Expressing and Enforcing Timing Constraints in a Dynamic Real-Time CORBA System*

Victor Fay Wolfe, Lisa Cingiser DiPippo,
Roman Ginis, Michael Squadrito,
Steven Wohlever and Igor Zykh
Department of Computer Science
University of Rhode Island
Kingston, RI 02881
lastname@cs.uri.edu

Russell Johnston
NCCOSC
RDT&E DIV (NRaD)
53140 Systems Street
San Diego, CA 92152
russ@nosc.mil

Distributed real-time applications have presented the need to extend the Object Management Group's Common Object Request Broker Architecture (CORBA) standard to support real-time. This paper describes our Dynamic Real-Time CORBA system, which supports the expression and enforcement of end-to-end timing constraints as an extension to a commercial CORBA system. The paper also describes performance tests that demonstrate the system's ability to enforce expressed timing constraints.

1 Introduction

Distributed object computing is becoming a widely accepted programming paradigm for applications that require seamless interoperability among heterogeneous clients and servers. The Object Management Group (OMG), an organization of over 700 distributed software vendors and users, has developed the Common Object Request Broker Architecture (CORBA) as a standard software specification for such distributed environments. The CORBA specification includes an *Object Request Broker* (ORB), which is the middleware that enables the seamless interaction between distributed client objects and server objects; *Object Services*, which facilitate standard client/server interaction with capabilities such as naming, event-based synchronization, and concurrency control; and the *Interface Definition Language(IDL)*, that defines the object interfaces within the CORBA environment.

Many distributed real-time applications, such as command and control, automated manufacturing, telecommunications, and simulation, are embracing the object-oriented paradigm and have a mandate to use an open systems design. The designers of many of these applications are considering CORBA for their architecture, but are finding it is currently inadequate to support real-time requirements. CORBA contains neither the services, nor the interface facilities to express and enforce end-to-end timing constraints on distributed client/server interactions.

In 1995, a Special Interest Group (SIG) was formed within the OMG with the goal of extending the CORBA standard with support for real-time applications. This SIG (RT SIG) is developing a whitepaper [otO96] that details requirements for extending/modifying CORBA to support real-time. The whitepaper

*This work is supported by the U.S. Office of Naval Research grant N000149610401.

describes requirements for the operating environment, for the ORB architecture, and for the CORBA Object Services.

Our research group at the University of Rhode Island and the U.S Navy's NRaD facility, along with collaborators from the MITRE Corporation, produced an early design of real-time capabilities in a CORBA system [KSTW94, WBTK95], which is a partial basis for the RT SIG whitepaper. We then implemented a prototype of a real-time CORBA environment as an extension to the Orbix CORBA system from Iona Technologies. This paper describes our prototype system and the issues and techniques for adding real-time capabilities to CORBA.

Our prototype is designed to support end-to-end timing constraints in flexible *dynamic* CORBA environments. A dynamic CORBA environment is one in which clients and servers may be added and removed, and where constraints may change. This type of environment prohibits complete *a priori* analysis of timing behavior. Our Dynamic Real-Time CORBA system implements a best-effort approach towards enforcing timing constraints through global priority-based scheduling across the CORBA system, but does not offer hard real-time guarantees.

The main concept behind our Dynamic Real-Time CORBA system is support for *Timed Distributed Method Invocations* (TDMIs). A TDMI is a client's request to a server object along with real-time constraint information for the request, such as deadline, importance, and quality of service. The components that we added to the Orbix CORBA system express and enforce TDMI constraints. These components include a real-time library with types for expression of real-time constraints, and extensions to the ORB and Object Services to enforce the constraints. A major addition in our system is an Object Service to assign and enforce a global priority across the entire CORBA system based on the real-time constraint information in the TDMI. Our prototype implementation of this Object Service uses a dynamic *Earliest Deadline First within Importance* scheduling policy on all schedulable entities in the CORBA system. Our system also provides a CORBA Real-Time Concurrency Control Service to enforce consistency of server objects through a form of locking with priority inheritance; and provides a Real-Time Event Service to allow real-time event-based synchronization and constraints.

In this paper, we describe our Dynamic Real-Time CORBA design and prototype implementation along with results of performance tests. We present background on CORBA and Real-Time CORBA requirements in Section 2. Section 2 also describes related RT CORBA research. Section 3 describes our design and implementation. The testing environment, tests, and test results are described in Section 4. Section 5 concludes by summarizing the concepts and techniques for adding real-time capabilities to CORBA and speculates on further enhancements towards developing RT CORBA.

2 Background

This section provides background on the CORBA architecture and on the OMG RT SIG's requirements for extending CORBA for real-time. The section also describes related work in real-time CORBA.

Track Table Server IDL

```
interface Track_table
{ readonly attribute short max_length;
    short put(in short index,in long data);
    long get(in short index);
}
```

Client Code

```
long retval;
Track_table *p;
p = bind("my_Track_table");
 retval = p->get(500);
```

Figure 1: Sample IDL and Client Code

2.1 CORBA

The CORBA standard developed by the OMG deals primarily with the basic framework for applications to access objects in a distributed environment. This framework includes an object interface specification and the enabling of remote method calls from a client to a server object. Issues such as naming, events, relationships, transactions, and concurrency control are also addressed in the CORBA 2.0 specification [OMG96]. Services such as time synchronization and security are expected to be addressed in later revisions.

CORBA is designed to allow a programmer to construct object-oriented programs without regard to traditional object boundaries such as address spaces or location of the object in a distributed system. The CORBA specification includes: an *Interface Definition Language(IDL)*, that defines the object interfaces within the CORBA environment; an *Object Request Broker* (ORB), which is the middleware that enables the seamless interaction between distributed client objects and server objects; and *Object Services*, which facilitate standard client/server interaction with capabilities such as naming, event-based synchronization, and concurrency control.

CORBA IDL. CORBA IDL is a declarative language that describes the interfaces to server object implementations, including the signatures of all server object methods callable by clients. As an example, consider an object that acts as a shared table for tracking data (represented as long integer values) for clients in a distributed system. CORBA IDL for a simple *Track_table* object is displayed in Figure 1. The IDL keyword `interface` indicates a CORBA object (similar to a C++ class declaration). A `readonly attribute` is a data value in the object that a client may read. This IDL example also specifies two methods: *put*, which stores a data value at a index into the table; and *get* which returns a data value given an index.

Figure 1 also displays possible client code in C to access the *Track_table* object in a CORBA environment. The client must first bind to the *Track_table* object before calling the *get* method on the server. This method invocation assumes that a *Track_table* server was previously implemented and registered with the CORBA ORB.

The ORB and Object Services An ORB provides the services that locate a server object implementation for servicing a client's request; establish a connection to the server; communicate the data making up the request; activate and deactivate objects and their implementations; and generate and interpret object

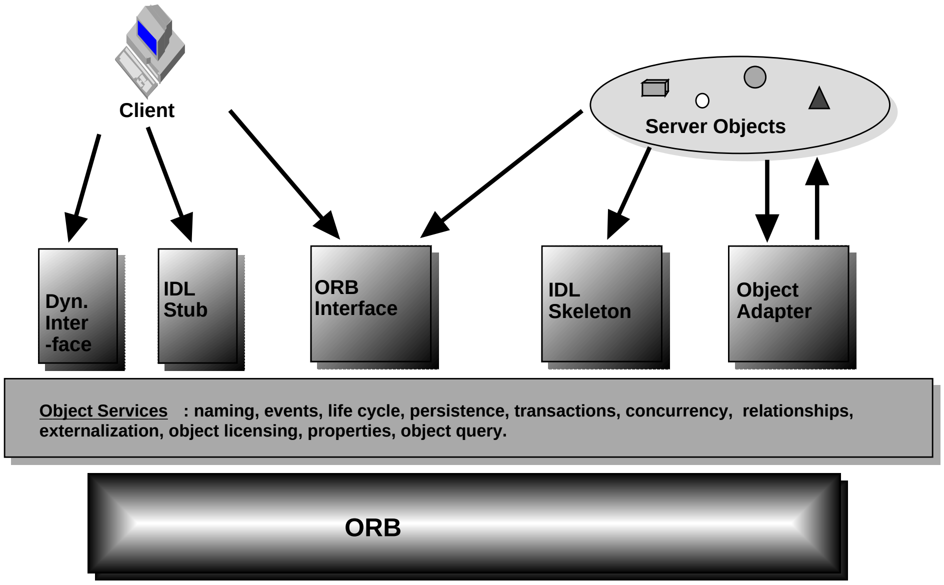

Figure 2: CORBA System Components

references.

