we detail the properties leading to closed formulas for bounding the response time of requests. The objective of this analysis is to determine the parameters that affect response time. Third, we investigate the message overhead in detail by message types to provide an understanding of the dynamics of the protocol and reason about the sources of overhead.

In addition to empirical evaluation and analysis, the following experiments also demonstrate the feasibility and applicability of the protocol for different application areas [2]. This is orthogonal to the aforementioned objectives and is elaborated in the respective contexts in the discussion below. The experimental setup and

[1] We could not find other protocols for distributed mutual exclusion with hierarchical locking models that follow a peer-to-peer paradigm. Current CORBA implementations, *e.g.*, TAO [2], do not support hierarchical locking. A comparison with centralized protocols did not seem fair due to the inefficiency of client-server approaches when scalability is to be assessed.

[2] Although fault-tolerance is an important property for such protocols, in this paper we restrict ourselves to the basic protocol. However, fault-tolerance can be ensured by extensions such as [26].

the protocol parameters are designed to match a unique set of applications in each case.

In each of the following experiments, nodes in the system execute an instance of an application (multi-airlines reservation) on top of the protocol. The data representing ticket prices are stored in a distributed table and shared amongst all the nodes. In case of our protocol, each entry of the data is associated with a lock. In addition, the entire table is associated with another lock (higher level of granularity). Each application instance (each node) will request the locks iteratively. The critical section time, the non-critical code time and the network latency experienced by messages (if applicable) were randomized with different mean values, depending on the type of experiment. The mode of lock requests was randomized so that the IR, R, U, IW and W requests are 80%, 10%, 4%, 5% and 1% of the total requests, respectively. This request distribution should reflect the typical frequency of request types for such applications in practice where read requests to a hierarchical database dominate writes. In addition, we subsequently show that changing the request distribution does not affect the asymptotic behavior of the protocol. To observe the scalability behavior, the number of nodes participating in the system is increased from 3 to 120, and the aforementioned experiment is repeated for each configuration.

4.1 Comparison between Hierarchical and Non-Hierarchical Locking

A first set of experiments focuses on the comparison of our hierarchical protocol with its non-hierarchical counterpart [28]. Experiments were conducted on Red Hat 7.3 Linux machines with 16 AMD Athlon XP 1800+ processors connected by a full-duplex FastEther TCP/IP switched LAN allowing disjoint point-to-point communications. Once the number of simulated nodes exceeds the number of physically available nodes, multiple processes share a physical node, where a process represents a simulated node. (The second set of experiments on the IBM SP will remove this restriction.) The critical section time, the non-critical code time and the network latency experienced by messages were randomized with mean values of 15 msec, 150 msec and 150 msec, respectively. Randomization occurs with a uniform distribution within a range of $\pm 33.33\%$ for these metrics.

Our protocol requests locks at both table and entry levels. In contrast, Naimi's protocol only acquires the lock at the entry level as it cannot distinguish different locking granularities. To access the entire table, our protocol will acquire a single lock associated with the table in the mode requested. We compare this overhead to two variants of Naimi's protocol at the application level. The first variant performs the *same work* in terms of functionality but requires a larger number of lock requests to do so. The second variant is Naimi's *pure* protocol with fewer number of lock requests, which, on a functional level, is not equivalent

since it provides access to fewer entries. (Instead of sharing a table, only a single entry is shared here.) The second variant serves as the reference for comparison. Figure 11 shows all three ways of organizing data and locks. For example, when the entire table is accessed, our protocol utilizes a single non-intention lock

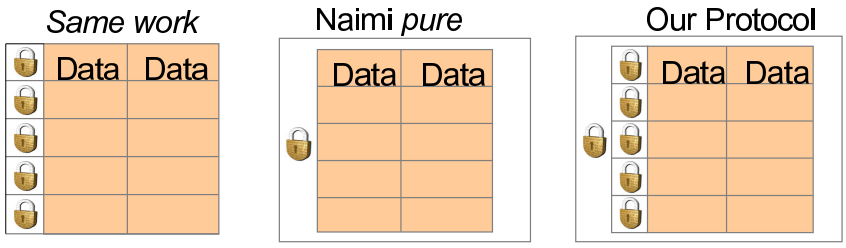


Figure 11: Data Structures for Hierarchical and Non-hierarchical Locking

on the table while Naimi's *same work* version acquires a lock on each individual table entry. Naimi's *pure* version, in contrast, acquires a single lock. When an individual entry of the table is accessed, our protocol has to acquire an intention lock on the table and the non-intention lock on the specific entry while both variants of Naimi acquire only a single lock.

