ABSTRACT

DESAI, NIRMIT V. Scalable Distributed Concurrency Protocol with Priority Support. (Under the direction of Assistant Professor Frank Mueller).

Middleware components are becoming increasingly important as applications share computational resources in large distributed environments, such as web services, high-end clusters with ever larger number of processors, computational grids and an increasingly large server farms. One of the main challenges in such environments is to achieve scalability of synchronization. Another challenge is posed by requirement for shared resources with a need for QoS and real-time suppoprt. In general, concurrency services arbitrate resource requests in distributed systems. But concurrency protocols currently lack scalability and support for service differentiation based on QoS requirements. Adding such guarantees enables resource sharing and computing with distributed objects in systems with a large number of nodes and supporting a wide range of QoS metrics.

The objective of this thesis is to enhance middleware services to provide scalability of synchronization and to support service differentiation based on priorities. We have designed and implemented middleware protocols in support of these objectives. Its essence is a novel, peer-to-peer, fully decentralized protocol for multi-mode hierarchical locking, which is applicable to transaction-style processing and distributed agreement.

We discuss the design and implementation details of the protocols and demonstrate high scalability combined with low response times in high-performance cluster environments as well as TCP/IP networks when compared to a prior protocol for distributed synchronization. The prioritized version of the protocol is shown to offer differentiated response times to real-time applications with support for protocols to bound priority inversion such as PCEP and PIP. Our approach was originally motivated by CORBA concurrency services. Beyond CORBA, its principles are shown to provide 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 and real-time applications. In general, the results of this thesis impact applications sharing resources across large distributed environments ranging from hierarchical locking in real-time databases and database transactions to distributed object environments in large-scale embedded systems including real-time applications.

Scalable Distributed Concurrency Protocol with Priority Support

by

Nirmit V. Desai

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

Raleigh

2003

Approved By:

Dr. Vincent Freeh	Dr. Gregory Byrd

Dr. Frank Mueller
Chair of Advisory Committee

To my parents
for their everlasting and unconditional
love and support

Biography

Nirmit Desai was born on 18^{th} December 1979, in Surendranagar, India. He received his Bachelor of Engineering in Information Technology from Gujarat University, India, in 2001. After working as a Software Engineer with Applied Software Inc., he opted for graduate studies at North Carolina State University. With the defense of this thesis, he is receiving Master of Science in Computer Science from NCSU in June, 2003.

Acknowledgements

A lot of people have played a very important role in the successful completion of this thesis. Firstly, I thank my advisor, Dr. Frank Mueller, for his continuous guidance, mentoring and support. This work could not have been a success without his insightful remarks and advice. I also extend my gratitude to Dr. Vincent Freeh and Dr. Gregory Byrd for being on my advisory committee and providing me with valuable suggestions.

I also thank my colleagues and friends Jaydeep Marathe, Kiran Seth and Yifan Zhu for instantly helping me on any issues, sharing their creative ideas and providing me with an excellent research environment.

My special thanks go to my roommates and friends Ajay Dudani, Arnav Jhala, Kunal Shah, Kurang Mehta, Sameer Desai and Sameer Rajyaguru. Their delightful company has brought me some of the best times during my stay at NC State.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Resource arbitration is rapidly becoming a commodity in distributed computing where, data, objects and devices are shared 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. Centralized approaches are inherently unreliable and suffer from the classical single point of failure problem. Also, more importantly, they have very limited scalability when applied to large distributed system environments. This is mainly due to the number of server(s) in the system remain constant while the population of requesters may be arbitrarily large and dynamic.

1.1.1 Middleware Concurrency Services

For the above-cited reasons, recent trends aim at peer-to-peer computing with location-transparent distributed objects and services. This paradigm is generally supported by middleware to provide distributed services. This middleware provides a software layer between the operating system and the application 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. The CORBA/IIOP and RT-CORBA specifications by Object Management Group (OMG) [19] define one such middleware. The corresponding OMG concurrency service specification [18] describes a hierarchical locking model coupled to the core middleware to enable scalability of synchronization. However, existing implementations of such middlewares do not fully implement the concurrency services as specified in [18]. They, either have fairly thin and primitive implementations of the distributed locking mechanisms or they do not support concurrency services at all. For example, TAO (The ACE ORB) described in [38], one of the best known and supported implementations of CORBA/IIOP middleware, supports a thin concurrency service without hierarchical locking support. For these reasons, current middleware concurrency service implementations remain poorly scalable by in large and mostly non-applicable to large scale distributed systems and services.

1.1.2 Clusters and Grid

Another relevant trend regarding our work relates to clusters and the Grid [13, 20]. 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 their 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, 13]. We see a similar trend in commercial computing areas, such as server-style 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. In such environments, the traditional approaches for fault-tolerance have to be enhanced or replaced due to the shrinking mean-time-to-failure (MTTF) margins with growing sizes of clusters and Grids as described in [15]. Redundant computing is emerging as an alternative in such environments to cope with the failures efficiently where multiple replicas of a computation exist in the system simultaneously.

1.1.3 Issues

One of the main challenges in the environments described above is to achieve scalability of synchronization. Synchronization might be required for various purposes in each case, *e.g.*, state management, consistency management, replication consistency, resource arbitration, serializability,

distributed agreement and distributed transactions. Another issue is service differentiation based on the priorities to ensure predictable response times for real-time applications. This thesis is motivated by these unique and emerging challenges in the area of large scale distributed systems and middlewares.

1.2 Approach

We first investigate the issue of scalability and discuss the design and implementation of a scalable protocol for synchronization supporting hierarchical locking. Next, we focus on providing support for real-time applications by means of supporting priorities of lock requests. We also discuss the problem of priority inversion and bound inversion duration by supporting priority inheritance protocols. The results demonstrate that the additional support for hierarchical locking and priorities does not affect the scalability of the protocol.

1.2.1 Scalability

We address the issues of scalability through novel middleware concurrency protocols. Though our protocols are compatible with existing standards, such as CORBA, the 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 [11, 12]. The use of transactions in such environments requires support for hierarchical locking services to arbitrate between requests with different locking modes at multiple levels of data structures within the shared or replicated data. Hierarchical locks have been studied in the context of database systems with a limited number of nodes [17, 26, 25, 23, 3]. However, hierarchical locking has not been studied in the context of distributed synchronization. The sheer sizes of clusters, server farms and the Grid lead us to consider hierarchical locking again, but this time under the aspect of scalability.

The problem of distributed mutual exclusion has been approached from different angles, broadly classified as token-based [32, 16, 37] and non-token-based [24]. The distributed mutual exclusion protocol by Naimi et al. [32] demonstrates one of the best known token-based, decentralized, distributed synchronization approaches. However, none of these approaches considered supporting hierarchical locking to accommodate large number of concurrent requests. The essence of our contribution lies in leveraging hierarchical locking with one such distributed synchronization

protocol. The resulting protocol supports a high degree of concurrency for a number of locking 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 with 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.

1.2.2 Priority support

We demonstrate how the middleware concurrency protocol for distributed objects and services can be extended to support real-time requirements. The use of such protocols in real-time systems requires support for priorities and should address the requirements imposed by real-time schedulability theory, namely, guaranteed end-to-end response times and bounded priority inversion. Priority support for distributed mutual exlusion has been studied in [30] without the support for hierarchical locking scheme. Contemporary distributed real-time systems exploit locking schemes in real-time database systems and transaction protocols, just to name a few [44, 39, 38]. The protocol developed in this thesis not only fits these requirements but also provides a more efficient solution to prioritized hierarchical locking in distributed systems than any of the previous protocols, to the best of our knowledge.

Real-time scheduling provides the theory for uniprocessor allocation of tasks and provides guarantees to meet deadlines in the presence of resources [28, 2, 41, 4, 6, 45]. Even for multi-processor environments with shared memory, certain guarantees can be provided if tasks with shared resources are scheduled on the same processor. We focus on a protocol to acquire resources in a distributed environment, bound the message overhead, and provide support to bound priority inversion so that this protocol can be used in arbitrary distributed scheduling methods for real-time systems. Thus, this work is applicable but not limited to rate-monotone priority assignments for static priority preemptive scheduling where the highest priority task ready to run is being executed on each processor. Our protocol is subject to static priorities associated with requests. In addition, extensions to this protocol to bound priority inversion are introduced to provide temporary adjustments of the priorities in a dynamic manner.

1.3 Outline

In this thesis, 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. The next chapter focuses on the priority support for the base protocol. We motivate and outline the set of new rules to support priorities. Protocol extensions for bounding priority inversion are developed. Our experiments are comprised of the aforementioned comparison with non-hierarchical locking on a Linux TCP/IP network on one hand and results for scalability and latency evaluations on an IBM SP cluster on the other hand. Also, a performance study of the prioritized version assesses the quantitative benefits of the novel approach. Finally, we discuss related work and summarize our contributions.

Chapter 2

The Unprioritized Protocols

In this chapter, we focus on the problem of achieving scalablity of synchronization and support for hierarchical locking without taking priorities into account. Therefore, the protocol introduced in this chapter is suitable for general distributed applications with a focus on the average throughput of the entire system. After introducing a non-hierarchical protocol by Naimi [32], we present our novel base protocol with a set of governing rules and tables. We provide examples covering important aspects of the operational details.

2.1 Prior Work

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 [8]. 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 [32]. 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 2.1, 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.

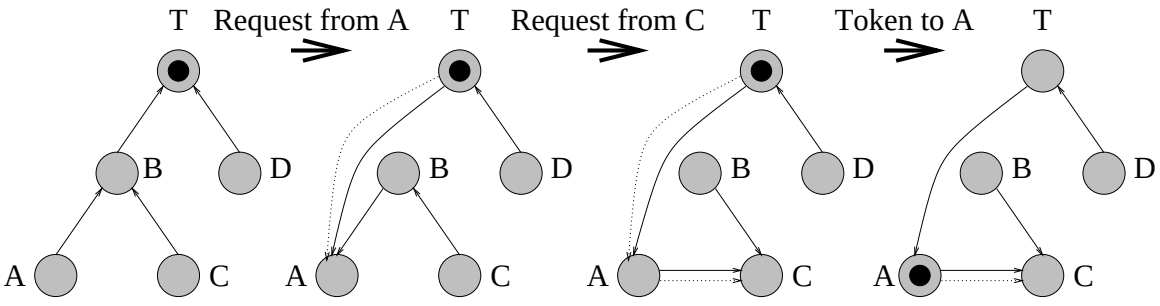


Figure 2.1: Non-hierarchical Example

Consider the process of request forwarding again. When a request passes intermediate nodes, the probable owner is set to the requesting node. This causes a transformation of the tree to a forest. The traversed nodes form a tree rooted in the requesting node, which will be the token owner of the future. Nodes that have not been traversed are still part of the original tree rooted in the current token owner. For example, when B forwarded A's request to T, B sets the probable owner

to A. At this point, one logical tree $C \rightarrow B \rightarrow A$ is rooted in A while another tree $D \rightarrow T$ is still rooted in T. Once the request reaches T, the separate trees are merged again (2nd tree of Figure 2.1).

Should k requests be in transit, then there may be up to $k + 1$ separate trees in the forest. Once all requests have arrived at current or future token owners, the forest collapses to a single tree again rooted in the last requester. The *next* pointers form a distributed queue of pending requests from the current token owner to the last requester. A new request would simply be appended at the end as described above.

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. In the worst case, requests can be forwarded via $n - 1$ messages and one additional message is needed to send the token. The model assures that requests are ordered FIFO. Our contribution in prior work was to alleviate shortcomings in priority support [29, 30]. 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.

2.2 Our Hierarchical Locking Protocol

This section introduces our 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 an O(log n) message complexity ensure scalability and are also demonstrated to 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 [18].

2.2.1 Compatibility between Lock Modes

The main objective of our approach is to ensure an optimal 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 locking 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 as described in [18]. 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 [17, 26]. 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 lock modes indicate a need for serialization of two requests. Let R be a resource and L_R be the lock associated with it. Table 2.1 shows the rules for granting L_R in different modes according to the specification of concurrency services [18]. 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 2.1.

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:

$$\Phi < IR < R < U = IW < W \tag{2.1}$$

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 2.1: Incompatibility of Lock Modes

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 2.1 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 A 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 sub-tree rooted in node A.

