Implementing Concurrency Control With Priority Inheritance in Real-Time CORBA

BY

Steven Wohlever, Victor Fay Wolfe, and Russell Johnston

June 1997

ABSTRACT

Distributed object computing environments (DOCEs) must support real-time processing if they are to provide the timely execution required by many complex applications such as telecommunications, automated manufacturing, aerospace automated control, medical patient monitoring, and multi-media. To this end, any concurrency control mechanism used by a real-time DOCE (RTDOCE) must enforce timing constraints.

One popular DOCE is the Common Object Request Broker Architecture (CORBA). This DOCE is currently gaining popularity in industrial, government, and academic projects. However, in order for it to be suitable for real-time computing, CORBA must be extended to support real-time characteristics including real-time concurrency control.

Concurrency control in a real-time environment can lead to *priority inversion*. Priority inversion occurs when a real-time activity blocks another, higher-priority real-time activity. This situation can lead to unbounded blocking time for the higher-priority activity. Therefore, any concurrency control mechanism used in a real-time system must ensure that priority inversion is bounded.

This report presents an implementation of a dynamic real-time distributed concurrency control mechanism that has been developed as part of a larger project at the University of Rhode Island that is designing a RTDOCE based on CORBA. This mechanism uses *basic priority inheritance* to bound priority inversion. This report describes the design and implementation of this mechanism in addition to presenting

tests results which illustrate that, while the mechanism cannot ensure that all timing constraints are met, it does contribute to a best effort approach for ensuring that high priority activities meet their timing constraints.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This report describes a concurrency control mechanism that was designed for use in Real-Time CORBA (RTCORBA), a real-time distributed object computing environment. It is a lock-based mechanism that uses *basic priority inheritance* to bound *priority inversion* that occurs during access to shared data. This report describes the implementation of the mechanism. It also presents results that demonstrate that the mechanism provides a correct implementation of basic priority inheritance, which contributes to RTCORBA's best effort approach for enforcing time constraints.

1.1 Motivation

Distributed object computing environments (DOCEs) must support real-time computing if they are to provide the timely execution required by many complex *real-time applications* such as telecommunications, automated manufacturing, aerospace automated control, medical patient monitoring, and multi-media [RTSIG96]. In the con-

text of this report, real-time computing is defined to be computing that is *predictable* in terms of the performance of the system. This is in contrast to real-time computing defined as *fast* computing. A real-time application is defined to be an application that must meet timing constraints in order to be correct [WBTK95]. In support of this requirement, any concurrency control mechanism used by a real-time DOCE (RTDOCE) must enforce timing constraints.

One popular DOCE is the Common Object Request Broker Architecture (CORBA). CORBA is a DOCE specification that was created by the Object Management Group (OMG), a group of over 600 DOCE vendors and users. This DOCE is currently gaining popularity in industrial, government, and academic projects. However, in order for it to be suitable for real-time computing, the current version of CORBA, CORBA 2.0, must be extended to support real-time characteristics.

CORBA 2.0 defines an interface to the *Concurrency Control Service* (CCS) in [OMG96]. The CCS is designed to provide lock-based concurrency control for CORBA objects. In order to support real-time computing, the CORBA CCS must be modified for use in a RTCORBA environment.

Concurrency control in a real-time environment can lead to *priority inversion* [SRL90]. Priority inversion occurs when a real-time activity blocks another, higher-priority real-time activity. This situation can lead to an unbounded blocking time for the higher-priority activity. Therefore, any concurrency control mechanism used in a real-time system must ensure that priority inversion is bounded. In addition, the concurrency control mechanism must address the issue of *deadlock*, the situation in which activities are indefinitely blocked due to locking conflicts. This will be

addressed further in Chapter 2.

Although work has been done in the area of real-time distributed concurrency control, it either does not address the issue of priority inversion or it requires *a priori* information about the priorities of the activities that will run. Therefore, there is a need for a real-time distributed concurrency control mechanism that will bound priority inversion and prevent deadlock while allowing for *dynamic* workloads. In the context of this project, a system supports dynamic workloads if it does not require *a priori* information about the activities that will run on it.

This report presents an implementation of a dynamic real-time distributed concurrency control mechanism that was developed for use in CORBA systems as part of a larger project at the University of Rhode Island that is designing a RTDOCE based on CORBA. This mechanism uses *basic priority inheritance* [SRL90] to bound priority inversion. This protocol does not require *a priori* information about the activities that will run in the CORBA environment. This makes it more suitable for systems that require dynamic workloads. This report describes the design and implementation of this priority inheritance mechanism in RTCORBA in addition to presenting test results. These results illustrate that, while the mechanism cannot ensure that all timing constraints are met, it does contribute to RTCORBA's best effort approach for ensuring that high priority activities meet their timing constraints.

1.2 Goal of Research

The main goal of this research is to develop a *Real-Time Concurrency Control Service* (RTCCS) for use in a prototype RTCORBA environment. The RTCORBA project is a *soft real-time* system in the sense that it is designed to make a best effort to meet timing constraints. This is in contrast to a *hard real-time* system in which violated timing constraints lead to catastrophic results. The RTCCS provides an extended form of read/write locking which allows for consistent, concurrent access to shared data resources. This service uses basic priority inheritance to bound priority inversion for dynamic workloads.

Another goal of this project is to develop the RTCCS to comply with established standards whenever applicable. In order to remain compliant with the CORBA 2.0 specification, this implementation complies with the specified interfaces for the CCS [OMG96] with a few minor extensions. In addition, the implementation is compliant with the real-time POSIX operating system interface standard [Gal95].

1.3 Approach

In order to achieve these goals, several existing algorithms for bounding priority inversion were examined. These included a number of *priority ceiling protocols* [SRL90, Raj91]. However, since these protocols require *a priori* information about the priorities of activities that will run, they are inappropriate for systems that have dynamic workloads. Therefore, basic priority inheritance, which does not need *a priori* information, was chosen to bound priority inversion in our implementation of the CCS.

1.4 Outline

Chapter 2 provides an overview of CORBA and the RTCORBA project at the University of Rhode Island. It then presents a review of lock-based concurrency control, priority inheritance, deadlock prevention, and priority ceiling protocols. The chapter concludes with a presentation of an existing real-time concurrency control mechanism for CORBA. Chapter 3 describes the design of the RTCCS while Chapter 4 describes its implementation. Chapter 5 presents the results of correctness and performance tests done using the RTCCS. Chapter 6 presents the contributions and limitations of this prototype RTCCS and discusses future work.

Chapter 2

Related Work

This chapter presents a brief summary of CORBA and the RTCORBA project at the University of Rhode Island. It then presents some background information on lock-based concurrency control, priority inheritance, and priority ceiling protocols. This includes a discussion of a priority ceiling protocol designed specifically for distributed systems. Finally, an existing implementation of a RTCCS for CORBA is examined.

2.1 CORBA

CORBA is a specification for a DOCE with a client/server architecture. Clients interact with servers through the *ObjectRequestBroker* (ORB), the central piece of middleware in CORBA. The ORB is responsible for passing requests from clients to servers and sending results from servers back to clients. These requests are targeted at specific objects that exist in the server process' address space. For example, a server may contain a number of database objects, each with its own data and inter-

face. When a server receives a client request, it invokes the specified method on the appropriate object. In the case of multithreaded servers, a thread is created to handle the request, allowing the server process to accept additional client requests.

Each object managed by the server has an interface specified in an *interface definition language* (IDL) which is part of CORBA 2.0. These IDL interfaces are used to generate *stubs* and *skeletons*. A stub is library code that is linked into a client to enable it to interact with the server. Likewise, the skeleton code, which is linked into the server process, enables the server to interact with the client. A client begins its interaction by *binding* to the appropriate server. At this point, the client gets an object reference to the server object it wishes to interact with. The stub code provides it with interfaces to all of the available methods on the server object's interface. When these methods are called, the ORB is responsible for routing requests and results to the proper servers and clients.

The CORBA 2.0 specification also defines a set of *Object Services*. Theses services include a naming service, transaction service, and concurrency control service, among others. The Object Services are intended to provide a variety of low-level services to clients and servers in the CORBA environment.

2.1.1 CORBA Concurrency Control Service

The OMG has recognized the merit of a standard interface for concurrency control that can be used to manage access to shared resources in an ORB environment by specifying interfaces to a *Concurrency Control Service*(CCS). This is one of the Object

Services specified in CORBA 2.0. The motivation behind the inclusion of the CCS is that it provides a standard interface to a concurrency control mechanism that can be used by programmers to manage access to shared resources that do not have built-in concurrency control. For example, the CCS can be incorporated into a linked list object that was originally designed for single-user usage such that it can support safe, concurrent access.

The CCS provides an extended form of object-level read/write locking. Locks can be acquired on behalf of a transaction or on behalf of a client operating outside of a transaction. In the first case, a second Object Service called the *Transaction Service* drives the release of locks as the transaction commits or aborts. The RTCCS developed for the RTCORBA project does not provide support for this type of locking. It does provide support for the second form of locking in which the user of the CCS (i.e., the client that obtained the locks) is responsible for releasing them. In this non-transactional mode of concurrency control, a *LockSet* object embodies the locks that can be obtained on a single resource as well as the methods for obtaining and releasing specific types of locks. The *LockSet* is also responsible for ensuring that conflicting locks are not held by different clients. Any shared resource in the distributed system that requires concurrency control has its own instantiation of a *LockSet* object. Clients that access the resource should do so only after acquiring the necessary locks from the appropriate *LockSet* object.

The CCS supports five types of locks. The first two are the standard *read* (R) and *write* (W) locks. The *upgrade* (U) lock is useful for avoiding a common form of deadlock that arises when two or more activities with read locks try to get write locks.

Consider the following scenario involving two clients, C_1 and C_2, that are accessing shared object O.

1. C_1 gets a R lock on O.

2. C_2 gets a R lock on O.

3. C_2 tries to get a W lock on O but is blocked by C_1's R lock.

4. C_1 tries to get a W lock on O but is blocked by C_2's R lock.

Since each client is blocked by the other, deadlock ensues. This can be remedied by requiring the clients to obtain an upgrade lock instead of a read lock.