Figure 12 assesses the scalability in terms of the average number of messages being sent for each lock request. We make several interesting observations. First, the message overhead of our protocol is lower than that of Naimi's variants. The lower overhead compared with *same work* is not surprising since more locks are acquired in a sequential fashion leading to long next queues. If we compare with the *pure* version, our protocol performs slightly more lock operations but incurs a lower message overhead. This demonstrates the strengths of our approach: Not only do we provide additional functionality for hierarchical locking, we also do so approximately at 20% fewer messages. Hence, protocol overhead for mode distinction is offset by savings due to local queuing and, most significantly, by allowing children to grant requests.

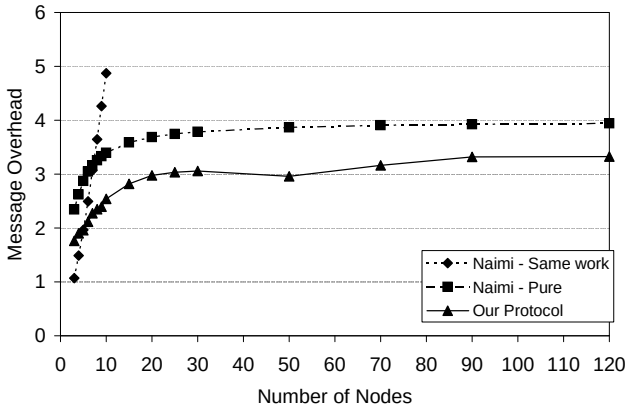


Figure 12: Scalability of Message Overhead

The second observation regards the asymptotic behavior of our protocol. After an initial increase, our protocol results in roughly 3.25 messages per request, even if more and more nodes are issuing requests. The depicted logarithmic behavior makes the protocol highly suitable for large networked environments. In contrast, Naimi's *same work* is superlinear in terms of message complexity, *i.e.*, when providing the same functionality. The multi-granular nature of our protocol combined with the message saving optimizations are the prime causes of this difference, which represents a major contribution of our work.

Figure 13 compares the request latency behavior, *i.e.*, the time elapsed between issuing a request and entering the critical section in response to a grant message. As stated above, in this family of applications, the network latency experienced by the messages might be higher than the network latency on our LAN testbed. We overcome this limitation by resorting to network latency simulation. In case of our protocol, the latency is averaged over all types of requests (*viz.* IR, R, U, IW and W). The average request latency for the same functionality increases superlinearly in case of Naimi's protocol compared to the linear behavior of our protocol. To avoid deadlocks, Naimi's protocol has to acquire locks in a predefined order, which adds a significant amount of overhead resulting in this behavior. The linearly increasing behavior of our protocol is the result of increasing interference with other nodes' requests as number of nodes increases. Hence, a request has to wait for a linearly increasing number of interfering critical sections. (A more detailed analysis of these trends will be discussed later.) Naimi's *pure* protocol has identical asymptotic behavior for the same reason. Our protocol has a better constant factor than that of Naimi's base protocol for a single lock. This is due to savings in lock requests granted by children as well as lock acquisitions that are resolved locally without sending messages when modes are changed in the presence of a prior owned mode (as described in Rule 2), which is compatible.

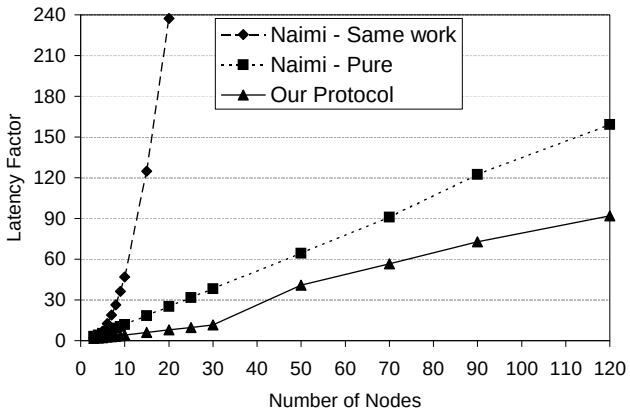


Figure 13: Request Latency (as a factor of point-to-point latency)

Let us return to the issue of linear response time behavior of our protocol. As the message overhead behavior is logarithmic and as each message being exchanged contributes to the response time for the request, one would expect the response time behavior to be logarithmic as well. However, while the message overhead behavior can be used to study the behavior of the protocol, it may not represent the request latency accurately. This is due to the fact that the request latency time has two components: The network delay experienced by each of the messages sent and the queuing delay incurred due to the request being locally queued at other nodes. While the former can be accurately estimated by the message overhead, the latter is not included in the message overhead. This means that response time behavior can be identical (in ideal case) or worse than message overhead behavior. It is therefore crucial to study the second component of the response time. The following experiments focus on assessing this behavior.