2.2.2 Local Queues, Intent Locks and Copysets

In our protocol for hierarchical locking, we employ a token-based approach. A novel aspect of our protocol 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 optimize 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 [27].

Definition 4:

The *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.

2.2.3 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 granter

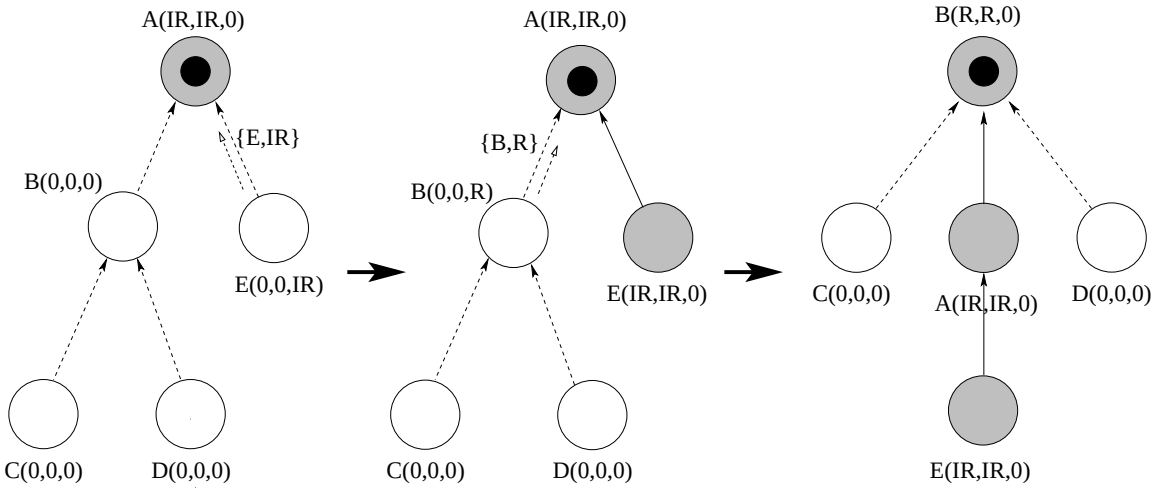
in case 2: if modes are compatible and if $M_O < M_R$, the token is transferred to the requester. Thus, the requester becomes the new token node and parent of the original token node. If $M_O \geq M_R$, the requester receives a granted copy from the token node and becomes a child of the token node. (See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

Table 2.2 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 2.1.

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 2.2: 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.



(a) E Requests IR (b) B Requests R (c) Final state

Figure 2.2: Request Granting Example

Example: Consider the initial state as shown in Figure 2.2(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.

2.2.4 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 2.3.

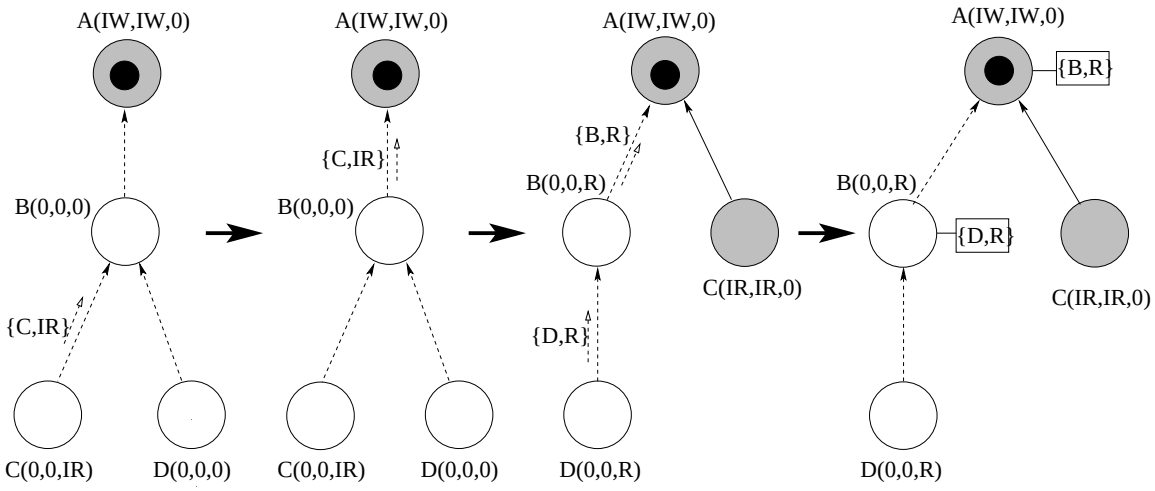
2. If the token node cannot grant a request, it will queue the request locally regardless of the state of its pending request.

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 2.3, local queuing and forwarding are indicated as *Q* and *F*, respectively. The aim here is to queue as many requests as possible to suppress message passing overhead without compromising FIFO ordering.

Example: In Figure 2.3(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 2.3 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 2.3 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

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 2.3: Queue / Forward Decision



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

Figure 2.3: Request Queuing/Forwarding Example

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 2.4 and 2.5 depict the pseudo-codes for 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).

The predicate *grantable(M_O, M_R)* indicates whether M_O and M_R satisfy Rule 3.1. *tokenable (M_O, M_R)* indicates whether M_O and M_R satisfy Rule 3.2 and the token needs to be transferred. Similarly, *compatible(M_1, M_2)* is the inverse of *conflicts(M_1, M_2)* that indicates whether

the compatibility matrix of Table 2.1 shows a conflict between M_1 and M_2. The pseudo-procedure Check_requests_on_queue() in Figure 2.5 handles locally queued requests as described in Rules 4 and 5.

2.2.5 Lock Release

The following rule governs the handling of lock releases. As an implementation detail, remember that the parent nodes keep track of the owned mode of their immediate children. (The request granter records the granted modes while granting requests and the requester becomes the child of granter).

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 is similar to Naimi's protocol in that queued requests are served upon a release. If a token node has received release notifications from its children and if the token 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 and so on.

The second part of the above rule ensures that release messages are only sent if necessary, *i.e.*, when owned 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 owned 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 2.6(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.

```
RequestLock(M_R)
    if Self ≠ Token_Node then
        if M_O ≥ M_R ∧ compatible(M_O, M_R) ∧ ¬M_R ∈ Frozen_Modes then [Rule 2]
            Acquire Lock
            Copyset ← Copyset + M_R
        else [Rule 2]
            M_P = M_R
            Send Request to Parent
            Wait for grant
    else
        if compatible(M_O, M_R) ∧ ¬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 [Table 2.4]
            Send Freeze to Children if required ([a])
            Wait for grant
HandleRequest(M_R)
    if Self ≠ Token_node then
        if grantable(M_O, M_R) then [Rule 3.1, Table 2.2]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send Grant with Frozen_Modes to Requester
        else if can_be_queued(M_P, M_R) then [Rule 4.1, Table 2.3]
            Queue ← Queue + M_R
        else [Rule 4.1, Table 2.3]
            Send Request to Parent
    else
        if tokenable(M_O, M_R) then [Rule 3.2, Table 2.1]
            if Requester ∈ Children then
                Children ← Children - Requester
            Parent ← Requester
            Send Token with Queue to Requester
        else if grantable(M_O, M_R) then [Rule 3.2, Table 2.2]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send Grant with Frozen_Modes to Requester
        else [Rule 4.2]
            Queue ← Queue + M_R
            Update Frozen_Modes [Table 2.4]
            Send Freeze to Children if required ([a])
```

[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

Figure 2.4: Pseudocode for Requesting and Granting Locks

```
ReceiveToken()
    if M_H = φ ∧ M_O ≠ φ ∧ granter ≠ parent then
        Send release of M_O to parent
    Parent ← NIL
    Copyset ← Copyset + M_P
    M_H ← M_P, M_P ← φ
    Children ← Children + Sender if required [Rule 2] ([b])
    Merge Queues ([c])
    Calculate Frozen_Modes from Queue
    Check_requests_on_queue [Rule 4]
    Signal grant
ReceiveGrant()
    if M_H = φ ∧ M_O ≠ φ ∧ granter ≠ parent then
        Send release of M_O to parent
    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]
    Signal grant
Check_requests_on_queue()
    if M_P = W ∧ M_O = U ∧ Copyset = phi then
        M_O ← W, M_H ← W
        Copyset ← Copyset + W - U
        Signal grant
    while Queue ≠ EMPTY
        M_R ← Queue.head
        if tokenable(M_O, M_R) then [Rule 3.2, Table 2.1]
            if Requester ∈ Children then
                Children ← Children - Requester
            Parent ← Requester
            Send Token with Queue to Requester
        else if grantable(M_O, M_R) then [Rule 3.2, Table 2.2]
            Children ← Children + Requester
            Copyset ← Copyset + M_R
            Send Grant with Frozen_Modes to Requester
        else
            Update Frozen_Modes [Table 2.4]
            exit_loop
```

[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 [29].

Figure 2.5: Pseudocode for Entering Critical Section on Grant/Token

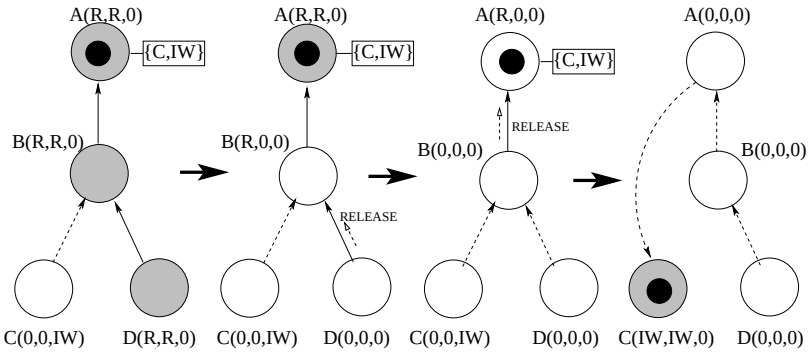


Figure 2.6: Lock Release Example

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).

```
RequestUnlock()
    Copyset ← Copyset - M_H
    M_H ← φ
    Update Frozen_Modes [Table 2.4]
    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 2.7: Pseudocode for Lock Release Handling

Figure 2.7 depicts the pseudo-code of the lock release handling. *RequestUnlock()* is a user API.

2.2.6 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.

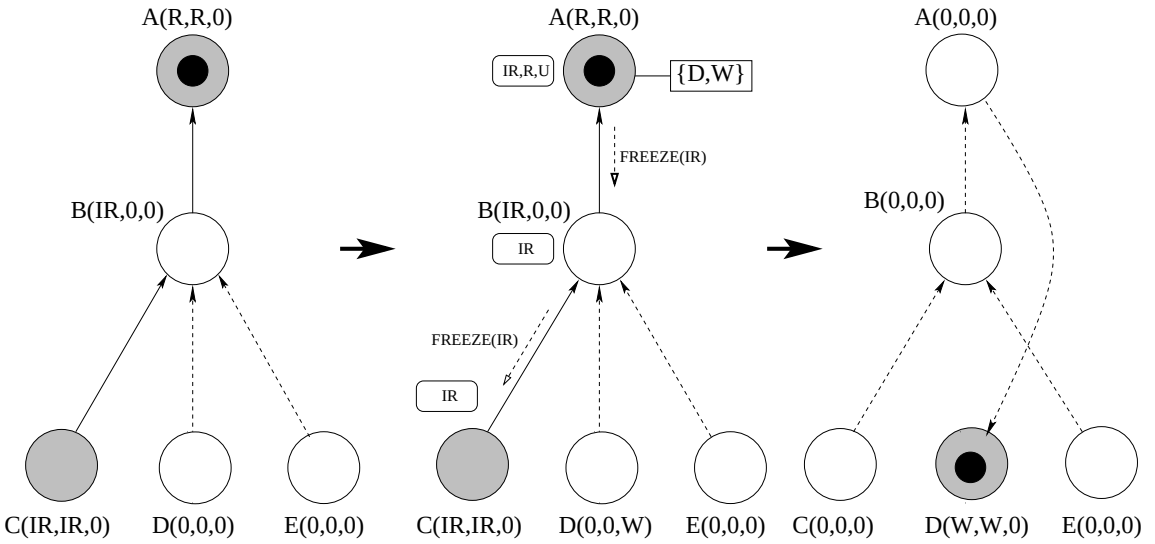


Figure 2.8: Frozen Modes Example

Consider a request by D for mode W in the state of Figure 2.8(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 2.8(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 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). The rationale behind the freezing mechanism can be formalized by the following invariant:
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, $\forall\ M_i\ \in\ M_{FT}$:

$$[grantable(M_{OT}, M_i)\ \vee\ tokenable(M_{OT}, M_i)]\ \wedge\ conflicts(M_i, M_R) \tag{2.2}$$

Token Node owning Mode M_{OT}	Requested Mode M_R				
	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 2.4: Rules for Freezing Lock Modes at the Token Node

Table 2.4 depicts an enumeration of frozen modes for all combinations of M_{OT} and M_R. For instance, if the token node is owning a lock in R and a W request is received and queued locally (as depicted in Figure 2.8(b)) then IR, R and U are the modes to be frozen at the token node according to invariant 2.2.

In order to extend fairness beyond the token node, 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. For instance, in Figure 2.8(b), had A not notified its child B about the frozen modes, B would potentially grant any future requests for IR. This will further delay the grant of the waiting $\{D, W\}$ and also violate FIFO queuing as

$\{D, W\}$ was received first. To avoid this version of FIFO violation, the token node notifies children about the frozen modes by sending a *freeze* message. The propagation of *freeze* messages can be formalized by the following invariant:

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, $\forall\ M_i\ \in\ M_{FC}$:

$$M_i\ \in\ M_{FT}\ \wedge\ grantable(M_{OC}, M_i) \tag{2.3}$$

These invariants are effected by the following rule which supplements Rules 2 and 3:

Rule 6:

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