Figure 2 illustrates the parts of a CORBA system. The client stubs and the server skeletons are produced by the IDL compiler. There is a stub and skeleton for each method on a server's interface. A method's stub is linked with the client code to hide the details of communicating with the server. The skeleton is linked with the server code to allow application developers to create servers without knowing the communication details. Server skeleton code is used by the ORB in forwarding method invocation requests to the server, and in returning results to the client. Using the stubs, the skeletons, the ORB, and a component called the Basic Object Adapter, the CORBA system handles all details of the distributed method invocation so that the distribution is essentially transparent to both the client and server application developers.

The CORBA standard contains specifications for Object Services that facilitate client/server interaction. These services include a naming service for binding a name to an object; an event service for notification of named events; and a concurrency control service for locking of resources to maintain consistency. A more complete list of the Object Services can be found in [OMG96].

2.2 Real-Time CORBA

The OMG RT SIG is currently defining the specifications for RT CORBA. The essence of its definition is:

> *Real-Time CORBA deals with the expression and enforcement of real-time constraints on end-to-end execution in a CORBA system.*

Consider a real-time scenario where a client needs to perform a *get* method from the $Track\_table$ server of Figure 1 within timing constraints. This interaction means that the client must have some way of expressing timing constraints on its request, and that the CORBA system must provide an ORB and Object Services that support enforcement of the expressed timing constraints. It also means that the underlying operating systems on the client and server nodes, along with the network that they use to communicate, must support enforcement of real-time constraints. Thus, there are two main categories of real-time CORBA requirements: requirements on the operating environment (operating systems and networks); and requirements on the CORBA run-time system. The operating environment requirements include requirements for synchronized clocks, for bounded message delay, for priority-based scheduling, and for priority inheritance of operating environment entities.

The requirements on the ORB and Object Services involve providing for specification and enforcement of end-to-end timing constraints on client/server interactions. Some of these requirements are:

- *Transmittal of Real-Time Method Invocation Information.* The standard should allow a client to attach timing constraint information, such as deadline, importance and quality of service, to a method invocation. This information will be available to the ORB, ORB Services, skeletons, and server implementations in order to enforce the real-time constraints.

- *Global Priority.* The ORB should establish global priorities for all execution so that the priorities of any tasks that compete for any resource in the real-time CORBA environment are set relative to each other.

- *Priority Queueing of All CORBA Services.* All real-time CORBA-level software should use priority based queuing. For instance, queues of requests for CORBA 2.0 services such as Naming should be priority queues.

- *Real-Time Events.* The real-time CORBA environment should provide the ability for clients and servers to determine the absolute time value of "events". These events may include the current time (provided by a Global Time Service), or named events provided by the CORBA 2.0 Event Service. Furthermore, events should be delivered in an order reflecting either the priority of the event or the priority of the event consumer, or both.

- *Priority Inheritance.* All real-time CORBA-level software that queues one task while another is executing should use priority inheritance. This requirement includes the locking done by the CORBA 2.0 Concurrency Control Service, but also includes simple queuing such as waiting for the Naming Service.

A concise summary of the real-time CORBA requirements can be found in [WDG+97] and a full listing can be found in the RT SIG whitepaper [otO96].

2.3 Related Work

There have been several real-time CORBA projects initiated over the past few years. One early approach to real-time CORBA was to install a non-real-time ORB on real-time operating systems. These ported ORBs did not take advantage of most of the operating system's real-time features. Furthermore, although implementation on a real-time operating system may be necessary for real-time CORBA, it is not sufficient to enforce end-to-end timing constraints in a distributed system.

Several projects have sought to realize "real-time ORBs" that are stripped-down, faster, versions of existing ORBs. They removed features like CORBA's Dynamic Invocation Interface and allowed special protocols with fixed point-to-point connections of clients to servers that by-passed most CORBA features. Such high performance might also be necessary in a real-time CORBA system, but it may not be sufficient for predictable enforcement of end-to-end timing constraints.

MITRE has done work [KSTW94, Ben96] to identify requirements for the use of real-time CORBA in command and control systems. They have prototyped the approach by porting the ILU ORB from Xerox to the Lynx real-time operating system. This system provides a static distributed scheduling service supporting rate-monotonic and deadline-monotonic techniques.

Researchers at Washington University in St. Louis are developing a high-performance endsystem architecture for real-time CORBA called TAO (The ACE ORB) [HGSP96]. The focus of this work is on hard real-time systems, requiring *a priori* guarantees of Quality of Service (QoS) requirements. The key components of TAO include a Gigabit I/O subsystem; a method for specifying QoS requirements; a real-time inter-orb protocol for transferring QoS parameters; a real-time scheduling service; a real-time object adapter with a real-time event service; and presentation layer components.

The CHORUS/COOL ORB is a flexible real-time ORB that is being developed by Chorus Systems [Sys96]. The design enforces a strict separation between resource management policy and mechanism. The design philosophy also calls for providing applications with full control over operating system-level resources. Given this philosophy, the goals of the CHORUS/COOL ORB include: a flexible binding architecture; producing minimum CORBA on a minimal ORB; and a real-time operating environment that provides access to fine grain resource management. The COOL ORB from Chorus Systems does not provide many real-time features itself, but rather relies on the CHORUS operating systems. A strength of the COOL ORB is that it imposes minimal overhead on top of the native operating systems.

The preliminary work on the projects described in this section, along with our research and development described in the next section, provided the basis for the RT SIG whitepaper. The development of this whitepaper, in turn, is providing the start to a common set of requirements for real-time CORBA. Approaches such as CORBA on a real-time operating system and fast CORBA are necessary parts to real-time CORBA development. Static scheduling across the system, such that provided by the MITRE prototype and the TAO CORBA system, among others, are important steps to supporting hard real-time applications. The incorporation of dynamic real-time, where clients and servers can be added or removed, timing constraints may change, and priorities are not fixed, is the subject of our work described in the remainder of this paper.