4.2 Effects of Concurrency Level and Response Time

A second set of experiments focuses on the effect of concurrency levels on the response time behavior, and at the same time, assesses scalability in high-performance clusters. Experiments were conducted on an IBM SP with 180 nodes, each of them comprised of a 4-way SMP with 375 MHz Power3-II CPUs connected *via* a Colony SPSwitch, an IBM proprietary low-latency interconnect utilized through user-level MPI. Results were obtained for nodes in single-CPU configuration to eliminate noise from application mixes sharing nodes. This limited us to experiments of up to 120 nodes due to administrative constraints. (In the presence of node sharing, we observed large perturbations of our results due to increased network latencies and memory contention.)

In the context of our protocol, the *concurrency level* refers to the number of simultaneously active requests in the entire system. Quantitatively, the concurrency level can be defined as

$$Conc(N) = N \times \frac{CriticalSectionTime}{NonCriticalCodeTime} \tag{4}$$

where N is the number of nodes in the system. Below, we describe experiments under different concurrency levels to study its impact on response time behavior.

Variable Concurrency Level

In these experiments, we kept the critical section time constant at a mean value of 15 msec and varied the length of non-critical code. Results are reported for ratios of one, five, ten and 25 for non-critical code time relative to the critical section time. Both metrics are randomized around these mean values to trigger different request orders and tree configurations in consecutive phases. Randomization occurs with a uniform distribution within a range of $\pm 33.33\%$ for both the metrics. These experiments are designed to assess

the protocol's properties for clusters with native network latencies, *i.e.*, we did not resort to simulation of network latency as in the previous experiments.

Figure 14 depicts the average number of messages incurred by requests for different constant ratios as the number of nodes in the system is increased. We observe an asymptotic overhead of 3.5, 5, 6.5 and around 9 messages for rations one, five, ten and 25, respectively. These results show that message overhead varies between architectures due to interconnect properties when compared with ratio ten results of 3.25 messages for the Linux cluster (critical section time 15 msec, non-critical code time 150 msec). Higher ratios result in lower concurrency level and longer propagation paths, which explains the increased message overhead. Most significantly, the message overhead shows a logarithmic behavior with a low asymptote, which confirms our claim for high scalability in the case of high-performance clusters.

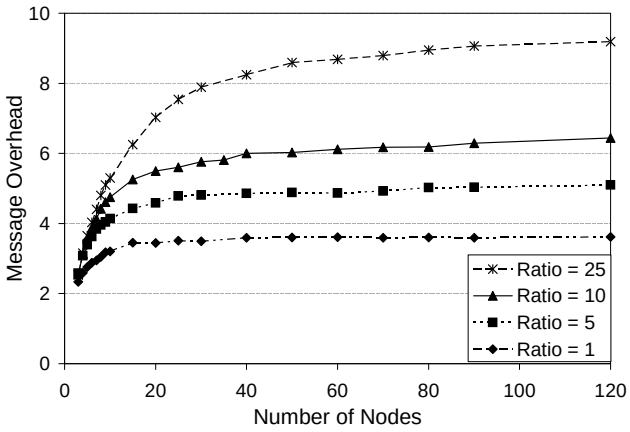


Figure 14: Message Overhead for Varying Non-Critial/Critical Ratios

Figure 15 depicts the average request latency in msec for different ratios as the number of nodes are varied for each ratio. One purpose of this experiment is to demonstrate the effect of the ratio of non-critical code time and critical section time, even though some critical sections cannot be parallelized and are subsequently subject to Ahmdahl's Law. Barring the initial curve, response time is clearly linear as we increase the number of nodes for each ratio. Though lower ratios (higher concurrency) result in much longer response times than that of higher ratios (lower concurrency), the asymptotic behavior is linear for all ratios. While the ratios (concurrency levels) are highly dependent on the type of applications, this result illustrates that, regardless of the application and concurrency level, the response time will be linear. It is also important to understand that, though the ratio is kept constant as we increase the number of nodes, the concurrency level changes due to the factor N in Equation 4.

