Chapter 1

Introduction

This thesis presents the necessary extensions to the CORBA standard and CORBA Services to support requirements of a real-time distributed system. It also describes a prototype implementation on which the tests were conducted, and analyzes the results.

1.1 Motivation

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) has developed the Common Object Request Broker Architecture (CORBA) as a standard software specification for such distributed environments. This standard specifies an Interface Definition Language (IDL) for the description of interfaces to the functional behavior of distributed components. The standard also specifies Object Services [2], which facilitate standard client-server interaction with a set of capabilities (i.e. Naming, Event, etc.), and an Object Request Broker (ORB), which is the middleware that allows for the seamless interaction between distributed client objects and server objects.

Many distributed real-time applications, such as automated factory control, avionics navigation and military target tracking, could benefit from a standard architecture like CORBA. 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. For example, the IDL describes the interface to the functional behavior of distributed components, but does not explicitly describe timing constraints for their behavior. Furthermore, system services provided by distributed environments offer little support for *end-to-end* real-time scheduling across the environment. In fact, some environments do not provide such basic services as synchronized clocks and bounded message latencies.

Recently, a Special Interest Group (SIG) has been formed within the OMG with the goal of examining the current CORBA standard and determining requirements for supporting real-time applications. Specifically, the real-time SIG is focusing on supporting the ability to express and enforce timing constraints by extending the current CORBA standard (CORBA/RT). The SIG has produced a white paper [1] that details the desired capabilities for a distributed object computing environment to support real-time. The real-time desired capabilities specified in the CORBA/RT SIG white paper are classified into three areas: desired capabilities for the operating environment; desired capabilities for the ORB architecture; desired capabilities for the object services and facilities.

This thesis presents object services desired capabilities that provide and support expressing of timing constraints. These desired capabilities are: expressing of timing

constraints and handling their violations on CORBA method invocations, Global Time Service and clock synchronization, and Real-Time Event Service.

1.2 Goal of Research

The goal of this research is to develop CORBA/RT desired capabilities involving object services and features for handling real-time client-server interaction. In Real-Time CORBA a client must have some way of expressing timing constraints on its request; and CORBA must provide object services that support enforcement of the expressed timing constraints. This involves designing and implementing a model of *Timed Distributed Method Invocations* (TDMIs), and designing and remodeling some of the CORBA Services to meet real-time requirements.

1.3 Approach Used

In order to achieve the goal, the current CORBA standard was examined and evaluated. Our research group has submitted the results to the OMG and proposed a model of TDMI along with the extensions to the CORBA Services to support that model. We are pleased to announce that most of our changes were accepted and incorporated into the SIG's white paper [1].

The goal of this research is to extend the current CORBA standard with the object Services to meet real-time requirements and desired capabilities without, actually, changing the standard itself.

1.4 Outline

Chapter 2 contains a review of real-time systems and their requirements, the CORBA standard and one of its implementations. Also, this chapter gives an overview of different approaches to build Real-Time CORBA. Chapter 3 describes the model of TDMIs, and the design of a CORBA Global Time Service with our approach for clock synchronization. This chapter also presents an overview of the CORBA Event Service and describes the Real-Time Event Service requirements. Chapter 4 describes the prototype implementation that was used to evaluate the model of TDMIs, Global Time and Real-Time Event Services. Chapter 5 presents and analyzes the results of the performance tests using simulated workloads. Chapter 6 explains the contributions and limitations of this thesis, and discusses future work.

Chapter 2

Related Work

This section provides background information on real-time systems and their requirements, the current CORBA standard and its implementation, and it summarizes known efforts to build Real-Time CORBA. It presents also the CORBA/RT SIG's desired capabilities involving Global Time and Real-Time Event Services as well as features for handling real-time client-server interactions.

2.1 CORBA standard and its Implementation

The basic notion behind CORBA is to provide a uniform way for any object to receive and respond to a request from any requester (client), either another object or even a traditional nonobject-oriented program. Once such a request is made, the ORB makes sure that the request is delivered to an appropriate receiving object, no matter where it is and how it is implemented. To provide all these capabilities the CORBA specification defines an architecture of interfaces that may be implemented in different ways by different vendors. The architecture was specifically designed to separate the concerns of interfaces and implementations (Figure 1). The main components of the architecture may be divided into three specific groups: client side, implementation (server) side, and the ORB core. The client and server sides represent interfaces to the ORB via the IDL.

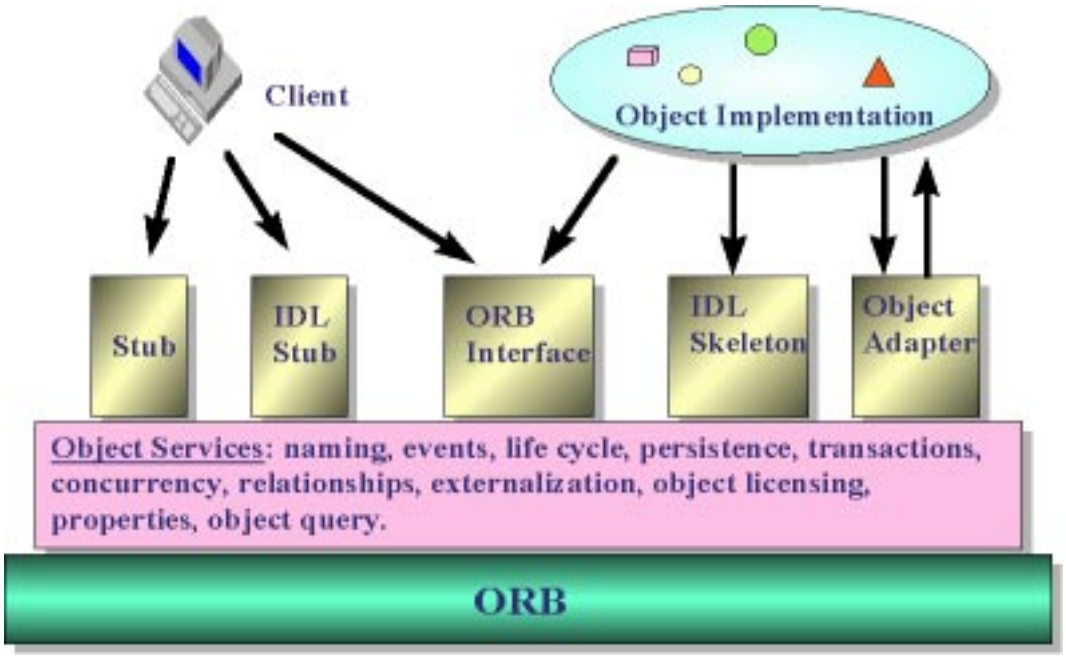


Figure 1: Object Request Broker Technology

As mentioned previously, the purpose of the IDL is to allow the language-independent expression of interfaces, including the complete signatures (name, parameters and their types, etc.) of methods or functions, and accessible attributes. An interesting aspect of the IDL is exceptions. Exception declarations define a struct-like data structure with attributes that can be used to pass information about an exception condition to a service requester. An exception is declared with its name, which is accessible as a value when the exception is raised, allowing the client to determine which exception has been received

The client-side architecture consists of three components:

- the Dynamic Invocation interface (Stub) - a mechanism for specifying request at runtime;

- the IDL stub interface - small piece of machine-language code, which is generated according to IDL interface definitions;
- the ORB service interface - a number of functions that may be accessed directly by the client code (e.g. retrieving a reference to an object).