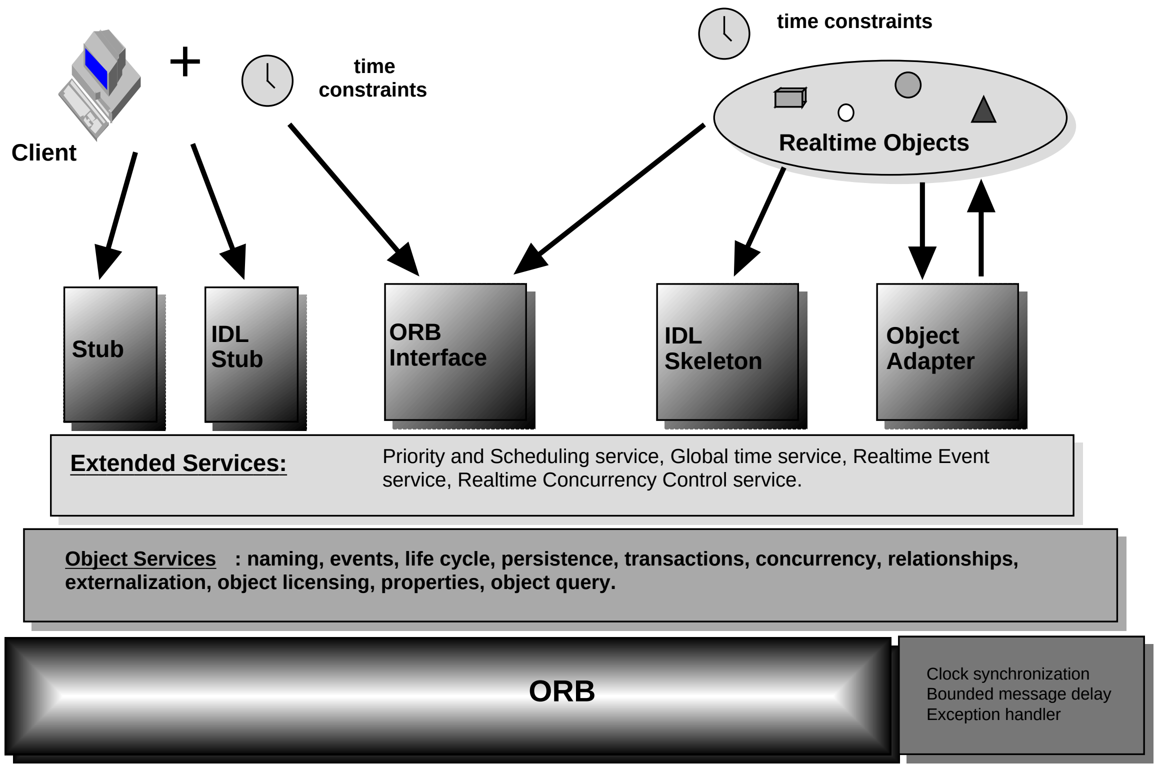

Figure 3: Dynamic Real-Time CORBA System

3 Dynamic Real-Time CORBA System

This section describes our Dynamic Real-Time CORBA system, shows how real-time constraints are expressed, and details how global dynamic real-time scheduling is performed.

A depiction of our Dynamic Real-Time CORBA system components is shown in Figure 3. This system is designed to augment an existing CORBA system; our prototype implementation augments the Orbix CORBA system from Iona Technologies. The extensions consist of minor changes to the basic ORB, modifications and additions to the Object Services, and modifications to the client-side and server-side high-level code, stubs, and skeletons.

The components of our Dynamic Real-Time CORBA system are implemented as a *Real-Time Daemon* process (RT Daemon) that executes on each real-time POSIX operating system in the system, and as a real-time library that provides type definitions, IDL definitions, and code that is used to link in with client and server code. The RT Daemon coordinates dynamic aspects of the system including changing global priorities, time synchronization, and supporting real-time events. The library code performs tasks such as initial priority assignment, handling of real-time information that is associated with all execution in the system, and handling of real-time exceptions. The remainder of this section details how this is done.

3.1 Underlying System

Our Dynamic Real-Time CORBA System assumes implementation on an underlying system that provides these features:

- Real-time operating systems on all nodes that provide:
  - Priority-based scheduling of tasks.
  - Priority-based queuing with priority inheritance within the operating system.
- Clock synchronization among nodes so that any two clocks are with a bounded skew ϵ of each other.
- Real-time networking that provides:
  - A message delay bound of δ.
  - Priority-based queuing of all network functions.

The current implementation uses only operating system features specified in the IEEE POSIX 1c real-time operating system standard [CLG90], which includes priority-based scheduling of threads and priority-based queueing with priority inheritance.

As depicted in Figure 3, our changes to the basic ORB include implementing a version of the NTP protocol to provide clock synchronization. Also depicted in the figure is a change to the ORB to ensure bounded message delays. Although the prototype implementation uses a dedicated network between two hosts to achieve bounded message delay, we have done research on providing bounded messages delays for a general real-time CORBA system on an ATM network; including the Latency Service described in Section 3.3.

3.2 Global Time Service.

For expressed timing constraints to be meaningful in a distributed system, a common global notion of time must be supported. Our Dynamic Real-Time CORBA system implements this by synchronizing the clocks and by providing a Global Time Service, which clients and servers can call using standard CORBA calls, to get the current time. Our Global Time Service calls simply make calls to the local operating system to get the (synchronized) time. The Global Time Service is described in detail in [Zyk97].

3.3 Latency Service.

Since bounded message delays are imperative to reasoning about behavior in a distributed real-time system, our Dynamic Real-Time CORBA system provides a Latency Service to allow clients and servers to determine various qualities of latency bounds in the network. The Latency Server accounts for varying "tightness" of bounds by allowing requests for bounds to specify the percentage of cases in which the latency bound must hold. A 100% specification requires that the bound must always hold. A specification of 98% specifies that the value of the returned latency must be greater than the actual latency 98% of the time. The Latency Service uses three methods to provide latency bounds:

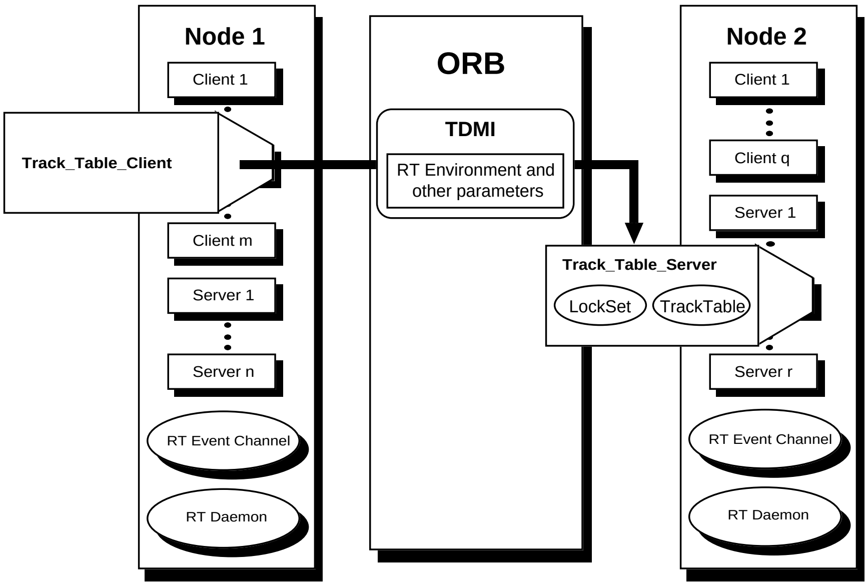

Figure 4: Timed Distributed Method Invocation in Our Dynamic Real-Time CORBA system

- *Estimated Latencies* - *A priori* measurements are used to establish latencies. The implementation of this form of service is a simple lookup in a table of latencies on each node.
- *Measured Latencies* - The Latency Service performs latency measurements when system changes occur. This method typically yields more accurate latency measurements, but requires significant overhead in both the call to the Latency Service and in background measurements taken by the Latency Service(s).
- *Analytical Latencies* - The Latency Service uses network parameters to calculate expected latencies. In our prototype, the Latency Service uses SNMP information provided by each node to make latency estimates.

Which form of service the Latency Service provides to a client is determined by the quality of the bound needed, as specified by a parameter in the Latency Service call. Typically, higher quality bounds, like measured and analytical bounds, require higher overhead and take longer to process. The implementation of the Latency Service that provides these capabilities on an ATM network is described in [Pal97].

3.4 Specification of Real-Time Constraints

Our Dynamic Real-Time CORBA system allows clients to express dynamic constraints on execution through *Timed Distributed Method Invocations (TDMIs).* In our prototype implementation these constraints include *deadlines* and *importance.* Importance is an ordinal application-level specification of the relative value to the system of an execution[1]. Note that although the current system supports only deadline and importance specifications, the design is flexible enough to allow other constraints including periods and various Quality of Service parameters to be expressed.

[1]Importance differs from *priority*, which is an implementation-level attribute used to order various executions.

Executions in our Dynamic Real-Time CORBA system use a *RT_Environment* structure and a *RT_Manager* class, both defined in the Real-Time Library, to convey real-time information (see Figure 4). A *RT_Environment* structure contains attributes that include the importance and deadline. Our Dynamic Real-Time CORBA run-time system attaches the *RT_Environment* structure to all executions in the system. Other parts of our Dynamic Real-Time CORBA run-time system examine this structure to acquire information necessary to enforce the expressed real-time requirements by doing things such as establishing priority and setting operating system timers.