1. C_1 gets a U lock on O.

2. C_2 tries to get a U lock on O but is blocked by C_1's U lock.

3. C_1 gets a W lock on O and writes to the object.

4. C_1 releases its W and U locks on O, allowing C_2 to continue unhindered.

The last two locks are the *intention read* (IR) and *intention write* (IW) locks. These two locks are useful when the resource that is being locked is hierarchical in nature (e.g., a database). The following scenario illustrates how the intention write lock is used to managed three clients, C_1, C_2, and C_3, as they access a database. It should be noted that the database and each of the records in the database requires its own $LockSet$.

1. C_1 gets an IW lock on the database.

Granted Mode	Requested Mode				
	IR	R	U	IW	W
Intention Read (IR)					*
Read (R)				*	*
Upgrade (U)			*	*	*
Intention Write (IW)		*	*		*
Write (W)	*	*	*	*	*

Table 2.1: Lock Incompatibilities

2. C_1 gets a W lock on record R_1 in the database.

3. C_2 gets an IW lock on the database.

4. C_2 gets a W lock on record R_2 in the database.

5. C_3 tries to get a W lock on the entire database but cannot since the database has been locked with an IW lock.

The changes being made by C_1 and C_2 are protected from interference by C_3 by the IW locks they obtained. The intention read lock works in a similar manner for read operations. Table 2.1 defines the compatibilities between the five types of locks (a * indicates a conflict between a requested lock and a granted lock).

2.2 Real-Time CORBA

The work done on this report was funded by the Distributed Hybrid Database Architecture project [JWS96] that is being developed as a joint effort between the U.S. Navy NRaD labs and the RTCORBA group at the University of Rhode Island. The RTCORBA group is currently working on a client/server implementation of a RT-DOCE based on CORBA. The efforts of the group are focused on the expression and enforcement of *end-to-end timing constraints* on databases in CORBA environments. In this context, enforcing end-to-end timing constraints means the DOCE enforces the timing constraints of a real-time activity at all stages of it execution throughout the DOCE.

Recently, a Real-Time Special Interest Group (RT SIG) in the OMG has begun to identify those capabilities which it feels would be desirable in a real-time version of CORBA (CORBA/RT) [RTSIG96]. The goal of the RTCORBA group is to design and implement a prototype of CORBA/RT that meets the desired capabilities for expressing and enforcing time constraints as specified in the RT SIG's white paper([RTSIG96]). One of these capabilities is the need for priority inheritance to bound priority inversion. Therefore, any concurrency control mechanism in CORBA/RT should make use of priority inheritance.

2.3 Lock-Based Concurrency Control

When a data resource is accessed by multiple users simultaneously, steps must be taken to ensure that this concurrent access does not leave the data in an inconsis-

tent state. One method of ensuring consistency of a shared resource is to require that clients of the resource obtain locks on the resource. *Exclusive locking* involves obtaining a lock on the entire resource, preventing any other client from accessing the resource for any reason. *Read/write* locking allows for more concurrent access by allowing multiple clients to read from the resource as long as there are no writers accessing the data. If a client desires to write to the resource, it is allowed to do so only when there are no readers or writers accessing the data. This project implements an extension of this method of locking.

2.4 Basic Priority Inheritance Protocol

The basic priority inheritance protocol was developed to solve the problem of unbounded priority inversion. In order to understand the need for priority inheritance, consider the following example which involves three real-time activities, A_1, A_2, and A_3, in ascending order of priority. Note that a high priority activity that is ready to run preempts any lower priority activities that are running. Let O represent a shared object that will be locked by both A_1 and A_3. The following scenario then becomes possible:

1. A_1 gets a write lock on O.
2. A_3 preempts A_1.
3. A_3 tries to get a read lock on O but is blocked by A_1's write lock.
4. A_1 resumes execution.

5. A_2 can preempt A_1 any number of times since it does not try to lock O.

Since A_1 can potentially be preempted by A_2 any number of times, A_3 can be blocked for an unbounded amount of time since A_1 cannot release the lock on O until it completes its critical section (i.e., the code executed under the protection of the lock). This is unbounded priority inversion. The priority inversion in this example can be bounded in the following way:

1. A_1 gets a write lock on O.
2. A_3 preempts A_1.
3. A_3 tries to get a read lock on O but is blocked by A_1's write lock.
4. A_1's priority is raised to that of A_3.
5. A_1 resumes execution.
6. Since A_2's priority is lower than A_1's new priority, A_2 cannot preempt A_1.
7. When A_1 completes its execution, it releases the write lock and resets its priority to its original value.
8. A_3 is no longer blocked, and since it has the highest priority, it is allowed to obtain the read lock.

This example makes use of basic priority inheritance. A_1 is said to *inherit* A_3's priority at step 4. It has been shown in [SRL90] that the basic priority inheritance protocol places an upper bound on priority inversion. The number of critical sections

that can block a particular activity A is given by the smaller of the following two values:

1. The number of activities with priorities lower than that of A.

2. The number of locks that can block the activity A.

This is an important result since it places a bound on a previously unbounded delay. However, this bound can be substantial if there is *chained blocking* [SRL90]. Chained blocking refers to the situation in which a high priority activity is blocked by multiple activities with lower priorities.

The previous example illustrates *direct blocking* [SRL90]. The basic priority inheritance protocol must also contend with *transitive*, or *push-through* [SRL90], blocking. Consider the following example which involves an object O with two locks L_1 and L_2 that do not conflict with each other but do conflict with themselves.

1. A_1 gets lock L_1 on O.

2. A_2 preempts A_1.

3. A_2 gets lock L_2 on O and tries to get lock L_1 on O but is blocked by A_1.

4. A_1's priority is raised to that of A_2.

5. A_1 resumes execution.

6. A_3 preempts A_1.

7. A_3 tries to get lock L_2 on O but is blocked by A_2.

8. A_2's priority is raised to that of A_3.

9. A_1 resumes execution at A_2's priority, not A_3's.

In this example, A_3 is indirectly blocked by A_1, but A_1's priority is not raised to that of A_3. This is transitive blocking. To handle this situation, whenever an activity's priority is raised due to priority inheritance, all activities that are currently blocking it must also be raised. This is referred to as *transitive priority inheritance*. In this example, when A_2's priority is raised to that of A_3, A_1's priority must also be raised to that of A_3.

Transitive blocking can also occur in the following way.

1. A_1 gets a lock L on object O.

2. A_1 starts activity A_2, and A_1 suspends until A_2 completes.

3. A_3 preempts A_2.

4. A_3 tries to get lock L but is blocked by A_1.

5. A_1's priority is raised to that of A_3.

6. A_2 resumes execution at its original priority, not A_3's.

In this example, A_3 is indirectly blocked by A_2, but A_2's priority is not raised to that of A_3. Both A_1 and A_2 should undergo priority inheritance since A_1 cannot release the lock until A_2 finishes.

2.5 Deadlock

Basic priority inheritance does not intrinsically prevent deadlock. Therefore, some additional mechanism is needed. Deadlock can occur if and only if all of the following four conditions are present in the system:

1. Mutual exclusion.

2. Hold and wait (situation in which an activity holds a lock while waiting to obtain another lock).

3. No preemption of locks (i.e., only the activity that holds a lock can release it).

4. Circular wait (e.g., activity A_1 holds a lock activity A_2 needs to continue execution, but A_2 holds a lock which A_1 needs to continue execution).

To prevent deadlock, one of these four conditions must be eliminated. Mutual exclusion and no preemption are needed to maintain consistent resources and therefore cannot be eliminated. Hold and wait can be eliminated by forcing an activity to obtain all of the locks it needs before it begins execution. However, this reduces the amount of concurrency in the system and can lead to starvation since an activity may wait indefinitely for a lock before it can start execution. Eliminating circular wait is the last option. This can be done by forcing an ordering of locks. This requires that all activities obtain locks in the same order. The issue then becomes how to order the locks such that the amount of concurrent access is maximized. This issue is not addressed by this project.

2.6 Priority Ceiling Protocols

Priority ceiling protocols [SRL90] differ from the basic priority inheritance protocol in that they bound priority inversion to at most one critical section locked by a lower priority activity. This eliminates the problem of chained blocking. These protocols also implicitly prevent deadlock and transitive blocking. Therefore, they perform better than basic priority inheritance which cannot provide such a low bound, nor can it implicitly prevent deadlock or transitive blocking. However, as will be seen, there is a loss of flexibility when using priority ceiling protocols. Below are the three basic steps needed by the priority ceiling protocol:

1. The protocol first determines the *priority ceiling* for each lock. The priority ceiling is the priority of the highest priority activity that will access the lock. This must be done off-line.

2. When an activity requests a lock, it is granted the lock only if its priority is higher than the priority ceilings of all locks currently held by other activities.

3. If an activity A_2 cannot obtain a lock because its priority is lower than or equal to the priority ceiling of a lock held by activity A_1 (a lower priority activity), then A_1's priority is raised to that of A_2 until A_1 releases the lock blocking A_2.

Priority ceiling protocols are less dynamic than the basic priority inheritance protocol since they must determine the priority ceilings for each lock *before* the locks can be used. Since these calculations rely on knowing the priorities of all activities that may request the locks, dynamic workloads are not possible.

2.6.1 Distributed Priority Ceiling Protocol

Work presented in [Raj91] demonstrates how a priority ceiling protocol can be used to bound priority inversion in a distributed environment. This method is called the *Distributed Priority Ceiling Protocol* (DPCP).

As previously mentioned, priority inversion occurs when a real-time activity is blocked by a lower-priority real-time activity. In a distributed environment, there is the additional concept of *remote blocking*. In this context, *remote* will be used to refer to an activity or resource that is on a processor other than the local one. Remote blocking occurs when a real-time activity that is trying to access a remote shared resource is blocked by a remote real-time activity, regardless of the remote activity's priority. This includes blocking caused by activities with equal or higher priorities. The justification for being concerned with the blocking time caused by a remote activity of equal or higher priority is that such blocking would not occur if there were no shared resources (i.e., remote blocking does not occur when there are no shared resources).

The DPCP assumes that binary semaphores are used for synchronization. A semaphore that is accessed by remote activities is called a *global semaphore*, and the critical section it protects is called a *global critical section*. There are also *local semaphores* (semaphores that are only accessed by activities on the same processor) and *local critical sections*. The DPCP assumes that global critical sections do not make nested accesses to local semaphores or to other global semaphores on other processors. Local critical sections likewise do not make nested accesses to global

semaphores. The DPCP also introduces the idea of *synchronization processors*, which are processors that execute global critical sections. This protocol requires that all global critical sections guarded by the same global semaphore are bound to the same synchronization processor.

