A Log(n) Multi-Mode Locking Protocol for Distributed Systems

Nirmit Desai and Frank Mueller
Computer Science, North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu

Abstract

Trends in high-end cluster are to deploy an ever larger number of processors to push the envelope on parallelizing scientific applications. A similar trend exists in server computing where more and more servers are composed into a farm. One of the main challenges in such environments is to achieve scalability of synchronization, specifically for locking. Our work focuses on scalable locking in distributed systems. At 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 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. We observed response times under 2 and 10 msec for up to 25 and 80 nodes for critical sections that are one 25th the size of non-critical code. Our approach benefits in large from high levels of concurrency upon resolving requests and dynamic path compression for request propagation. 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.

1. Introduction

High-end clusters have increased considerably in size over the past years, ranging up to 8,000 processors with future projections over 10,000, *e.g.*, for IBM's Blue Gene Light [1]. We see a similar trend in commercial computing areas, such as server computing. Servers are increasingly organized in ever larger server farms in response to requirements for high availability and faster response times. For sever-style computing, availability and adequate response times are of utmost importance to be able to compete. Replicated information allows load balancing of requests between servers, both within a server farm and across geographically distant servers farms. Furthermore, distributed agreement, originally designed for distributed database systems, has recently been studied for cluster computing [7, 8]. 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 [9, 13, 12, 2]. 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. We present a hierarchical locking protocol that supports a high degree of concurrency for a number of access modes, *i.e.*, 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 shown by our experimental results.

The paper is structured as follows. First, a brief review a non-hierarchical locking protocol is given for later comparison. Next, the novel peer-to-peer hierarchical locking protocol is defined step by step. Then, experiments to compare the two protocols and assess their properties in large systems are conducted. Finally, related work is discussed and the paper's contributions are summarized.

2. Mutual Exclusion for Distributed Systems

Concurrent requests to access shared resources have to be arbitrated by means of mutual exclusion, *e.g.*, to provide hierarchical locking and support transaction processing, which is realized via message passing for distributed systems. Several non-hierarchical locking protocols, distinguished as broadcast *vs.* point-to-point and token *vs.* non-token based, have been proposed [4]. 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 [17]. 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 Fig. 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. After a request from C, T returns from its critical section and sends the token along the *next* vertex to A and deletes the *next* pointer.

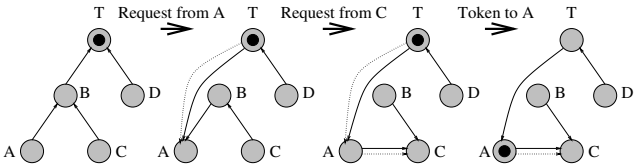


Figure 1. Non-hierarchical Locking

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 [15, 16]. In this work, we develop a novel protocol for hierarchical locking building on our past results and demonstrate its suitability to deliver short latencies and low message overhead in cluster environments.

3. Peer-to-Peer Hierarchical Locking

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 [10]. 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.

3.1. Compatibility between Lock Modes

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 [9, 13]. 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 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. Tab. 1(a) shows when L_R in modes M_2 cannot be granted by another node in mode M_1 due to incompatibility according to the specification of concurrency services [10].

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 Tab. 1(a).

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. 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. Queues, Copysets, Grants and Release

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

Mode M_2:	(a) Incomptabible					(b) No Child Grant					(c) Queue/Forward					(d) Freezing Modes at Token				
Mode M_1	IR	R	U	I	W	IR	R	U	I	W	IR	R	U	IW	W	IR	R	U	I	W
No lock – Φ						X	X	X	X	X	F	F	F	F	F					
Intent Read – IR					X		X	X	X	X	Q	F	F	F	F					IR,R,U,IW
Read – R				X	X			X	X	X	F	Q	F	F	F				R,U	IR,R,U
Upgrade – U			X	X	X			X	X	X	F	F	Q	Q	Q				R	IR, R
Intent Write – IW		X	X		X		X	X		X	F	F	F	Q	F		IW	IW		IR, IW
Write – W	X	X	X	X	X	X	X	X	X	X	Q	Q	Q	Q	Q					

Table 1. Rules of Hierarchical Locking for Mode M_1 relative to Mode M_2

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 maximize the degree of concurrency.