As an example of expressing real-time constraints, consider a client making a TDMI to a server that contains a table of tracking data for some tracking application. The client requires that the results of its query to the tracking table be returned within a specified deadline.

The following is the IDL for the table:

```
#include "rt_info.idl"

struct Track_Record {
     // contains track ID, position, etc.
};

interface Track_Table {
     void         Put(in Track_Record track, in RT_Environment rt_env);
     Track_Record Get(in long track_id, in RT_Environment rt_env);
};
```

The two methods on the table's interface enable clients to insert ($Put()$) and retrieve ($Get()$) track data. The code for a client of the table looks as follows:

```
     #include Track_Table.hh             // header file generated by IDL compiler
     #include RT_Manager.h               // header file for RT_Manager class
     #include Track_Table_i.h            // header file for table implementation
       :
(1)  RT_Manager rt_mgr;                  // create instance of RT_Manager
     Track_Table*  Track_Table_Obj; // declare pointer to table
       :
     int main()                          // main procedure of a CORBA client
     {
       :
       // bind to the appropriate Track_Table (in this case, the
       // one managed by the server named Track_Table_Server).

(2)    Track_Table_Obj = Track_Table::_bind("Track_Table_Server");
       CORBA::Long track_id = 42;

       try {
         :
         :     // set constraints and scheduling parameters
(3)      rt_mgr.Set_RT_Constraint_Now(BY,REL,3,0); // deadline = NOW + 3 seconds
```

```
(4)      rt_mgr.Start_RT_Invocation();
         // start TDMI:  1) calculate Transient Priority
         //              2) call RT Daemon and register as an active client
         //              3) map Transient Priority to this node's priority
         //                 set and change this thread to the new priority
         //              4) arm the timer

(5)      Track_Record track = Track_Table_Obj->Get(track_id, rt_mgr.Get_RT_Env());
(6)      rt_mgr.End_RT_Invocation();
         // finish TDMI 1) call RT Daemon and deregister as a client
         //             2) disarm the timer
         //             3) restore this thread to its original priority
       }

(7)    catch(const RT_Exception &rtp) {       // catch RT_Exception
         cout << ‘‘RT_Exception Raised :’’  << rtp.reason << endl;
       }
       :
     }
```

The client first creates a *RT_Manager* object (Label 1 in the above code). It then binds to the appropriate server (Label 2). Next, it calls the *RT_Manager* functions necessary to set timing constraints and scheduling parameters (Label 3). In our example, we set a relative deadline of 3 seconds in the *Set_RT_Constraint*() method. The bulk of the work is done inside of the *Start_RT_Invocation*() (Label 4) function and is transparent to the client. *Start_RT_Invocation*() calls the RT Library functions to register the client with the RT Daemon on its node, to calculate the priority for the client (described in Section 3.6.1), and to arm a timer in the real-time operating system to expire at the client's deadline.

After the above sequence is complete, the client makes the TDMI to the table (Label 5). The *RT_Environment* that is sent with the call contains the timing information computed by the *RT_Manager*. Figure 4 depicts a typical TDMI such as this one. Also in Figure 4, note the existence of a RT Daemon on each node.

When the TDMI request is received by the server skeleton, the execution is scheduled on the server's node by server library code going through a procedure similar to that of the client: registering the thread as a server with its local RT Daemon, establishing a priority, and arming a timer. How this is done is described further in Section 3.6. If the client has not missed its deadline when the TDMI returns, then *End_RT_Invocation*() (Label 6) disarms the client's deadline timer and performs some clean up. If the timer expires (i.e., the deadline is missed), a new real-time CORBA exception of type *RT_Exception* is raised in the client. The client catches this exception (Label 7) and performs any necessary recovery operations.

3.5 Real-Time Event Service.

An important aspect of expressing and enforcing real-time constraints and providing synchronization is the use of real-time events. The current CORBA 2.0 Event Service allows for the exchange of named events in the CORBA system. For instance, a client might synchronize with another client by waiting for the first client to generate a CORBA event. Our Dynamic Real-Time CORBA system has a modified *Real-Time*

Event Service that prioritizes the delivery of events and delivers the time that the event occurred. Prioritized events are based on the global priorities of the producers and consumers as set by the Priority Service. The (global) time of an event occurrence is important to allow expression of timing constraints relative to events. The following example illustrates this concept.

3.5.1 Example of Real-Time Events For Expressing Timing Constraints

In the above real-time CORBA TDMI example of Section 3.4, the deadline for the client request was based on an absolute time that is relative to the current time (three seconds from the current time). Now consider the case where the client's deadline is event-driven. Let the deadline for the TDMI on the tracking table server be $NewContact + 3secs$, where $NewContact$ is a named event that occurs when a new contact is entered into the table. In this case, the client first has to create a *RT_Event* object, specify a real-time event ID number, importance of the event, and event source (a server name). The revised code for an event-driven client is:

```
      :
      try {
        :
        RT_Event  rt_event("NewContact",5);  // create RT_Event object with event name & ID 5
        rt_event.Set_Importance(1000);
        rt_event.Set_Server_Name("Track_Table_Server");
        rt_event.Set_Push_Consumer();        // act as a push consumer
        :
        // set constraints and scheduling parameters
        :
        // deadline = the absolute time when the named Event "NewContact" happened + 3 secs
(1)     rt_mgr.Set_RT_Constraint_Event(BY,&rt_event,3,0);

(2)     rt_mgr.Start_RT_Invocation();        // start TDMI

        Track_Record track = Track_Table_Obj->Get(track_id, rt_mgr.Get_RT_Env());

        rt_mgr.End_RT_Invocation();          // finish TDMI
      }
      :
```

The *Set_Time_Constraint_Event*() function call (Label 1 in above code) causes the client to wait, with infinite deadline, for a notification that the real-time event has occurred. The deadline for the client's request is determined by the absolute time when that event occurred plus 3 seconds. This timing constraint is stored in the *RT_Environment*, and the rest of the work is done inside of the *Start_RT_Invocation*() function as previously described (Label 2).

3.5.2 Implementation of Real-Time Event Service

The implementation of a RT Event Service is based on IP multicasting [Moy] and takes advantage of multi-threading in the local operating systems. Each node has a CORBA *Event_Channel* [OMG96] component in its RT Daemon that is configured to "listen" to a pre-defined IP multicast group. Each real-time event has a unique event ID number, which is mapped to the IP address for the multicast group. Suppliers transport real-time event data to each RT Event Channel by multicasting to its IP address. Event consumers can wait for delivery of real-time events to the IP multicast groups associated with the events, or they can invoke the local RT Event Channel to retrieve the real-time event. In the prototype, each RT Event Channel buffers the incoming events in priority order so that consumers can look for the buffered high priority real-time events first. If the real-time event data is not in the buffer, then the RT Event Channel raises a RT exception to the consumer, which is handled as described in Section 3.4. Further details on the Real-Time Event Service are provided in [Zyk97].

3.6 Global Priority Service and Real-Time Scheduling

Dynamic real-time scheduling is done by establishing a global priority assignment for all execution in the Dynamic Real-Time CORBA system. Each client communicates its scheduling parameters to the Global Priority Service, and in turn receives a global priority for its execution. These priorities are dynamic and may change over the lifetime of the execution.

We call an execution's priority at an instant in time its *transient priority*. A transient priority is an integer that is derived by the Global Priority Service based on the information in the *RT_Environment* for the execution. The Global Priority Service ensures that the transient priority is meaningful relative to all other transient priorities in our Dynamic Real-Time CORBA system. That is, much like a single real-time operating system assigning priorities within its local domain, the Global Priority Service assigns priorities that are meaningful across the real-time CORBA domain. The RT Daemon on each node maps each execution's transient priorities to the priorities on the various schedulable entities that the RT Daemon manages. For instance, in the prototype, the RT Daemon maps a client's transient priority to one of the 60 real-time priorities that the local Solaris operating system allows. An execution's transient priority is dynamic and may change during the execution for several reasons that we describe below, including re-calculation, aging, and inheritance.