One aspect of the client-side interface is shared by object implementations: the ORB services. The other two components on the implementation side are:

- the IDL skeleton interface - is an up-call interface through which the ORB calls the method skeletons of the implementation, on a request by a client;
- the Object Adapter - is the means by which server implementations access most of the services provided by the ORB (e.g. generation and interpretation of object reference).

The CORBA specification is not explicit about what services an adapter must support, but it is clear that the adapter is intended to isolate object implementations from the ORB core as much as possible.

Objects in CORBA are created and destroyed dynamically in response to the issuance of requests. Objects can also participate in any of the normal types of relationships, with perhaps the most important being subtype/supertype relationships or inheritance. Inheritance between object interfaces is specified syntactically by using the OMG's IDL . The object model in CORBA is strongly typed. As in C++, types are used to restrict and characterize operations. Unlike languages such as Smalltalk, these types are not first-order objects, and cannot be manipulated as objects.

The major part of the CORBA standard is Object Services - some software designed to provide a particular set of operations applicable to broad classes of objects. For example, a given object service might store and retrieve objects, or it might manage

relationships among objects, or it might protect objects from unauthorized access by other objects, etc. The Object Services are the key to expanding the functionality and interoperability of objects beyond the simple request management capabilities of the ORB.

However, for the OMG's CORBA to be truly usable as an industry standard for a wide range of commercial applications, much more work needs to be done, and the OMG formed different SIGs for that purpose. There are a number of implementations of CORBA available now on the market, one of which is ORBIX from IONA Technologies [15].

ORBIX represents the distillation of ten years research in the area of distributed systems. Because the software was built from scratch, it conforms faithfully to the OMG's CORBA specification. It does not contain vestiges of an old product trying to comply with the CORBA standard, and its architecture presents CORBA in a most natural way to C++ developers. The product has been tried, tested and deployed by corporations across the globe. It is used by leading software providers for the banking, telecommunications, engineering and government sectors. With ORBIX, programmers can develop distributed applications using object-oriented client-server technology, and use object technology to compose new applications from existing components and subsystems.

2.2 Real-Time Systems

In a real-time system, timing constraints must be met for the application to be correct. This requirement typically comes from the system interacting with the physical environment. The environment produces stimuli, which must be accepted by the real-time system within timing constraints. The environment further requires control output, which must be produced within timing constraints.

One of the main misconceptions about real-time computing is that it is equivalent to fast computing. Sometimes researchers challenge this myth by arguing that computing speed is often measured in average case performance, whereas to guarantee timing behavior, in many real-time systems worst case performance should be used. That is, in a delicate application, such as a nuclear reactor or avionics control, where timing constraints **must** be met, worst case performance must be used when designing and analyzing the system. Thus, although speed is often a necessary component of a real-time system, it is often not sufficient. Instead, predictably meeting timing constraints is sufficient in real-time system design.

Real-time System Requirements. Real-time systems require that timing constraints be expressed, enforced, and their violations handled. The unit of time-constrained execution is called a *task*. For example, in a real-time database, time-constrained transactions are considered tasks. Timing constraint expression can take the form of start times, deadlines, and periods for tasks. Timing constraint enforcement requires predictable bounds on task behavior. The handling of timing constraint violations depends on the tasks requirements: whether they are *hard*, *firm* or *soft* real-time. A task with a hard real-time constraint has disastrous consequences if its constraint is violated. Many constraints in life-critical systems, such as nuclear reactor control and military vehicle control, are hard real-time

constraints. A task with a firm real-time constraint has no value to the system if its constraint is violated. Many financial applications have firm constraints with no value if a deadline is missed. A task with a soft real-time constraint has decreasing, but usually non-negative, value to the system if its constraint is violated. For most applications, most tasks have soft real-time constraints. Graphic display updates are one of many examples of tasks with soft real-time constraints.

Expressing Timing Constraints. Most real-time systems specify a subset of the following constraints:

- An *earliest start time* constraint specifies an absolute time before which the task may not start. That is, the task must wait for the specified time before it may start.
- A *latest start time* constraint specifies an absolute time before which the task must start. That is, if the task has not started by the specified time, an error has occurred. Latest start times are useful to detect potential violations of planned schedules or eventual deadline violations before they actually occur.
- A *deadline* specifies an absolute time before which the task must complete.

Frequently, timing constraints will appear as *periodic execution constraints*. A periodic constraint specifies earliest start times and deadlines at regular time intervals for repeated instances of a task.

This thesis describes the design and implementation of expressing and supporting of soft real-time constraints in a CORBA-compliant distributed computing environment.

2.3 Approaches

There have been related efforts to use CORBA in real-time environments. One approach has been to produce a CORBA specification or product implementation that supports faster performance. For instance, several U.S. military systems from Lockheed/Martin and Boeing use high-performance CORBA implementations for real-time applications. Another approach taken by some vendors is to port their CORBA implementations to the real-time operating systems (OS). This section gives overviews of these two main baselines and points out some of their weaknesses.

2.3.1 "Fast" CORBA

Research work at Washington University [7], [8] extends the ORB architecture to provide forms of Quality of Service (QoS) guarantees by reducing performance overhead during CORBA method invocations. This approach uses the Real-Time Object Adapter, which is responsible for real-time scheduling and dispatching of the ORB operations, and provides multiplexing-demultiplexing optimizations at different levels over an Asynchronous Transfer Mode (ATM) network using a lightweight transport mechanism. This research group strongly required that the underlying operating system and network provide resource-scheduling mechanisms to support real-time guarantees. For instance, the operating system must support scheduling mechanisms that allow the highest priority task to run to completion. Furthermore, real-time tasks should be given precedence at the network level to prevent them from being blocked by low priority applications.

The same research group proposed the architecture to enhance the real-time capability of the CORBA Event Service [8], [9]. These extensions introduce several components augmenting the Event Service to support events scheduling and minimize dispatch latency, based on a priori knowledge of participated *consumer(s)/supplier(s)* and

periodic rate-based events. The interfaces of the designed Real-Time (RT) Event Service include QoS parameters that allow consumers and suppliers to specify their execution requirements and characteristics. These parameters are used by the event dispatching mechanism to integrate with the system-wide real-time scheduling policies to determine dispatching ordering and preemption strategies.

In a real-time system, some consumers can execute whenever an event arrives from any supplier. Other consumers can execute only when an event arrives from a specific supplier, or when multiple events have arrives from a particular set of suppliers. The RT Event Service provides filtering and correlation mechanisms that allow consumers to specify logical OR and AND event dependencies. When those dependencies are met, the RT Event Service dispatches all events that satisfy the consumers' dependencies. In some real-time systems, consumers may require periodically the same amount of execution time. The designed RT Event Service allows consumers to specify event dependency timeouts and will propagate temporal events in coordination with system scheduling policies.

Unfortunately, the proposed extensions do not provide any ability to specify priority/importance of the real-time events and do not enforce distribution of those events in priority order. Also, the architecture does not support the mechanism for expressing and enforcing event-driven deadlines on CORBA method invocations.

2.3.2 CORBA on a Real-Time Operating System

Another approach, taken by ORB vendors like Iona and Chorus Systems [16], is to port their non-real-time CORBA implementation to real-time operating systems (i.e.