In the DPCP, the priority ceiling of a local semaphore is the priority of the highest-priority activity that can access it. The priority ceiling of a global semaphore is found by summing the priority of the highest-priority activity that can access it with the *base priority ceiling*, a priority which is higher than the priority of the highest-priority activity in the entire distributed system. This forces the priority ceilings of all global semaphores to be higher than the priority of the highest-priority activity in the distributed system while maintaining the same relative priority ordering between all global semaphores. This means that global critical sections are executed at a higher priority than all tasks outside critical sections. This is required in order for the remote blocking time of an activity in a global critical section to be a function of critical sections only.

2.6.2 Comments About Priority Ceiling Protocols

Although priority ceiling protocols do provide bounds on priority inversion, prevent deadlock and transitive blocking, they are not equipped to handle dynamic workloads. They require *a priori* information about the priorities of the activities that will run in order to compute priority ceilings. The basic priority inheritance protocol can not bound priority inversion as low as priority ceiling protocols, but it does not require the

a priori information about activity priorities. This is important for the RTCORBA project which needs to support dynamic workloads.

2.7 Programmable Concurrency Control Service

One related effort to extend CORBA's CCS for use in real-time systems is the *Programmable Concurrency Control Service* (PCCS) presented in [BG96]. This work attempts improve upon the real-time performance of the standard CCS by increasing the amount of concurrent access that can take place. This is done by replacing the object-level read/write locking that is specified in CORBA with user-specified method-level locking. The compatibility semantics of these locks can be defined by the programmer, allowing the programmer to provide a finer locking granularity than read/write locking. This functionality is implemented in the form of a *LockTable* object. Methods on the *LockTable's* interface allow the programmer to explicitly specify the compatibilities between locks. In this way, the generic semantics of the CCS *LockSet* can be replaced with semantics that allow for more concurrent access.

For example, suppose a shared resource has two attributes, *speed* and *heading*. In addition, suppose it has two methods on its interface, *write_speed* and *write_heading*, for writing to these attributes. Let A_1 be an activity that wishes to call the *write_speed* method and A_2 be an activity that wishes to call the *write_heading* method. Since the two methods do not access the same attribute, there would not be any conflict if they ran concurrently. However, if the standard CORBA CCS is used, these method invocations of A_1 and A_2 cannot execute concurrently since only one can hold a write

lock on the resource at a time.

The benefits of using the PCCS are effectively the same as using one CCS $LockSet$ for each attribute in the resource and one for the entire resource. The $intention\_write$ and $intention\_read$ locks ensure that reducing the locking granularity to the attribute level will not lead to inconsistencies in the resource as a whole. However, the PCCS does have the advantage that only one $LockSet$ is needed per resource. In terms of its real-time performance, the PCCS is not adequate. In its current form, the PCCS does not address the issues of unbounded priority inversion or deadlock. It is real-time in the sense of being faster than the standard CCS, not more predictable.

Chapter 3

Design of the RTCCS

This chapter presents the design issues of the RTCCS. The first section briefly addresses the desirability of priority inheritance in CORBA/RT as expressed by the CORBA RT SIG. The second section outlines the design decisions made by the RTCORBA group during the development of the RTCCS.

3.1 Priority Inheritance in CORBA/RT

Although the CCS provides a good starting framework for concurrency control in DOCEs, it is not sufficient for real-time systems. For example, since read/write locking is used to maintain the consistency of a resource, clients of the resource may be suspended if their requests conflict with other clients currently using the resource. This situation can lead to unbounded priority inversion. In order to be useful for real-time applications, the CCS must be extended such that it can bound priority inversion.

Previous approaches to extending the real-time capabilities of the CCS have not taken this problem into account. For example, the implementation presented in [BG96] increases the amount of concurrent access to a resource by replacing the standard object-level locking with method-level locking. However, it does not address the issue of priority inversion. [RTSIG96] specifies that priority inheritance should be used for any resource access which can lead to unbounded priority inversion.

3.2 Real-Time Concurrency Control Service

The solution to the priority inversion problem for the RTCORBA project was to extend the standard CCS by implementing basic priority inheritance within the *LockSet* object. When a client requests a lock on a resource, its priority is compared to those of all clients holding conflicting locks on that resource. For all clients that hold conflicting locks and have lower priorities than that of the requesting client, the RTCCS raises their priorities (and the priorities of any clients that may be blocking them) to the requesting client's priority. The requesting client is then suspended.

Whenever a lock is released, the releasing client resets its priority to that of the highest priority client it still blocks (this is possible since clients can hold several types of locks simultaneously). If it no longer blocks any higher priority clients, the releasing client is reset to its original priority. Finally, the highest priority blocked client that can now get its lock is allowed to obtain the lock and continue execution.

3.2.1 Implicit and Explicit Locking

When using the CCS, two forms of locking are possible: *implicit locking* and *explicit locking*. Implicit locking is done within the methods on the resource's interface. For example, any method that writes to the resource's data must obtain a write lock from the appropriate *LockSet* object first. This requires that the calls to the *LockSet* methods be made in the implementation of the resource's methods. This simplifies the usage of the resource since client objects do not need to know the locking semantics of the resource. However, there is a loss of flexibility when using purely implicit locking. For example, if a client wishes for a block of method calls to be protected by the same lock, implicit locking is insufficient. Explicit locking provides more flexibility since it allows the client that is using the resource to request and release locks when needed. This is done by explicitly obtaining the necessary locks from the appropriate *LockSet* object. However, this requires that the client knows which *LockSet* to use. More importantly, the client must have knowledge of the locking semantics for the resource being accessed (i.e., the client must know which locks are required for each method on the resource's interface). Aside from the burden this places on the client, breaking the encapsulation of the resource is not desirable from an object-oriented design perspective.

3.2.2 Transitive Priority Inheritance

Another issue that must be addressed is that of transitive blocking (see Chapter 2). The two forms of transitive blocking involve a high priority activity A_3 that is

indirectly blocked by a lower priority activity A_2 that is either:

1. holding a lock that is blocking activity A_1, the activity that is directly blocking activity A_3

2. running under a lock held by A_1.

In either case, a *transitive blocking chain* is formed in which an activity (e.g., A_3) is indirectly blocked by another activity further down the chain (e.g., A_2). Note that this is not the same as chained blocking in which an activity is blocked by multiple activities. The difficulty with transitive priority inheritance is the fact that these blocking chains can become arbitrarily long, especially when activities are allowed to lock multiple resources. This can require a great deal of overhead in terms of data structures and CPU time. Therefore, this implementation of the RTCCS was designed with the following limitations:

1. No "child" activities can be created under a lock.

2. An activity can only hold locks on one resource at a time.

The first restriction disallows explicit locking in the sense that only code local to the activity that holds the lock can run while the lock is held. The second restriction is a special case of the first restriction since obtaining additional locks after the initial one would constitute starting "child" activities under the initial lock. The only transitive blocking that is allowed in this project is that which occurs within a $LockSet$. That is, blocking chains are allowed to form as long as all of the clients in the chain are

clients of the same *LockSet* object and do not start any "child" activities while they hold locks.

3.2.3 Design Details

Before discussing the design of the RTCCS, a brief description of the RTCORBA project as a whole would be beneficial to the reader. The URI RTCORBA project is researching the following four topics as they relate to CORBA:

- Real-time method invocations
- Real-time events
- Global priority assignment
- Priority inheritance

Real-Time Method Invocations

In a DOCE, all server executions are initiated by method invocations made by clients. In a real-time application, a client must be able to specify timing constraints on method invocations. The CORBA/RT white paper specifies that timing information should be made available to the ORB, object services, and server implementations. There are five forms of client-side timing constraints that must be expressible: deadlines, earliest start times, latest start times, periodic, and quality of service (QoS) constraints.

In order to provide this capability, the URI RTCORBA group has defined a new structure in IDL called *RT_Environment*. This data structure is used to pass a vari-

ety of information from clients to servers during *timed distributed method invocations* (TDMIs). This includes time constraints, importance information, and an identification tag specifying the identity of the client (includes thread ID, process ID, and IP address). The RTCORBA work presented in [DGSWWZ97] and [Zykh97] provides additional functionality for TDMIs. This includes the *RT_Manager_Server* and *RT_Manager_Client* classes which provide the framework upon which the TDMIs are structured. These classes are integral to the design and implementation of the RTCCS.

Real-Time Events

Events may occur in a DOCE (e.g., radar contact made with an airplane), and a distinct set of clients in the DOCE may be interested in the event. In a real-time environment, these clients may need to know the absolute time that an event occurred so that time-constrained responses can be made (e.g., within 1 second of detecting airplane, update controller's display). To this end, [RTSIG96] specifies that the real-time CORBA Event Service must provide the ability for CORBA clients and servers to determine the absolute time when an event has occurred.

Global Priority Assignment

One way to provide distributed real-time scheduling is through the enforcement of *global priority*. Global priority can be represented as an ordinal quantity that is attached to every method invocation and is interpreted in a homogeneous fashion by the schedulers and queues throughout the CORBA system. That is, if method

invocation A has a higher global priority value than method invocation B, method invocation A should always be serviced first.

Enforcement of global priority requires the use of real-time schedulers and priority-based queues throughout the distributed system. A real-time scheduler typically strives to execute the highest priority task first and a priority queue typically places the highest task at its head. If these conditions are violated anywhere in the path of a real-time method invocation, unbounded priority inversion may occur and no guarantees can be made about the real-time behavior of any of the components involved. [RTSIG96] calls for a Global Priority Service that is available to establish priorities for all executions in the entire distributed CORBA system.

The RTCORBA project makes use of a *Pserver* (priority server) running on each node in the RTCORBA system. All real-time processes and threads, including those required by the RTCCS, register with the local Pserver, are assigned priorities based on their timing constraints, and are "aged" as new real-time activities enter the system [DGSWWZ97]. "Aging" the priorities of the activities is needed in order to maintain the correct relative ordering of the activities.

