Scalable Distributed Concurrency Services for Hierarchical Locking

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

Abstract

Middleware components are becoming increasingly important as applications share computational resources in distributed environments. One of the main challenges in such environments is to achieve scalability of concurrency control. Existing concurrency protocols lack scalability. Scalability enables resource sharing and computing with distributed objects in systems with a large number of nodes. We have designed and implemented a novel, scalable and fully decentralized middleware concurrency control protocol. Our experiments on a Linux cluster indicate that an average number of three messages is required per lock request on a system with as many as 120, which is a logarithmic asymptote. At the same time, the response time for the requests scales linearly with the increase in concurrency level. A comparison to another scalable concurrency protocol shows that our protocol results in significantly superior asymptotic savings in message overhead and response time for large number of nodes. While our approach follows the specification of general CORBA concurrency services for large-scale data and object repositories, the principles are applicable to any distributed concurrency services and transaction models. The results of this work impact scalability for distributed computing facilities ranging from embedded computing with distributed objects over peer-to-peer computing environments to arbitrating accesses in very large database environments.

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 their 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 constitutes the enabling technology for distributed object services, such as resource arbitration in distributed systems. 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. Its sole reliance on decentralized approaches makes it widely applicable to distributed objects and peer-to-peer paradigms in general, *e.g.*, in the context of web caching or embedded computing with distributed objects. In the following, we outline the protocol details and compare our results with other protocols to demonstrate feasibility and performance. Our approach is unique in several respects. We rely on fully decentralized protocols instead of a client/server model to ensure scalability, which is of utmost importance for large-scale distributed computing environments. We support request arbitration through strict priority ordering to ensure fairness. And finally, our model is compatible with hierarchical locks, such as those used in transaction processing. As a result, our protocol distinguishes itself from others through its low message complexity combined with a high degree of concurrency for issued requests. This paper is structured as follows. At first, a scalable protocol for distributed mutual exclusion is introduced. Then, our new protocol is presented through a set of rules and examples. Next, a performance study assesses the quantitative benefits of this novel protocol. Finally, we discuss related work and summarize our contributions.

2. Distributed Mutual Exclusion

Common resources in a distributed environment may require that they are used in mutual exclusion, *e.g.*, to provide hierarchical locking and transaction processing. In a distributed environment, mutual exclusion has to be provided via a series of messages passed between nodes that share a certain resource. Several algorithms to solve mutual exclusion for distributed systems have been developed [3]. 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.

In token-based algorithms for mutual exclusion, a single token, representing the lock object, is passed between nodes within the system [14]. 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 holds the token for mutual exclusion. The 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.

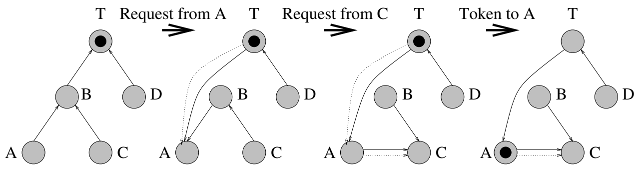


Figure 1. Non-hierarchical Locking

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 number of nodes. The model assures that requests are ordered FIFO, which supports the predictable response times for requests [11, 12]. Our contribution in prior work was to alleviate this shortcoming with a novel protocol. In this proposal, we develop a novel protocol for hierarchical locking building on our past results.

3. Protocol for Scalable Concurrency Services

In the following, we present a protocol for mutual exclusion with a number of access modes in support of concurrency services for distributed computing [6]. This protocol is unique in its low message complexity and high concurrency level. These attributes provide scalability and low response times for hierarchical locking requests at the same time. Here also, nodes form a logical tree structure by maintaining their local parent pointers. However, the next pointers are not needed. The root node of the tree holds the token and is referred to as the *token* node. All other nodes are *non-token* nodes. Nodes having either the token or a granted copy can access (Read, Write) shared resources with the mode of grant (R/W).