```
HandleFreeze(Modes)[Rule 6]
    Frozen_Modes ← Frozen_Modes + Modes
    Send Freeze to Children if required (a)
```

Figure 2.9: Pseudocode for Freezing Mechanism

The pseudo-code of this freezing mechanism is augmented to almost all the other routines where requests are granted (Routines in Figure 2.4, 2.5 and Figure 2.11). Figure 2.9 depicts the pseudo-code for the freeze message handler. Through this freezing mechanism, ultimately, all children and grandchildren will release their modes, and a release message will arrive at the token node for each of its immediate children. Thus, the FIFO policy is preserved.

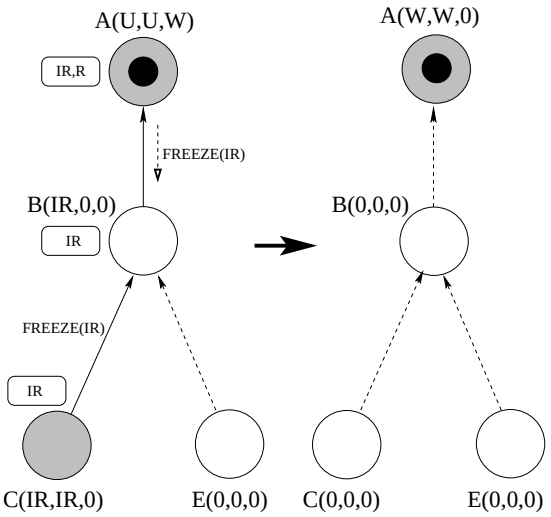
2.2.7 Upgrade Request

Upgrade locks facilitate the prevention of potential deadlocks by very simple means if the resulting reduction in the concurrency level is acceptable [18]. 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 2.10(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 2.11 is the pseudo-code of the *RequestUpgrade()* user API.



(a) A requests Upgrade (b) Upgrade complete

Figure 2.10: Request Upgrade Example

RequestUpgrade() [Rule 7]
    if Copyset $\leftarrow \phi$ **then**
        Release U
        Acquire W
    else
        $M_P \leftarrow W$
        Update Frozen_Modes [Table 2.4]
        Send Freeze to Children if required (^{a})
        Wait for grant

Figure 2.11: Pseudocode for Request Upgrade

2.3 Summary

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.

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.

Chapter 3

The Prioritized Protocol

In this chapter, we describe the prioritized version of our protocol. First, we motivate priority support for the hierarchical locking protocol described in the previous chapter along with a brief description of prior work to support priorities for distributed synchronization. Next, we describe the design of the prioritized protocol derived from our basic protocol. We also address the problem of bounding the priority inversion duration by employing inheritance protocols such as PCEP and PIP.

3.1 Motivation

In this section, we motivate priority support for our basic hierarchical locking protocol described in the previous chapter. We also describe in brief, a prior approach by Mueller [30] to support priorities of requests for distributed mutual exclusion and motivate supporting hierarchical locking along with request priorities.

3.1.1 Prior Work

The protocol by Mueller [30] descibed here is an enhancement over the protocol by Naimi et al. [32] descibed briefly in Sectio 2.1 for supporting priorities. In a prioritized environment, requests should be ordered first by priority and then (within each priority level) FIFO. Assume that a request by node N is issued at priority $\xi(N)$. The basic idea for priority support is to accumulate

priority information on intermediate nodes during request forwarding. Thus, if the request by N passes along a node I, the probable owner for requests with priority $\leq \xi(N)$ is set to N. (Higher priorities denote more important requests.)

The *next* list approach of Naimi's protocol is replaced by a *queue* of locally stored requests to avoid race conditions. Each request seen by a node is stored in this local *queue* ordered first by decreasing priority, and then in FIFO order. When the token arrives at a requesting node, the requests are gradually removed from the *queue* of other nodes.

The example in Figure 3.1 depicts a sequence of token requests of A, B, C and D with priority 1, 2, 1 and 3, respectively. The token resides in node T while the requests arrive. For the

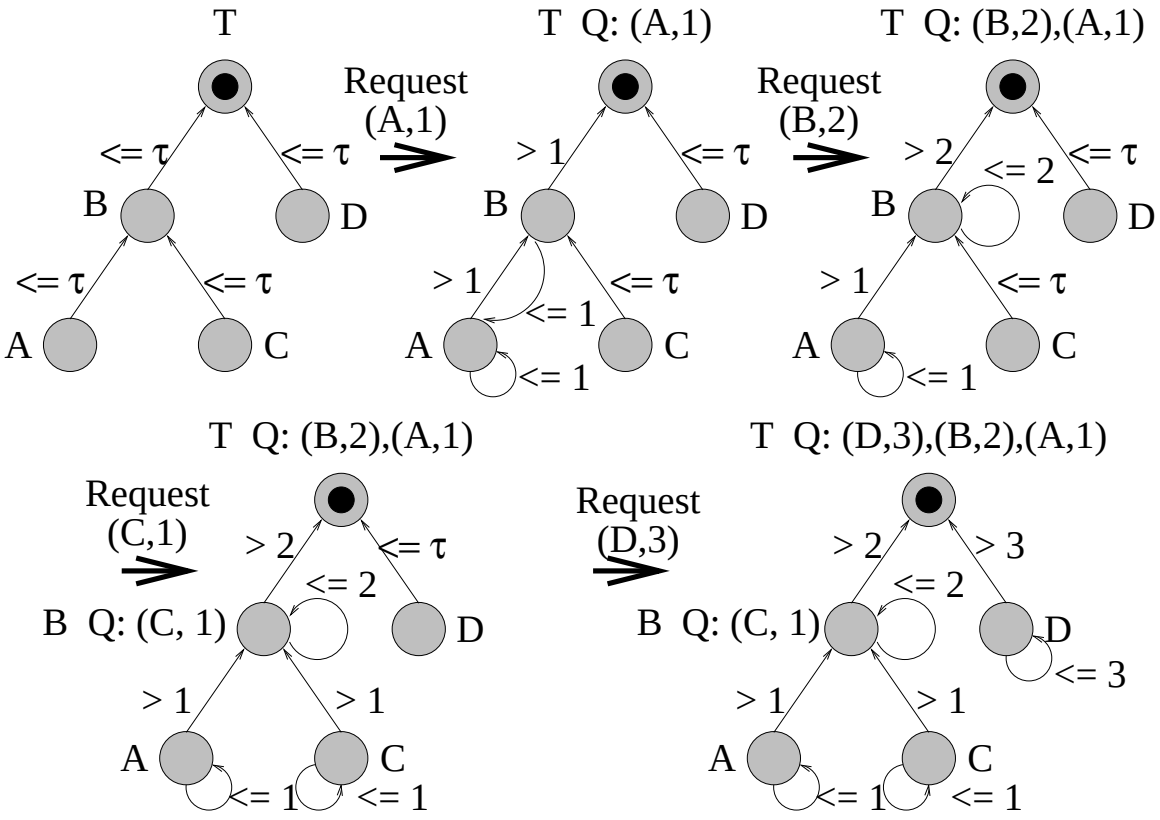


Figure 3.1: Non-hierarchical Priority Support Example

sake of simplicity, we assume that the initial logical structure is a tree whose edges (directly or indirectly) point to the token owner for all priorities (labeled as top τ). Each request results in new forwarding information for certain priority levels depicted by labeled edges. For example, request $(A, 1)$ results in edges $A \rightarrow A$ labeled ≤ 1, which indicates that future requests received by A with

priority ≤ 1 are handled by A. Other edges include $A \rightarrow B(> 1)$, $B \rightarrow T(> 1)$ and $B \rightarrow A(\leq 1)$.

Any request is locally logged, then the probable owner is determined as the last node whose request was seen with greater or equal priority. The probable owners are updated so that edges with less or equal priority than the current request are replaced by the new request. If the token is not present locally, the request is added to the local queue and a message is sent to the probable owner. When a node R issues a request at priority $\xi(R)$, this request propagates along the chain of probable owners until it reaches a node I with a pending request at priority $\geq \xi(R)$. It is then locally queued at I. The basic idea is that node I has to be served before our request. Thus, R's request may be stored locally until the token arrives. Once the token arrives, R's request should be served after I's if no other requests have arrived since then. This is also depicted in Figure 3.1, where local queues are associated with nodes T and B. Node B stores C's request at priority one since B has an outstanding request at priority two that will be served first.

While this protocol supports priorities of requests with a simple and elegant solution, it does not differentiate between lock access modes. Also, it does not study the problem in the context of hierarchical locking scheme specified in [18]. Since hierarchical locking schemes constitute a solution to the problem of scalability, this protocol may not be suitable for large distributed environments with focus on scalability. Our prioritized protocol described below borrows some of the basic mechanisms from this simple protocol and extends our basic hierarchical locking protocol described in the previous chapter to support request priorities.

3.1.2 Basic Protocol

Let us revisit Figure 2.8 of our basic protocol from the viewpoint of real-time applications. Possibly, timing requirements of node E can be more stringent than those of node D. In that case, requests issued by E should have higher priority than those issued by D. We should have granted $\{E, IR\}$, overruling the frozen modes, thereby giving preference to E over D. Enforcing FIFO vmay result in unbounded priority inversion, and the QoS of the upper layer applications will be compromised significantly. While the base protocol works best for non-realtime applications and ensures FIFO policy, it does not handle priorities adequately.

Let $\beta(N)$ and $\xi(N)$ be the base and effective priorities of node N, respectively. Each request made by node N has priority $\xi(N)$, the effective priority of N, at the time of request. Without the support for a priority inheritance scheme, $\xi(N)$ and $\beta(N)$ would not be distinguished. The basic approach of supporting prioritized requests is to redesign the

1. **Freezing Mechanism:** Overrule the frozen modes if the received request has higher priority than the request waiting in queue.

2. **Queue Reordering:** Order the local queue of requests first according to their priority $\xi(N)$ and then according to FIFO.

3. **Queue/Forward Policy:** Change the local queuing policy to take into account the priority of received request Vs. the priority of pending self-requests in the case of Rule 4.1.

We describe each of these improvements and their motivations in the following subsections.

3.2 Freezing Mechanism

As illustrated in Figure 2.8, the current implementation of the freezing mechanism potentially results in unbounded priority inversion. Here, we examine the freezing mechanism outlined in the previous chapter and provide the extensions to overcome the shortcomings.

In the base protocol freezing mechanism, sets M_{FT} and M_{FC} calculated by invariants 2.2 and 2.3 were unified with a set $Frozen\_Modes$ recording the state for each of the six modes mentioned in inequation 2.1 at each node. The only values each of the elements in $Frozen\_Modes$ could take were $FROZEN$ and NOT_FROZEN. Rule 6 used this information and approved a grant/token for requested mode M_R if the set $Frozen\_Modes$ had NOT_FROZEN as the state value of the element corresponding to M_R. This meant that any request, regardless of its priority, could cause a set of elements of $Frozen\_Modes$ to assume $FROZEN$ state if it were queued at the token node. Any request for any of those modes in the future, regardless of their priority, would be disapproved by Rule 6. This violates the strict priority policy according to which, higher priority requests should never block for lower priority requests. The solution to the problem is quite intuitive and elegant. Let *threshold* be the priority of the highest priority request causing the freeze of some modes in $Frozen\_Modes$. The elements of the $Frozen\_Modes$ set can be made to record the *threshold* of the requests that caused the freeze. Table 3.1, the prioritized version of Table 2.4 outlines the improved mechanism.