Another interesting observation is that the curves are initially superlinear with any given ratio. Each curve

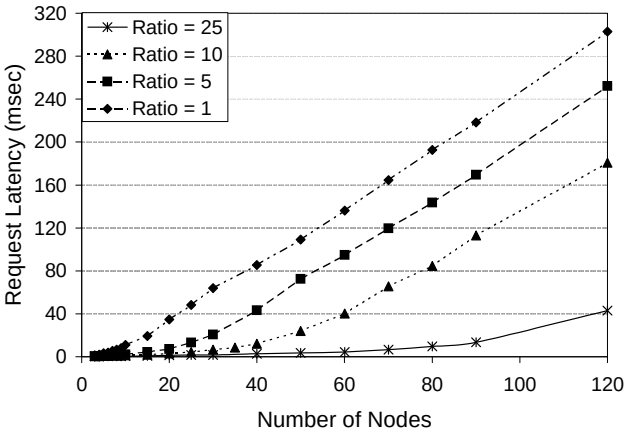


Figure 15: Absolute Request Latency

becomes linear after some point. Specifically, the number of nodes at which the curves become linear is smaller for lower ratios (higher concurrency). To understand this behavior and to answer the questions raised so far about the linear response time behavior, we model the system as follows:

As the number of nodes in the system increases, the average number of simultaneous requests also increases. This causes more conflicts between incompatible request types, thereby increasing the queuing delay. However, the tree height increases logarithmically, and so does the propagation path of requests making the message overhead logarithmic. Hence, with an increase in the number of nodes, an increased queuing delay is added while message overhead increases logarithmically. Ultimately, the response time increases superlinearly and, consequently, the queuing delay should increase superlinearly as well.

Without restricting generality, consider a point in time during the execution of the protocol. The probability of a request of mode M being active at such a point of time is $Np(M)$, where N is the number of nodes in the system, and $p(M)$ is the probability of an M request given by the request distribution discussed before. This is due to the fact that request types are randomized, and each node has an independent randomized stream. Hence, the probability of each of the modes of the requests increases linearly with the number of nodes in the system. Let $c(N)$ be the function representing the number of conflicts present in the system at this point in time. By the compatibility matrix of Table 1, we infer:

$$
\begin{aligned}
c(N) \quad = \quad & Conc(N)p(W)\{Conc(N)p(IR) + Conc(N)p(R) + \\
& Conc(N)p(U) + Conc(N)p(IW) + Conc(N)p(W)\} + \\
& Conc(N)p(IW)\{Conc(N)p(R) + Conc(N)p(U) + \\
& Conc(N)p(W)\} + \\
& Conc(N)p(U)\{Conc(N)p(U) + Conc(N)p(IW) + \\
& Conc(N)p(W)\} + \\
& Conc(N)p(R)\{Conc(N)p(IW) + Conc(N)p(W)\} + \\
& Conc(N)p(IR)\{Conc(N)p(W)\}
\end{aligned}
$$

The probabilities of each request type, *i.e.*, $p(IR), p(R), p(U), p(IW)$ and $p(W)$ depend on the application. For our experiments, these probabilities are 0.80, 0.10, 0.04, 0.05 and 0.01, respectively, as stated before. Hence, the probabilities are constant while $Conc(N)\ \alpha\ N$. By simplifying $c(N)$, we derive

$$c(N) = C_0N^2 + C_1N^2 + C_2N^2 + C_3N^2 + C_4N^2 \tag{5}$$

where C_i are constant factors. Equation 5 indicates that $c(N)\ \epsilon\ \Theta\ (N^2)$. The queue length at nodes will follow the $c(N)$ trend adding a superlinear trend to the logarithmic message overhead. However, the maximum queue length at any node is still limited by N, the number of nodes, in the worst case. This means that, at values of N satisfying $c(N) \leq N^2$, we see the superlinear behavior but after that, it becomes linear as $c(N)$ is bounded by N and N is linearly increasing. This is evident in the results above. Another important point is that the asymptotic behavior of the response time is independent of the request distribution, as seen by the simplification given by Equation 5.

Constant Concurrency Level

The previous section explained the linear behavior (in the asymptotic case) of the response time dependent on the concurrency level $Conc(N)$. Hence, if $Conc(N)$ was kept constant, $c(N)$ were constant as well and the response time behavior would scale logarithmically. To verify this hypothesis, we designed our experiments to keep $Conc(N)$ constant. We modeled this effect by increasing the number of nodes, N, to $N + K$. This elongates the non-critical code time from NCT to $NCT\frac{N+K}{N}$ and allows us to maintain the concurrency at a predefined constant level. Conceptually, the number of active requests at any point of time is constant regardless of the number of nodes in the system.[3]

Figure 16 shows the response time behavior obtained for two different constant concurrency levels, both of which result in linear lac-ency factors. This behavior can be explained by considering the message overhead