The RTCCS depends on the services of the Pserver to obtain priority information about activities and to request that the priorities of activities be changed. This functionality is needed in order to implement priority inheritance. The manner in which this is done will be explored in Chapter 4.

```
module CosConcurrencyControl {

  enum lock_mode {
       read, write, upgrade, intention_read, intention_write
  };

  exception LockNotHeld{};

  interface LockSet {
     void lock(in lock_mode mode);
     boolean try_lock(in lock_mode mode);
     void unlock(in lock_mode mode);
           raises(LockNotHeld);
     void change_mode(in lock_mode held_mode, in lock_mode new_mode);
           raises(LockNotHeld);
  };

};
```

Figure 3.1: Subset of CORBA Concurrency Control Service (CCS) IDL

Design of the RTCCS

The design phase for the RTCCS entailed specifying how the CCS was to be implemented and extended to support priority inheritance. Figure 3.1 is the subset of the CCS (shown here in its IDL format as found in [OMG96]) that was implemented and extended.

One of the goals during the development of the RTCCS was to ensure that the interface to the CCS was changed as little as possible. This was to ensure that the new RTCCS could be easily incorporated into existing applications that use the standard CCS. The only change to the standard interface is that a *RT_Environment* is passed into each TDMI. This parameter contains information about the time constraints of

```
#include "rt_info.idl"
module CosConcurrencyControl {

  enum lock_mode {
       read, write, upgrade, intention_read, intention_write
  };

  exception LockNotHeld{};

  interface LockSet {
     void    lock(in lock_mode mode, in RT_Environment rt_env);
               raises(RT_Exception);
     boolean try_lock(in lock_mode mode, in RT_Environment rt_env);
               raises(RT_Exception);
     void    unlock(in lock_mode mode, in RT_Environment rt_env);
               raises(LockNotHeld, RT_Exception);
     void    change_mode(in lock_mode held_mode, in lock_mode new_mode,
                         in RT_Environment rt_env);
               raises(LockNotHeld, RT_Exception);
  };

};
```

Figure 3.2: Real-Time Concurrency Control Service (RTCCS) IDL

the locking client. As will be seen in Chapter 4, this information is needed by the *LockSet* to determine when priority inheritance is needed. In addition, each method can raise a *RT_Exception* exception. This exception is used to indicate that a timing constraint has been violated during the TDMI. The revised IDL for the RTCCS is shown in Figure 3.2.

The design of the RTCCS makes use of several simplifying restrictions:

1. Only implicit locking is allowed.

2. A client can only obtain locks on one *LockSet* at a time.

3. A client cannot start "child" activities while the client holds a lock.

4. Locks must be ordered.

The first restriction requires that only the methods on a resource's interface be allowed to request locks from the resource's $LockSet$. The next two restrictions prevent all transitive blocking except that which arises between clients of the same $LockSet$. Finally, the last restriction supports the prevention of deadlock.

3.3 Summary of RTCCS Design

Although the DPCP bounds priority inversion, prevents transitive blocking, and prevents deadlock, this protocol requires foreknowledge about the activities that will be running on the system. In dynamic environments in which this information is not available until runtime, priority ceiling protocols are inappropriate. The PCCS attempts to improve real-time performance by using method-level locking rather than object-level locking. This may increase the amount of concurrent access, but it does not address the problems of priority inversion and deadlock.

The RTCCS for this project makes use of basic priority inheritance to bound priority inversion that arises during TDMIs in the RTCORBA environment. Support for priority inheritance relies on the global priority assignment and enforcement mechanism that have been developed for the RTCORBA project. Timing information passed to the RTCCS by the TDMIs enables the RTCCS to determine when priority inheritance needs to be done.

Basic priority inheritance does not prevent deadlock or transitive blocking. Deadlock is prevented by ordering locks. In order to manage transitive blocking, some form of transitive priority inheritance is needed unless certain restrictions are made. This is a desired capability in [RTSIG96]. However, this implementation of the RTCCS does not provide full support for transitive priority inheritance due to the complex nature of the problem. Instead, it implements priority inheritance among activities accessing the same shared resource.

Chapter 4

Implementation of the RTCCS

This chapter presents the implementation of the RTCCS that has been incorporated into the RTCORBA project. The first section describes the implementation of the RTCCS and the underlying configuration of the development environment. The second section presents an example of how the RTCCS is used.

4.1 RTCCS Implementation

Implementing the RTCCS consisted of first coding the subset of the CORBA CCS as specified in Chapter 3. This involved defining the IDL interface for the RTCCS (see Figure 3.2) and coding the implementation of its methods in C++. Support for priority inheritance was then added. The remainder of this section addresses how this support was integrated into the CCS.

4.1.1 *LockSet* Data Structures

Each *LockSet* object maintains information about the clients interacting with it. When a request for a lock is received, the *LockSet* uses this information to determine if the lock can be granted (i.e., there is no conflict with granted locks) or, in the case where there is a conflict, whether or not priority inheritance is needed.

The implementation of the RTCCS *LockSet* object maintains a linked list of clients (*clientList*) that are either requesting locks or are holding locks. For each client, the list maintains the following information:

- The client's original *RT_Environment*.
- The *RT_Environment* of the active *LockSet* thread, if any, that is associated with the client.
- *lockList*, a list of locks currently held by or requested by the client. Each item in this list includes the lock type, if it is granted or still only requested, and a count of the number of locks of that type granted to the client.
- *blockedClients*, a list of clients currently blocked by this client. Each item in this list includes a pointer to the blocked client's node in the *LockSet*'s *clientList*, the *RT_Environment* of the blocked client, and the type of lock this client is blocking with.
- *blockingClients*, a list of clients that are currently blocking this client. Each item in this list include a pointer to the blocking client's node in the *LockSet*'s

clientList, the *RT_Environment* of the blocked client, and the type of lock this client is blocked by.

Note that client C_1 *blocks* client C_2 if and only if C_1 holds a lock that conflicts with C_2's request and C_1's priority is less than that of C_2. In a similar manner, C_2 is *blocked* by C_1 if and only if C_1 holds a lock that conflicts with C_2's request and C_1's priority is less than that of C_2.

Each client in the list has a unique thread ID, process ID, and IP address. This information is recorded in the *RT_Environment* that is stored with each entry in *clientList*. Therefore, the *RT_Environment* can be used to search *clientList* for a particular client. In addition, it is used by the RTCCS to obtain the priorities of activities from the local Pserver to test for priority inversion during lock requests. The *RT_Environment* for the active *LockSet* thread is needed in order to changed the priority of the thread if priority inheritance is required. For example, if a client that is suspended in a *lock* thread blocks a higher priority client, the *lock* thread must also undergo priority inheritance.

The information in the *lockList* is used to detect locking conflicts whenever any client requests a lock. The counts maintained for each lock type are used to determine when a client is no longer interacting with the *LockSet* (allowing the *LockSet* to remove the client from its *clientList*).

The information stored in *blockedClients* is needed in order to restore the priority of a client that had undergone priority inheritance when it releases a lock. When a client releases a lock, it must reset its priority to that of the highest priority client

it still blocks. If *blockedClients* is empty, the client's priority is reset to its original value. The pointer that is maintained in each node is used to identify and remove the corresponding *blockingClients* entry in a blocked client whenever the blocking client releases a lock.

The list *blockingClients* is needed for the situation in which transitive blocking occurs. Recall that transitive blocking is allowed to take place as long as it is contained among the clients interacting with the same *LockSet*. If a client C_1 holds a lock that blocks a higher priority client C_2, but C_1 is blocked by other clients (those in its *blockingClients*), the priorities of C_1 and the clients in its *blockingList* have to be raised to that of C_2. This is a recursive process in the sense that the clients blocking the clients in C_1's *blockingClients* list are also raised. This continues until all blocking clients, whether they directly or indirectly block C_2, are raised to C_2's priority.

The pointer maintained in each node of *blockingClients* is needed in case a blocked activity violates a timing constraint while it is blocked. If this occurs, the activity releases any locks it may hold. The pointers in *blockingClients* can be used to identify and remove the corresponding *blockedClients* entries in the blocking clients.

4.1.2 Synchronization Constructs

Each *LockSet* object uses a mutex to control access to *clientList*. In addition, each *LockSet* uses a condition variable to synchronize clients as they request locks. This will be illustrated in the next subsection.

4.1.3 Implementation of *LockSet* Methods

The RTCORBA project makes use of multithreaded servers to allow the servers to manage multiple client requests at once. A typical server process contains one shared resource (e.g., a tracking database) and a *LockSet* object that provides concurrency control for the resource. Another scenario has the *LockSet* and resource located in separate server processes. The methods on a resource's interface are responsible for obtaining whichever locks they require to maintain the resource in a consistent state. If a client invokes a method on the resource's interface (e.g., *get_speed_of_plane*), the implementation of the resource's method is responsible for invoking the appropriate method on the *LockSet* interface (e.g., *get_speed_of_plane* gets a read lock). Therefore, while the resource has clients of its own and acts as a "server", the resource acts as a "client" of a *LockSet* object.

The *lock* Method

When a resource method requires a lock, the thread that is executing the resource method request invokes the *lock* method on the *LockSet* that is managing access to the resource's shared data. The *lock* method performs the following steps:

1. Register the *lock* thread with the local Pserver (sets priority) and start timer (start of real-time invocation with deadline specified in *RT_Environment* parameter).

2. Start an atomic code block (explained below).

3. Lock the *LockSet*'s mutex.

4. If the calling client is not in *clientList*, add it to *clientList*.

5. If the calling client holds the requested lock, increment its count for that lock, else:

   (a) Add an entry to the client's *lockList* indicating that the specified lock has been requested.

   (b) While there is a conflict (make call to $lockConflict$), wait on the $LockSet$'s condition variable. Each time the thread is awakened, it must call *lockConflict* to determine if it can proceed or if it must wait on the condition variable again.

   (c) Increment the client's lock count for the requested lock and set its status to granted.

6. Unlock the $LockSet$'s mutex.

7. End the atomic block.

8. Turn off the timer and deregister the *lock* thread from the local Pserver.

If the deadline is missed during the *lock* method invocation, the *lock* thread is deregistered from the Pserver and a *RT_Exception* is thrown to the calling client. The implementation of the *try_lock* method differs from that of the *lock* in that a call to *try_lock* only tries to obtain the lock once (i.e., it does not wait on the condition variable and no priority inheritance is done). If the lock can be granted, *try_lock* returns with a value of TRUE. Otherwise, it returns FALSE.