Hence, Rule 6 can be replaced by its prioritized version as:

Rule 6$_{PRIO}$:

A request for mode M_R can only be granted if $\xi(M_R) > Frozen\_Modes(M_R).threshold$.

Token Node owning Mode M_{OT}	Requested Mode M_R				
	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					
$\forall M_i \epsilon Frozen\_Modes : Frozen\_Modes(M_i).threshold = max\,[Frozen\_Modes(M_i).threshold,\ \xi(M_R)]$					

Table 3.1: Rules for Freezing Lock Modes at the Token Node (Prioritized)

3.3 Reordering Queue

The following rule gives preference to higher priority requests when requests are queued locally.

Rule 8:

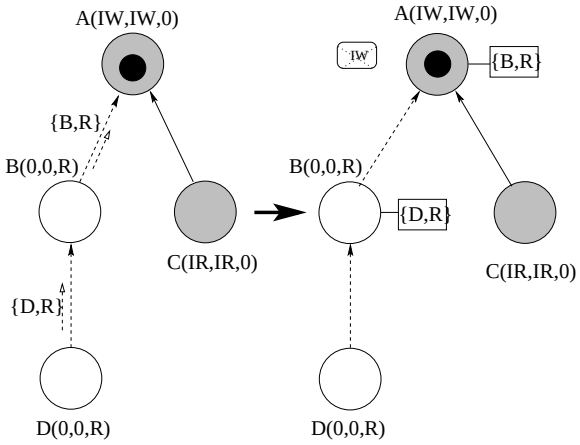
1. While queuing a request with priority $\xi(N)$ locally, it is queued first according to the non-increasing order of priority and then according to non-increasing order of *usage times* and finally increasing order of their *receipt times* (described in [29, 30]).

2. While receiving the token and merging the local queue with the queue received from the original token node, the queue is sorted according to the order described in Rule 8.1.

3.4 Queue/Forward Policy

The essence of the policy to queue requests locally is to enable path compression for requests, thereby reducing the message overhead without violating FIFO. Consider the initial state shown in Figure 3.2(a) and request $\{B, R\}$ from B followed by a request $\{D, R\}$ from D. According to Rule 2, both will be forwarded to their respective parents. When A receives $\{B, R\}$, it cannot grant it due to Rule 3.2 and queues it locally according to Rule 4.2. While B is waiting, it receives the request $\{D, R\}$ and cannot grant it according to Rule 3.1 but queues it according to Rule 4.1 and Table 2.3. The rationale behind this table is as follows: Had $\{D, R\}$ been forwarded to A instead of queuing locally at B, A would not be able to grant it and $\{D, R\}$ would never precede $\{B, R\}$ in

the queue at A due to FIFO policy. This means that $\{B, R\}$ will always be granted before $\{D, R\}$. By queuing $\{D, R\}$ at B, we reduce the amount of request propagation messages and promote the distribution of queued requests to multiple sites instead of a single large queue at the token node. Ultimately, when $\{B, R\}$ is granted, B is guaranteed to be able to grant the locally queued $\{D, R\}$ due to Rule 3.1.



(a) B Requests R, D Requests R

(b) {D,R} Queued by B

Figure 3.2: Request Queuing/Forwarding Example (Prioritized)

Consider real-time applications with $\xi(D) > \xi(B)$. According to Rule 8.1, $\{D, R\}$ will precede $\{B, R\}$ in the queue at A. This means that queuing $\{D, R\}$ locally at B results in (potentially unbounded) priority inversion. This leads us to extend the Table 2.3 as shown in Table 3.2, bold faces indicating extensions. Essentially, to queue the request locally, $\xi(pending\_request) \geq \xi(received\_request)$ should be met in conjunction to the conditional of Table 2.3.

3.5 Example

We describe an example here that illustrates the prioritized protocol and its policies we just described. To simplify the details (without restricting the generality), assume that $\xi(A) = 1$, $\xi(B) = 2$, $\xi(C) = 3$, $\xi(D) = 4$ and $\xi(E) = 5$, with larger values denoting higher priorities. Assume an initial state of the system as shown in Figure 3.3(a). B is waiting for its request $\{B, IW\}$ to be served while A has queued $\{B, IW\}$ due to incompatibility. $Frozen\_Modes$ at A reflect the

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	**CQ**	F	F	F	F
Read – R	F	**CQ**	F	F	F
Upgrade – U	F	F	**CQ**	**CQ**	**CQ**
Intention Write – IW	F	F	F	**CQ**	F
Write – W	**CQ**	**CQ**	**CQ**	**CQ**	**CQ**
Conditional Queuing CQ = IF($\xi(M_P) \geq \xi(M_R)$) THEN 'Q' ELSE 'F'					

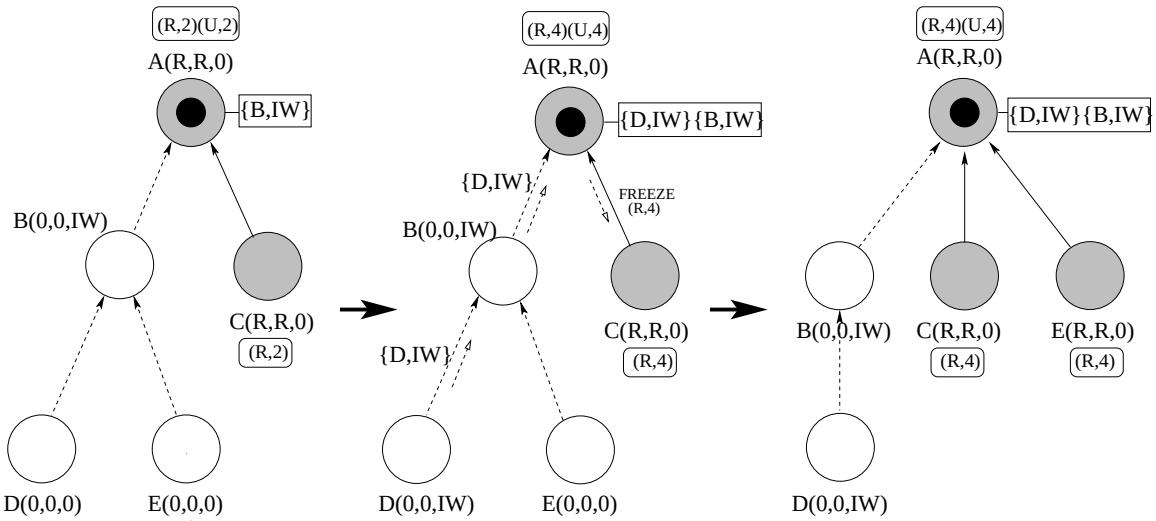
Table 3.2: Queue / Forward Decision (Prioritized)

modes R and U frozen at thresholds $\xi(B)$ according to invariant 2.2 and Table 3.1. A has also sent a freeze message with $(R, 2)$ according to invariant 2.3. Next, consider a request $\{D, IW\}$ from D as shown in (b). D forwards the request to its parent B. B cannot grant it due to Rule 3.1 and also cannot queue it locally as the conditional of Table 3.2. Hence, Rule 4.1 is not satisfied ($\xi(D) > \xi(B)$) and $\{D, IW\}$ is forwarded to A. A cannot grant it due to incompatibility and locally queues $\{D, IW\}$ ahead of $\{B, IW\}$ as per Rule 8.1. Queuing of $\{D, IW\}$ at A causes the thresholds of $Frozen\_Modes$ R and U to be boosted to $\xi(D)$ as per Table 3.1. Notice that we have to notify C about this boost of thresholds to prevent C from granting requests with priority $\xi(request) \leq \xi(D)$. As a result, C has the threshold of R boosted to $\xi(D)$.

Next, consider a request $\{E, R\}$ from E as depicted in (c). E forwards it to its parent B, B cannot queue it or grant it and forwards it to A. A can grant a copy to the requester as Rule 3.2, and Rule 6_{PRIO} permits this action since $\xi(E) > Frozen\_Modes(R).threshold$ holds at A. Since E is the higher priority node, it receives a grant immediately and becomes a child of A (Rule 3, operational specification 2). Notice that E needs to be sent the $Frozen\_Modes$ that satisfy invariant 2.3, which is $(U, 4)$ in this case.

3.6 Bounding Priority Inversion

The examples we have presented so far are in the context of a single lock L_i. However, in realistic applications we always need a set of locks L to protect the data structures at same or at different levels of the hierarchy. This means that there is a logical tree formation for each of such



(a) D Requests IW (b) E Requests R (c) Final state

Figure 3.3: Prioritized Protocol Example

$L_i \in L$. The policies we described in the previous section allowed us to bound priority inversion local to one lock, such as L_i. But considering the scenarios in the global context with all such L_i, still poses a potential for unbounded priority inversion.

Consider Figure 3.3(a) again. Queuing of $\{B, IW\}$ by A may result in unbounded priority inversion as $\xi(A) < \xi(B)$ and B is waiting for A. A similar situation arises when A queues $\{D, IW\}$ as depicted in Figure 3.3(b).

Obviously, A cannot grant any of these requests straight away due to the conflict between its owned mode R and the requested modes IW. To see how the current policies will lead to unbounded priority inversion in such situations, consider the logical tree for some other lock L_j as shown in Figure 3.4(a). C is waiting for $\{C, W\}$ to be granted, while D and B are owning/holding IW and $\{C, W\}$ is queued due to incompatibility. As a result, D has $Frozen\_Modes$ $(IR, 3)(IW, 3)$ as per Invariant 2.2 and Table 3.1. B has been notified about $(IW, 3)$ as per Invariant 2.3. Let us assume L_i enters the state described in Figure 3.3(b) prior to L_j entering the state described in Figure 3.4(b). As shown Figure 3.4(b), A sends its request $\{A, IR\}$ to D and D queues it locally as the Rule 6_{PRIO} disapproves the grant (even though Rule 3.1 allows it). Additionally, $\{A, IR\}$ is queued after $\{C, W\}$ due to Rule 8.1. Once D and B release their locks, the token will be transferred to C (according to Rule 5) and $\{A, IR\}$ will still be queued at C due to incompatibility, as shown in Figure 3.4(c). Figure 3.5 presents the time line perspective of the scenario, wherein

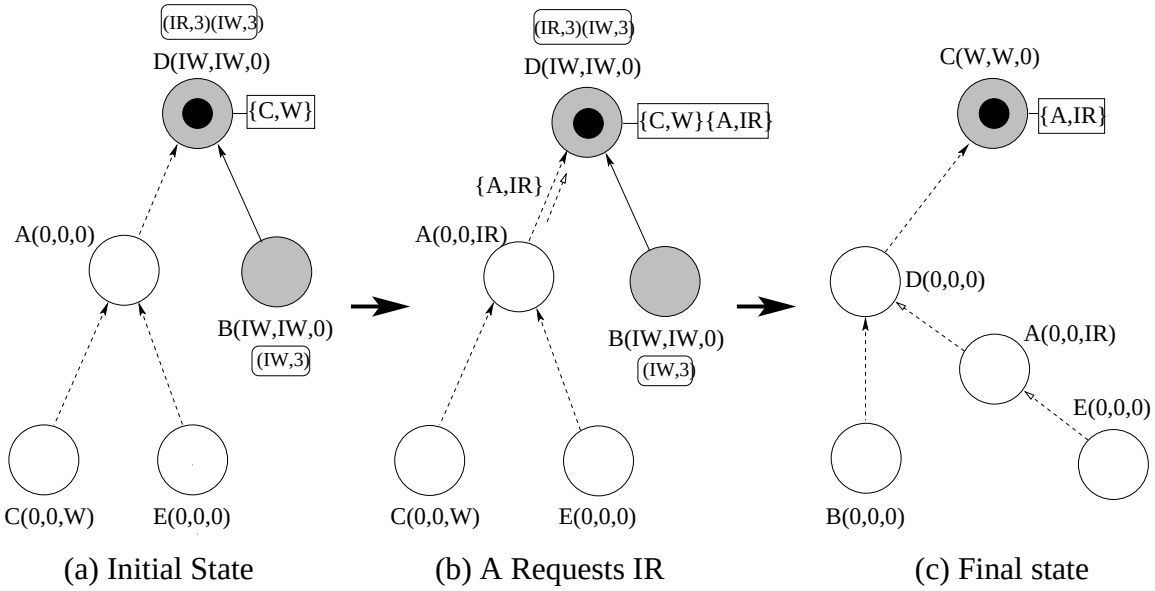


Figure 3.4: Priority Inversion Example

upper segments correspond to activity on L_i while lower segments sorrespond to activity on L_j for each node.

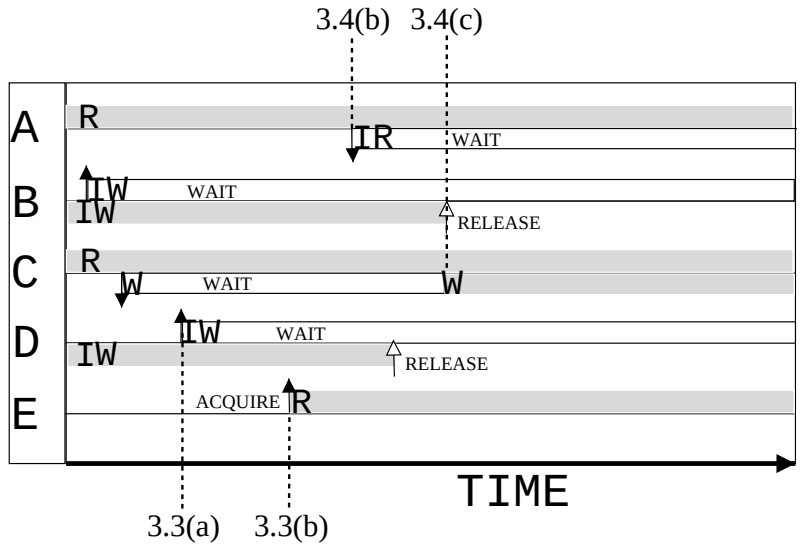
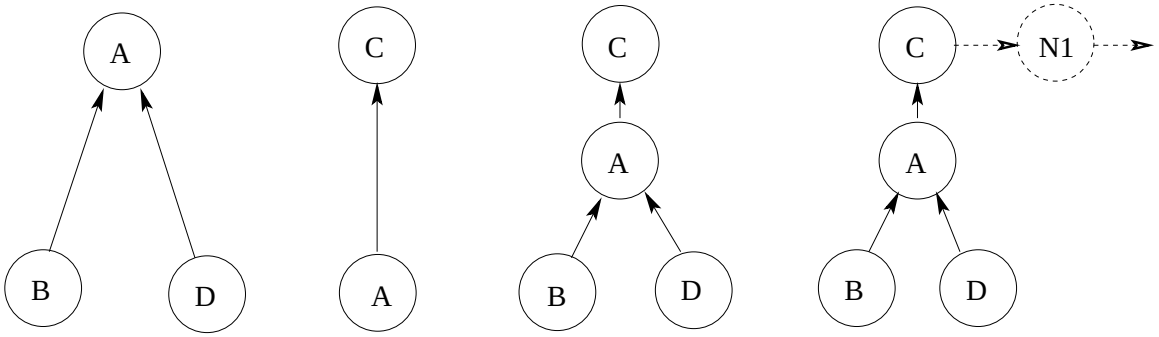


Figure 3.5: Time Line Representation of Priority Inversion Scenario

If we construct a wait-for graph of the local and global contexts as shown in Figure 3.6,

B and D are waiting for A in the context of L_i (as shown in Figure 3.6(a)) while A is waiting for C in the context of L_j (as shown in (b)). Hence, in a global context B and D wait for C transitively, as shown in (c). In general, an unlimited number of future requests from nodes N_i such that $\xi(A) < \xi(N_i) < \xi(D)$ can be queued at D in the state of Figure 3.4(a). In summary, D can potentially be waiting for an unbounded number of lower priority requests causing unbounded priority inversion as shown in Figure 3.6(d).