3.6.1 Transient Priority Calculation

Our Global Priority Service uses a uniform function for all clients and servers in the system to compute transient priority using the attributes in the *RT_Environment* that is associated with the execution. Our prototype uses a global *earliest-deadline-first within importance* priority assignment scheme. That is, the prototype's transient priority function orders priorities based on the *importance* attribute first, and then based on the *deadline* attribute. A transient priority is a seven digit value, where the millions digit represents importance, and the lower order digits represent a time difference (multiplied by 100,000) between the maximum allowable deadline and the deadline specified in the *RT_Environment* for the execution. For instance, if the maximum deadline is 10 seconds, then execution with importance level 2 and a deadline of 3 seconds has a transient priority of 2,700,000. Changing the calculation of transient priorities based on other scheduling policies, such as global rate-monotonic priority assignment, is facilitated by the function's central implementation in the Global Priority Service.

The implementation of the Global Priority Service in our prototype is accomplished through a combination of the RT Daemon and code from the RT Library. The library code calculates the initial transient priority, the RT Daemon handles mapping an execution's transient priority to a priority on a local node, and also handles changing the transient priority.

3.6.2 Priority Mapping

The RT Daemon on each node maps the transient priority to the priorities available on the local real-time operating system. The function that performs the mapping must be written for each operating system individually because of the variability in ranges of real-time priorities present on different systems (e.g., Solaris has 60 local priorities, and LynxOS has 256). In the prototype, which uses RT Solaris operating systems, the RT Daemon must map the (wide) range of transient priorities into the 60 local priorities. The mapping is done by using a statistical model of the likely deadlines and calculating transient priorities such that TDMIs are probabalistically evenly distributed among the local priorities. For example, if there were 60 executions to be scheduled on a Solaris node, the mapping would reduce the probability that two executions would be at the same priority. Unfortunately mapping of a large range of transient priority values into a smaller range of priorities can cause more than one transient priority to be mapped to a single local priority value, which could cause some execution to be out of deadline order.

3.6.3 Dynamic Transient Priority Re-Calculation

An execution may have different transient priorities at different times during its lifetime. For instance, a real-time CORBA client could have an initial transient priority based solely on its importance with no deadline. It then might enter phases of its execution that must be done under deadlines (as specified by a `Set_RT_Constraints` *RT_Manager* method call in the client). Thus, each `Start_RT_Invocation` call must re-calculate the transient priority for the execution that makes the call. Similarly, the `End_RT_Invocation` method call, recalculates a transient priority using the deadline (if any) that was in effect before its associated `Start_RT_Invocation` call.

Another re-calculation of transient priority is done when a client makes a TDMI to a server. Assume that the client's deadline constraint is d_{client}. This means that the return message from the server with results for the client must be received by the client by d_{client} as measured on the client's clock. Recall from Section 3.1 that we assume synchronized clocks with maximum skew ϵ, and assume maximum network message delay δ. To calculate the transient priority at which a server should execute on behalf of the client, the server uses the deadline $d_{server} = d_{client} - \delta - \epsilon$ to pessimistically allow for δ message delivery time back to the client and an ϵ clock skew between its clock and the client's clock. Since this deadline is tighter than the client's deadline on whose behalf the server is executing, the TDMI will usually have a higher transient priority when executing in the server than it will while executing in the client.

3.6.4 Transient Priority Aging

Another change in an execution's transient priority is performed by the RT Daemons in our Dynamic Real-Time CORBA system enforcing *aging* of transient priorities. Aging is the process of increasing priority as time goes on, which is necessary in dynamic earliest-deadline-first scheduling. Each RT Daemon keeps track of the transient priorities on its node. A RT Daemon increases an execution's transient priority if, due to the passage of time, the execution's transient priority is too low compared to a newly-arrived execution on the node which the RT Daemon controls. Note that in our prototype the aging facility can be "turned off" for real-time scheduling policies that do not require aging, such as a static rate-monotonic-based policy.

Another source of possible of transient priority change is priority inheritance in the Real-Time CORBA Concurrency Control Service, as described next.

3.7 Real-Time Concurrency Control Service.

CORBA 2.0 provides a Concurrency Control Service to maintain consistent access to servers. Our Dynamic Real-Time CORBA system includes a Real-Time Concurrency Control Service that implements *priority inheritance* [Raj91]. When a TDMI requests a lock on a resource from the Real-Time Concurrency Control Service, the TDMI's execution priority is compared to those of all TDMIs holding conflicting locks on that resource. Conflicting TDMIs with lower priorities are raised to the requesting TDMI's priority, and the requesting TDMI is suspended. Whenever a lock is released, the releasing TDMI resets its priority to that of the highest priority TDMI it still blocks (this is possible since clients can hold several locks of different types). If it no longer blocks any higher priority TDMIs, then the releasing TDMI is reset to its original priority. Finally, the highest priority blocked TDMI that can now run is allowed to obtain its lock and continue execution. In designing the Real-Time Concurrency Control Service, we needed to consider several issues including whether to allow explicit locking and how to handle transient priority inheritance. We now address these issues and then provide an example of the use of the Real-Time Concurrency Control Service.

3.7.1 Implicit vs. Explicit Locking

When using the CORBA Concurrency Control Service, two forms of locking are possible: *implicit locking* and *explicit locking*. Implicit locking is done within the method code. This placement of lock calls simplifies the usage of the resource because clients 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 to execute a sequence of method calls that are 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 high-level client code explicitly making CORBA Concurrency Control Service calls to obtain the necessary locks. However, explicit locking requires the client to know which internal locks 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.

3.7.2 Transitive Priority Blocking

Another issue that must be addressed is that of *transitive blocking*. There are two forms of transitive blocking in which a high priority activity A_3 is indirectly blocked by a lower priority activity A_2. If A_1 is the activity that directly blocks A_2, then either:

1. A_2 is holding a lock that is blocking activity A_1; or
2. A_2 is executing 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 other activities. The difficulty with transitive priority blocking is the fact that these blocking chains can become arbitrarily long, especially when activities are allowed to lock multiple resources. This locking can require a great deal of overhead to implement. Therefore, our prototype implementation of the Real-Time Concurrency Control Service 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 lock would constitute starting "child" activities under the initial lock. The only transitive blocking that is allowed in our prototype is that which occurs within a *LockSet*, which is a CORBA 2.0 lock object for a single resource. That is, blocking chains are allowed to form as long as all of the clients in the chain are clients of the same resource and do not start any "child" activities while they hold locks.

3.7.3 Real-Time Concurrency Control Interface

One of the goals of our Dynamic Real-Time CORBA system was to ensure that the interfaces to various CORBA Object Services were changed as little as possible. The only change to the CORBA Concurrency Control interface is that a *RT_Environment* is passed into each TDMI for use in implementing priority inheritance. 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 following CORBA IDL shows a subset of the Concurrency Control Service that was implemented and extended for real-time.

```
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);
  };

};
```

The revised IDL for the Real-Time Concurrency Control Service is shown here:

```
#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);
  };

};
```

The design of the Real-Time Concurrency Control Service 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 requested in a pre-determined order.

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.

3.7.4 Example With Real-Time Concurrency Control

Consider the case where the tracking table requires concurrency control to handle multiple clients and to maintain its data in a consistent state. The following code illustrates how the table's *Get* method implements implicit locking using the Real-Time Concurrency Control Service provided by our Dynamic Real-Time CORBA system.

```
Track_Record TrackTable::Get(CORBA::Long track_id, const RT_Environment& rt_env)
{
     Track_Record track;

     try {
       RT_Env_Mgr rt_mgr(rt_env);

       rt_mgr.Begin_RT();

       {
         CosConcurrencyControl::LockSet* LockSetObj;
(1)      LockSetObj = CosConcurrencyControl::LockSet::_bind("Track_Table_Server");

(2)      LockSetObj->lock(CosConcurrencyControl::read, rt_mgr.Get_RT_Env());

         // Code for retrieval of TrackRecord with the specified ID

(3)      LockSetObj->unlock(CosConcurrencyControl::read, rt_mgr.Get_RT_Env());
       }

       rt_mgr.End_RT();
     }

     catch(const RT_Exception &rtp) {}

     return track;
}
```

The method first binds to the *LockSet* object (Label 1 in above code) that is associated with the table (both the *LockSet* and table are managed by the same server, namely, *Track_Table_Server*). The *Get* method then makes the request for a read lock on behalf of the client (Label 2). Thus, the *Get* method invocation becomes a client of the *LockSet* object. Below is the sequence of steps that occur within the *LockSet* object when a lock is requested:

1. The *Get* method invocation requests a lock (CORBA call to *LockSet* object).

2. The *LockSet* object grants the requested lock if it does not conflict with any locks currently held (skip Steps 3-5). This simply involves incrementing a counter within the *LockSet* object (each *LockSet* client has one counter per lock type).