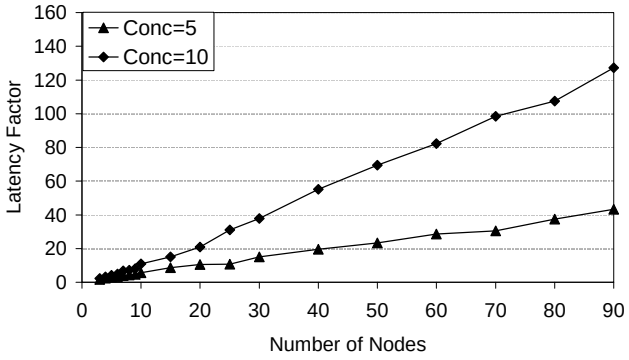


Figure 16: Response Time for Constant Concurrency

[3]We realize that this is not a realistic scenario. However, the purpose of the experiment is to help us assess the response time behavior.

behavior shown in Figure 17. By maintaining the constant concurrency level, the logarithmic behavior of message overhead ceases to persist. This behavior is due to the fact that, even though the conflicts between

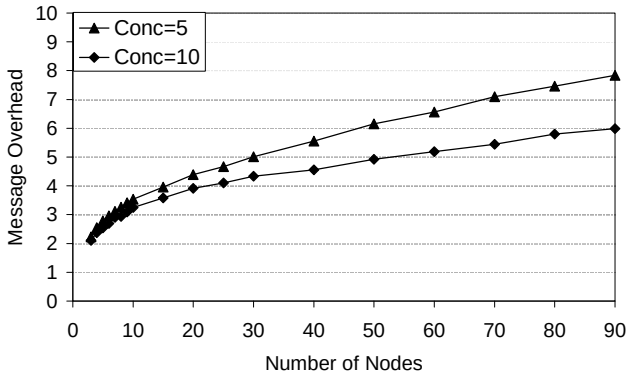


Figure 17: Message Overhead for Constant Concurrency

requests are kept constant, the potential number of granters of the requests in the tree (copyset) also remains constant regardless of the number of nodes in the tree. As a result, longer propagation paths are traversed before requests reach the root and a grant message is sent. With linearly increasing numbers of nodes, this overhead is observed resulting in (close to) linear message overhead. This can be verified by the breakdown of the message overheads shown in Figure 18. While all other types of messages demonstrate logarithmic

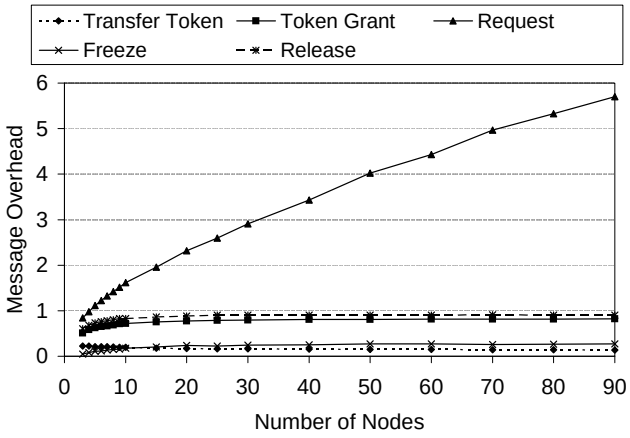


Figure 18: Breakdown of Message Overhead

behavior, the request propagation message increases linearly with an increase in number of nodes. Since the message overhead is one of the factors affecting response time, the response time behavior is also linear in spite of constant queue length.

In conclusion, the analysis of the response time allows us to predict the worst-case response time for the requests and provides bounds that can be exploited by QoS guarantees.

4.3 Message Overhead Breakdown

Figures 19, 20, 21, 22 and 23 depict the change in message overhead for the message types request, grant, transfer token, release and freeze, respectively. In each case, changes are shown for varying number of nodes and different ratios. Explanations about each of these figures illustrate the behavior of the protocol and its properties.

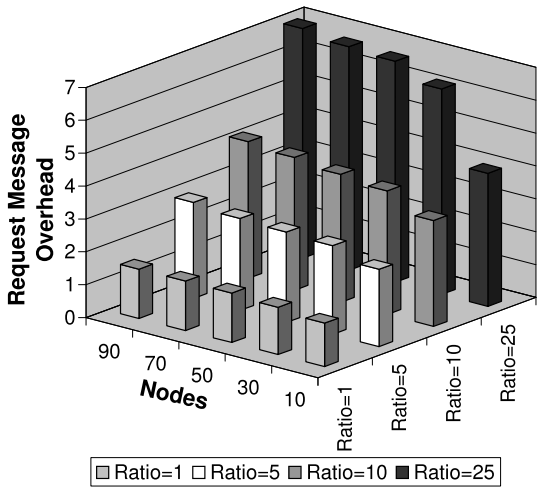


Figure 19: Request Messages

Request messages increase with the number of nodes and with higher rations, as seen in Figure 19. These increases are primarily due to longer propagation paths of requests. For a larger number of nodes, propagation paths increase due to initial request forwarding. With higher ratios, this effect becomes even more significant since propagation path length increases with longer non-critical code fragments. This is the single largest contributor to message overhead (up to seven messages).