(a) Context of Li (b) Context of Lj (c) Global Context (d) Worst Case

Figure 3.6: Wait-for Graphs

In the following section, we discuss various priority inheritance protocols as a solution to the problem of unbounded priority inversion. We also describe the implementation of two of the most prominent inheritance schemes, *viz.* PCEP and PIP, for our protocol.

3.7 Priority Inheritance

Two basic methods for bounding priority inversion can be incorporated into the prioritized protocol. First, early priority boosting can be discussed in the context of the priority ceiling emulation protocol (PCEP). Second, the protocol can be enhanced for dynamic priority changes due to resource contention to support the priority inheritance protocol (PIP). For both cases, changes to the prioritized protocol are identified.

3.7.1 Priority Ceiling Emulation Protocol(PCEP)

Priority ceiling protocols in general require that a priority value, the *ceiling* C_i, be associated with a resource R_i and the corresponding lock L_i. C_i is defined as the priority of the highest priority node contending for L_i. The original priority ceiling protocol (PCP) determines the ceiling off-line for static priority scheduling [40]. The dynamic priority ceiling protocol (DPCP) determines the ceiling online at the time of resource contention for dynamic priority scheduling [10]. Both PCP and DPCP only grant a resource to a requester if its priority is greater than the ceilings of all resources in use by other tasks. Should a requester block due to this rule, its priority is inherited by the node that holds the requested resource. While both protocols avoid chained blocking (such as induced by the transitivity property of PIP), they still require the following properties:

DPCP only: The state of all resources must be inspected to determine the maximum ceiling value of held resources.

PCP and DPCP: A node's priority may be changed during execution of a critical section.

The first property cannot be realized efficiently in the absence of shared memory. In a distributed system a number of messages have to be exchanged to collect information about global state while any other activities potentially modifying this state have to be suspended. Thus, DPCP is not suitable for distributed environments. The second property may result in multiple priority changes while holding a resource. This property can be realized by extending the prioritized protocol for reissuing requests, which will be discussed in the context of the priority inheritance protocol. Thus, PCP is suitable for distributed environments and can be realized by extensions of the prioritized protocol.

A third variant, the priority ceiling emulation protocol (PCEP), offers an even easier solution. This protocol combines the general idea of the original PCP [40] with early priority boosting to the ceiling level immediately upon granting resource requests as proposed in the stack resource protocol [4]. It also requires that the ceilings be computed statically (as for PCP). When granting a resource R_i, PCEP boosts the priority of a requester node N to $max(\xi(N),\ C_i)$. When releasing a resource R_i by node N, which still holds a subset L_h of L resources, PCEP lowers the effective priority of N to $max(\beta(N),\ max(C_h))$. The protocol avoids chained blocking, avoids priority changes while holding a resource, and may reduce the number of context switches for a task on node N. On the other hand, it may experience *false blocking* in situations where the previous protocols would not have blocked. The most important advantage of PCEP is its simplicity in terms

of implementation. As a result, PCEP has been adopted by industry standards such as POSIX and Ada95 [21].

PCEP can be used with minimal changes to the prioritized protocol presented so far. It can be extended as (bold face extensions in Figure 3.7):

1. Boost the requester's priority upon acquiring the resource in *ReceiveToken* or *ReceiveGrant* as described before.

2. Lower its priority upon releasing the resource in *RequestUnlock* as described before.

$\mathbf{ReceiveToken}()$
    ...
    $\xi(\mathbf{self}) \leftarrow \mathbf{max}(\xi(\mathbf{self}), \mathbf{C_{lock}})$
    ... //All the other operations
$\mathbf{ReceiveGrant}()$
    ...
    $\xi(\mathbf{self}) \leftarrow \mathbf{max}(\xi(\mathbf{self}), \mathbf{C_{lock}})$
    ... //All the other operations
$\mathbf{RequestUnlock}()$
    ...
    $\xi(\mathbf{self}) \leftarrow \mathbf{max}(\beta(\mathbf{self}), \mathbf{C_{held}})$
    ... //All the other operations

Figure 3.7: Extensions for Ceiling Emulation

Implementing PCEP in the example of Figure 3.3(a) will boost $\xi(A) = C_i = \xi(E)$ as soon as A acquires R on L_i. Consequently, the request $\{A, IR\}$ is never queued in Figure 3.4(b). Instead, it is granted immediately due to Rule 3.1 and Rule 6_{PRIO} and, hence, priority inversion is bounded.

These modifications do not require any additional message overhead. The resulting protocol provides the most efficient solution to bounding priority inversion in the protocols for real-time distributed concurrency control to our knowledge.

3.7.2 Priority Inheritance Protocol (PIP)

The PIP protocol requires that the priority of a node holding a resource be boosted to some value upon detection of a contention for this resource [40]. The protocol also requires transitive boosting when node A requests resource R_i held by node B and node B has issued a request for

resource R_j held by node C. This situation is commonly known as chained blocking. In this case, C will inherit A's priority (if $\xi(A) > \xi(C)$) since node C causes node A to be blocked indirectly through node B. Transitive priority changes may cause a chain of priority boosts along the contention dependencies of resources.

Also, considering the global context of locks L, a node N may have its effective priority boosted due to a contention for multiple locks in L that N is holding. Hence, when N releases one of its locks, its effective priority should be the highest of the effective priorities due to all the locks it is still holding. To calculate this new effective priority, we keep track of the highest boosted priority for each of the locks and call it the effective priority of a lock $\xi(lock)$. We describe the extensions as follows (see bold-face extensions in Figure 3.8):

HandleRequest: If the request for M_R is being queued at the token node (a contention is detected) then

1. Send the freeze messages with frozen thresholds to its children. (This notifies the children holding conflicting modes for this lock for boosting their priority as stated in Lemma 1.)

2. If the request conflicts with token node's (self) held mode, boost self's effective priority to $\xi(self)$ = max($\xi(self)$, $\xi(M_R)$) and boost the effective priority of this lock to $\xi(lock)$ = max($\xi(lock)$, $\xi(M_R)$). If self's effective priority is changed, re-issue all the pending requests for all the locks $L_i \in L$. (This boosts self's effective priority in case of a conflict.)

HandleFreeze: If a child receives a freeze message from its parent and if the child is *holding* the lock in any mode, a conflict between a child's held mode and a recently queued requested mode at the token node is detected. The child boosts its effective priority to $\xi(self)$ = max($\xi(self)$, max(Frozen_Modes.threshold)). If self's effective priority is changed, the child reissues all the pending requests for all the locks $L_i \in L$. The child also boosts the effective priority for this lock to $\xi(lock)$ = max($\xi(lock)$,max(Frozen_Modes.threshold)).

RequestUnlock: When a lock is released, self's effective priority is lowered to $\xi(self)$ = max(β (self), max($\xi(\forall L_i \in L$: IF(Held(L_i)) THEN $\xi(L_i)$ ELSE NIL))). Once the lock is released, the effective priority for this lock is lowered to the lowest priority NIL, *i.e.*, $\xi(lock) = NIL$.

HandleReissuedRequst: If a reissued request is received that is already present in our local queue, the queued request's effective priority is updated to the reissued request's effective priority. This may require reordering of the queue according to Rule 8.1. Also, if we are not the token node and the reissued request has higher priority than our pending request, we can no longer queue the request