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 [14].

Definition 4: The *copyset* of a node is a set of nodes simultaneously holding a common lock, which is owned by the parent in a mode stronger than the children's modes.

In the following, we refer to a token owner and its children to establish the relation of a copyset. The next rules govern the dispatching and granting 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. Otherwise, the local copyset is updated and the critical section is entered without messages.

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 protocol specification further requires:

in case 1: that 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. 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.)

Tab. 1(b) depicts legal owned modes M_1 of non-token nodes for granting a mode M_2 according to Rule 3, indicated by the absence of an X. For the token node, compatibility represents a necessary and sufficient condition. Thus, access is subject to Rule 1 in conjunction with Tab. 1(a).

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. A dotted arc from A to B depicts ancestor relations known by A but not (yet) by B while solid arcs indicate mutual awareness. The token is depicted as a solid circle inside a node. Example: Consider the ini-

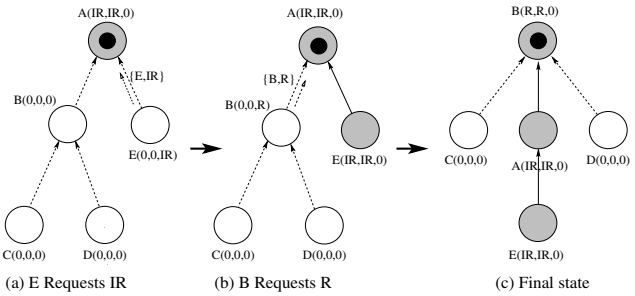


Figure 2. Request Granting

tial state as shown in Fig. 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.

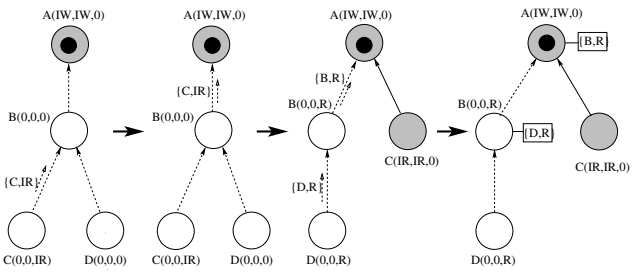
When a node issues a request that cannot be granted right away due to mode incompatibility, Rule 4 applies:

Rule 4:

1. If a non-token node with pending requests for mode M_1 cannot grant a new request for M_2, it either forwards (F) the request to its parent or locally queues (Q) according to Tab. 1(c).
2. If the token node cannot grant a request, it will queue the request locally regardless of its pending requests.

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. The aim here is to queue as many

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



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

Figure 3. Queue/Forward Request

Example: In Fig. 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 Tab. 1(c) 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 Tab. 1(c) 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).

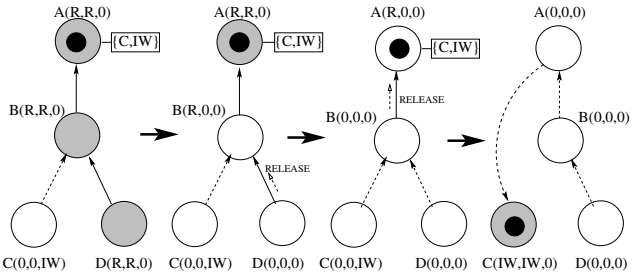
The following rule governs the handling of lock releases.

Rule 5:

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 ensures queued locks are served upon a release based on local queues instead of a distributed queue employed in [17]. The second part results in release messages to the root of the copyset if necessary, *i.e.*, when modes change. 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 Fig. 4(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) 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).