Chorus/ClassiX, Solaris [17], Lynx [18], VxWorks [19]). However, the products that we have investigated make only limited use of the operating system's real-time features (i.e. real-time priorities, scheduling policies, etc.) and do not support any expression and enforcement of timing constraints, scheduling parameters, and QoS requirements. For instance, the COOL ORB from Chorus Systems doesn't provide any real-time features of its own. The prime vertues of the COOL ORB is to impose minimal overhead (time and memory size) compared to using the native operating system. Besides from minimal overhead in terms of time and memory, COOL ORB provides thread safe libraries, using fine grained internal locking. When running on CHORUS/ClassiX, the full Application Programming Interface (API) of CHORUA/ClassiX is available to COOL ORB applications to precisely control real-time aspects.

Both the "fast CORBA" and "CORBA on a real-time operating system" approaches, although often necessary parts of the solution, are not sufficient for supporting timing constraints. Support for the expression of and system-wide enforcement of timing constraints is also necessary.

Chapter 3

Real-Time CORBA

As mentioned before, the desired real-time capabilities stated in the CORBA/RT white paper have been classified into three areas. The desired operating environment features include priority-based scheduling, multi-threading, and synchronized clocks. My prototype implementation was developed on a real-time POSIX compliant operating system that satisfies most of the CORBA/RT operating environment desired capabilities. The CORBA/RT desired capabilities that involve the architecture of the ORB include specification for modularity of the ORB as well as for a generic scheduling framework. This chapter will focus on CORBA/RT capabilities that are essential to support the

expression of timing constraints and scheduling parameters. This includes the model of Timed Distributed Method Invocations, Global Time, and Real-Time Event Services.

3.1 The Need for Real-Time Extensions

Before detailing the CORBA/RT desired capabilities that have been addressed, I will illustrate the need for them by describing a typical client-server interaction in CORBA and by highlighting the changes that are necessary for the interaction to support real-time.

Non-Real-Time Example. In CORBA, the ORB is responsible for managing the communication between clients and servers. The CORBA specifications also provide for object services that are necessary to facilitate the processing of the requests. To illustrate a client-server interaction, I introduce an example to which I will refer throughout this thesis. Assume that a table containing tracking data exists on one node and is represented as a CORBA server. A client on another node wishes to retrieve data from the table using the server's *Get* operation. The following sequence of steps illustrates what will occur in such a client-server interaction:

1. The client initiates a bind to a CORBA server.
2. The ORB finds the server that will handle the request and binds the client to the server. Binding involves returning an object reference for the server.
3. The client invokes a Get method on the server, and the ORB delivers the request to the server.
4. The request is scheduled on the server's node using whatever scheduling policy is specified for that node.

5. The Concurrency Control Service ensures that the client's access to the server is consistent with access by other clients. It provides read/write locking of the entire server object.
6. When the server completes the *Get* operation requested by the client, the server sends a response back to the client through the ORB.

Real-Time Example. The same client-server interaction in a real-time scenario involves a client that wishes to access data from the tracking table server within timing constraints. This interaction has two main differences from the interaction above: the client must have some way of expressing timing constraints on its request; and CORBA must provide object services that support enforcement of the expressed timing constraints (Figure 2). For instance, suppose the client requires that the *Get* operation on the server containing the tracking table be performed within a specified amount of time. Then the interaction would be as follows:

1. The client initiates a bind to a CORBA server.
2. The ORB finds the server that will handle the request and binds the client to the server.
3. The client specifies its constraints, such as timing constraints and relative importance, for a *Get* method invocation. It then invokes the Get method. Timing constraints may refer to events that have occurred or will occur in the future. The client then sends the request to the ORB for service by the tracking table server.
4. A Global Priority Service examines the request and provides a global priority to the request so that it can be scheduled relative to all other real-time clients and servers.

5. The ORB receives the request and finds the server to handle the request.
6. A Real-Time Concurrency Control Service provides locking with bounded priority inversion so that the client's access to the server is consistent.
7. When the server completes the *Get* operation, it sends the response back to the client through the ORB. If the response is not received within timing constraints, an exception is raised for the client.

This sequence of steps highlights some of the extensions that are necessary for a real-time client/server interaction. In a distributed computing environment all server executions are initiated by method invocations made by clients. In real-time applications, a client must be able to specify timing constraints on method invocations.

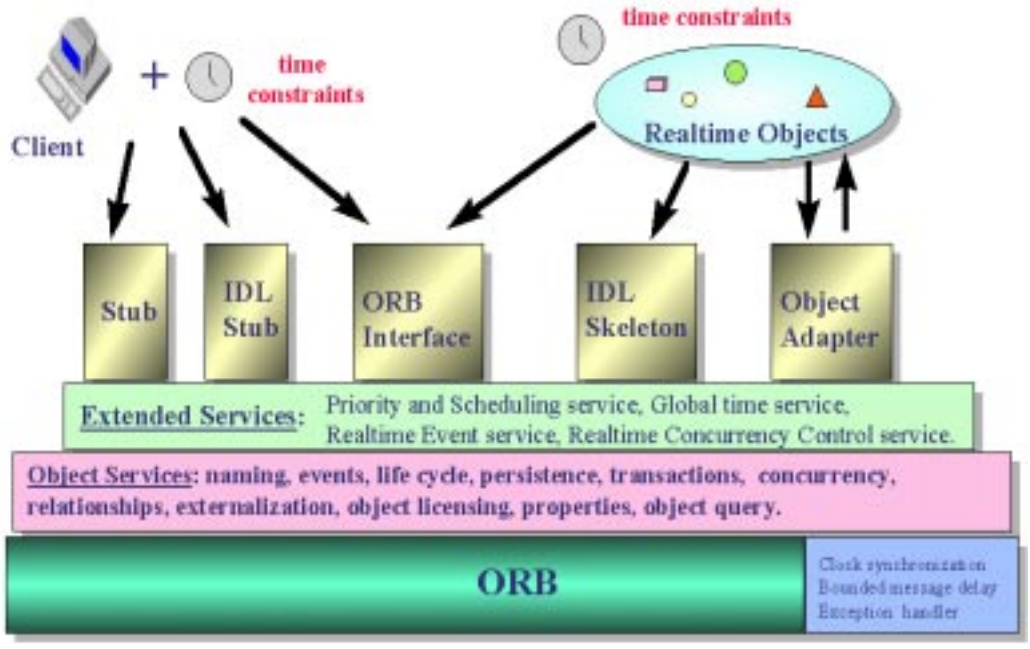


Figure 2: Real-Time CORBA

In the next section, I will describe the design of the model of TDMIs that provides such capabilities.

3.2 Timed Distributed Method Invocation

The CORBA/RT white paper states that in a real-time system timing constraints imposed on an execution of a task must be met for the task to be correct. In order to provide support for full expression of timing constraints on method invocations, it is necessary to design a model of Timed Distributed Method Invocations. The CORBA/RT white paper established five forms of client-side timing constraints: deadlines, earliest start times, latest start times, periodic, and execution time constraints for method invocations. The introduction of timing constraints adds another dimension to real-time computing: it may need to yield a lower QoS. For instance, in systems where timely but less accurate results are better than late exact results, some imprecision may be tolerated.