```
HandleRequest(M_R)
    IF Self ≠ Token_node THEN
        IF grantable(M_O, M_R) ∧ Rule 6_PRIO THEN [Rule 3.1]
            ...
        ELIF queuable(M_P, M_R) THEN [Rule 4.1,Table3.2]
            ...
        ELSE [Rule 4.1]
            ...
    ELSE
        IF tokenable(M_O, M_R) THEN [Rule 3.2]
            ...
        ELIF grantable(M_O, M_R) THEN [Rule 3.2]
            ...
        ELSE [Rule 4.2]
            Queue ← Queue + M_R
            IF conflicts(M_H, M_R) THEN
                ξ(self) ← max(ξ(self), ξ(M_R))
                ξ(lock) ← max(ξ(lock), ξ(M_R))
                IF changed(ξ(self)) THEN
                    re − issue all the pending requests
                    for all the L_i ∈ L
            Update Frozen_Modes [Inv 2.2]
            Send Freeze to Children [Inv 2.3]
HandleFreeze(Frozen_Modes)
    IF M_H ≠ φ THEN
        ξ(self) ← max(ξ(self), max(Frozen_thresholds))
    IF changed(ξ(self)) THEN
        re − issue all the pending requests
        for all the L_i ∈ L
    ξ(lock) ← max(ξ(lock), max(Frozen_thresholds))
    Send Freeze to Children [Inv 2.3]
RequestUnlock()
    ...
    ξ(self) ← max(β(self), max(∀ L_i ∈ L : ξ(L_i)))
    ξ(lock) ← NIL
HandleReissuedRequest()
    IF req ∈ Queue THEN
        ξ(req) ← ξ(new_req)
        Rearrange the queue [Rule 8.1]
        IF Self ≠ Token_node THEN
            IF ξ(M_P) < ξ(new_req) THEN
                Dequeue and forward the request [Tab 3.2]
        Update Frozen_Modes [Table 3.1]
        Send freeze to children [Inv 2.3]
```

Figure 3.8: Extensions for Inheritance

locally (Table 3.2). Hence, the request is dequeued and forwarded to the parent. Also, updating the priority of the queued request may cause the frozen thresholds to be boosted and sent to the children according to Table 3.1 and Invariant 2.3.

Notice that while sending freeze messages to children with necessary thresholds (as given by Table 3.1 and Invariants 2.2 and 2.3), we assume that whenever there is a potential conflict between the request being queued and the held mode of the children, a freeze message will always be sent to those children. Lemma 1 explicitly states this requirement:

Lemma 1 *If a child owns lock L_i in mode M_{OC} and a request for mode M_R is received by the token node owning L_i in M_{OT} such that the modes are in conflict, then a freeze message will be sent to the child.*

Proof:

conflicts(M_{OC}, M_R) $\Rightarrow$ conflicts(M_{OT}, M_R) [as $M_{OC} < M_{OT}$].

Let M_{FT} be a set of all modes to be frozen at the token node according to Invariant 2.2, $\forall M_i \in M_{FT}$:

$$conflicts(M_i, M_R) \ \wedge \ [grantable(M_{OT}, M_i) \ \vee \ tokenable(M_{OT}, M_i)] \tag{3.1}$$

Let M_{FC} be a set of all modes that will be sent to the child according to Invariant 2.3, $\forall M_j \in M_{FC}$:

$$grantable(M_{OC}, M_j) \wedge conflicts(M_j, M_R) \wedge [grantable(M_{OT}, M_j) \ \vee \ tokenable(M_{OT}, M_j)] \tag{3.2}$$

If the M_{FC} calculated in constraint 3.2 is non-empty, the token node will send a freeze to the child.

By substituting $M_j = M_{OC}$ in constraint 3.2, the constraint is satisfied as $\forall M_{OC}$:

$grantable(M_{OC}, M_{OC})$ [U and W mode owners cannot be children], conflicts(M_{CO}, M_R) is true and grantable(M_{OT}, M_{OC}) is always true for a child of a token node.

$\Rightarrow$ M_{FC} is non-empty.

⇒ *a freeze message will be sent to the child*.

Chapter 4

Experiments

Our experimental results not only serve the purpose of empirical evaluation and analysis, but they also demonstrate the feasibility and applicability of the protocol for different application areas. The performance and applicability of our protocol are elaborated in the respective contexts in the discussion below. The experimental setup and the protocol parameters are designed to match a unique set of applications in each case.

4.1 Environment Setup

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 among 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) are randomized with different mean values, depending on the type of experiment. The modes of lock requests are 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 in the unprioritized case, and from 8 to 48 in the prioritized case, and the aforementioned experiment is repeated for each configuration.

4.2 Unprioritized Experiments

Our unprioritized 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. [32], 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.

4.2.1 Comparison

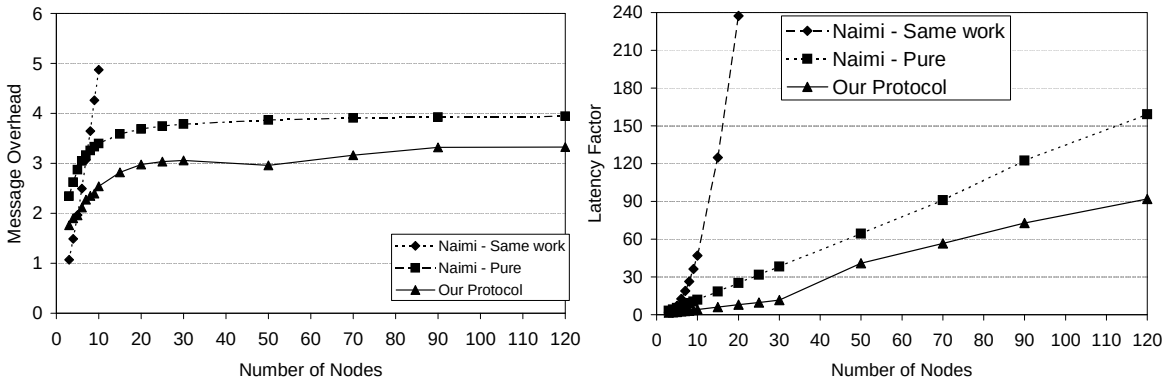
The first set of experiments focuses on the comparison of our hierarchical protocol with its non-hierarchical counterpart [32]. Experiments are conducted on Red Hat 7.3 Linux machines with 16 AMD Athlon XP 1800+ processors connected by a full-duplex FastEther TCP/IP switched LAN allowing disjoint point-to-point 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 virtual 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 are 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

[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.

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. For example, when the entire table is accessed, our protocol utilizes a single non-intention lock 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 4.1(a) assesses the scalability in terms of the average number of messages being sent for each lock request.



(a) Message Overhead Comparison

(b) Response Time Comparison

Figure 4.1: Comparison with Naimi's Protocol

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 with approximately 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 in their subtrees.

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 4.1(b) 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.

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.

4.2.2 Effects of Concurrency Level

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 are 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 are 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

$$Conc(N) = N \times \frac{Critical\ Section\ Time}{NonCritical\ Code\ Time} \tag{4.1}$$

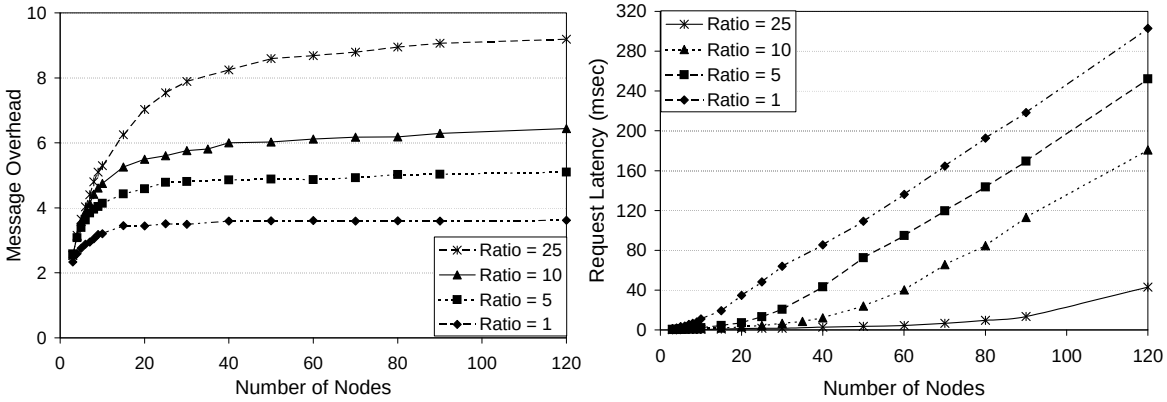
where N is the number of nodes in the system. Below, we describe experiments under different concurrency levels to study its impact on response time behavior.

Variable Concurrency Level

In these experiments, we kept the critical section time constant at a mean value of 15 msec (safe from OS timer granularity limitations) 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 4.2(a) 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 approximately 9 messages for ratios one, five, ten and 25, respectively. These results show that message overhead varies between



(a) Message Overhead

(b) Response Time

Figure 4.2: Varying Non-Critical/Critical Ratios

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 4.2(b) depicts the average request latency in msec for different ratios as the number of nodes are varied for each ratio.

One purpose in this experiment is to demonstrate the effect of the varying ratio of non-critical code time and critical section time, even though some critical sections cannot be parallelized and are subsequently subject to Amdahl'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 4.1.

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). 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 $N \times p(M)$, 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 2.1, we infer:

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

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

$$c(N) = C_0N^2 + C_1N^2 + C_2N^2 + C_3N^2 + C_4N^2 \tag{4.2}$$

where C_i are constant factors. Equation 4.2 indicates that $c(N) \in \Theta\ (N^2)$. 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 4.2.

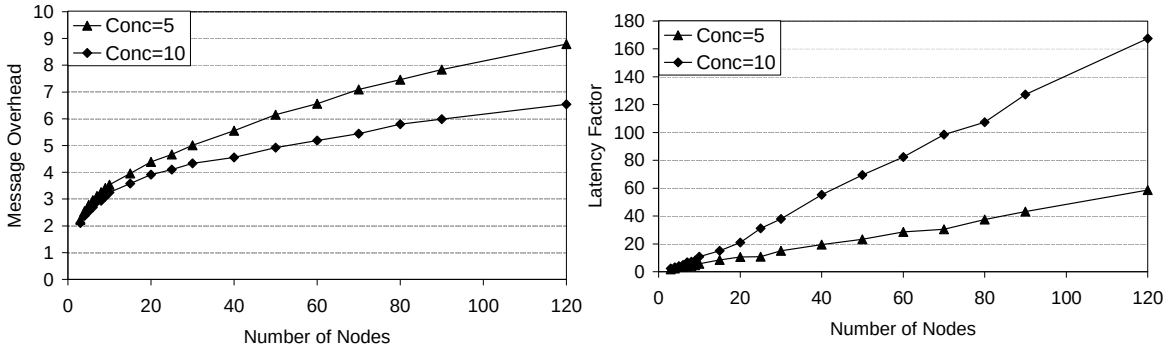
Constant Concurrency Level

The previous section explained the linear behavior (in the asymptotic case) of the response time dependent on the concurrency level $Conc(N)$. Hence, if $Conc(N)$ were kept constant, $c(N)$ would be 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 non-critical code time from NCT to $NCT\frac{N+K}{N}$ as we increase the number of nodes from N, to $N + K$. This 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.[2]

Figure 4.3(b) shows the response time behavior obtained for two different constant concurrency levels, both of which result in linear latency factors.

This behavior can be explained by considering the message overhead behavior shown in Figure 4.3(a). 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 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 4.4(a).

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



(a) Message Overhead

(b) Response Time

Figure 4.3: Constant Concurrency Level

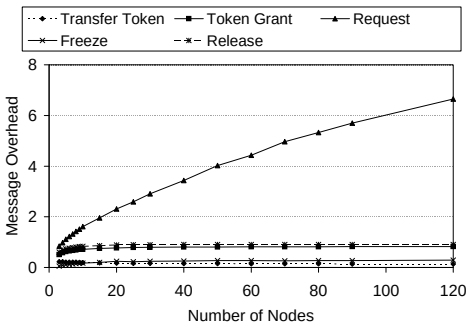
While all other types of messages demonstrate logarithmic behavior, the request propagation message increases linearly with an increase in the number of nodes. Since the message overhead is one of the factors affecting response time, the response time behavior is also linear in spite of constant queue 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 to provide QoS guarantees.

4.2.3 Message Overhead Breakups

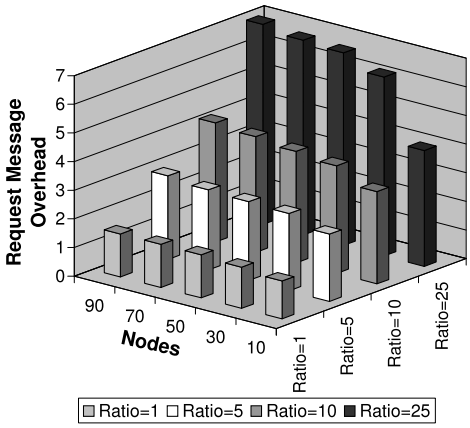
Figures 4.4(b), 4.4(c), 4.4(d), 4.4(e) and 4.4(f) depict the change in message overhead for the message types request, grant, transfer token, release and freeze, respectively. In each case, changes are shown for varying number of nodes and different ratios. Explanations about each of these figures illustrate the behavior of the protocol and its properties.

Request messages increase with the number of nodes and with higher ratios, as seen in Figure 4.4(b). 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).

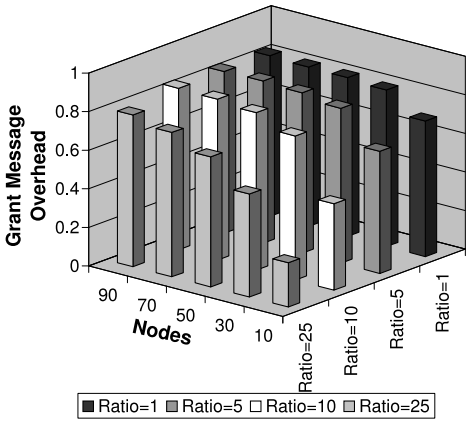
A similar trend is shown for token grant messages in Figure 4.4(c). With more contention,



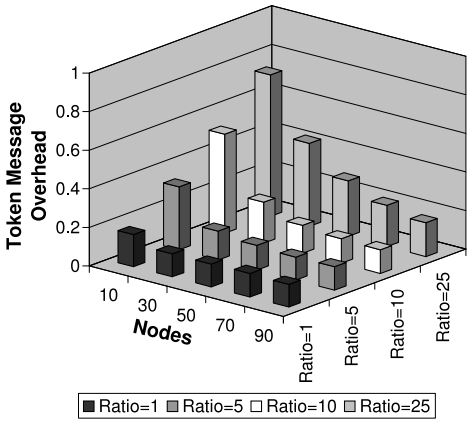
(a) Message Overhead Breakup



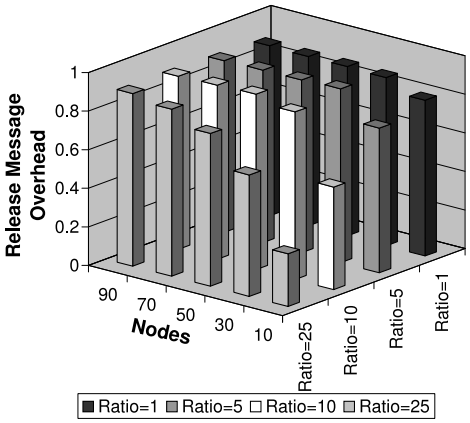
(b) Request Message



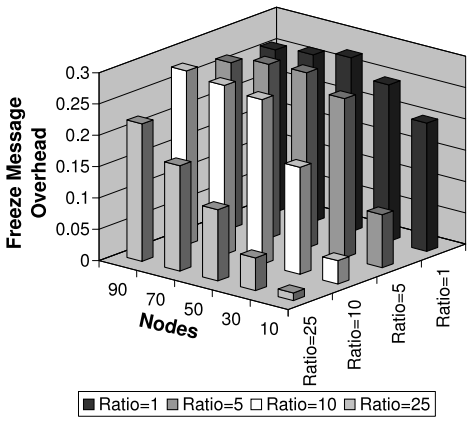
(c) Grant Message



(d) Token Transfer Message



(e) Release Message



(f) Freeze Message

Figure 4.4: Message Overhead Breakup

more concurrent requests can be resolved by allowing children within a copyset to grant requests. The overall contribution is below one message.

The token transfers in Figure 4.4(d) increase at high ratios when concurrency is low, *i.e.*, during low contention, the token node 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).

Release messages in Figure 4.4(e) 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.