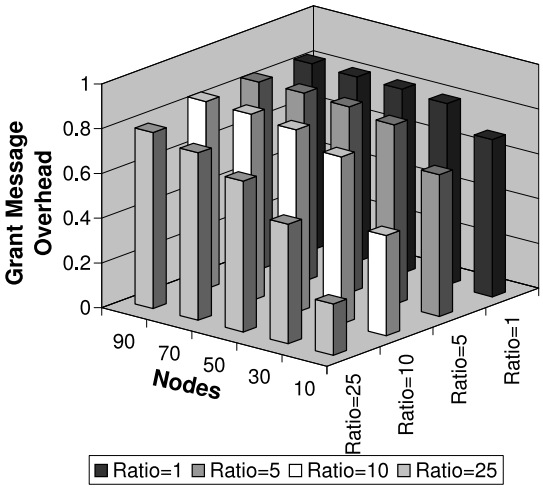


Figure 20: Grant Messages

A similar trend is shown for token grant messages in Figure 20. The more contention, the more concurrent requests can be resolved by allowing children within a copyset to grant requests. The overall contribution is below one message.

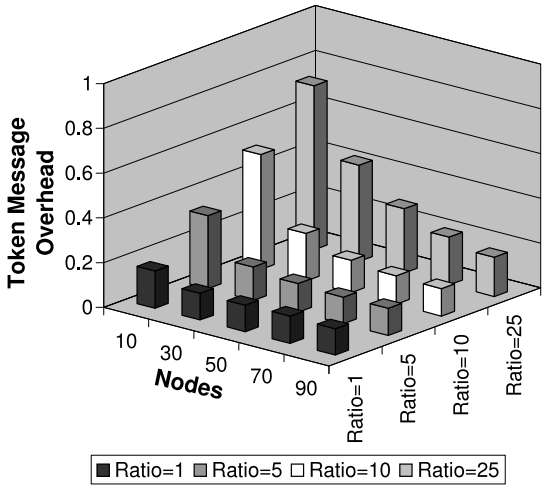


Figure 21: Token Transfer Messages

The token transfers in Figure 21 increase at high ratios when concurrency is low, *i.e.*, during low contention, the root is unlikely to be able to grant a copy . Hence, requests are likely to be resolved by transferring the token. Similarly, contention increases with the number of nodes, which increases the possibilities of copy grants, thereby lowering the transfer of tokens. The total contribution of token transfers remains relatively low (below one message on the average).

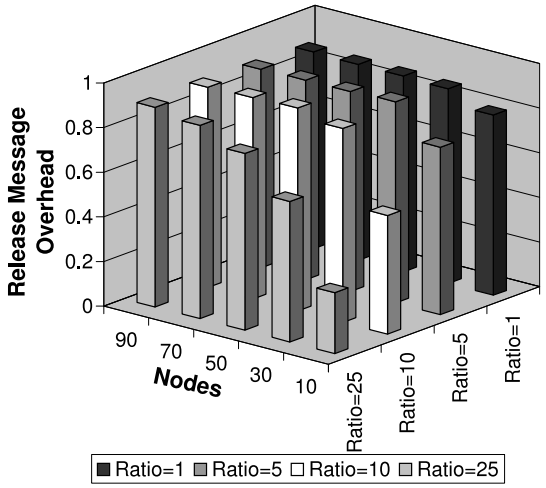


Figure 22: Release Messages

Release messages in Figure 22 show the inverse behavior of token transfers. At low contention (high ratios and few nodes), few release messages are generated since the copy set remains small. This number increases

with the number of nodes and even more dramatically with lower ratios. Its total contribution is still below one message on the average.