In order to implement timing constraints on the execution of a TDMI, it is necessary to take into account the delivery time of the request and the reply. Also, each method call has to carry its constraint information indicating its deadline, importance, QoS parameters, etc. This information will be used by the clients, ORB(s), CORBA Object Services, underlying OS, and servers' implementations to set priorities, alarms, and whatever else is needed to enforce the specified requirements and to interact based on time. To make this timing information meaningful across nodes, all clocks in the system must be synchronized to within a bounded skew of each other. This should be provided by a Global Time Service. In some real-time systems a client may want to start a method

invocation only after a particular event has occurred. Thus, a CORBA client should be able to specify a “within 10 seconds of completion of Task A” deadline. The client needs the time that the named event “completion of Task A” has occurred. This service should be provided by a Real-Time Event Service. I will elaborate on each of these services in the next sections.

3.2.1 The design of the Model

In order to provide for full expression of timing constraints on method invocations, a model of Timed Distributed Method Invocations has been designed and implemented. In this model, all timing information is packaged in a structure called a *Real-Time Environment* (*RT_Environment*), as shown in the TDMI of Figure 3.

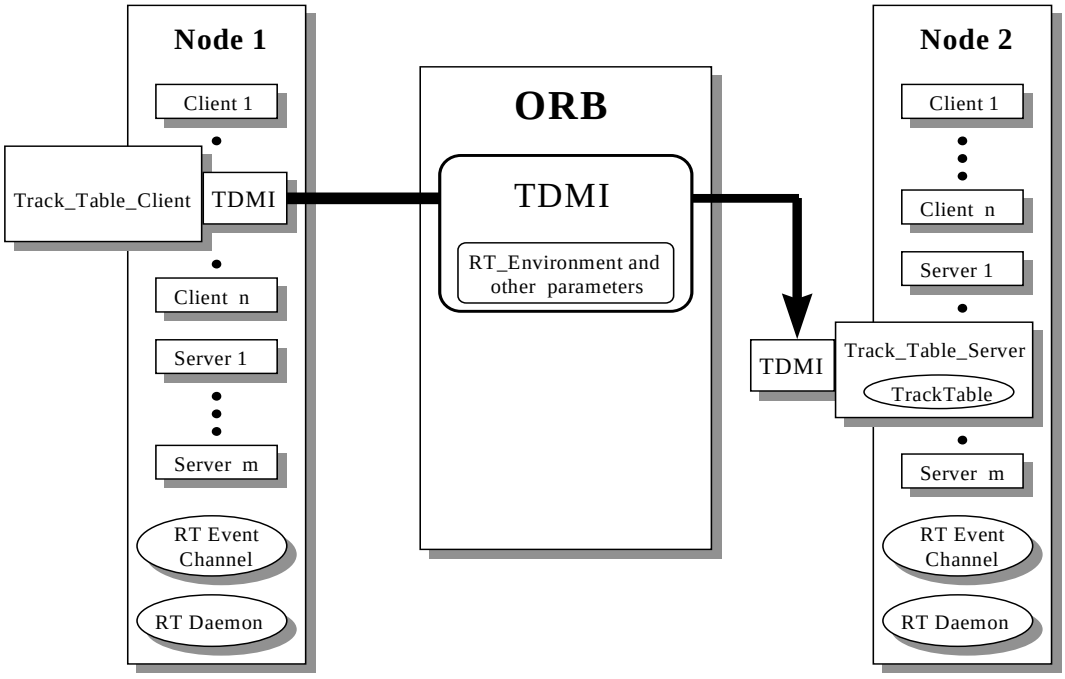


Figure 3: Implementation of a Timed Distributed Method Invocation

Following the model, a CORBA client will start at a base priority that is established from static timing and importance information available when the client is dispatched. The client runs at this base priority whenever it is not executing a TDMI. To configure a TDMI, the client specifies its timing constraint parameters, its QoS parameter, and its scheduling parameters, such as deadline, and importance, in the Real-Time Environment. This environment is attached to every method call and is used by the *Global Priority Service*, the ORB, and the *Real-Time Concurrency Control Service* to enforce the specified timing constraints.

3.3 Global Time Service

Currently, the OMG is working on the Object Time Service specifications, which should be released in the near future. The main requirements stated in the Request for Proposal (RFP) [13] are very close to ones that have been described above, but implementation issues are not addressed at all. This section will give the detailed description of the designed service and will present our approach for clock synchronization mechanism.

3.3.1 Description of the Service

As mentioned previously, the Global Time Service is required to support Timed Distributed Method Invocations. This service must ensure that all clocks in the distributed system are synchronized to within a known skew of each other to provide a consistent notion of time. While this service is not a desired capability specified in the CORBA/RT white paper, I believe that it is necessary. Clients and servers must be able to

call this service to get the “current global time”. Also, in some embedded systems the Global Time Service must allow the ability to specify delays, time-outs, and deadlines in terms of the absolute and the relative time. For instance, a client should be able specify a deadline such as *within 10 seconds of the current global time* or *by April 24, 1997, at 5:00 AM*.

3.3.2 Clock Synchronization

While clock synchronization is not a desired capability specified in the CORBA/RT white paper, I believe that it is also necessary in a real-time system. Although in many systems the clock synchronization function has not been decoupled from the applications (e.g. the distributed versions of the applications synchronize by messages), research and experience have led us to believe that solving the synchronization problem independently from the applications design can provide significant simplification of the system.

A generalized view of the algorithm employed by each clock looks something like:

```
do forever {
       exchange clock values
       determine adjustment for this interval
       determine local time to apply correction
       when this time arrives, apply correction
}
```

The general algorithm is parameterized by a convergence function that determines both the magnitude and time of adjustment.

3.3.2.1 Network Time Protocol (NTP)

One approach to clock synchronization is the Network Time Protocol (NTP) [5]. The NTP specified in the Request For Comments (RFC)-1305 [6] can be used to synchronize

computer clocks in the global Internet. It provides comprehensive mechanisms to access national time and frequency dissemination services, organize the time-synchronization subnet, and adjust the local clock in each participating subnet peer. In most places of today's Internet, NTP provides accuracy of 1-50 ms, depending on the characteristics of the synchronization source and network paths [7].

The RFC-1305 [6] specifies the NTP protocol machine in terms of events, states, transition functions and actions and, in addition, optional algorithms to improve the timekeeping quality and mitigate among several, possibly faulty, synchronization sources. To achieve accuracy in the low milliseconds over paths spanning major portions of the Internet, these intricate algorithms, or their functional equivalents, are necessary.

NTP is designed for use by clients and servers with a wide range of capabilities and over a wide range of network delays and jitter characteristics. Most users of the Internet NTP synchronization subnet use a software package including the full suite of NTP options and algorithms, which are relatively complex, real-time applications. Typical NTP configurations utilize multiple redundant servers and diverse network paths, in order to achieve high accuracy and reliability. Some configurations can include cryptographic authentication to prevent accidental or malicious protocol attacks. The NTP software has been tested and ported successfully to a wide variety of hardware platforms ranging from supercomputers to personal computers.

3.4 Event Service

A standard CORBA request is a synchronous execution of an operation by an object. If the operation defines parameters or return values, data is communicated between the client and the server. In some systems, a decoupled communication model between objects may be required. For instance, in response to some events, a diverse set of participating objects may want to respond in different ways. The most effective way of supporting this is to implement a mechanism by which the interested parties can be informed that the event has occurred.