The atomic block is used to block signals during critical sections. When a deadline is missed in the RTCORBA environment, a signal is raised by the timer in the thread that missed the deadline. If the signal arrives while the thread is in a critical section (e.g., holding a lock on a mutex), steps must be taken to ensure that the thread leaves whatever shared data it is accessing in a consistent state. In the case of the *LockSet* object, this situation can arise during invocations of any of its four methods. Methods in the *RT_Manager_Server* class allow the thread to block the processing of signals during these critical sections [Zykh97]. The signals must also be blocked because of a problem in Orbix 2.0.1MT, the commercial CORBA product on which this project was developed. If an exception (e.g., *RT_Exception*) is thrown in a thread or process that is making a CORBA call, a run-time exception is raised and the program exits. The workaround for this requires that CORBA calls in a real-time activity be made from inside atomic blocks.

The *lock* implementation uses a Boolean function called *lockConflict*, which is passed the identity (i.e., a *RT_Environment*) of a client requesting a lock and the type of lock it is requesting. The function returns TRUE if the requested lock can be granted and FALSE if it cannot. The implementation of *lockConflict* is based on the locking semantics specified in Table 2.1. This is a modular design, allowing a different implementation of *lockConflict* to be inserted in place of the one implemented for this project. The *lockConfict* function performs the following steps:

1. For the specified lock, check if any other clients hold conflicting locks and do priority inheritance if necessary. This is done by calling *otherClientHoldsLock*

for each lock that can conflict with the requested lock.

2. Return TRUE if conflict exists. Otherwise, return FALSE.

The function *lockConflict* makes uses of a method on the interface of *clientList* called *otherClientHoldsLock*. The *otherClientHoldsLock* method performs the following steps:

1. For each client (excluding the requesting client) that holds a conflicting lock, if the client's current *or* original priority is lower than that of the requesting client, then:

   (a) Raise the priorities of the blocking client and all clients blocking it to the requesting client's priority if the blocking client's current priority is *lower* than that of the requesting client. This includes raising the priority of any *LockSet* threads in which the blocking clients may be suspended.

   (b) Add the blocking client to the requesting client's *blockingClients* list.

   (c) Add the requesting client to the blocking client's *blockedClients* list.

2. If another client holds the specified lock, return TRUE. Otherwise, return FALSE.

The requesting client's priority is compared to the blocking client's original priority in the event that the blocking client has a higher *inherited* priority but is suspended (i.e., since it is suspended, clients with lower priorities can run). By comparing the requesting client's priority with the blocking client's original priority, the necessary entries can be added to the appropriate *blockingClients* and *blockedClients* lists.

This ensures that the blocking client's priority is not reset to its original value if, when it releases the lock that forced its priority inheritance, it still holds a lock that blocks the requesting client.

If a resource method invocation that is holding locks on a *LockSet* object misses its deadline, the RTCCS must ensure that all of the locks are released. This is done by calling the *cleanup* method in the *lockSetManager* class. Each resource method that can possibly acquire locks on a *LockSet* object directly must instantiate an instance of a *lockSetManager* object. This class acts as the resource's interface to the *LockSet* (i.e., the resource does not directly invoke the *LockSet*'s methods). One of the goals of this research project was to remain as CORBA compliant as possible. In its current form, the specification for the CORBA CCS *LockSet* interface does not include a method that allows a client to release all of its locks. Therefore, the *lockSetManager* class was created to act as an intermediary between a client and the *LockSet* object. The *lockSetManager* class interface is an extended form of that specified for the *LockSet* object. In addition to the usual methods (i.e., *lock*, *unlock*, *try_lock*, *change_mode*), the *lockSetManager* has a *cleanup* method. As locks are obtained and released by a client, its *lockSetManager* object (one for each *LockSet* object it interacts with) maintains counts of the types of locks currently held by the client. The *cleanup* method simply uses this information to release all locks the client holds.

The *unlock* Method

When a resource method that obtained a lock no longer needs the lock, the thread that is executing the method invokes the *unlock* method on the *LockSet* that is managing access to the resource's shared data. The *unlock* method performs the following steps:

1. Register the *unlock* thread with the local Pserver (sets priority) and start timer (start of real-time invocation). This is done only if this call to *unlock* is not made from *lockSetManager::cleanup* (see note below).

2. Start an atomic code block (prevents processing of signals). This is done only if this call to *unlock* is not made from *lockSetManager::cleanup*.

3. Lock the *LockSet*'s mutex.

4. If the calling client is not in *clientList*, then:

   (a) Unlock the *LockSet*'s mutex.

   (b) End the atomic block and deregister the *unlock* thread. This is done only if this call to *unlock* is not made from *lockSetManager::cleanup*.

   (c) Throw a *LockNotHeld* exception to the calling client.

5. If the calling client does not hold the specified lock, then:

   (a) Unlock the *LockSet*'s mutex.

   (b) End the atomic block and deregister the *unlock* thread. This is done only if this call to *unlock* is not made from *lockSetManager::cleanup*.

   (c) Throw a *LockNotHeld* exception to the calling client.

6. Decrement the appropriate lock count.

7. If this lock count is now zero, then:

   (a) Remove the lock from the calling client's *lockList*.

   (b) Update the calling client's *blockedClients* list by removing all clients that the calling client no longer blocks.

   (c) Restore the priority of the calling client to the priority of the highest priority client still blocked by this client, or reset to original priority if no clients are blocked. Currently, this step is done even if this client's priority was never changed.

   (d) If the calling client no longer holds any locks, remove it from *clientList*.

   (e) Send a broadcast signal to the $LockSet$'s condition variable (allows waiting clients to obtain mutex in priority order and attempt to get their locks).

8. Unlock the $LockSet$'s mutex.

9. End the atomic block. This is done only if this call to *unlock* is not made from *lockSetManager::cleanup*.

10. Turn off the timer and deregister the *unlock* thread from the local Pserver. This is done only if this call to *unlock* is not made from *lockSetManager::cleanup*.

If the deadline is missed during the *unlock* method invocation, the *unlock* thread is deregistered from the Pserver and a *RT_Exception* is thrown to calling to the client. This is done only if this call to *unlock* is not made from *lockSetManager::cleanup*.

Calls to *unlock* that are made from *cleanup* are not assigned priorities based on time constraints. This is because the call to *cleanup* is made as a result of violating a constraint. Therefore, calls to *cleanup* are executed at the maximum system priority. This ensures that locks held by the client that violated a constraint are released as quickly as possible. Since the calls to *unlock* made by *cleanup* do not interact with the Pserver, there is no need to register them with the Pserver. In addition, since these calls do not use timers, there is no need for atomic blocks.

Finally, the implementation of the *change_mode* method simply tries to unlock the specified lock using the *unlock* method. It then uses the *lock* method to obtain the new lock. Any exceptions raised during these two operations are passed onto the calling client.

4.2 Operating System's Relation to RTCORBA

This project was developed and tested on two Sun SPARCstations on an isolated LAN running Solaris 2.5 and IONA Technologies Orbix 2.0.1MT, a commercial CORBA 2.0 compliant DOCE. Solaris 2.5 is a POSIX [Gal95] compliant operating system, meeting the requirements of POSIX.1, POSIX.4, and POSIX.4a [Sun95]. Although it has support for real-time scheduling of threads, it does not provide priority-based scheduling for mutexes or any form of priority inheritance for mutex queues. This introduces a source of unbounded priority inversion in the case of the $LockSet$ methods since they lock a mutex. One workaround to this problem is to raise the priority of any thread that tries to lock the mutex to the system maximum. This eliminates the

priority inversion caused by the mutex, but it bypasses the sense of consistent global priority that RTCORBA tries to enforce (e.g., a low priority client that requests a lock should not be scheduled at a high priority unless priority inversion is first detected). Currently, this version of the RTCCS does not make use of this workaround.

The condition variable used by the *LockSet* provides better support for priority-based scheduling. When the the condition variable receives a broadcast signal, all threads blocked on the condition are awakened. They are then scheduled according to the current scheduling policy as they attempt to lock the mutex. In this project, this means that the highest priority thread will get the mutex first. The remaining threads will likewise obtain the mutex in descending order of priority. The order in which threads within the same priority level obtain the mutex is undetermined.

4.3 The RTCCS Mechanism by Example

This section presents an example of how the *LockSet* is incorporated into a resource. Below is the IDL specification for a tracking database object.

```
#include "rt_info.idl"
interface track_db {
     void set(in long track_id, in track_record t,
              in RT_Environment rt_env);
     track_record get(in long track_id, in RT_Environment rt_env);
};
```

This object stores tracking information (e.g., speed and heading) for objects (e.g., airplanes). It has two methods, *set* and *get*, that allow clients to access the data it stores. The method *set* is passed a track ID, a new tracking record to insert, and a *RT_Environment* parameter which contains the timing constraints for the method invocation. The *get* method is passed the track ID of the record to retrieve and a *RT_Environment* parameter, and it returns the appropriate tracking record.