3. Otherwise, the *LockSet* object determines which clients hold locks that are blocking the requesting client.

4. The *LockSet* uses a function in the RT Library to raise the priority of each blocking client to that of the requesting client.

5. The requesting client then waits on the *LockSet*'s condition variable.

Once the *Get* method invocation finishes reading from the table, it releases the read lock (Label 3). The following is the sequence of steps that occurs when the lock is released:

1. The *Get* method invocation requests to release a lock (CORBA call to *LockSet* object).

2. The *LockSet* object releases one instance of the specified lock for that client (remember that a client can hold multiple instances of a given lock). This simply involves decrementing the appropriate counter. If the client does not hold the indicated lock, an exception is raised.

3. The *LockSet* uses a function in the RT Library to set the releasing client's priority to that of the highest priority client still blocked by the releasing client. If it does not block any clients, its priority is reset to its original value.

4. Finally, the *LockSet* object sends a broadcast signal to its condition variable (the intention is to wake the highest priority client that can run).

Further details of the Real-Time Concurrency Control Service can be found in [Woh97].

3.8 Summary

Our Dynamic Real-Time CORBA system has been prototyped as an extension to Orbix, but was designed without access to Orbix source code and using only real-time POSIX operating system features. Thus, it is suitable for use as the basis of extending many different CORBA systems to support real-time. Our extensions are packaged as a Real-Time Library of definitions and linkable code, and a Real-Time Daemon that executes on each node in the system. We have provided the prototype to companies including Iona Technologies, Tri-Pacific Corporation, and Computing Devices International, as well as several U.S Navy programs for use in their applications and products.

4 Performance Tests

We have performed tests on the prototype implementation of our Dynamic Real-Time CORBA system in order to demonstrate how it enforces expressed timing constraints. We ran several tests to indicate raw performance numbers for individual features of the system. We also ran tests to determine how many client deadlines the prototype missed under varying system conditions. In this section we describe the testing environment, as well as the tests we performed and the results of the tests.

Due to the limited size of our prototype, the tests described here are not meant to portray the capabilities of a full-scale implementation of our Dynamic Real-Time CORBA system. Rather, these tests are meant to illustrate how such an implementation might perform under varying conditions.

All of the tests described below were performed on the two-node isolated network described in Section 3. A network delay of approximately 1.2 ms was measured on the system.

4.1 Overhead Performance Tests

The tests described here were performed to determine the amount of overhead in the system that was due to changes we made to the Orbix system to add real-time capabilities. We discuss tests involving Timed Distributed Method Invocations, the Real-Time Event Service, and the Real-Time Concurrency Control Service. The results described here are summaries of extensive testing performed on each of these Services. For more detailed descriptions of these tests and results, see [Woh97, Zyk97].

4.1.1 TDMI Overhead Tests

The results of tests described here are averages over 25 trials, with an error of 1% or less. The tests had a client running on a Sun Sparc IPX station, and a server running on a Sun Sparc Station 5. When a client sends a TDMI to a server, the first source of overhead is the addition of the *RT_Environment* structure to the method invocation. The extra data copying, moving, dereferencing and transmission that must be done by the stubs, skeletons and the ORB amounts to approximately 3 ms per method invocation.

On both the client node and the server node, the bulk of the overhead of the TDMI is in the setting up the real-time information. On the client side, most of the overhead was produced by the following two methods:

- *Start_RT_Invocation*(): The *RT_Manager_Client* method that registers a client with a Real-Time Daemon, calculates and assigns the transient priority, and arms a timer with a signal handling function. The latency introduced by this method averaged 25.2 ms.
- *End_RT_Invocation*(): The *RT_Manager_Client* method that deregisters a client with a Real-Time Daemon, changes the transient priority to its base priority, and disarms the timer. The latency introduced by this method averaged 10.7 ms.

On the server side most of the overhead was produced by the following two methods:

- *START_RT*(): The *RT_Manager_Server* method that registers a server's thread with a Real-Time Daemon, calculates and assigns the transient priority, and arms a timer with a signal handling function. The latency introduced by this method averaged 11.4 ms.

	Consumer on same node	Consumer on different node
Event Response Time	90.6 ms	96.6 ms

Table 1: Real-Time Event Channel Overhead

	One Client	**Two Clients**	
	Write Lock	Low Prio Client unlock	High Prio Client
PI Enabled	102.62 ms	82.60 ms	129.90 ms
PI Disabled	80.39 ms	65.22 ms	83.54 ms

Table 2: Priority Inheritance Overhead Measurements

- *END_RT*(): The *RT_Manager_Server* method that deregisters a server's thread with a Real-Time Daemon, changes the transient priority to its base priority, and disarms the timer. The latency introduced by this method averaged 6.1 ms.

4.1.2 Real-Time Event Service Overhead Tests

The implementation of the Real-Time Event Service was tested by periodically starting up the suppliers on one node that would send real-time events to the consumers on another node via the Real-Time Event Channels. Table 4.1.2 shows the event response time, the time between a supplier generating a real-time event and a consumer receiving the event, for each of these tests.

The tests for the Real-Time Event Channel involved a supplier running on the Sun Sparc Station 5, and two consumers, one on the same node, and the other on the Sun Sparc IPX. An analysis of the results shown in Table 4.1.2 indicates that 85-90% of the overhead was due to network communications via IP Multicasting.

We also performed a set of tests in which a consumer periodically invoked the Real-Time Event Channel on its node. The overhead for this test was 161.0 ms. These results include the time to make a *bind*() call to the Real-Time Event Channel, which itself took approximately 43.4 ms.

4.1.3 Concurrency Control Overhead Tests

The majority of the overhead that was added to the Real-Time Concurrency Control Service comes from priority inheritance. We ran two tests to compare the overhead involved when priority inheritance was enabled with overhead when there was no priority inheritance. The results of each test were compiled by averaging over 100 trials. The first test involved a single client that requested a write lock from a server on another node. The second test involved two clients both trying to access the same write lock on a server. In this test, a low priority client first requested and got a write lock. Then the high priority client requested the write lock, but was blocked. The low priority client released the lock and then the high priority client was granted the lock.

Table 2 displays the overheads for each of the tests that we performed. A more detailed breakdown of the overhead for these tests can be found in [Woh97]. Notice in Table 2 that there are no reported overhead numbers for the low priority client getting the write lock in the two client test. This is because there is nothing added to this operation for priority inheritance. In all three reported cases, a difference in overhead comes from the mechanism used to update the priority of a low priority client for priority inheritance, and the mechanism for restoring the priority when a low priority client releases its lock. The results for the high priority client also reflect a difference due to a query the RT Daemon to detect the priority of the blocking client.

In all of the overhead performance tests, a majority of the overhead came from CORBA calls that were made through the Orbix system. Unfortunately, we did not have access to the source code for Orbix, and so we had to work with the system that was not designed for real-time.

4.2 Missed Deadline Performance Tests

In this set of tests, we compared our prototype with an implementation of a non-real-time ORB (Orbix 2.0.1) on a real-time operating system (Solaris 2.5). In this baseline system, all clients executed at the same real-time priority, and so scheduling reverted to a round-robin scheme. We chose this comparison for several reasons. First, the Orbix on Solaris implementation provides a good baseline for a “first step” towards real-time CORBA. In fact, there have been claims that running a CORBA implementation on a real-time operating system can be called Real-Time CORBA [WDG+97]. Second, there are no other existing dynamic real-time CORBA implementations available for comparison.