3.1. Concurrency of Locking Modes

An exclusive locking scheme was introduced in the previous section. However, in practice, systems generally require the distinction between lock modes to increase the potential for parallelism. Most systems distinguish read (R) locks and write (W) locks with parallel and exclusive access, respectively. A special case is an upgrade (U) lock, which represents an exclusive read lock that is followed by an upgrade request for a write lock. Upgrade locks avoid conflicts between threads that intend to modify (write) objects after having held a read lock. Hierarchical locking schemes enhance parallelism by distinguishing 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. Hierarchical locking introduces *intent* locks for read (IR) and write (IW) access at a higher level [5, 9]. By exploiting intent locks, multiple threads may first acquire an intent write (IW) lock on a database and then a disjoint write (W) lock 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. 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. In general, lock requests may proceed concurrently if modes for a lock are compatible. For the above-mentioned five lock modes, lock compatibility is defined by concurrency services supported by the operating system, the database or distributed object middleware. Compatibility between two lock modes indicates that these locks may be acquired concurrently, *i.e.*, the compatibility property reduces serialization of locks by supporting an increased level of parallelism. In the following, we refer to the Concurrency Services of CORBA, a *de facto* standard hierarchical locking model, as the underlying model without restricting the generality of our protocol [6].

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 [6].

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).

Mode M_2:	(a) Incomptabible					(b) No Child Grant				
Mode M_1	IR	R	U	IW	W	IR	R	U	IW	W
Φ						X	X	X	X	X
IR					X		X	X	X	X
R				X	X			X	X	X
U			X	X	X			X	X	X
IW		X	X		X		X	X		X
W	X	X	X	X	X	X	X	X	X	X

Table 1. Rules for Comaptibility and Granting

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:

$$\Phi < IR < R < U = IW < W \quad (1)$$

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. Granting Lock Modes

Our protocol for hierarchical locking with the above lock modes follows a token-based approach but differs from previous approaches in several ways. While previous work relied on a distributed FIFO queue, we combine multiple local queues for incompatible requests. 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 [10].

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.

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. 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).

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. 2(a).
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.

The following rule governs the handling of lock releases.

Rule 5:

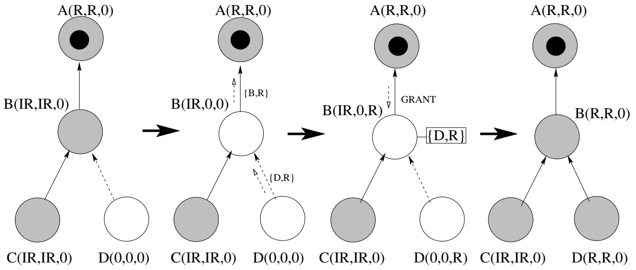
1. When the token node releases a lock or receives release from one of its children, it considers the locally queued requests for granting under constraints of Rule 3.
2. When a non-token node A releases a lock or receives a release in some mode M_R, it will send a release message to its parent only 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 [14]. 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.

M_2	(a) Queue/Forward					(b) Freezing Modes at Token				
M_1	IR	R	U	IW	W	IR	R	U	IW	W
Φ	F	F	F	F	F					
IR	Q	F	F	F	F					IR,R,U,IW
R	F	Q	F	F	F				R,U	IR,R,U
U	F	F	Q	Q	Q				R	IR, R
IW	F	F	F	Q	F		IW	IW		IR, IW
W	Q	Q	Q	Q	Q					

Table 2. Rules for Queuing and Freezing

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, 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.



(a) Initial State (b) B Released IR (c) B,D Issued R (d) Final State

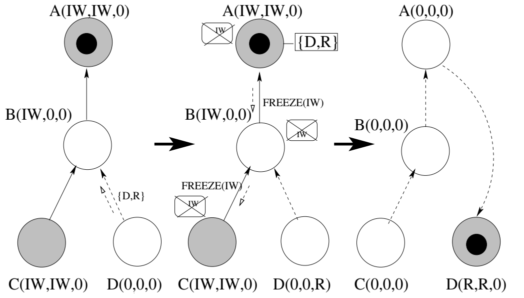
Figure 2. Example for Grant, Release, Queue