3.4.1 CORBA Event Service

The current CORBA Event Service specifications define two roles for objects (the *supplier* and the *consumer*) and propose the *Event Channel* architecture for decoupling the communication between those objects. There are two approaches for initiating event communication: the *push* model and the *pull* model. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. An Event Channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. Event data are communicated between suppliers and consumers via Event Channels by issuing standard CORBA requests.

3.4.2 Real-Time Event Service Requirements

Applications with significant real-time performance requirements can benefit from using real-time asynchronous and group communications. Unfortunately, the CORBA Event Service specifications do not address issues important for real-time applications, such as priorities/importance of events, scheduling, dispatch latency, and bounded event response

time [12]. Furthermore, the Event Service must provide the ability for CORBA clients and servers to determine the absolute time when the event occurred. Some events may have different priorities on the system (i.e. System level and User defined events), while events within the same priority may have different levels of importance. The Event Service must support the event notification mechanism with regard to the priority and importance of the events occurring on the system. The Event Service architecture should be extended to introduce Event Channels with dynamically configurable policies for supporting QoS requirements. For instance, to meet QoS, scheduling policies often require that priorities must be assigned to Event Channel operations (i.e. push, pull). Event Channels must support high-performance real-time concurrency that will provide "worst-case" guarantees on dispatching latency and event response time. Thus, it is necessary to design a Real-Time Event Channel architecture that supports processing of real-time events in priority order by explicitly representing and enforcing priority/importance information associated with *real-time events*.

3.4.2.1 IP multicasting transport protocol (MTP)

Multicast is a term for describing multiple hosts communicating with each other within functional groups over LAN/WAN. These multicast groups can be used for teleconferencing, or by other multiple peer applications.

One way of designing the underlying communications of the Real-Time Event Service is to use IP Multicasting Transport Protocol (MTP) mechanism. The large body of the networking and distributed systems literature and technology that has been introduced during the last decade has influenced the MTP design. MTP provides reliable

delivery of client data between one or more communicating processes, as well as a predefined principal process. In addition to transporting data reliably and efficiently, MTP over IP provides the synchronization necessary for web members to agree on the order of receipt of all messages and can agree on the delivery of the message even in the face of partitions.

Our design and implementation of the Real-Time Event Service takes advantage of the UDP based IP Multicast [8] on TCP/IP to share information between applications. Each real-time event will have a unique event ID number, which will be mapped to the corresponding IP multicast group address. An IP multicast group (Class D Internet address) consists of a 32-bit number. The high order 4 bits are 1110, which identify the Internet address as a IP multicast group address. The remaining 28 bits contain the multicast group ID. Thus, IP multicast groups are in the range 224.0.0.0 to 239.255.255.255. CORBA clients/servers would join and leave different groups and can communicate by talking or listening via Real-Time Event Channels.

Chapter 4

Implementation

The Real-Time CORBA prototype was designed and implemented as a part of the Distributed Hybrid Database Architecture project developed by the U.S. Navy NRaD, and our research group at University of Rhode Island.

This thesis was concerned with the implementation of TDMIs that support expression of timing constraints, scheduling parameters and QoS requirements on database access in CORBA environments. Also, this thesis was concerned with the design and implementation of Global Time and Real-Time Event Services that are essential for a real-time distributed system.

The prototype implementation is being developed on two Sun Sparc work stations running Sun's Solaris 2.5 operating system. This operating system conforms to IEEE Std 1003.1-1990 (System API as amended by IEEE Std 1003.1b-1993 Realtime Extension), IEEE Std 1003.1c-1995 (Threads Extension) and IEEE Std 1003.1i (Technical Corrigenda to Realime Extension) that allow to write portable multi-threaded applications.

4.1 Timed Distributed Method Invocations Implementation

Following the rules of object-oriented design [14], the implementation of the semantics of a TDMI is encapsulated in a base C++ class called *Base_RT_Manager.* This class defines the set of *virtual* and *pure virtual* functions that will be overwritten by *Real-Time Mangers* on the client side and the server side. Thus, the actual implementation of those functions, along with some specifics of a client and a server internals, are defined in the corresponding C++ subclasses that inherit from the same base class: *RT_Manager_Server* and *RT_Manager_Client*. I will now elaborate on each of these implementations.

4.1.1 Server Side Implementation

As mentioned above, a server side implementation is defined in the RT_Manager_Server C++ class, which provides the following methods:

```
static void RT_Manager_Init();
void START_RT();
void END_RT();
void STOP();
```

The first method will *bind* a CORBA server process to the *Real-Time Daemon* (real-time scheduler) on its local node. The next three methods will be executed by the threads (can be thought of as a light weight process (LWP)) created by the server to handle clients requests, as shown in Figure 4. The method `START_RT()` will register with a Real-Time Daemon on its local node to calculate and assign a global priority called *Transient Priority* to the TDMI. Also, a timer with the proper signal handling will be armed according to the client's deadline (minus network delay). The `END_RT()` method disarms the alarm, and communicates with the Real-Time Daemon to change the TDMI's

Transient Priority to its base priority. If the server misses its deadline, a CORBA exception of type *RT_Exception* will be thrown from a signal handler to the calling

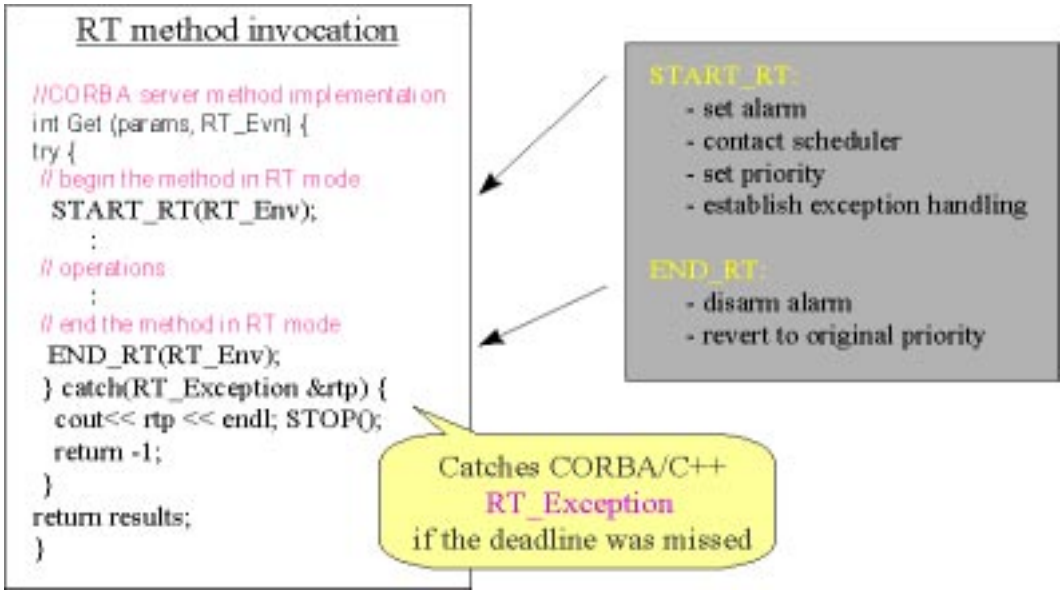


Figure 4: TDMIs implementation: server side

thread. That thread can catch the exception and will execute the `STOP()` method to de-register with the Real-Time Daemon and release the resources, for example, read/write locks, etc.