Freeze messages in Figure 4.4(f) 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.

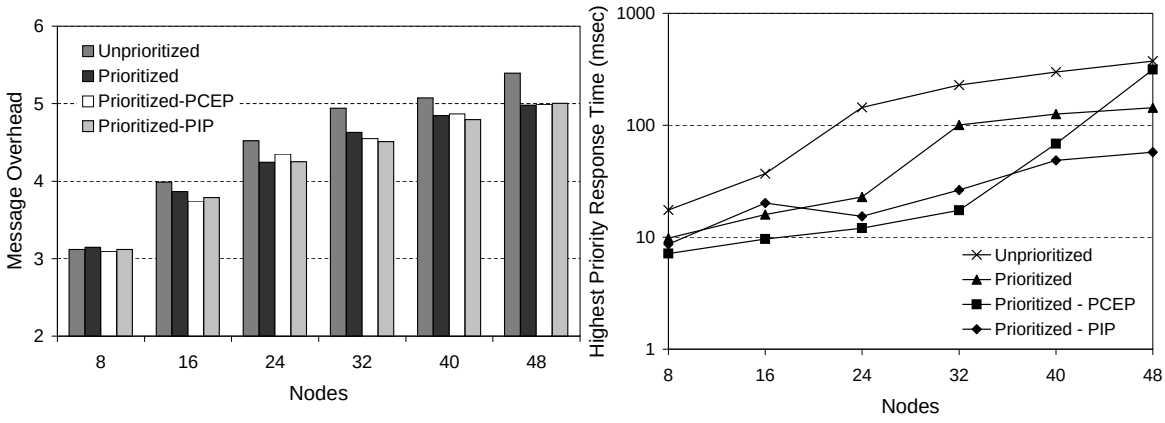
4.3 Prioritized Experiments

Here, we experiment with the prioritized version of the protocol. First, we evaluate the performance of our protocol relative to the unprioritized protocol described in chapter 2, which is the best known scalable protocol for distributed concurrency services with hierarchical locking support. Second, we analyze the effect of supporting PCEP and PIP on the response times relative to the protocol version without any priority inheritance support. We also demonstrate clear separation in response times among distinct priority levels in all three cases. We further show that the message overhead introduced due to reissued requests in PIP is negligible. Third, we investigate the queuing mechanism in detail and quantify and compare the degree of preference given to the higher priority requests in the cases of no inheritance, PCEP and PIP.

As with the previous comparison experiments, all the prioritized experiments are 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. Each node is assigned one of the 8 available priority levels equally divided amongst the number of nodes. The priority of a request without support for inheritance is the base priority of the requesting node while, in case of inheritance versions PCEP and PIP, priority of the request made by a node is determined by its effective priority at that instant.

4.3.1 Comparison

In this subsection, we compare the message overhead scalability and response time behavior of the unprioritized protocol with the prioritized protocol as well as PCEP and PIP versions of the prioritized protocol. In case of the unprioritized version we compute the response averaged over all requests while in all other cases, we consider the response time averaged for the higher priority requests. The objective is to demonstrate the scalability in terms of message overhead as well as in terms of response time as experience by the higher priority requests.



(a) Message Overhead

(b) Response Time

Figure 4.5: Comparison with Unprioritized Protocol

Figure 4.5(a) compares the message overheads of our protocols. Note the logarithmic increase in the message overhead (number of messages exchanged for each request) as we increase the number of nodes in the system stabilizing at around 5 messages for 48 nodes. Notably, the

additional support for priorities comes without any penalties in terms of message overhead. Also, the additional messages for reissued requests in case of PIP is negligible. This confirms our claim for the scalability in terms of message overhead.

Figure 4.5(b) shows the comparison of response times. There are several interesting trends to be studied here. First, as expected, the response time for the unprioritized version is always higher (by about an order of magnitude) than that for the higher priority requests in all of the prioritized versions.

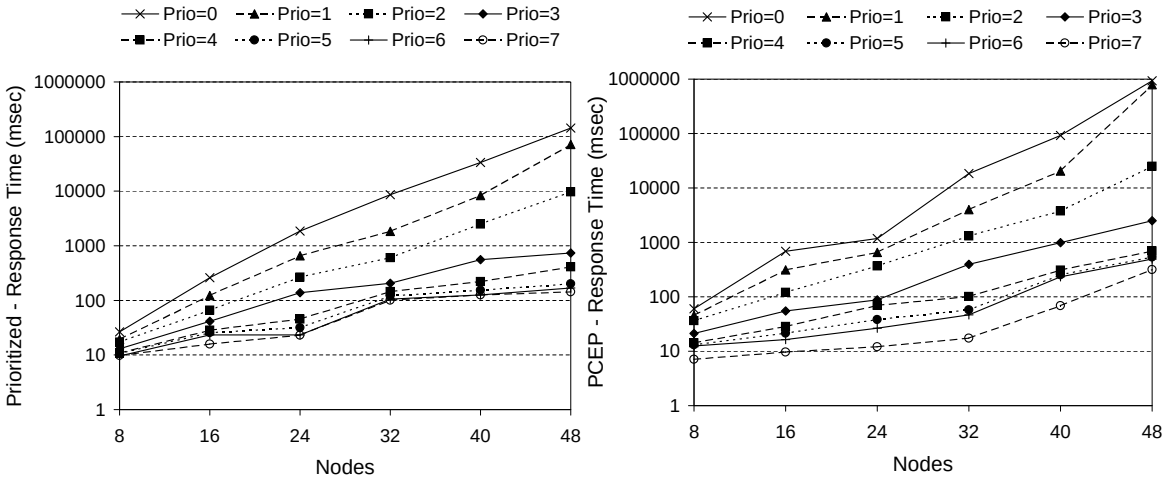
Second, till 32 nodes, the prioritized version without inheritance support suffers from severe inversion problems and has higher response times compared to the one with support for PIP/PCEP. However, for more than 32 nodes in case of PCEP, the problem of false blocking becomes more severe and response times grow faster. As all the nodes may access any of the shared resource, the ceilings of all the resources are at the highest priority in the system (*i.e.*, 7). And following the hierarchical locking protocol, the effective priority of each node is boosted to the highest priority on acquiring the outer lock on the table, regardless of the future possibilities of a potential conflict. Hence, as requests from all nodes contend for the inner locks on the individual tuples, they have the same priority level, *i.e.*, the boosted highest priority. The lost differentiation amongst priority levels inflicts higher response times for higher priority requests.

Third, PCEP minimizes the priority inversion and performs better than PIP till about 32 nodes for the same reason. Beyond 32 nodes, the problem of false blocking becomes more severe and the inversion problem dominates the results.

4.3.2 Effects of Inheritance Protocols

In this subsection, we take a closer look at the response time behavior for individual priority levels in the prioritized version of the protocol (with and without the support for inheritance protocols). For all three cases, our experiments show that lower priority requests have response times several magnitudes higher than that for higher priority requests, and they grow much faster. On the other side, higher priority requests have almost better than linear response times growing predictably and slowly as the number of nodes increase.

Figure 4.6(a) shows the response times for each priority level in case of the prioritized version without inheritance support. Here, there are some occurrences of anomalies as the response times are quite similar for some of the higher priority requests and they do cross each other. This is caused by unbounded priority inversion that disregards the priority distinction.



(a) Response Time for Prioritized Protocol

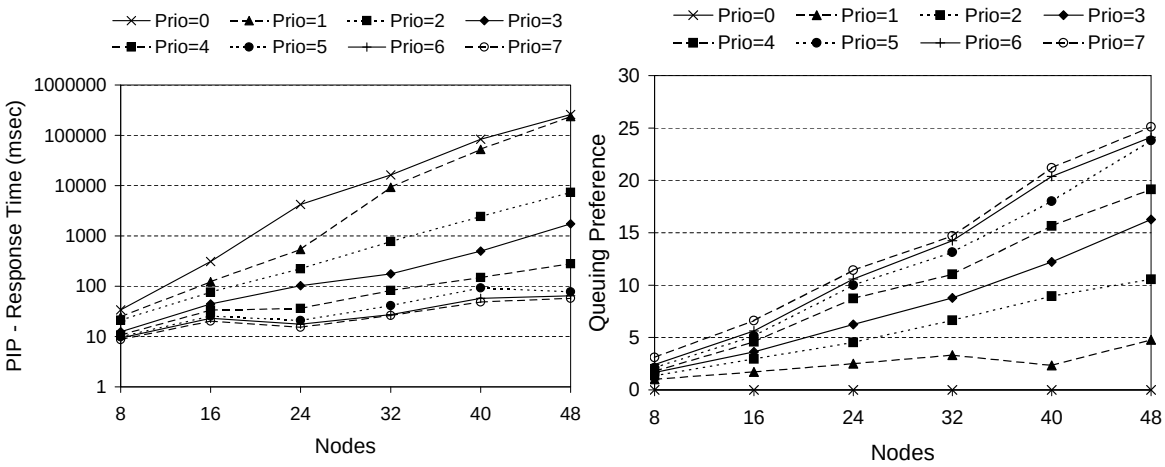
(b) Response Time for Prioritized Protocol with PCEP

Figure 4.6: Pure Prioritized and PCEP

Figure 4.6(b) shows the response times for each priority level in case of the prioritized version with PCEP support. Till 32 nodes, the individual levels are evenly separated and have quite predictable behavior. This is due to the tight bound achieved by PCEP on the priority inversion. But for more than 32 nodes, the problem of false blocking inflicts higher and faster growing response times for each priority level as explained before. However, we still observe a degree of separation among individual levels as the bound on inversion is still in effect.

Figure 4.7(a) shows the response times for each priority level in case of the prioritized version with PIP support. Here, the response times for higher priority requests only increase insignificantly (are almost constant) as we increase the number of nodes. Also, we do not see any anomalies (intersecting curves) as inversion is bounded under PIP without false blocking. However, as compared to PCEP, the separation among the levels is less pronounced. This can be expected as PCEP offers bounded priority inversion while PIP does not. Since, PIP avoids false blocking, it scales well with large numbers of nodes.

We draw several conclusions from the results above. First, the prioritized versions of the protocol certainly offer response times several orders of magnitude lower for higher priority requests. Second, PCEP and PIP support can control the priority inversion problem and make the protocol more predictable. Third, PCEP has limited scalability when applied to hierarchical locking



(a) Response Time for Prioritized Protocol with PIP

(b) Queuing Preference Degree Vs. Concurrency

Figure 4.7: PIP and Queuing

schemes due to the problem of false blocking introduced due to uniform resource ceilings.

4.3.3 Queuing Preference Degree

In this subsection, we study the queuing behavior for the prioritized version of the protocol and comment on the suitability of prioritized protocols with respect to the concurrency levels. Quantitatively, the concurrency level is defined in Section 4.1 $conc(N)$ defines the number of concurrent requests across the system at any time instant. As we increase the number of nodes in the system, the degree of concurrency increases, and so does the potential for conflicts among the requests. Let the degree of queuing preference denote the average number of nodes in the queue bypassed by a higher priority request while getting queued. The queue is ordered according to Rule 8. The motivation for this experiment is as follows:

The prioritized scheme proposed here tries to give preference to higher priority requests by queuing them in the order of their priorities and freezing a set of modes at a threshold as described in Section 3. However, freezing of modes and queuing occur only if the requests conflict. In case of low concurrency and fewer conflicts, a higher priority request and a lower priority request are not differentiated and receive identical treatment from the protocol. However, it should be obvious that in a system with a large number of nodes, concurrency is likely to be higher. Conversely,

if the protocol is employed in a system with sparse requests and low concurrency, differentiation of priority is not necessary, *i.e.*, all the requests will have ideal response times regardless of their priorities. This property of the protocol favors higher priority requests as the concurrency in the system increases, as seen by the increasing preference in Figure 4.7(b).

4.4 Summary

Summarizing our experiments, First, the basic protocol with the support for hierarchical locking, demonstrates very high scalability as compared to oen of the best prior protocols. protocols. This gains represent the technical strength of our approach of hierarchical locking in the context of distributed synchronization. Second, the scalability of the protocol is shown to be resilient to the concurrency level of the system, maintaining logarithmic behavior of message overhead. Third, we showed that neither the priority support nor the support for PCEP and PIP affect the scalability of the protocol. Finally, we compared and contrasted the properties of PCEP and PIP.

Chapter 5

Related Work

A number of algorithms exist to solve the problem of mutual exclusion in a distributed environment. Chang [8] and Johnson [22] give an overview and compare the performance of such algorithms. Goscinski [16] proposed a prioritized algorithm based on broadcast requests using a token-passing approach. Chang [7] developed extensions to various algorithms for priority handling that use broadcast messages [48, 42] or fixed logical structures with token passing [37].

Our protocol differs from Chang's extensions of [48] and [37] by not requiring broadcasts at all and lower average message complexity, respectively, as detailed below. Chang, Singhal and Liu [9] use a dynamic tree similar to Naimi *et al.* [31, 32]. 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 thesis. Other mutual exclusion algorithms (without token passing) employ global logical clocks and timestamps [24]. 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 re-

quests per node to provide more concurrency within a multi-threaded environment [29]. None of the above algorithms have been studied with regard to their applicability to concurrency services, neither have they considered hierarchical locking models to achieve optimal concurrency to the best of our knowledge.