Example: Consider the initial configuration of nodes as depicted in Figure 2(a). Suppose B releases IR. It will not send a release message to A due to Rule 5.2. It still owns IR due to C holding/owning IR, as shown in Figure 2(b). Next, suppose B requests R and D requests R in that order. B sends a request $\{B, R\}$ to A according to Rule 2, and while $\{B, R\}$ is in transit, B receives the request $\{D, R\}$ which is sent due to Rule 2. B will queue the request locally according to Rules 3.1 and 4.1, as shown in Figure 2(c). Eventually, A receives $\{B, R\}$, grants a copy according to Rule 3.2, and B, at the receipt of a grant token message, can grant the locally queued $\{D, R\}$ request according to Rules 3.1 and 4.OP. The final state is shown in Figure 2(d). Observe the significant amount of savings in terms of the number of messages being passed due to these rules compared to eager request/release messages.

3.3. Fair Queuing

The protocol described so far 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 undesirable and may lead to starvation due to the described race.

Consider a request by D for mode R in the state of Figure 3(a). $\{D, R\}$ will reach A according to the rules described above, and A will queue it according to Rules 3.2 and 4.2, as shown in Figure 3(b). Once A and C release their locks for IW, respectively, the token will be forwarded to D due to Rules 4.OP and 3.2 , as depicted in Figure 3(c). While D waits for $\{D, R\}$ to advance, A may grant other IW requests from other nodes according to Rule 3.2. As mentioned before, accepting IW requests potentially violates the FIFO policy since $\{D, R\}$ arrived first. After $\{D, R\}$ is received, we should be waiting for release of IW modes since they are not compatible with R. This prevents A from granting the pending $\{D, R\}$ request. If, however, subsequent IW requests are granted, the $\{D, R\}$ request may starve. To avoid starvation (and ensure fair FIFO queuing), the token node A, after receiving $\{D, R\}$, will not grant any other requests compatible with the waiting request (R in this case). Other modes (IW in this case) are said to be *frozen* when certain requests (R in this case) are received, depending on the mode owned by the token node (IW in this case).



(a) D Requests R (b) Frozen State (c) C,A Released IW

Figure 3. Example for Frozen Modes

Another problem is posed by the following scenario. Potential granters of requests, which are incompatible with the waiting request (B and C in this case), may continue granting subsequent requests (IW in this case) without knowing about the waiting request at token node ($\{D, R\}$ in this case). As a result, the token node will await a release, which will be delayed. Hence, the waiting request may starve.

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

The operational specification for our protocol further requires that the token node notify children about the frozen modes. This ensures that potential granters of any mode incompatible with the requested mode will no longer grant such requests.