The C++ implementation for the *set* method is given below. It illustrates how the calls to *lock*, *unlock*, and *cleanup* are incorporated into the method's implementation.

```
void track_db_i::set(CORBA::Long track_id, track_record t,
                     const RT_Environment& rt_env,
                     CORBA::Environment &) {
(1)  RT_Manager_Server rt_mgr(rt_env);
     RT_Environment local_rt_env = rt_mgr.Get_RT_Env();
     try {
(2)     rt_mgr.START_RT();
(3)     rt_mgr.Start_Atomic_CORBA_Call();
(4)       lock_set.lock(CosConcurrencyControl::write, local_rt_env);
(5)     rt_mgr.End_Atomic_CORBA_Call();
(6)       // CODE FOR SETTING RECORD
        rt_mgr.Start_Atomic_CORBA_Call();
(7)       lock_set.unlock(CosConcurrencyControl::write, local_rt_env);
        rt_mgr.End_Atomic_CORBA_Call();
```

```
(8)     rt_mgr.END_RT();

     } catch (const RT_Exception &rtEx) {}
(9)         rt_mgr.STOP();
(10)        lock_set.cleanup(local_rt_env);

     } catch(const CosConcurrencyControl::LockNotHeld) {
            rt_mgr.STOP();
            lock_set.cleanup(local_rt_env);

     } catch (CORBA::SystemException &sysEx) {}
            exit(1);

     } catch(...) {
            exit(1);
     }
}
```

At step 1, the *RT_Manager_Server* for this thread is initialized based on the time constraints of the calling client [Zykh97]. At step 2, the real-time method invocation starts (i.e., the thread is registered with the local Pserver and the timer for the invocation is started). The atomic block that prevents signals from being delivered during the *lock* call is started at step 3 and ends at step 5. Note that *lock_set* (at

step 4) is an instance of *lockSetManager*. It provides this thread with an interface to the appropriate *LockSet* (this information is initialized in the constructor for the *track_db_i* object). Step 6 represents the code that sets the specified record in the database. The call to *unlock* at step 7 is protected by an atomic block. Finally, at step 8, the timer for the thread is stopped and the method invocation ends. If a *RT_Exception* or *LockNotHeld* exception is raised during the execution of the method, the thread is deregistered from the local Pserver (step 9) and the *cleanup* method is called to release all locks held by this thread (step 10).

Chapter 5

Evaluation

This chapter presents a description of the testing procedure for this report. The testbed that was used is described in Section 1. A review of the the tests which were executed is given in Section 2. The results of the testing are listed in Section 3. Finally, an analysis of the results is presented in Section 4.

5.1 Testbed Construction

All testing for this report was conducted using two Sun Sparc workstations on an isolated LAN. One workstation (a SPARCstation 5) acted as the server node and ran three CORBA servers. The first server managed access to a shared resource that implemented a two-dimensional grid of integers. The interface for this server consisted of a method for writing to and reading from the grid. The second server managed the *LockSet* object that the grid server used for concurrency control. The third server was the node's local Pserver. The second workstation (a SPARCstation

IPX) was a client node which managed the client activities that interacted with the grid server. The client node also ran its own local Pserver.

5.1.1 Tests for Correctness

The first set of tests were designed to demonstrate the correctness of the implementation of priority inheritance in the RTCCS. These tests involved two or three clients running concurrently. Simple priority inheritance, transitive priority inheritance, and the performance of the *cleanup* method were tested. The test data for these tests was collected using an array of structures defined in the *LockSet*. Access to this array was protected by the *LockSet*'s mutex. Each structure in the array recorded the ID of the grid server thread that was invoking the *LockSet* method, the current priority of that thread, the name of the method being invoked (e.g., *lock*), and an identifier of the point in the method at which the record was made (e.g., after priority was updated during a call to *unlock*). In this way, a timeline of the changes to the client priorities and the order in which locks were obtained and released was constructed. The contents of the array were displayed to the workstation screen once each test completed.

5.1.2 Execution Overhead

A number of tests were run in order to quantify the execution time of the RTCCS methods. In particular, the overhead introduced by calls to *lock* and *unlock* were measured. This involved running tests with the priority inheritance enabled and

disabled. Testing was done with only one client (i.e., no lock contention) and with two clients (i.e., lock contention). The test data for these tests was collected in a manner similar to that used for the correctness tests. An array in the *LockSet* recorded the ID's of the invoking grid server thread, the name of the invoked method, and the start and end times of the specific segments of the method. This set of tests required that the priority inheritance mechanism be disabled for certain tests. This was done by commenting out the relevant code segments. This included the following:

- The code in *otherClientHoldsLock* that compares/changes priorities.
- The code in *otherClientHoldsLock* that updates the *blockingClients* and *blockedClients* lists.
- The code in *unlock* that updates a client's *blockedClients* list when a lock is released.
- The code in *unlock* that updates a client's priority after it releases a lock.

5.2 Test Details

5.2.1 Tests for Correctness

Several tests were done to verify that the priority inheritance mechanism functions as expected. These tests included the following:

1. Test for priority inheritance between two clients.
2. Test for priority inheritance between three clients.

3. Test for transitive priority inheritance between three clients.

4. Test that priority inheritance takes place when locking clients are suspended (e.g., waiting on a condition variable).

5. Test that locks are released when a time constraint is violated.

This testing took place in the RTCORBA environment with priority inheritance enabled.

Test 1 involved running two clients such that the low priority client blocked a high priority client. This was done by forcing the low priority client to obtain and hold a *write* lock on the grid server. The high priority client then requested a *write* lock.

Test 2 involved running three clients (with three different priorities) which requested *write* locks on the grid server. This was done such that the following sequence of steps took place:

1. The low priority client requested and obtained the lock.

2. The medium priority client requested lock but was blocked by the low priority client.

3. The high priority client requested lock but was blocked by the low priority client.

The Test 3 required three locks with the following characteristics:

- Lock L_1 that does not conflict with any other locks.

- Lock L_2 that conflicts with itself.

- Lock L_3 that conflicts with itself.

This was implemented by temporarily changing the locking semantics in the RTCCS. Once these semantics were implemented, the test was run with the following sequence of steps:

1. The low priority client requested and obtained both L_1 and L_2.

2. The medium priority client requested L_3 and L_2. It obtained L_3 but was blocked by the low priority client when it requested L_2.

3. The high priority client requested L_3 but was blocked by the medium priority client.

Test 4 involved the same set of locks specified for Test 3. This test involved the following sequence of steps:

1. The low priority client requested and obtained L_2 and L_3 and suspended itself on a condition variable (defined within the grid server).

2. The high priority client requested L_3 but was blocked by the low priority client.

3. The medium priority client requested L_2 but was blocked by the low priority client.

Test 5 was the same as Test 2 except the low priority client was forced to miss its deadline while it held the lock.

5.2.2 Execution Overhead

The first set of overhead tests involved running one client that invoked the method on the grid server's interface that writes to the grid. This method's implementation obtains a *write* lock on the corresponding *LockSet* object. The following average execution times were measured:

- Time for a *lock* operation with priority inheritance.
- Time for a *lock* operation with no priority inheritance.
- Time for an *unlock* operation with priority inheritance.
- Time for an *unlock* operation with no priority inheritance.

Some additional time measurements for specific code segments in these methods were measured (see results section for details). Note that these times were calculated within the *LockSet*'s methods (e.g., difference between start time of *lock* method and its end time).

The second set of tests involved running two clients concurrently. The lower priority client was allowed to obtain a *write* lock. While it held the lock, the higher priority client requested the *write* lock and was blocked. The low priority client then released the lock, allowing the high priority client to obtain it. The following average execution times were measured for both clients:

- Time for a *lock* operation with priority inheritance.
- Time for a *lock* operation with no priority inheritance.

- Time for an *unlock* operation with priority inheritance.
- Time for an *unlock* operation with no priority inheritance.

In the case of the high priority client, the time measurements taken in the *lock* method did not include the time spent waiting on the $LockSet$'s condition variable. Some additional time measurements for specific code segments in these methods were measured (see results section for details).

5.3 Results

5.3.1 Correctness Tests

All five tests in this category were successful. Test 1 demonstrated that the low priority client inherited the high client's priority while it held the *write* lock. The low priority client was reset to its original priority when it released the lock.

Test 2 demonstrated that the low priority client's priority was first raised to medium (when it blocked the medium priority client) and then to high (when it blocked the high priority client). When the low priority client released the lock, its priority was reset to its original value. The high priority client was then able to obtain the lock. Once it released it, the medium priority client obtained the lock.

Test 3 demonstrated that the low priority client's priority was raised to medium when it blocked the medium priority client with L_2. The medium priority client was then raised to high when it blocked the high priority client with L_3. Transitive priority inheritance then took place, and the low priority client, which was blocking

the medium priority client, was raised to high. When the low priority client released L_2, it was reset to its original priority value. When the medium priority client released L_3, it was reset to its original priority value.

Test 4 demonstrated that the medium priority client identified the low priority client as a blocking client even though the low priority client had a higher inherited priority. In this way, when the low priority client released L_3, its priority was reset to that of the medium priority client, not its original priority. This was necessary since the low priority client still held L_2 which was blocking the medium priority client.

Test 5 demonstrated that the low priority client correctly released its lock when the deadline was missed. This allowed the high priority client to obtain the lock. When the high priority client released the lock, the medium priority client was allowed to obtain the lock.

5.3.2 Execution Overhead

The tables at the end of this chapter contain the execution overheads measured during testing. Each table represents the results of 100 runs of each test, and contains information about the following operations whenever applicable:

- A *lock* operation.
- An *unlock* operation.
- An operation to query the Pserver for the current priority of an activity (called *query* here).

- An operation to update the priority of an activity during registration with the Pserver (called *register* here).
- An operation to deregister an activity from the Pserver (called *deregister* here).
- An operation to update the priority of an activity due to priority inheritance (called *update* here).
- An operation to update the priority of an activity and its *LockSet* thread when the activity releases a lock (called *restore* here).

For each operation, the following data is listed:

- Average execution time in milliseconds.
- Error in milliseconds (calculated in a 95% confidence interval).
- Standard deviation (σ) in milliseconds.
- Minimum execution time in 100 trials.
- Maximum execution time in 100 trials.

5.4 Analysis

5.4.1 Correctness Tests

The results from the correctness tests indicate that the RTCCS provides support for priority inheritance within the specifications set for this report. Both the simple case of priority inheritance (Tests 1 and 2) and transitive priority inheritance within a

LockSet (Test 3) are handled correctly by the RTCCS. In addition, it was demonstrated in Test 4 that the mechanism works correctly even when clients are allowed to suspend themselves while they hold locks. Finally, Test 5 shows that the *cleanup* method performs as expected.

5.4.2 Execution Overhead

Each call to *lock* and *unlock* requires one call to the *query* function, one call to the *register* function, and one call to the *deregister* function. All three of these calls require invoking a CORBA method on the local Pserver. The call to *query* is needed to obtain the latest time constraint information for the server thread that invoked the *LockSet* method. This information is then used to register the *LockSet* thread with the Pserver and set its priority. The *deregister* function is used to deregister the thread from the Pserver. These calls are required regardless of whether or not priority inheritance is enabled. When priority inheritance is enabled, the *unlock* call requires two additional calls to *update* to reset the priorities of the server and *LockSet* threads of the client that is releasing a lock. This cumulative time is represented by a call to the *restore* function. When the high priority client calls *lock*, it requires an additional two calls to *query* (to detect that the client blocking it has a lower priority) and one call to *update* in order to raise the priority of the low priority grid server thread.