4.2.1 Testing Environment.

We chose to use percentage of missed deadlines as our performance metric because the goal of our prototype is to provide for the expression and enforcement of timing constraints in the CORBA environment. Missed deadline percentage is a clear measure of how well timing constraints (deadlines) are enforced.

To examine different system workloads, we varied the number of clients that execute concurrently. We used an increasing range of start times to implement this. That is, in some tests, all clients start at the same time, producing a very high workload. In other tests the start times for the clients were randomly chosen from a wider range of values. Overall, we examined five different start time ranges to represent different workloads. The five ranges from which start times were randomly generated for clients to produce varying system workloads were: 1-1 seconds (all clients start at the same time); 1-4 seconds (all clients start between 1 and 4 seconds after an initial start time); 1-7 seconds; 1-10 seconds; and 1-13 seconds.

Table 3 displays the parameters that we used in our testing. Each test involved two nodes. On one of the nodes resided a single server on which all of the method invocations were performed. There were certain system limitations that required a small number of servers, however for the purposes of our testing, one server was sufficient to demonstrate the capabilities of the system. There were 6 clients the other node, each

Parameter	Baseline Value
# Nodes	2
# Clients	6
# Servers	1
Client Start Time	1 - 13 sec
Client Deadline	6 sec
Write Probability	50%

Table 3: System Parameters

accessing the server on the first node. Each client started some time between one second and 13 seconds after an initial start time. The range of start times was varied in order to illustrate different system workloads. For the baseline test, each client had a deadline of 6 seconds. One of the suites of tests we performed (see Section 4.2.2) indicates how system performance changes as the clients' deadlines change. Each client performed a single operation, either a read or a write on the server. The *Write Probability* of a client represents the probability that the operation performed by the client is a write. Write probability is a measure of contention for the server, since the concurrency control for the server must not allow more than one client to hold a write lock. In the baseline test, the write probability was 50%. That is, a client's operation was just as likely to be a read as a write. Another suite of tests was performed to illustrate the effect of write probability on the performance of the system (see Section 4.2.2).

4.2.2 Tests and Results.

Along with the baseline test that we performed, we also ran two suites of tests to illustrate how deadline and write probability affect the ability to meet timing constraints. Each suite of tests was made up of two tests for each start time range. Each test was the result of averaging over 15 random trials that produced a 95% confidence level with an error of at most 5%.

Baseline Test. The parameters for the baseline tests are listed in Table 3. We performed this test to illustrate how the system performs under average conditions.

Figure 5 displays the results for the baseline test. The x-axis of the graph represents system workload. The numbers that label the axis represent the outside number of the start time range. For instance, we can see that as system workload decreases (start time range increases) the percentage of missed deadlines decreases. This is expected since the contention for the CPU also decreases. The figure also shows that our Dynamic Real-Time CORBA system consistently meets more deadlines than the non-real-time CORBA implementation.

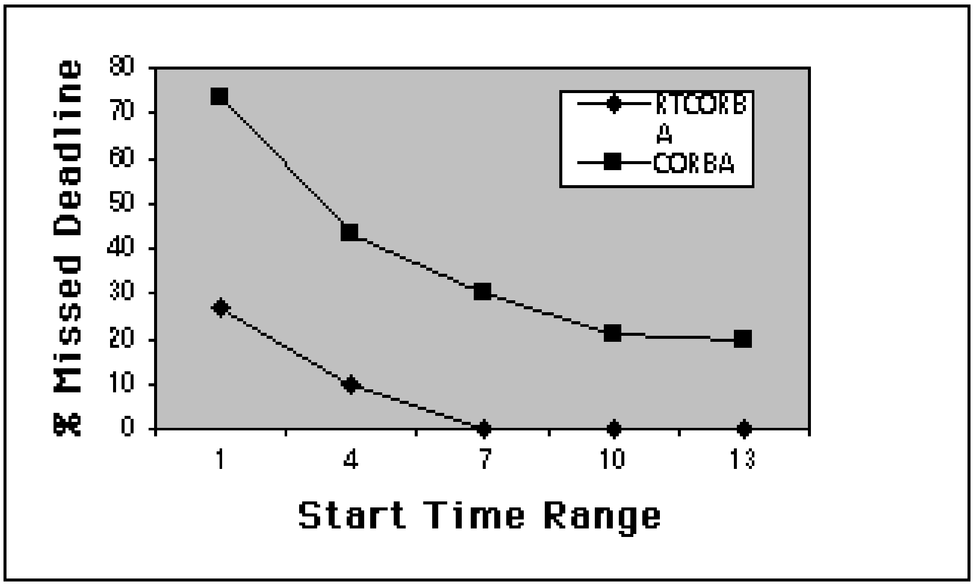

Figure 5: Baseline Test

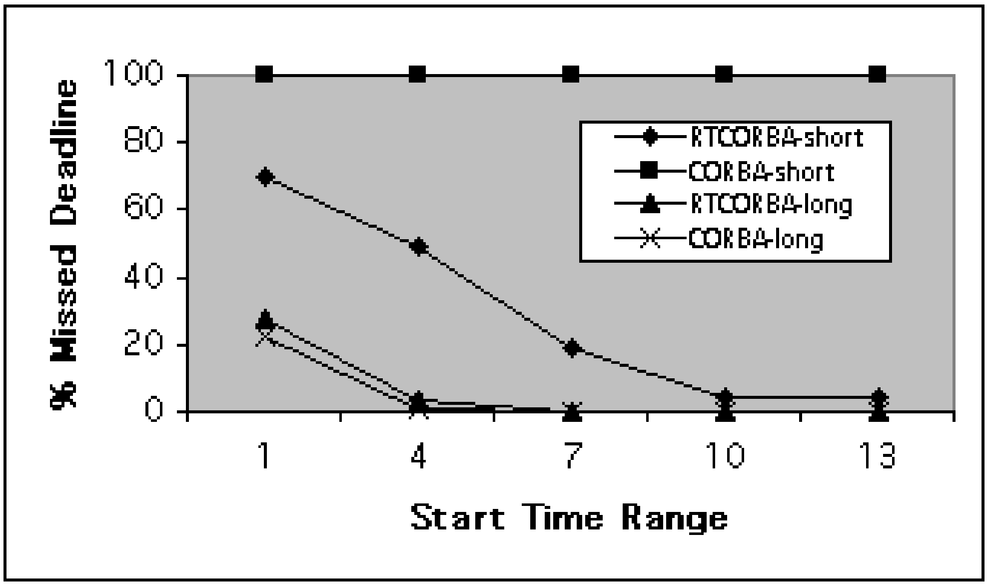

Figure 6: Deadline Tests

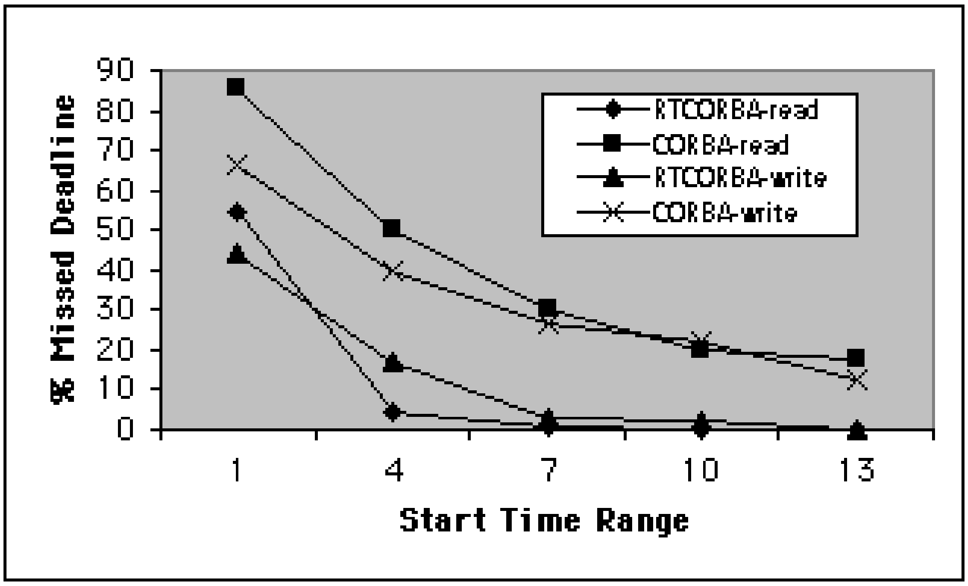

Figure 7: Write Probability Tests

Deadline Tests. We performed this suite of tests to illustrate the effect of deadline on the ability to meet timing constraints. Because the clients in the baseline test all had a 6 second deadline, which turned out to be a medium length deadline, this test examined short deadlines (4 seconds), and long deadlines (8 seconds).