This rule supplements Rules 1 and 3. Tab. 2(b) depicts an enumeration of frozen modes for all combinations. For example, if the token node is owning a lock in IW and a R request is received and queued locally (as depicted in Figure 3(b)) then IW is the modes to be frozen at the token node. Through this freezing mechanism, ultimately, all children and grandchildren will release the modes, and a re-

```
RequestLock(M_R)
    if Self ≠ Token_Node then
        if M_O ≥ M_R ∧ compatible(M_O, M_R) ∧
            Frozen_Modes then [R 2]
            Acquire Lock
            Copyset ← Copyset + M_R
        else [R 2]
            M_P = M_R
            Send Request to Parent
    else
        if compatible(M_O, M_R) ∧
            Frozen_Modes then [R 3.2]
            Acquire Lock
            Copyset ← Copyset + M_R
        else [R 4.2]
            Queue ← Queue + M_R
            M_P ← M_R
            Update Frozen_Modes [T 2.b]
            Send Freeze to Children if required (^a)
HandleRequest(M_R)
    if Self ≠ Token_node then
        if grantable(M_O, M_R) then [R 3.1, T 1.b]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send grant to Requester
        else if queuable(M_P, M_R) then [R 4.1, T 2.a]
            Queue ← Queue + M_R
        else [R 4.1, T 2.a]
            Send Request to Parent
    else
        if tokenable(M_O, M_R) then [R 3.2, T 1.a]
            if Requester ∈ Children then
                Children ← Children - Requester
            Parent ← Requester
            Send Token to Requester
        else if grantable(M_O, M_R) then [R 3.2, T 1.a]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send Grant to Requester
        else [R 4.2]
            Queue ← Queue + M_R
            Update Frozen_Modes [T 2.b]
            Send Freeze to Children if required (^a)
ReceiveGrant()
    Parent ← Sender [R 3.1]
    Copyset ← Copyset + M_P
    M_H ← M_P, MP ← φ
    Frozen_Modes ← Frozen_Modes + Parent_Frozen
    Check_requests_on_queue [R 4]
ReceiveToken()
    Parent ← NULL
    Copyset ← Copyset + M_P
    M_H ← M_P, M_P ← φ
    Children ← Children + Sender if required [R 2] (^b)
    Merge Queues (^c)
    Check_requests_on_queue [R 4]
RequestUnlock()
    Copyset ← Copyset - M_H
    M_H ← φ
    if Changed Mode of Self then [R 5]
        Send Release to Parent
    Check_request_on_queue [R 5]
HandleRelease(M_R, M_O) [R 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
HandleFreeze(Modes)[R6]
    Frozen_Modes ← Frozen_Modes + Modes
    Send Freeze to Children if required (^a)
RequestUpgrade() [R 7]
    if Copyset ← φ then
        Release U
        Acquire W
    else
        M_P ← W
        Update Frozen_Modes [T 2.b]
        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 [R 3.2, T 1.a]
            if Requester ∈ Children then
                Children ← Children - Requester
            Parent ← Requester
            Send Token to Requester
        else if grantable(M_O, M_R) then [R 3.2, T 1.b]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send Grant to Requester
        else
            Update Frozen_Modes [T 2.b]
            exit_loop
```

[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 [11].

Figure 4. Pseudocode of the Protocol

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

3.4. Upgrade Mode Precedes Write Mode

An upgrade lock is a special read lock that is used if a node needs to read data that, subsequently, will be written. An upgrade lock conflicts with upgrade locks held by other nodes. An upgrade lock successfully obtained by a node indicates that no other upgrade lock is currently being held, which prevents any new upgrade locks from being obtained on the same data. Upgrade locks may be utilized to prevent the potential for deadlocks [6]. A node wishing to read and, subsequently, write the data will request the U mode to read the data in exclusive mode instead of a shared R mode. Once the read operation is performed, an upgrade to W mode is requested. 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).

The algorithmic details of the protocol are depicted in Figure 4. 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.

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. Performance Evaluation

We evaluate the performance of our protocol and compare it with the protocol by Naimi et al. [14], which has the best known average case message complexity, $O(log\,n)$. To the best of our knowledge, a protocol for distributed mutual exclusion with hierarchical locking model does not exist.

The protocol was implemented and the experiments were conducted on Red Hat 7.3 Linux with AMD Athlon XP 1800+ machines connected by a FastEther TCP/IP LAN with a full-duplex switch allowing disjoint point-to-point communication in parallel. Each node in the system ran an instance of an application, a multi-airline reservation system, exploiting the protocol. The data about the ticket prices was stored in a table and shared amongst all the nodes. In case of our protocol, each entry of the data was associated with a lock. In addition, the entire table was associated with another lock (higher level of granularity). For Naimi's protocol, the lock for the entire table is not needed as this protocol does not distinguish between different levels of lock granularities. Each application instance will request the locks iteratively. 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 msec, 150 msec and 150 msec, respectively. 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. Each time, the aforementioned experiment was repeated.

To access the entire table, our protocol will acquire a single lock associated with the table in the mode requested. To achieve the same functionality, Naimi's protocol will acquire all the locks associated with the individual table entries one by one, in order. To access a single entry in the table, our protocol will acquire the lock associated with the table in intention mode and the lock associated with the requested table entry in the requested mode. In contrast, Naimi's protocol will directly acquire the lock associated with the entry itself. The results obtained in this manner are presented as Naimi's *same work* when comparing to our protocol.