The current implementation of RT_Manager_Server C++ class was based on:

- POSIX high resolution clocks to set up a timer for each thread (LWP) of execution;
- POSIX Real-Time signals to deliver a notification to the LWP upon the timer expiration. These signals are processed and delivered by the operating system first, in priority order, and they can carry extra information, like an integer value or a pointer;

- The C++ exception mechanism used to notify the thread that it missed its deadline. C++ exception handling is a new feature that has been recently added to the C++ environment. The idea of using this exception handling is to provide a standard (ANSI C++) method of dealing with exceptional conditions on a higher level. The C++ Sparc compiler and ORBIX provides compile and run-time support for that mechanism.

4.1.2 Client Side Implementation

A client side implementation is defined in the RT_Manager_Client C++ class. This class inherits and implements the methods of its base class providing the same functionality as the RT_Manger_Server class. In addition, it defines the following extra methods:

```
Set_Time_Constraint_Now(Constraint_Type ,Time_Type , long sec, long nsec);
Set_Time_Constraint_Event(Constraint_Type,RT_Event& , long sec, long nsec);
Set_Importance(unsigned short );
Set_QoS(unsigned short );
Start_RT_Invocation();
End_RT_Invocation();
```

The first four methods in the RT_Manager_Client are used to set parameters in the *RT_Environment* (e.g., relative/absolute deadline or event driven timing constraints, importance level, and QoS). The method `Start_RT_Invocation()`will be executed by the client's main() function to initiate processing of the CORBA request in the form of a TDMI. This method will register the client with a Real-Time Daemon on its local node, and will calculate and assign a global priority called *Transient Priority* to the TDMI. Also, a timer with a signal handling function will be armed according to the specified deadline. `End_RT_Invocation()` communicates with the Real-Time Daemon to change the TDMI's Transient Priority to its base priority. If the client misses its deadline, a CORBA exception of type *RT_Exception* will be thrown from a signal handler to the

client process. The client can catch the exception and will execute the `STOP()` method to de-register with the Real-Time Daemon and cleanup the internal structures and resources.

The methods of the RT_Manager_Client are responsible for assembling the *RT_Environment* that will be attached to the TDMI for use by the ORB, the object services, and the server implementation. The RT_Environment data structure has the following form:

```
struct RT_Environment {
       TDMI_info tdmi;
       ClientID cid;
       unsigned short qos;
       unsigned short importance;
       long tpriority;
};
```

The *TDMI_info* field stores the timing constraint parameters. Additional information used for scheduling is also included in the structure: qos for Quality of Service, importance, and tpriority for the Transient Priority. The *ClientID* field stores the client's process ID, the client's thread ID and the IP address of the client's node. The RT_Environment, along with a new type of CORBA exception (RT_Exception) used for processing the TDMI in real-time, are contained in a header IDL file called *rt_info.idl*.

In Figure 3 (Page 20) the client object on Node 1 has specified a TDMI to be sent to the real-time tracking table server object on Node 2. All of the timing information is packaged in the RT_Environment structure that is being sent through the ORB.

4.2 Global Time Service Implementation

I chose to implement a Global Time Service in a "quick-and-dirty" way with the assumption that all clocks in the system are synchronized within a known bounded skew. Following this design, a client or a server can determine the current time by referring to

its own local clock without making expensive CORBA calls. For clock synchronization I used the *xntpd3.5* software package developed at the University of Delaware. This software distribution contains a fully compliant implementation of the NTP Version 3 protocol, including an autonomous protocol daemon that disciplines the local host clock, as well as a set of supporting utility programs used to debug and manage one or more NTP servers in a network.

The idea behind xntpd is to effectively synchronize the time of a computer client or server to another server or reference time source, such as a radio or satellite receiver or modem. It provides accuracy typically within a millisecond on LANs and up to a few tens of milliseconds on WANs relative to a primary server.

For our prototype, I chose to configure the xntpd software to work in a broadcast (point to multipoint) mode. In this mode a broadcast server periodically sends a message to a designated IP broadcast address or IP multicast group address, and ordinarily expects no requests from clients. A broadcast client listens on this address and ordinarily sends no requests to servers. For this purpose, an IP broadcast address has its scope limited to a single IP subnet, since routers do not propagate IP broadcast datagrams. Some broadcast servers may elect to respond to client requests as well as send unsolicited broadcast messages, while some broadcast clients may elect to send requests only in order to determine the network propagation delay between the server and client.

The standard NTP *timestamp* format is described in RFC-1305 [6]. In conformance with standard Internet practice, NTP data are specified as an integer or fixed-point quantities, with bits numbered from 0 starting at the left, or high-order, position. Unless specified otherwise, all quantities are unsigned and may occupy the full

field width with an implied 0 preceding bit 0. Since NTP timestamps are cherished data and, in fact, represent the main product of the protocol, a special timestamp format has been established. NTP timestamps are represented as a 64-bit unsigned fixed-point number, in seconds relative to 0h on 1 January 1900. The integer part is in the first 32 bits and the fraction part in the last 32 bits. In the fraction part, the non-significant low-order bits should be set to 0. This format allows convenient multiple-precision arithmetic and conversion to UDP/TIME representation (seconds), but does complicate the conversion to Internet Control Message Protocol (ICMP) Timestamp message representation (milliseconds). The precision of this representation is about 200 picoseconds, which should be adequate for even the most exotic requirements. More specific considerations on the NTP are beyond the scope of this thesis.

As mentioned above, currently OMG is working on specifications for an Object Time Service for CORBA. In the future, if such a service is desirable in a particular real-time application, our Global Time Service an easily be replaced with the implementation of the native CORBA's Object Time Service.

4.3 Real-Time Event Service Implementation

A *real-time event* is a CORBA event, generated by a supplier (or by the system), that includes a timestamp, representing the absolute time at which the event occurred, along with information, such as its source, priority, and importance that are used to regulate distribution of the real-time events in the system. This information is critical for processing real-time events efficiently. For instance, an event associated with a CORBA

server failure is likely to be more important than notification of e-mail delivery. Thus, the priority of that event should be higher and its event data should be propagated first. In my design, all real-time events are delivered through a *Real-Time Event Channels* (*RT EventChannels*) that substitutes CORBA Event Channels by explicitly representing and enforcing timing information associated with events. Figure 5 depicts the functionality of the designed RT Event Channel. To achieve better performance in the Real-Time Event Service, my implementation takes advantage of a multithreaded environment provided by the operating system and its support of the IP multicasting API.

The event data associated with a real-time event carries the necessary information (event name, its id and its source: host name, server name) to uniquely identify the real-time event, and its timestamp information. It also contains the priority and importance associated with real-time events. Consumers, suppliers and Real-Time Event Channels use this information to filter, correlate and propagate the real-time events.

Recall that there are two models for the distribution of events: the push model and the pull model. The push model allows a supplier of events to initiate the transfer of the event data to consumers. The pull model allows a consumer of events to request the event data from a supplier. The design and implementation of those models in our prototype for Real-Time CORBA with TDMIs requires additional considerations.