(a) B Releases R (b) D Releases R (c) A Releases R (d) Final state

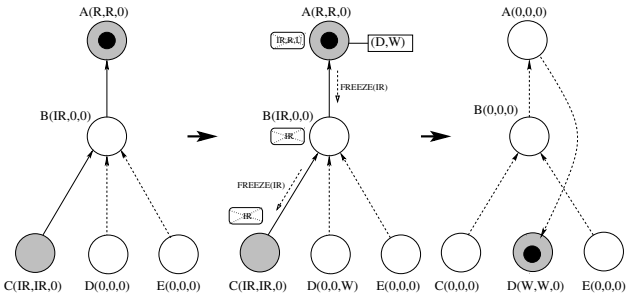
Figure 4. Lock Release

3.3. Fairness and Absence of Starvation

The objective of providing a high degree of concurrency conflicts with common concurrency guarantees. 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 Fig. 5(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 Fig. 5(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.

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



(a) D requests W (b) A sends Freeze (c) A,B,C Release R,IR,IR

Figure 5. Frozen Modes

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.

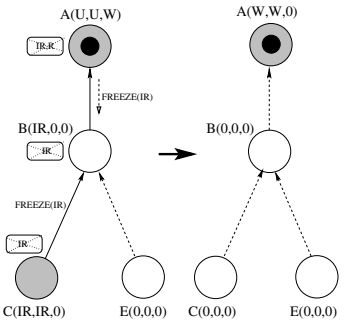
In order to extend fairness beyond the token holder, mode freezing is transitively extended to the copyset where required by modes. This ensures that potential granters of any mode incompatible with the requested mode will no longer grant such requests. We ensure transitive freezing by the operational specification for this rule, which states that the token node notify children about the frozen modes. This rule supplements Rules 2 and 3. Tab. 1(d) depicts the frozen modes for all combinations. For example, if the token node is owning a lock in $M_1 = R$ and a $M_2 = W$ request is received and queued locally (as depicted in Fig. 5(b)) then IR, R and U are the modes to be frozen at the token node. Via the freezing mechanism, all children and grandchildren will ultimately release the modes. Immediate children send a release message to its parent including the token node. This preserves the FIFO policy.

3.4. Upgrade Request

Upgrade locks facilitate the prevention of potential deadlocks by very simple means if the resulting reduction in the concurrency level is acceptable [10]. 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 Fig. 6(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.



(a) A requests Upgrade (b) Upgrade complete

Figure 6. Request Upgrade

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. These rules and tables serve as a specification of the algorithm (see [6] for details). Rules and tables greatly facilitate the implementation since it reduces a seemingly complex protocol to a small set of rules defined over lookup tables in practice.

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, *i.e.*, local knowledge is sufficient to ensure correctness.

4. Measurements

In performance evaluation, we first evaluate the performance of our protocol relative to the protocol by Naimi et al. [17], which has the best known average case message complexity of $O(log\ n)$. [1] Second and foremost, we demonstrate the feasibility of the protocol for high performance clusters by investigating the effects of node scaling and the

[1] We could not find other protocols for distributed mutual exclusion with hierarchical locking models that follow a peer-to-peer paradigm, and a comparison with centralized protocols did not seem fair due to the inefficiency of client-server approaches when scalability is to be assessed.

non-critical-to-critical ratio. In the experiments, nodes in the different test beds execute an instance of an application (multi-airlines reservation) on top of the protocol. The data representing ticket prices is stored in a 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) requests the locks iteratively. The length of a critical section, the inter-request idle time and the network latency experienced by messages were randomized with different mean values, depending on the 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. These parameters reflect the typical frequency of request types for such applications in practice where reads dominate writes. To observe the scalability behavior, the number of nodes was varied.

4.1. Comparison between Hierarchical and Non-Hierarchical Locking

The first set of experiments focuses on a comparison of our hierarchical protocol with its non-hierarchical counterpart [17]. Experiments were conducted under Red Hat 7.3 Linux on 16 AMD Athlon XP 1800+ machines connected by a full-duplex FastEther switch utilized through TCP/IP, where additional nodes were simulated by processes sharing a machine. The length of a critical section, the inter-request idle time and the network latency experienced by messages were randomized with mean values of 15, 150 and 150 msec, respectively.