Tables 5.1 through 5.8 at the end of this chapter contain the results of the overhead tests. It should be noted that the *lock* times for the low priority client in Tables 5.3

and 5.4 are longer than those in Tables 5.1 and 5.2. This is most likely due to the fact that two clients are running on the system concurrently, and the low priority thread is most likely preempted for a time while the high priority client is serviced. The large standard deviations for a number of the operations in this set of tests is most likely due to this preemption. An indication of this is that the execution times tended to group around either the minimum or maximum times, suggesting preemption occurred in some instances but not in others. Figures 5.1 and 5.2 illustrate this. Finally, Tables 5.9 and 5.10 contain the percentages of the execution times spent in CORBA calls to the Pserver for the various tests.

The results presented in the tables are intended to show that the largest overhead involved in calls to the *LockSet*, with or without priority inheritance, lie in the CORBA calls to the Pserver. The large percentages of execution time spent in these calls (70-97%) indicate that optimizing these method calls could significantly improve the performance of the mechanism. However, this is beyond the scope of this project and would have to be addressed by the vendor (IONA in this case) of the CORBA implementation (Orbix 2.0.1MT) the project is using.

Operation	Average(ms)	Error(ms)	σ(ms)	Minimum(ms)	Maximum(ms)
lock	28.01	0.23	1.07	27.65	37.56
unlock	49.30	0.14	0.66	48.80	53.34
query	6.46	0.06	0.38	6.29	10.78
register	12.60	0.04	0.24	12.22	14.57
deregister	6.25	0.01	0.09	6.14	7.14
update	11.12	0.02	0.11	11.04	11.80
restore	22.28	0.05	0.21	22.15	23.48

Table 5.1: Overheads For One Client Running in Isolation With Priority Inheritance Enabled

Operation	Average(ms)	Error(ms)	σ(ms)	Minimum(ms)	Maximum(ms)
lock	28.03	0.24	1.12	27.61	38.27
unlock	27.03	0.09	0.41	26.68	29.25
query	6.45	0.06	0.40	6.27	11.34
register	12.60	0.03	0.22	12.03	14.53
deregister	6.28	0.02	0.13	6.14	7.29

Table 5.2: Overheads For One Client Running in Isolation With Priority Inheritance Disabled

Operation	Average(ms)	Error(ms)	σ(ms)	Minimum(ms)	Maximum(ms)
lock	29.04	0.35	1.62	27.56	37.30
query	6.48	0.13	0.61	6.26	10.96
register	13.33	0.16	0.76	12.49	18.21
deregister	6.61	0.29	1.35	6.15	11.67

Table 5.3: Overheads For Low Priority *lock* Client With Priority Inheritance Enabled

Operation	Average(ms)	Error(ms)	σ(ms)	Minimum(ms)	Maximum(ms)
lock	32.52	1.57	7.31	27.55	47.35
query	7.51	0.45	2.12	6.26	11.71
register	15.09	0.72	3.35	12.46	21.85
deregister	7.24	0.41	1.93	6.13	11.56

Table 5.4: Overheads For Low Priority *lock* Client With Priority Inheritance Disabled

Operation	Average(ms)	Error(ms)	σ(ms)	Minimum(ms)	Maximum(ms)
unlock	55.88	0.17	0.78	55.20	61.19
query	6.51	0.05	0.24	6.37	8.22
register	13.98	0.04	0.17	13.82	14.64
deregister	6.23	0.03	0.13	6.11	6.84
update	13.08	0.08	0.61	12.15	14.44
restore	27.02	0.07	0.34	26.62	28.39

Table 5.5: Overheads For Low Priority *unlock* Client With Priority Inheritance Enabled

Operation	Average(ms)	Error(ms)	σ(ms)	Minimum(ms)	Maximum(ms)
unlock	38.41	0.11	0.50	38.06	42.35
query	6.50	0.04	0.21	6.39	8.18
register	13.96	0.03	0.13	13.83	14.53
deregister	6.35	0.01	0.07	6.30	6.83

Table 5.6: Overheads For Low Priority *unlock* Client With Priority Inheritance Disabled

Operation	Average(ms)	Error(ms)	σ(ms)	Minimum(ms)	Maximum(ms)
lock	54.93	0.09	0.44	54.57	58.00
unlock	49.47	0.28	1.32	48.61	61.15
query	6.25	0.03	0.25	5.93	7.48
register	13.06	0.11	0.71	12.03	14.38
deregister	6.19	0.01	0.67	6.14	6.79
update	11.21	0.02	0.15	11.09	11.97
restore	22.47	0.05	0.22	22.28	23.18

Table 5.7: Overheads For High Priority Client With Priority Inheritance Enabled

Operation	Average(ms)	Error(ms)	σ(ms)	Minimum(ms)	Maximum(ms)
lock	29.80	0.10	0.48	29.47	33.36
unlock	27.55	0.08	0.37	27.19	29.42
query	6.40	0.03	0.17	6.28	7.62
register	13.52	0.05	0.31	13.07	15.03
deregister	6.27	0.02	0.14	6.15	7.01

Table 5.8: Overheads For High Priority Client With Priority Inheritance Disabled

Operation	Percentage
Isolated *lock* call	90.4
Isolated *unlock* call	96.5
Low priority client *lock* call	91.0
Low priority client *unlock* call	96.2
High priority client *lock* call	89.6
High priority client *unlock* call	97.0

Table 5.9: Percentages of Execution Times Spent in CORBA Calls With Priority Inheritance Enabled

Operation	Percentage
Isolated *lock* call	90.4
Isolated *unlock* call	93.7
Low priority client *lock* call	91.8
Low priority client *unlock* call	69.8
High priority client *lock* call	87.9
High priority client *unlock* call	95.1

Table 5.10: Percentages of Execution Times Spent in CORBA Calls With Priority Inheritance Disabled

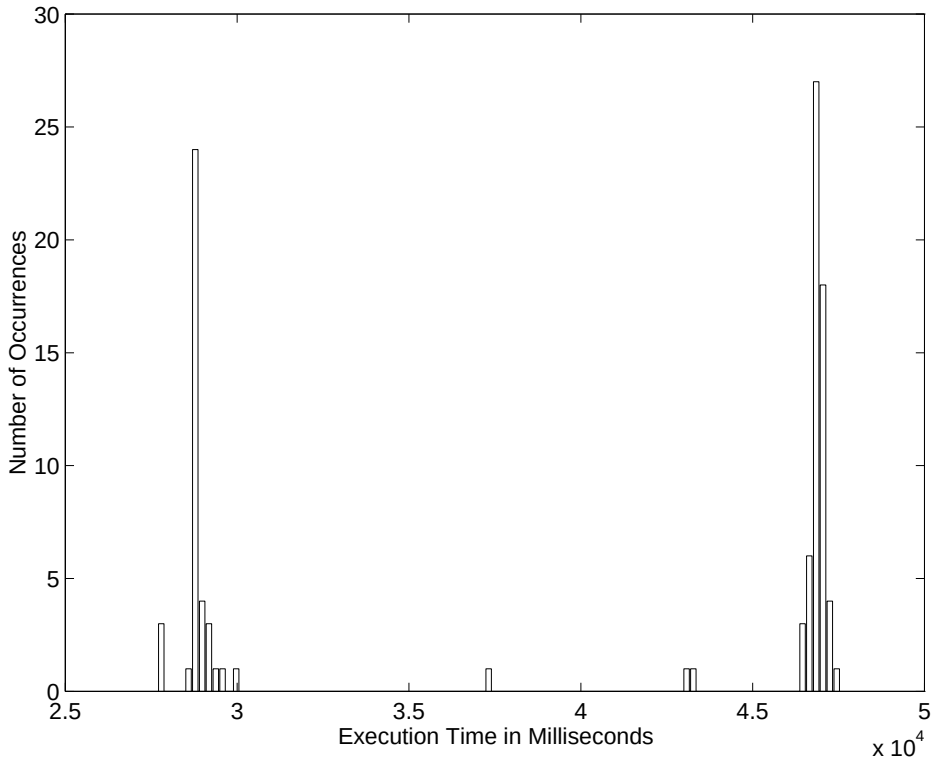

Figure 5.1: Low Priority Client *lock* Execution Times With Priority Inheritance Enabled

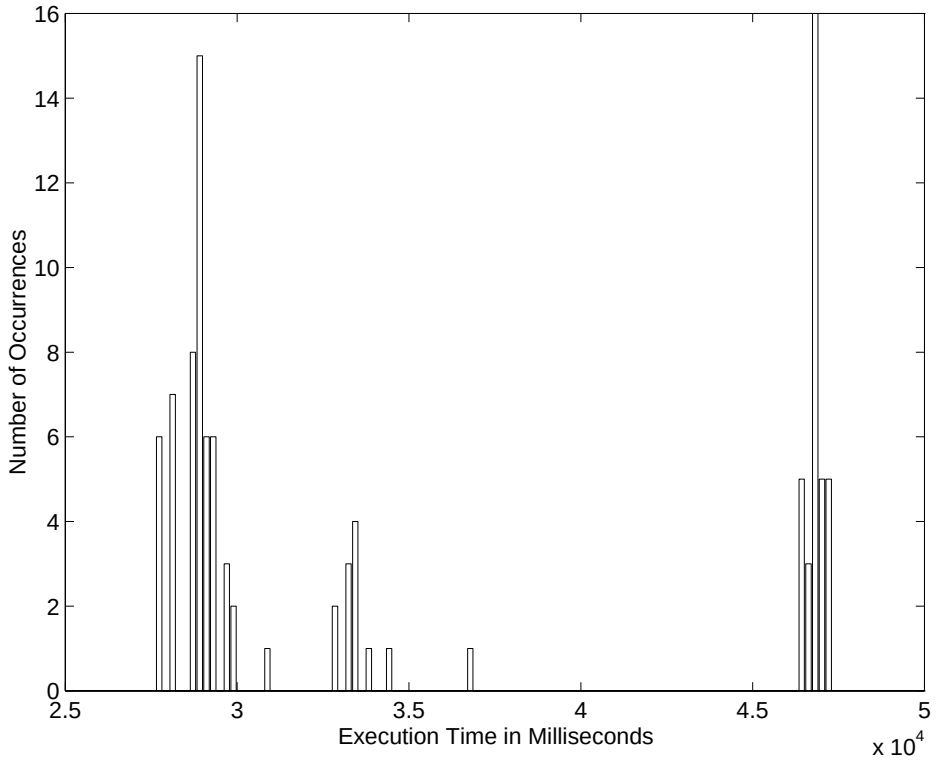

Figure 5.2: Low Priority Client *lock* Execution Times With Priority Inheritance Disabled

Chapter 6

Conclusion

6.1 Contributions

CORBA is gaining popularity in industrial, academic, and government projects. As its influence extends into specialized fields such as real-time computing, it will have to evolve to meet their particular needs. The RTCORBA project at the University of Rhode Island is a first step towards showing that it is feasible to extend the current specification of CORBA such that it can support real-time computing. The RTCCS is a crucial part of this project in that it provides a means to bound priority inversion that arises during access to shared resources. This was the main goal of this report. The tests for correctness indicate that the RTCCS does provide support for priority inheritance. The execution overheads demonstrate a gain in performance could be achieved if CORBA calls to the local Pserver were optimized. This is beyond the scope of the URI RTCORBA project. It is an issue that would have to be addressed by the vendor (IONA) of the CORBA implementation (Orbix 2.0.1MT) that the

project is using.