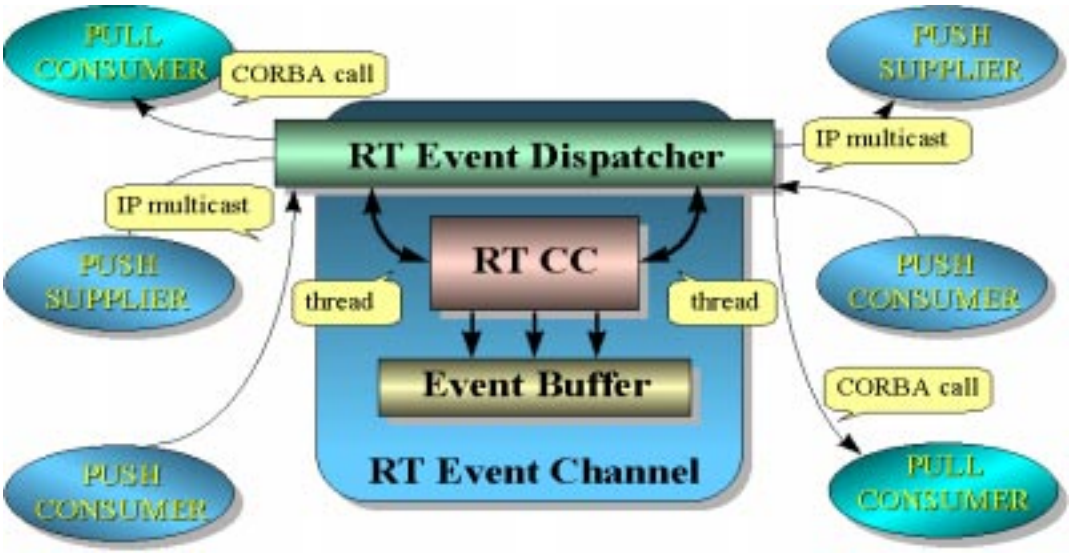


Figure 5: Real-Time Event Service

Our design and implementation of TDMIs in Real-Time CORBA supports a *dynamic scheduling* policy (currently *Earliest Deadline First* [20] with aging [21]). Each CORBA call has to be scheduled on its local node by a Real-Time Daemon based on its scheduling policy. In order to meet real-time guarantees in our system we can only allow method invocations to be made in the form of a TDMI. This imposes strong restrictions and makes it almost inappropriate to implement the Real-Time Event Service using the standard CORBA paradigm (interfaces, invocations, etc.).

4.3.1 Push model Implementation

In the push model, suppliers put a timestamp and “push” events to all interested push consumers via the RT Event Channel(s). The Push suppliers also deposit their events in the RT Event Channel(s) in order to keep a record of events that have occurred. The Push consumers are notified by the RT Event Channel(s) when a real-time event has been

delivered and stored. The underlying communication mechanism in our implementation of a push model is based on IP multicasting. The push suppliers “multicast” the real-time event data to the specified RT Event Channel(s). The RT Event Channel accepts and creates threads to dispatch all incoming real-time events. The *Real-Time Event Dispatcher* gets the priority/importance of the event, calculates and assigns a real-time OS priority to the corresponding thread (Figure 5). Next, the thread gets access to the RT Event Channel’s *Event Buffer* to store the event data. In order to provide the “best effort” on bounded event response time, our RT Event Channel makes use of Real-Time Concurrency Control. This Concurrency Control manages access to the event buffer using exclusive locking semantics with bounded priority inversion. It ensures that priority inversion is bounded and provides a best effort on minimizing event response time. The filtering of the real-time events will take place if the Event Buffer has stored the same event data already. In this case, the timestamp and the source information are updated, the event counter is incremented, and the renewed event data is “multicasted” to the interested push consumers. If the Event Buffer did not have that event data yet, the data will be stored and then “multicasted”. Each real-time event is uniquely mapped to the corresponding IP address of a multicast group based on its *event_ID* number. The push consumers will join the desired multicast group and listen for delivering of the real-time event from the RT Event Channel(s). Upon delivery of the event, the push consumers will perform type checking to insure validity of the real-time event. In the case of success, the consumers will leave the multicast groups and continue their execution.

4.3.2 Pull model Implementation

It is still an open question if the pull model can be, and should be, used in a real-time system. Recall that this model allows consumers of events to request the event data from supplier(s) by making CORBA requests via Event Channel(s). Moreover, as it was mentioned above, this approach to implement the pull model does not fit in the design of TDMIs in Real-Time CORBA. So, I have chose to implement only pull consumers that would request RT Event Channel(s) for the real-time events.

Our pull model relies on the RT Event Channel for distribution of events. Pull consumers query the RT Event Channel(s)' Event Buffer(s) for a particular real-time event via the *RT_Event_Channel* interface:

```
interface RT_Event_Channel {

        any try_pull(out boolean has_event) raises (RT_Exception);
        void pull(inout any event_data) raises (RT_Exception);

};
```

This interface supports two methods that can be invoked by the pull consumers: `try_pull` and `pull` (Figure 5). The `try_pull` method will be called periodically by the consumers untill the first real-time event has occurred. The Real-Time Event Dispatcher will assign the highest real-time OS priority to the threads that handle these periodic requests. Upon success, the event data will be returned along with a flag, which indicates that a real-time event has been delivered. The `pull` method will be invoked by the pull consumers that are pulling for a particular real-time event, and the type information (i.e. event name, its id, source, priority/importance) of the event will be passed as a parameter to the method call. The Real-Time Event Dispatcher will calculate and assign a real-time OS priority to the thread based on the priority/importance of the specified real-time event. If the real-time event was not found in the Event Buffer, a CORBA exception of type

RT_Exception will be raised in the consumer side. In order to provide the “best effort” on bounded event response time, our RT Event Channel makes use of the same Real-Time Concurrency Control with exclusive locking semantics.

4.4 Example of a TDMI

To illustrate how all of the parts of the prototype implementation work together, we present an example of a typical client-server interaction. Recall the example we described earlier in which a table containing tracking information is represented as a server on one node, and a client on another node wants to read certain data from the table with specified timing constraints. 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_Client.h   // header file for RT_Manager_Client class
 #include Track_Table_i.h       // header file for table implementation
       :
(1) RT_Manager_Client     rt_mgr;      // create instance of RT_Manager_Client
     Track_Table*   Track_Table_Obj;    // declare pointer to table
       :
     int main()                         // main procedure of a CORBA client
     {
        // RT init call
         RT_Manager_Init();
       :
       // 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
                  // deadline = NOW + 3 seconds
(3)      rt_mgr.Set_Time_Constraint_Now(BY,REL,3,0);
(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_Client` object (Label 1 in the above code). It then binds to the appropriate server (Label 2). Next, it calls the `RT_Manager_Client` 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_Time_Constraint_Now()` 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 functions to register with the Real-Time Daemon, calculate the Transient Priority for the client, set the client to a new priority and arms the timer according to the client's deadline.