We also compare to Naimi's *pure* protocol. In this case, the multi-granularity is eliminated, *i.e.*, a single lock in the system is requested by the participating nodes. This is equivalent to the original results presented by Naimi et al. [14]. The results below compare the protocols on the basis of Naimi's *same work* as well as Naimi's *pure* requests. The comparison with the *same work* is important and of particular interest. The comparison with the *pure* version serves as a baseline for the purpose of assessing the overhead and additional benefits of our protocol. In other words, we can only compare protocol overhead with *pure*. However, *pure* cannot provide the functionality of our protocol with the simulated work load.

Figure 5 assesses the scalability of the protocol as the number of nodes in the system is increased. Message overhead here refers to the average number of messages being sent for each lock request. The message overhead of our

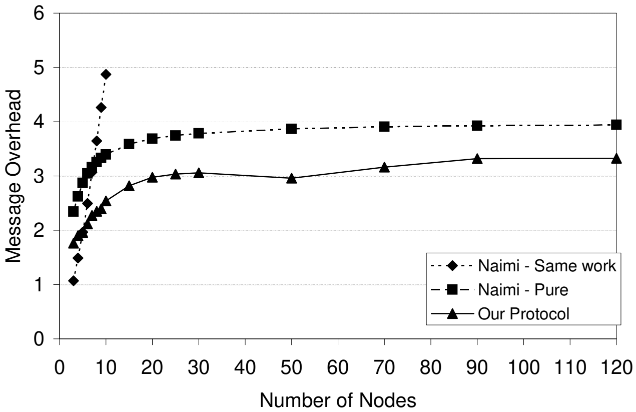


Figure 5. Scalability Behavior

protocol remains constant after an initial increase reaching an asymptotic threshold of about 3 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. 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. Also, Naimi's *pure* protocol has the same asymptotic behavior and scales logarithmically but at a slightly higher cost of up to 4 messages. This shows that, even though our protocol is actually providing more functionality, it still outperforms the base protocol without support for hierarchical locking by a 20% lower message rate. This behavior stems from the use of local queues and, most significantly, from allowing children to grant requests.

Figure 6 compares the request latency behavior. The latency is depicted as a multiple of the base average network latency (150 msec). 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 Naimi's *same work* increases superlinearly 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 the number of nodes increases. Hence, a request has to wait for a linearly increasing number of interfering critical sections. 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 lock requests granted by children as well as lock acquisitions that are resolved locally without messages when modes are changed in the presence of a prior owned mode (as described in Rule 2), which is compatible.

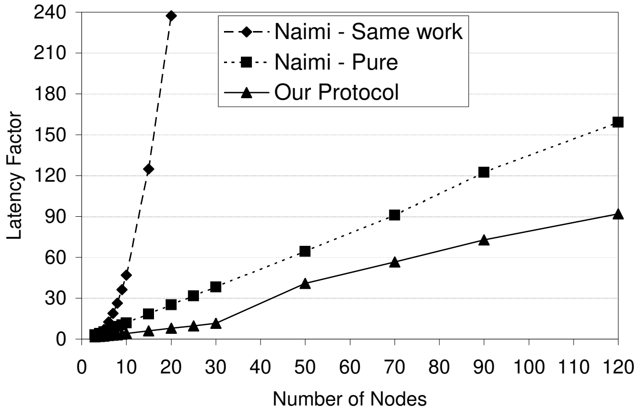


Figure 6. Request Latency (as a factor of point-to-point latency)

Figure 7 shows the breakdown of the message overhead in terms of individual types of messages. The number of request messages is increasing remarkably for small numbers of nodes. This is due to the fact that, as the number of nodes increases, the height of the tree within the copyset structures increases. Hence, requests experience longer propagation paths. The number of messages remains constant for larger number of nodes as the probability of the request being queued locally or being granted a copy by a node in the middle of the propagation path also increases.