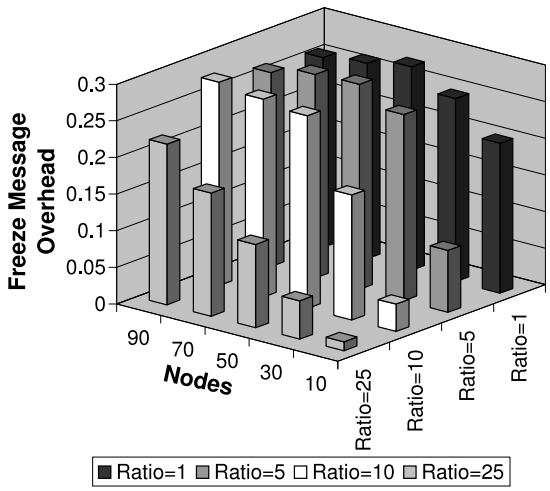


Figure 23: Freeze Messages

Freeze messages (Figure 23) occur predominantly for low ratios implying high contention. The trend is less pronounced for increases in the number of nodes. In general, the share of freeze messages is insignificant (below 0.3 messages). This demonstrates that the price of avoiding starvation is small compared to the overall functionality of the protocol.

Overall, we demonstrated the scalability of our hierarchical locking protocol through its logarithmic message overhead. We observed low latencies in the order of single-digit milliseconds to resolve requests with (realistically) high ratios. This makes our protocol highly suitable for its usage in conjunction with transactions, *e.g.*, in large server farms as well as in large-scale clusters that require redundant computing.

5. RELATED WORK

A number of algorithms exist to solve the problem of mutual exclusion in a distributed environment. Chang [6] and Johnson [18] give an overview and compare the performance of such algorithms. Goscinski [15] proposed a prioritized algorithm based on broadcast requests using a token-passing approach. Chang [5] developed extensions to various algorithms for priority handling that use broadcast messages [35, 32] or fixed logical structures with token passing [30]. Our protocol differs from Chang's extensions of [35] and [30] by not requiring broadcasts or shared memory at all and lower average message complexity, respectively, as detailed below. Fu and Tzeng's mutual exclusion scheme assumes support for shared memory multiprocessors [13]. In contrast, we provide a solution for mutual exclusion in distributed systems where no shared resources exist and communication is realized through message passing. Chang,

Singhal and Liu [7] use a dynamic tree similar to Naimi *et al.* [27, 28]. 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 refer to the older algorithm in this paper. Other mutual exclusion algorithms (without token passing) employ global logical clocks and timestamps [20]. 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 an average 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, which results in dynamic path compression. Furthermore, Raymond needs an average of $O(log\ n)$ messages to send the token to a requester, where our algorithm only requires one message. The modified version of Raymond's algorithm by Fu *et al.* [14] is, in its essence, similar to our local queues but with just one entry. The above algorithms use synchronous message passing with the exception of Raymond's algorithm. In contrast, our algorithm operates asynchronously. It has even been adapted to allow multiple requests per node to provide more concurrency within a multi-threaded environment [24]. None of the above algorithms have been studied with regard to their applicability to concurrency services, to the best of our knowledge.

Hierarchical locks and protocols for concurrency services have been studied in the context of database system [16, 22, 21, 19, 3]. Most concurrency services rely on a centralized approach with a coordinator to arbitrate resource requests or a combination of middle-tier and sink servers [31, 4]. These approaches do not dynamically adapt to resource requests while our protocol does. Our work is unique in this sense. Efforts on predictable ORB behavior have mostly focused on priority support for CORBA's interaction with message passing and thread-level concurrency [29] applicable to real-time database systems [33, 34]. In contrast, we make scalability a first-class property of protocols that implement CORBA-like services, such as hierarchical locking defined through the concurrency services. The applicability of our work reaches from large clusters to server-style computing.

6. CONCLUSION

We presented a novel peer-to-peer protocol for multi-mode hierarchical locking, which is applicable to transaction-style processing and middleware services. We demonstrate high scalability combined with lower response times in high-performance cluster environments. A first set of experiments shows that our protocol overhead is lower than that of a competitive non-hierarchical locking protocol. These benefits are due to several enhancements leading to fewer messages while assuring a higher degree of concurrency. A second set of experiments on an IBM SP shows that the number of messages approaches an asymptote at

15 nodes for ratios up to 10, from which point on the message overhead is in the order of 3-6 messages per request. Higher ratios of 25, *i.e.*, longer non-critical code fragments for constant size critical sections, result in higher message overheads approaching an interpolated asymptote around 9 messages at a node sizes up to 120. At the same time, response times increase linearly at high ratios and nearly linearly at lower ratios with a proportional increase in requests and, consequently, higher concurrency levels. In practice, non-critical fragments are substantially larger than critical sections, *i.e.*, higher ratios are the rule. For higher ratios, our approach yields particularly low response time, *e.g.* for a ratio of 25, response times below 10 msec are observed for critical sections in the range of up to 80 nodes. Our results include detailed assessments of the overhead of the protocol by message type, by concurrency level and ratios of non-critical and critical code sections.