Hierarchical locks and protocols for concurrency services have been studied in the context of database system [17, 26, 25, 23, 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 [38, 5]. 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 [33] applicable to real-time database systems [43, 46]. In contrast, we make priority support and scalability a first-class property of protocols that implement CORBA-like services, such as hierarchical locking defined through the concurrency services. Our support to bound priority inversion is compatible with POSIX/CORBA synchronization operations but goes beyond these standards in its unprecedented support for hierarchical locks.

Another related prior work is our own algorithm on prioritized non-hierarchical locking and its support for PCEP and PIP [30]. This prior work focused on distributed mutual exclusion support equivalent to binary semaphores with a detailed operational base protocol and priority inheritance extensions. Our new approach not only supports the more sophisticated paradigm of hierarchical locking, it also shows how rules and tables significantly simplify the protocol while ensuring real-time properties.

The application area of our protocol lies in distributed real-time systems. A number of approaches have been proposed to schedule real-time tasks on multi-processors and in distributed systems. An optimal algorithm for non-preemptive scheduling with precedence constraints and exclusion relations is quite powerful [49] based on segments (critical sections) combined with off-line schedules to ensure mutual exclusive execution of these segments. Our approach differs through its support for preemption and arbitrary ordering between critical sections, which results in a more flexible schedule. Other approaches assume that tasks with shared resources be scheduled on the same processor. Rajkumar *et al.* describe a method to schedule tasks statically assigned to processors in a shared-memory system with proxy (remote) execution of critical sections for remote resources [34]. Remote resources allocation schemes in multiprocessors under real-time scheduling often utilize heuristic methods since the general problem of distributed scheduling is NP-complete [36]. Heuristic algorithms have also been proposed for distributed systems to select nodes for tasks when

they become ready and consider the local availability of shared resources [35]. Another approach assumes that tasks are independent but subtasks may be scheduled on different processors while the communication between subtasks can be modeled as a task as well [47]. Our approach starts with assumptions similar to Rajkumar *et al.* in that it requires static or dynamic priority preemptive scheduling and strict priority scheduling applicable but not limited to rate-monotonic priority assignments [28]. Our model differs in that the location of resources or task interdependencies is arbitrary. We focus on a protocol to acquire hierarchically organized resources in a distributed environment, bound the message overhead and provide support to prevent priority inversion so that this protocol can be used in arbitrary distributed scheduling methods for real-time systems.

Chapter 6

Conclusions

We presented a novel peer-to-peer protocol for multi-mode hierarchical locking, which is applicable to transaction-style processing, real-time, distributed and middleware services. We demonstrate high scalability combined with lower response times in high-performance cluster environments. We also address the problem of priority inversion by supporting two of the most prominent inheritance protocols: PCEP and PIP. 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 code 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 120 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.

Our prioritized experiments on up to 48 computing nodes indicate that adding support for priorities does not affect the scalability of our protocol. It also shows that the higher priority

requests have predictable response times. Support of PCEP bounds priority inversion but introduces the problem of false blocking. While the PIP solution does not suffer from false blocking, it allows slightly greater amount of priority inversion to happen.

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.

Thus, the results of our work impact real-time applications sharing resources across large distributed environments ranging from hierarchical locking in real-time databases and database transactions to distributed object environments in large-scale embedded systems.

Appendix A

Glossary of Terms

Base priority: A static priority assigned to each node

Basic protocol: Our hierarchical locking protocol without priority support

Compatible modes: Modes that do not conflict with each other according to Table 2.1

Concurrency level: Number of concurrent requests in the system at a time instant

Copy grant: Granting of lock access without transferring the token

Copy set: Set of all nodes holding a common lock simultaneously at a time instant

Effective priority: Current priority of the node; possibly higher than its base priority in case of priority inheritance

Held mode: Mode in which the critical section is enetered

Hierarchical locking: Locks are associated to hierarchies of data structures a.k.a. multi-granularity locking

Message overhead: Average number of messages sent in the system per lock request

Mode strength: Degree of incompatibility with other modes in Table 2.1

Naimi's same work experiments: Experiments with Naimi's protocol where entire data structure is shared with fine grain locks association

Naimi's pure experiments: Experiments with Naimi's protocol where single data item is shared as in coarse grain locks association

Non-hierarchical protocol: Naimi's token-based distributed mutual exclusion protocol

Owned mode: Strongest held mode in the subtree of the node

Pending mode: Mode for which the request is made and grant is awaited

Prioritized protocol: Our hierarchical locking protocol with priority support

Queuing preference: Average number of nodes bypassed by a higher priority request while being queued

Ratio: Ratio of non critical code time to critical code time

Request latency: Response time for lock request

Token grant: Granting of lock access and transferring token to the requester

Token node: Root node of the tree having token

Unprioritized protocol: Our hierarchical locking protocol without priority support

Bibliography

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

[2] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *J. of Real-Time Systems*, 8:173–198, 1995.

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

[4] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[5] 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.

[6] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

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

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

[9] 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.

[10] M.I. Chen and K.J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. Technical Report UIUCDCS-R-89-1511, University of Illinois at Urbana-Champaign, Department of Computer Science, April 1989.

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

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

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

[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. Geist and C. Engelmann. Development of naturally fault tolerant algortihms for computing on 100,000 processors. *J. Parallel Distrib. Comput.*, page submitted, 2002.

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

[17] 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.

[18] Object Management Group. Concurrency service specification. http://www.omg.org/technology/documents/formal/concurrency_service.htm, April 2000.

[19] Object Management Group. Corba/iiop specification. http://www.omg.org/technology/documents/formal/corba_iiop.htm, December 2002.

[20] H. C. Hoppe and D. Mallamann. Eurogrid - european testbed for grid applications. *General article on the EUROGRID project in GRIDSTART Technical Bulletin*, October 2002.

[21] ISO/IEC. *ISO/IEC 8652L 1995 (E), Information Technology – Programming Languages – Ada*, February 1995.

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

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

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

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

[26] 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.

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

[28] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, January 1973.

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

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

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

[32] 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.

[33] 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.

[34] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–269, December 1988.

[35] Krithi Ramamritham and John A. Stankovic. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Transactions on Computers*, 38(8):1110–1123, August 1989.

[36] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

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

[38] 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.

[39] L. Sha, R. Rajkumar, S. H. Son, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, July 1991.

[40] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols - an approach to real-time synchronization. Technical report, Carnegie Mellon University, Departments of CS, ECE and Statistics, Pittsburgh, Pennsylvania, 1987.

[41] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, September 1990.

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

[43] 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.

[44] S. H. Son, S. Park, and Y. Lin. An integrated real-time locking protocol. In *International Conference on Data Engineering*, pages 527–534, Los Alamitos, Ca., USA, February 1992. IEEE Computer Society Press.

[45] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems*. Kluwer, 1998.

[46] 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.

[47] J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. In *Int. Conference on Distributed Computing Systems*, pages 38–45, May 1996.

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

[49] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, 19(2):139–154, February 1993.