The number of transfer token messages is decreasing initially before reaching a constant level. A transfer of token

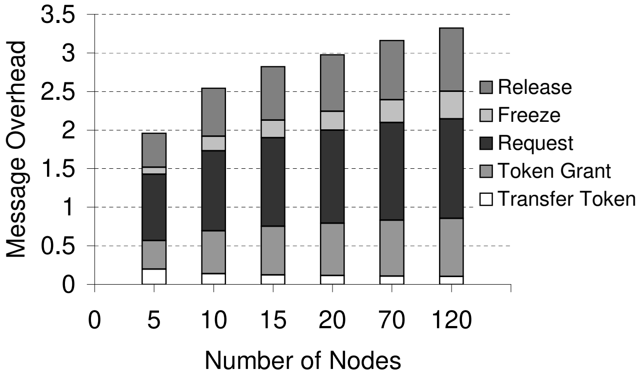


Figure 7. Message Behavior

is only possible if the requested mode is not frozen. But with larger number of nodes, longer queues are formed at the token node, and more and more modes are frozen as a result. Consequently, it becomes increasingly improbable to be able to transfer the token immediately. The number of transfer token messages is constant after some point because frozen modes need not be considered while transferring a token to a queued request. Copy grant messages increase and stabilize due to the fact that all the requests result either in token transfers or in copy grants with or without being queued. As the transfer tokens become less and less probable, copy grants become more and more probable. Release messages increase and remain almost constant after an initial stage due to the fact that every copy grant is accompanied by one or more releases (due to the propagation path) when the lock is released, whereas a token node does not send a release message after releasing a lock. Hence, the behavior of a release message is similar to that of copy grant. The number of freeze messages also increases initially and then remains constant due to the fact that a mode, once frozen, will not be sent a freeze message again for. There are a constant number of modes that can be frozen (at most five).

5. Related Work

Hierarchical locks and protocols for concurrency services have been studied in the context of database system [9, 8, 1]. Most concurrency services rely on a centralized approach with a coordinator to arbitrate resource requests or a combination of middle-tier and sink servers [17]. These approaches do not dynamically adapt to resource requests while our protocol does. Efforts on predictable ORB behavior have mostly focused on priority support for CORBA's interaction with message passing and thread-level concurrency [15] applicable to real-time database systems [19]. 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. Algorithms for distributed mutual exclusion with dynamic properties have other limitations. Chang [3] and Johnson [7] give an overview and compare the performance of such algorithms. Chang [2] developed extensions to various algorithms for priority handling that require broadcast messages [20, 18] or Raymond's non-adaptive logical structures with token passing [16]. Chang, Singhal and Liu [4] use a dynamic tree similar to Naimi *et al.* [13, 14]. 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 non-adaptive 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. The applicability of our work reaches from distributed object scenarios to the peer-to-peer computing paradigm in general.

6. Conclusion

We developed a protocol in support of concurrency services that enhances middleware services to provide scalability of synchronization while allowing a high degree of concurrency. We discussed the technical challenges in terms of design and provided implementation details. For as many as 120 nodes in the system, our protocol has message overhead of 3 messages Vs. that of 4 messages for Naimi's base protocol, which is one of the lowest complexity protocol for distributed mutual exclusion. Importantly, the logarithmic asymptotic behavior of message overhead is preserved in spite of the additional support for hierarchical locking modes in our protocol, which backs our claim for better scalability. Also, The response time as a factor of point-to-point latency is 90 for our protocol Vs. 160 for Naimi's protocol for 120 nodes. In summary, our protocol is highly scalable and enhances concurrency in distributed resource allocation following the specification of general concurrency services for large-scale data and object repositories. While our approach is motivated by CORBA, its principles are applicable to any distributed concurrency services and transaction models.

References

[1] 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.

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

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

[4] 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.

[5] 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.

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

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

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

[9] S.-Y. Lee and R.-L. Liou. A multi-granularity locking model for concurrency control in object-oriented database systems. *TKDE*, 8(1):144–156, 1996.

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

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

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

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

[14] 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.

[15] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. 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.

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

[17] 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.

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

[19] R. M. Sivasankaran, J. A. Stankovic, D. F. Towsley, B. Purimetla, and K. Ramamritham. Priority assignment in real-time active databases. *VLDB Journal: Very Large Data Bases*, 5(1):19–34, Jan. 1996.

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