While our protocol requests locks at both table and entry levels, 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 resembles Naimi's *pure* protocol with an equivalent number of lock requests, which, on a functional level, is not equivalent since it provides access to fewer entries. In particular, when we acquire a single lock on the table with our protocol in a non-intention mode, we request locks associated with each individual table entry for Naimi's same work. For Naimi's pure version, only a single lock is requested. In contrast, table locking in intention mode for our protocol does not require any locks in either of Naimi's variants, it suffices to only request locks at the entry level for all protocol variants.

Fig. 7 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* does not surprise since more locks are acquired in a sequential fashion leading to long next queues. If we compare with the *pure* version, our protocol performs the same number of 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.

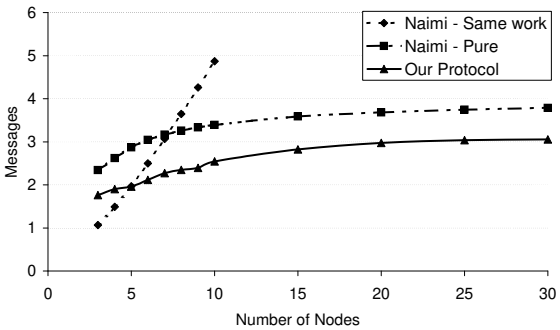


Figure 7. Scalability of Message Overhead

The second observation regards the asymptotic behavior of our protocol. After an initial increase, our protocol results in roughly three messages, 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.

Fig. 8 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 utilizing simulation with the aforementioned parameters for network latency. Latencies are averaged over all types of requests (*viz.* IR, R, U, IW and W). The average request latency

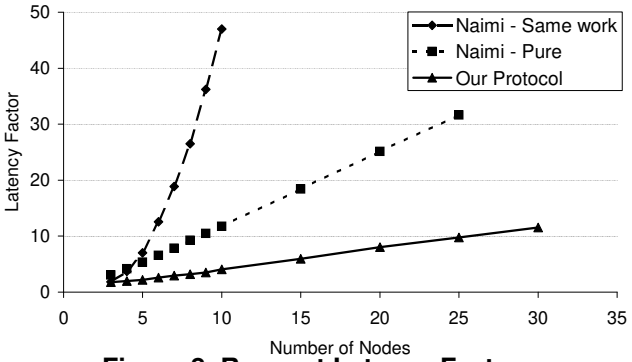


Figure 8. Request Latency Factor

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, as explained next.

4.2. Evaluation for High-Performance Clusters

The second set of experiments focuses on observed latencies and scalability in high-performance clusters. Experiments were conducted on an IBM SP with 180 nodes (4-way SMPs with 375 MHz Power3-II CPUs) connected *via* an IBM proprietary Colony SPSwitch 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 experiments, we kept the critical section length constant at 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 relative to the critical section length. Of course, both metrics are still randomized around their average values to trigger different request orders in consecutive phases.

Fig. 9 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

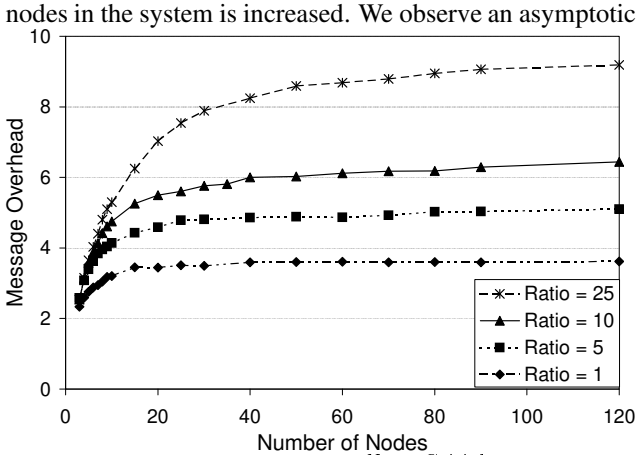


Figure 9. Messages for $\frac{Non-Critial}{Critical}$ Ratios

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.