6.2 Comparison with Related Work

This implementation of the RTCCS draws on a large body of existing research. However, it is unique in that it brings together the concepts of priority inheritance and a implementation of a standard service for a DOCE, namely, a concurrency control service. The PCCS presented in [BG96] is an extended form of the CORBA CCS. However, it does not provide enough support for real-time environments. In a similar way, the DPCP has taken a proven idea, the priority ceiling protocol, and extended it to the realm of distributed computing. One drawback, however, is its reliance on *a priori* information about the activities which will run on the system. The RTCCS presented in this report bounds priority inversion in a distributed dynamic real-time environment, and is based on the popular CORBA interface.

6.3 Limitations and Future Work

Throughout this report, it has been noted that this implementation of the RTCCS has limitations. Most noticeable is its limited support for transitive priority inheritance. Ideally, the RTCCS should be able to allow for any form of transitive blocking and be able to handle it. This would eliminate the restriction on explicit locking and starting "child" activities under a lock. This extension may require that certain design restriction be place on the end-users of the RTCCS in order to ensure that the blocking

chains remain manageable. At the very least, the RTCCS would have to be extended such that a given *LockSet* object would be cognizant of the states of other *LockSets* and activities throughout the RTCORBA environment. In this way, a *LockSet* could apply priority inheritance to activities outside its address space. It would also be feasible to extend the RTCCS to support forms of locking other than simple read/write locking. For example, results from research in the area of *object-based semantic real-time concurrency control* [DW93], where method-level locks, whose compatibilities are semantically defined, are employed, could be integrated into the RTCCS in place of read/write locking.

Further testing of RTCORBA's performance is ongoing. The goal of this testing is to provide a measure of the prototype's overall ability to reduce the number of violated timing constraints. The RTCCS plays a part in this by bounding priority inversion. This testing involves running randomly generated clients with a variety of deadlines (characterized as short, medium, or long) and start times (allows for variable system workloads). In addition, the types of methods each client invokes (i.e., some methods require *read* locks while others require *write* locks) are varied such that a variety of data contention rates are achieved. The metric by which each test run is judged is the percentage of clients which miss deadlines. These results are then compared the results of similar tests run in a non-real-time CORBA environment (i.e., CORBA without any mechanism for enforcing time constraints).

The implementation of the RTCCS presented in this report provides a foundation for further work in the area of dynamic real-time concurrency control in DOCEs. It does have limitations in its current form. However, the solutions provided here in

addition to the unresolved issues raised during this project contribute to this area of research.

List of References

[BG96] P.G. Bosco and E. Grasso. A programmable concurrency control service for CORBA. *Third International Workshop on Services in Distributed and Networked Environments SDNE '96*, Macau, June 1996.

[DGSWWZ97] L.C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, V.F. Wolfe, I. Zykh. *Expressing and Enforcing Timing Constraints in a Real-Time CORBA System*, University of Rhode Island Technical Report TR97-252, Kingston, RI, February 1997.

[DW93] Lisa B. Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[Gal95] Bill O. Gallmeister. *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

[HGSP96] T. Harrison, A. Gokhale, D. Schmidt, and G. Parulkar. Operating system support for a high-performance, real-time CORBA. In *International Workshop on Object-Orientation in Operating Systems: IWOOOS 1996 Workshop*, Seattle, WA, October 1996.

[JWS96] Russell Johnston, Victor Fay Wolfe, and Mark Steele. Common object request broker architecture (CORBA), JTA/TAFIM compliant, distributed object technology. Naval Command, Control and Ocean Surveillance Center Research, Development, Test and Evaluation Division, San Diego, CA, Sept 1996.

[OMG96] Object Management Group. *CORBAservices: Common Object Services Specification.* Object Management Group, Inc., Framingham, MA, 1996.

[Raj91] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, Boston, MA, 1991.

[RTSIG96] The Realtime Platform Special Interest Group of the OMG. *CORBA/RT White Paper.* Object Management Group, Inc., Framingham, MA, 1996.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. In *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[Sun95] Sun Microsystems, Inc. *Solaris 2.5 Introduction*, Sun Microsystems, Inc., Mountain View, CA, 1995.

[WBTK95] Victor Fay Wolfe, John Black, Bhavani Thuraisingham, and Peter Krupp. Towards distributed real-time method invocations. In *IEEE Proceedings of the International High Performance Computing Conference*, New Delhi, India, December 1995.

[Zykh97] Igor Zykh. *Timed Distributed Method Invocations in CORBA*, University of Rhode Island Technical Report TR97-254, Kingston, RI, April 1997.

Bibliography

Bosco, P.G. and E. Grasso. A programmable concurrency control service for CORBA. *Third International Workshop on Services in Distributed and Networked Environments SDNE '96*, Macau, June 1996.

DiPippo, L.C., R. Ginis, M. Squadrito, S. Wohlever, V.F. Wolfe, I. Zykh. *Expressing and Enforcing Timing Constraints in a Real-Time CORBA System*, University of Rhode Island Technical Report TR97-252, Kingston, RI, February 1997.

DiPippo, Lisa B. Cingiser, and Victor Fay Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

Gallmeister, Bill O. *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

Harrison, T., A. Gokhale, D. Schmidt, and G. Parulkar. Operating system support for a high-performance, real-time CORBA. In *International Workshop on Object-Orientation in Operating Systems: IWOOOS 1996 Workshop*, Seattle, WA, October 1996.

IONA Technologies Ltd.,*Orbix 2 Programming Guide.* IONA Technologies Ltd., Dublin, Ireland, 1995.

IONA Technologies Ltd. *Orbix 2 Reference Guide.* IONA Technologies Ltd., Dublin, Ireland, 1995.

Johnston, Russell, Victor Fay Wolfe, and Mark Steele. Common object request broker architecture (CORBA), JTA/TAFIM compliant, distributed object technology. Naval Command, Control and Ocean Surveillance Center Research, Development, Test and Evaluation Division, San Diego, CA, Sept 1996.

Object Management Group. *The Common Object Request Broker: Architecture and*

Specification. Object Management Group, Inc., Framingham, MA, 1996.

Object Management Group. *CORBAservices: Common Object Services Specification*. Object Management Group, Inc., Framingham, MA, 1996.

Pohl, Ira. *Object-Oriented Programming Using C++*. The Benjamin/Cummings Publishing Company, Inc., New York, 1993.

Rajkumar, Ragunathan. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Boston, MA, 1991.

The Realtime Platform Special Interest Group of the OMG. *CORBA/RT White Paper*. Object Management Group, Inc., Framingham, MA, 1996.

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

Sun Microsystems, Inc. *Solaris 2.5 Introduction*, Sun Microsystems, Inc., Mountain View, CA, 1995.

Wolfe, Victor Fay, John Black, Bhavani Thuraisingham, and Peter Krupp. Towards distributed real-time method invocations. In *IEEE Proceedings of the International High Performance Computing Conference*, New Delhi, India, December 1995.

Zykh, Igor. *Timed Distributed Method Invocations in CORBA*, University of Rhode Island Technical Report TR97-254, Kingston, RI, April 1997.

Bibliography

[BG96] P.G. Bosco and E. Grasso. A programmable concurrency control service for CORBA. *Third International Workshop on Services in Distributed and Networked Environments SDNE '96*, Macau, June 1996.

[DW93] Lisa B. Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1993.

[Gal95] Bill O. Gallmeister. *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Inc, Sebastopol, CA, 1995.

[HGSP96] T. Harrison, A. Gokhale, D. Schmidt, and G. Parulkar. Operating system support for a high-performance, real-time CORBA. In *International Workshop on Object-Orientation in Operating Systems: IWOOOS 1996 Workshop*, Seattle, WA, October 1996.

[JWS96] Russell Johnston, Victor Fay Wolfe, and Mark Steele. Common object request broker architecture (CORBA), JTA/TAFIM compliant, distributed object technology. Naval Command, Control and Ocean Surveillance Center Research, Development, Test and Evaluation Division, San Di ego, CA, Sept 1996.

[OMG96] Object Management Group. *CORBAservices: Common Object Services Specification.* Object Management Group, Inc., Framingham, MA, 1996.

[Raj91] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, Boston, MA, 1991.

[RTSIG96] The Realtime Platform Special Interest Group of the OMG. CORBA/RT white paper. Object Management Group, Inc., Framingham, MA, 1996.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept 1990.

[STDW96] Michael Squadrito, Bhavani Thurasingham, Lisa Cingiser DiPippo, and Victor Fay Wolfe. Towards priority ceilings in semantic object-based concurrency control. In *1996 International Workshop on Real-Time Database Systems and Applications*, March 1996.

[WBTK95] Victor Fay Wolfe, John Black, Bhavani Thuraisingham, and Peter Krupp. Towards distributed real-time method invocations. In *IEEE Proceedings of the International High Performance Computing Conference*, New Delhi, India, Dec. 1995.