The results of this test are displayed in Figure 6. For long deadlines, both implementations performed well, with our Dynamic Real-Time CORBA system performing slightly better than the non-real-time CORBA implementation at high system workload. As workload increased, both implementations missed no deadlines. This result is due to the fact that given a long enough deadline, all of the clients had time to complete, whether or not timing constraints were enforced.

When deadlines were short, however, there was a marked difference between the performances of our Dynamic Real-Time CORBA system and the non-real-time CORBA implementation. The non-real-time CORBA missed every deadline, while our system missed more at high system workload and virtually none at low system workload. This result illustrates how the enforcement of timing constraints enables more real-time clients to meet deadlines in a tight deadline situation. Because the non-real-time CORBA implementation schedules clients in a round-robin fashion, none of them was assigned enough CPU time to complete.

Write Probability Tests. In order to illustrate how resource contention affects the system's ability to meet timing constraints, we varied the write probability of clients accessing the server. The baseline test used a medium write probability of 50%. Therefore, this suite of tests examined the effect of low write probability (0%), in which none of the clients requested write locks on the server, and high write probability (100%), in which every client requested a write lock.

The results of this test are illustrated in Figure 7. It is clear, again, that our Dynamic Real-Time CORBA system missed fewer deadlines than the non-real-time CORBA implementation. One result that might seem surprising is that under high system workload, both systems missed more deadlines when clients requested only read locks, than when clients requested only write locks. This is surprising because one would expect that the blocking involved in write locking would cause more deadlines to be missed. In the non-real-time CORBA implementation, the explanation for this result lies in the fact that when one client holds a write lock, and blocks the others, the client holding the lock has an advantage over the others in a round-robin scheduling scheme. That is, while the client with the lock holds the lock, it will be the only execution that can run, and therefore it will be likely to meet its deadline. In the case of our Dynamic Real-Time CORBA system, the writing clients miss fewer deadlines than the reading clients only at the highest system workload. This is because in this test, all clients started at the same time and had the same deadline. Thus, under earliest-deadline-first, each client ran at the same priority, and therefore the scheduling policy reverted to round-robin.

5 Conclusion

This paper has presented our Dynamic Real-Time CORBA system, which is based on the requirements specified by the OMG's Real-Time SIG for CORBA. The focus of our work has been on the expression and enforcement of timing constraints on end-to-end client/server interactions. Clients can express timing constraints through TDMIs and the system enforces the timing constraints through various extensions and additions to the CORBA standard. The Global Priority Service ensures that all CORBA requests are scheduled at all points in the distributed system according to the same (real-time) policy. The Real-Time Concurrency Control Service provides CORBA object-level locking with bounded priority inversion. The Real-Time Event Service enforces the distribution of real-time events in priority order with real-time enforcement of event response time. The Global Time Service provides a common notion of time across the system.

The performance results presented here demonstrate the overhead involved with several aspects of the implementation as well as the ability of our Dynamic Real-Time CORBA system to enforce expressed timing constraints. Because our prototype implementation includes two nodes, and has other system resource limitations, the results of the tests must be seen as proof of concept and not as definitive results of the design of our system. Full-scale development will be performed by companies to which we have provided our prototype, including: Iona Technologies, Tri-Pacific Inc, and Computing Devices International.

Our Dynamic Real-Time CORBA system provides the groundwork for a full dynamic Real-Time CORBA design. However, there is still work to be done. Modifying the priority assignment to perform other forms of real-time scheduling, such as a global rate-monotonic policy is interesting. We have designed the system to facilitate such a modification by replacement of one priority-calculation module in the Priority Service. Also, currently, our scheduling heuristic uses only importance and deadline information in the calculation of the global transient priorities. We are investigating which other Quality of Service (QoS) parameters can be factored in as scheduling parameters and how to use them to generate the transitive priorities. *Performance polymorphism*, another form of QoS enforcement where the ORB decides which method of a server to invoke based on specified QoS parameters and current system conditions, is another area of current interest. A look at our RT CORBA example of Section 3 indicates another area of RT CORBA research and development: the need for a Real-Time Name Service. Currently, the only part of our example that does not have real-time support is the CORBA Name Service `bind` call. A Real-Time bind that chooses servers based on real-time criteria is needed. Another area of current work is in Real-Time Concurrency Control Service, where we are incorporating explicit locking in addition to the implicit locking described in Section 3. We are also extending our Real-Time Concurrency Control Service to use our results in *object-based semantic real-time concurrency control* [DW93, STDW96, DW97], where method-level locks, whose compatibilities are semantically defined, are employed.

Tackling the substantial requirements posed by using CORBA in a real-time environment is a monumental undertaking, but necessary if standard, open, distributed computing environments are to be used in real-time applications. Work that has been done on porting CORBA products to real-time operating systems, and on using high-performance CORBA, is necessary for supporting some aspects of real-time, but neglects expression and enforcement of distributed end-to-end real-time constraints. The results presented in this paper are important steps towards achieving this goal.

References

[CLG90] William Corwin, C. Douglass Locke, and Karen Gordon. Overview of the IEEE POSIX p1003.4 real-time extension to POSIX. *IEEE Real-Time Systems Newsletter*, 6(1):9–18, Winter 1990.

[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.

[DW97] Lisa Cingiser DiPippo and Victor Fay Wolfe. Object-based semantic real-time concurrency control with bounded imprecision. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):135–147, Jan-Feb 1997.

[Ben96] E. Bensley et. al. Object-oriented approach for designing evolvable real-time command and control systems. In *WORDS '96*, February 1996.

[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.

[KSTW94] Peter Krupp, Alice Schafer, Bhavani Thuraisingham, and Victor Fay Wolfe. On real-time extensions to the common object request broker architecure. In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) '94 Workshop on Experiences with the Common Object Request Broker Architecture (CORBA)*, Sept. 1994.

[Moy] J. Moy. Request for comments - no. 1584; proteon, inc. Network Working Group.

[OMG96] OMG. *CORBAservices: Common Object Services Specification.* OMG, Inc., 1996.

[otO96] The Realtime Platform Special Interest Group of the OMG. CORBA/RT white paper. ftp site: ftp://ftp.osaf.org/whitepaper/Tempa4.doc, Dec 1996.

[Pal97] Robert Pallack. A study and development of real-time CORBA on ATM. Technical Report URI-TR-97-255, University of Rhode Island Dept. of Computer Science, May 1997. Masters' Thesis.

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

[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.

[Sys96] Chorus Systems. Chorus/cool-orb r3 product description. Technical Report CS/TR-95-157.3, June 1996.

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

[WDG+97] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, and Russell Johnston. Real-time CORBA. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, June 1997.

[Woh97] Steven C. Wohlever. Concurrency control in a dynamic real-time distributed object computing environment. Technical Report URI-TR-97-253, University of Rhode Island Dept. of Computer Science, May 1997. Masters' Thesis.

[Zyk97] Igor Zykh. Timed distributed method invocations in CORBA. Technical Report URI-TR-97-254, University of Rhode Island Dept. of Computer Science, May 1997. Masters' Thesis.