Fig. 10 depicts the average request latency for different ratios as the number of nodes are varied for each ratio.

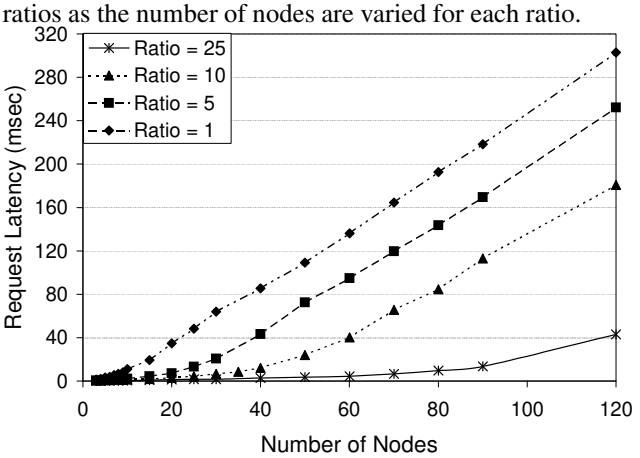


Figure 10. Absolute Request Latency

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. Another interesting observation is that the curves are initially superlinear with any given ratio. Each curve becomes linear after some point. Specifically, the number of nodes at which the curves become linear is smaller for lower ratios (higher concurrency). By modeling response times in terms of network latencies and queuing delays, we analytically derived complexity bounds of the protocol [6]. The model and additional measurements indicate that the superlinear behavior is due to queuing delays, which are initially dominating the algorithm. Overall, we demonstrated the scalability of our hierarchical locking protocol through its logarithmic message overhead and observed low latencies in the order of single-digit milliseconds to resolve requests. 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 [4] and Johnson [11] give an overview and compare the performance of such algorithms, such as Raymond's fixed logical structures with token passing [18]. Chang, Singhal and Liu [5] use a dynamic tree similar to Naimi *et al.* [17]. Most

of the prior work, except Raymond's and Naimi's, have an average message complexity larger than $O(log\ n)$ for a request. Raymond's algorithm uses a fixed logical structure while we use a dynamic one, which results in dynamic path compression. 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 [9, 13, 12, 2]. Most concurrency services rely on a centralized approach with a coordinator to arbitrate resource requests or a combination of middle-tier and sink servers [19, 3]. These approaches do not dynamically adapt to resource requests while our protocol does. Our work is unique in the sense that we make scalability a first-class property of protocols that implement CORBA-like services, such as hierarchical locking defined through the concurrency services.

6. Conclusion

We presented a novel 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 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 3-5 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 8 messages at a node sizes around 30. 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 2 msec are observed for critical sections in the range of up to 25 nodes. 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.

References

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

[2] B. R. Badrinath and K. 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.

[3] D. Brunsch, C. O'Ryan, and D. C. Schmidt. Designing an efficient and scalable server-side asynchrony model for CORBA. In C. Norris and J. B. F. Jr., editors, *Workshop on Optimization of Middleware and Distributed Systems (OM-01)*, volume 36, 6 of *SIGPLAN Notices*, pages 223–229, June 18 2001.

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

[5] 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.

[6] N. Desai and F. Mueller. Scalable hierarchical locking for distributed systems. *J. Parallel Distrib. Comput.*, page (submitted), 2003.

[7] D. Dolev and D. Malik. The transis approach to high availability cluster communication. *Commun. ACM*, 39(4):64–70, Apr. 1996.

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

[9] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks in a large shared data base. In D. S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases*, pages 428–451, Framingham, Massachusetts, 22–24 Sept. 1975. ACM.

[10] O. M. Group. Concurrency service specification. http://www.omg.org/tech-nology/docu-ments/formal/concurrency_service.htm, Apr. 2000.

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

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

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

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

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

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

[17] 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, Apr. 1996.

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

[19] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), Apr. 1998.