Overall, the high degree of scalability and responsiveness of our protocol is due in large to a high level of concurrency upon resolving requests combined with dynamic path compression for request propagation paths. Besides its technical strengths, our approach is intriguing due to its simplicity and its wide applicability, ranging from large-scale clusters to server-style computing.

7. REFERENCES

[1] Top 500 list. http://www.top500.org/, June 2002.

[2] DOC Group at Washington University. Tao: The ace orb. http://www.cs.wustl.edu/ schmidt/TAO.html.

[3] B. R. Badrinath and Krithi Ramamritham. Performance evaluation of semantics-based multilevel concurrency control protocols. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):163–172, June 1990.

[4] Darrell Brunsch, Carlos O'Ryan, and Douglas C. Schmidt. Designing an efficient and scalable server-side asynchrony model for CORBA. In Cindy Norris and James B. Fenwick Jr., editors, *Proceeding of the Workshop on Optimization of Middleware and Distributed Systems (OM-01)*, volume 36, 6 of *ACM SIGPLAN Notices*, pages 223–229, New York, June 18 2001. ACM Press.

[5] Y. Chang. Design of mutual exclusion algorithms for real-time distributed systems. *J. Information Science and Engineering*, 10(4):527–548, December 1994.

[6] Y. Chang. A simulation study on distributed mutual exclusion. *J. Parallel Distrib. Comput.*, 33(2):107–121, March 1996.

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

[8] N. Desai and F. Mueller. A log(n) multi-mode locking protocol for distributed systems. In *International Parallel and Distributed Processing Symposium*, page (accepted), April 2003.

[9] N. Desai and F. Mueller. Scalable distributed conucrrency services for hierarchical locking. In *International Conference on Distributed Computing Systems*, page (accepted), May 2003.

[10] D. Dolev and D. Malik. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[11] C. Engelmann, S. Scott, and G. Geist. Distributed peer-to-peer control in harness. In *ICCS*, 2002.

[12] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, CA, 1999.

[13] S. Fu and N. Tzeng. A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, pages 628–639, 1997.

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

[15] A. Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *J. Parallel Distrib. Comput.*, 9(1):77–82, May 1990.

[16] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks in a large shared data base. In Douglas S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases*, pages 428–451, Framingham, Massachusetts, 22–24 September 1975. ACM.

[17] Object Management Group. Concurrency service specification. http://www.omg.org/tech-nology/docu-ments/formal/con-currency_service.htm, April 2000.

[18] T. Johnson. A performance comparison of fast distributed mutual exclusion algorithms. In *Proc. 1995 Int. Conf. on Parallel Processing*, pages 258–264, 1995.

[19] U. Kelter. Synchronizing shared abstract data types with intention locks. Technical report, University of Osnabrueck, 1985.

[20] L. Lamport. Time, clocks and ordering of events in distributed systems. *Comm. ACM*, 21(7):558–565, July 1978.

[21] John Lee and Alan Fekete. Multi-granularity locking for nested transactions: A proof using a possibilities mapping. *Acta Informatica*, 33(2):131–152, 1996.

[22] Suh-Yin Lee and Ruey-Long Liou. A multi-granularity locking model for concurrency control in object-oriented database systems. *TKDE*, 8(1):144–156, 1996.

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

[24] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, 1998.

[25] F. Mueller. Priority inheritance and ceilings for distributed mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 340–349, December 1999.

[26] F. Mueller. Fault tolerance for token-based synchronization protocols. In *Workshop on Fault-Tolerant Parallel and Distributed Systems*, April 2001.

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

[28] M. Naimi, M. Trehel, and A. Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, April 1996.

[29] Carlos O'Ryan, Douglas C. Schmidt, Fred Kuhns, Marina Spivak, Jeff Parsons, Irfan Pyarali, and David L. Levine. Evaluating policies and mechanisms to support distributed real-time applications with CORBA. *Concurrency and Computation: Practice and Experience*, 13(7):507–541, June 2001.

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

[31] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), April 1998.

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

[33] Rajendran M. Sivasankaran, John A. Stankovic, Donald F. Towsley, Bhaskar Purimetla, and Krithi Ramamritham. Priority assignment in real-time active databases. *VLDB Journal: Very Large Data Bases*, 5(1):19–34, January 1996.

[34] John A. Stankovic and Sang H. Son. Architecture and object model for distributed object-oriented real-time databases. In *Proceedings of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, April 1998.

[35] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Systems*, 18(12):94–101, December 1993.