After the above sequence is complete, the client makes the CORBA call to the table. The *RT_Environment* that is sent with the call contains the timing information computed by the `RT_Manager_Client`. At this point, the request is scheduled on the server's node as described below. If the client has not missed its deadline during the CORBA call, then `End_RT_Invocation()` disarms the clock and performs some clean up. If the timer expires (i.e., the deadline is missed), a CORBA exception of type

RT_Exception is raised in the client. The client catches this exception and performs any necessary recovery operations. The server side implementation for the method `Get()` would look like:

```
Track_Record
TrackTable::Get(CORBA::Long track_id, const RT_Environment& rt_env)
{
     Track_Record track;
     try {
(1)    RT_Manager_Server rt_mgr(rt_env);
(2)    rt_mgr.START_RT();
          :
       // Code for retrieval of TrackRecord with the specified ID
          :
(3)    rt_mgr.END_RT();
     }
(4)  catch(const RT_Exception &rtp) { rt_mgr.STOP(); }
     return track;
}
```

The server's thread creates a `RT_Manager_Server` object (Label 1) passing the client's RT_Environment as an argument to the constructor. The bulk of the work is done inside of the `START_RT()` (Label 2) method and is transparent to the server. This method determines the network delay, calls the functions to register with the Real-Time Daemon, calculate the Transient Priority for the server's thread, set the thread to a new priority and arm the timer according to the new deadline. After the above sequence is complete, the server performs the operations. If the server has not missed its deadline during the service, then `END_RT()`(Label 3) disarms the clock, performs some clean up, and the results are sent back to the client. If the timer expires (i.e., the deadline is missed), a CORBA exception of type *RT_Exception* is raised in the server. The server thread catches this exception (Label 4), and performs any necessary cleanup and recovery operations.

Real-Time Events. In the above example, the deadline for the client request was based on an absolute time that is relative to the global current time. However, deadlines can also be event-driven. For example, assume that in the above example the deadline for the

request was *NewContact* + 3 secs, 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 name, ID number, priority/importance of the event, and event source (a server name). Also, the client may choose to act as either a pull or a push consumer.

The revised code for an event-driven client, in which the client is a push consumer, is given below:

```
       try {
       :
        // create RT_Event object with event name & ID 5
       RT_Event  rt_event("NewContact",5);

       rt_event.Set_Priority(10);
       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
(3)    rt_mgr.Set_Time_Constraint_Event(BY, rt_event, 3, 0);

(4)    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 3 in above code) causes the client to wait, with infinite deadline, for a notification that the real-time event has occurred (push model). In case of a pull model, the `Set_Time_Constraint_Event()` function would create a thread, with a priority calculated from the priority of the expected real-time event, that would pull for the specified real-time event. If no such real-time event can be found, a CORBA exception of type RT_Exception will be thrown to the client. In the case of success the deadline for the client's request will be 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()`(Label 4) function as previously described.

Chapter 5

Evaluation

After the implementation was completed, several tests were done to show that the TDMI performs correctly and with the expected results. Then a suite of random TDMIs with

absolute, relative, and event-driven deadlines were executed to evaluate the performance of the TDMI's model as well as the Global Time and Real-Time Event Services. Also, a set of tests were executed to find the raw performance numbers of the system. For the testing, a grid example, which comes with ORBIX as a demo, was used. In this example, grid server manages one resource (a table) that has an interface with two methods (read/write). Grid clients can invoke those methods updating some values in the table.

5.1 Testbed Construction

A set of tests were generated from the following parameters: importance and a base priority; priority/importance of the real-time events; deadline of TDMI (short, medium, long, and event-based); start time of clients, and the method that a client invokes (read/write). For showing the correctness of TDMIs, each set of parameters were tested under varying loads (low, medium, and high) which were represented by the range of start times of clients. Each test was performed on our RTCORBA on Solaris, with expression and enforcement of timing constraints.

The analysis of the performance was based on the raw data determined for each test. Corresponding graphs with the raw data can be found in the Appendix (Page 58). In this thesis, the results of each test were averaged and analyzed over 25 trials producing in error of at most 1% in most cases.

All testing was performed on two Sun Sparc workstations (IPX and Sparc Station 5) on an isolated LAN with a fixed number of CORBA clients and servers on each

computer. Network delay was measured under different loads using the standard UNIX utility program called *ping*, and was found to be approximately 1.2 millisecond (ms).

5.2 Testing the Model of TDMIs

The implementation of TDMIs was tested by periodically starting up a client on one node (on a Sparc IPX station) that would send a request to a server on another node (on a Sparc Station 5). The purpose of this testing was to determine correctness and the overhead produced by the design and implementation of TDMIs. Recall that in the prototype implementation of TDMIs, an extra parameter (structure RT_Environment) must be added to all method invocations to be executed in real-time. Thus, an extra data copying, moving, dereferencing and transmission must be done by the Stubs/Skeletons/ORB, which is about extra 3 ms of a latency per method invocation.

Correctness. The correctness of the implementation was tested by running a set of clients with short (0-5 sec.), medium (5-10 sec.), and long (10-14 sec.) deadlines (relative and absolute). The clients and server's threads were forced to miss their deadlines at different moments of their execution - at the very beginning, in the middle, and at the end of TDMIs. As mentioned earlier, CORBA exceptions of type RT_Exception were raised and processed successfully by the clients and the server's threads, showing that the deadlines were actually missed in all cases.

Client side. The client side implementation was tested, as previously described, on a Sun Sparc IPX station. As expected, most of the overhead and latency was produced by the following two methods:

- `Start_RT_Invocation()` - RT_Manager_Client method that registers a client with a Real-Time Daemon, calculates and assigns the Transient Priority, arms a timer with a signal handling function. The latency introduced by this method is about 25.2 ms.
- `End_RT_Invocation()` - 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 is about 10.7 ms.

Server side. The server side implementation was tested, as previously described, on a Sun Sparc Station 5. Again, as expected, most of the overhead and latency was produced by the following two methods:

- `START_RT()` - RT_Manager_Server method that registers a server's thread with a Real-Time Daemon, calculates and assigns the Transient Priority, arms a timer with a signal handling function. The latency introduced by this method is about 11.4 ms.
- `END_RT()` - 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 is about 6.1 ms.

Analyzing the source code and its functionality, it was determined that each of these latencies is due to CORBA calls to the Real-Time Daemons on the local nodes. Unfortunately, the current implementation of ORBIX can not handle those requests (inter-process communications) in a more efficient way. In fact, ORBIX uses its

robustness and produces the overhead of a distributed computing environment to process method invocations within the same node.

5.3 Testing the Global Time Service and Clock Synchronization

As previously described, the Global Time Service was implemented with the assumption that all clocks on the system are synchronized within a known skew. Thus, it was very important to determine the performance of the clock synchronization utility. During the testing of our prototype implementation, the Sun Sparc Station 5 was configured to be a synchronization source (a time keeping server), and the Sun IPX station was configured to be a broadcast client. The xntpd3.5 utility provides special programs that allowed me to get some statistics and performance information on clock synchronization. Using those programs, it was determined that the xntpd3.5 utility was stable under different system loads, and was able to keep clocks offset within 3.2 ms with synchronization distance (broadcast delay) of 1.2 ms. Taking into consideration that clocks resolution on these two Sparc stations is about 10 ms, the imprecision of the utility can be tolerated.

Since all the clocks in the system are synchronized within a bounded skew, clients and servers can refer to their local clocks to determine the current global time. The overhead of this operation is only one operating system call that returns a data structure with the current time in it.

5.4 Testing the Real-Time Event Service

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. The purpose of this testing was to determine correctness and the overhead produced by design and implementation of the Real-Time Event Service.

Correctness. The correctness of the implementation was tested by running a set of suppliers and push/pull consumers. The suppliers were simultaneously generating real-time events with different priorities and importance on two Sun Sparc stations. The Real-Time Event Channels, one on each node, accepted all the events, stored them in the Event Buffer, and forwarded them to the consumers in priority order as expected. Upon delivery, the push/pull consumers stored the absolute time when the events were received, and that information was used to demonstrate the correctness.

Pull consumers were invoking the Real-Time Event Channels, trying to pull for the specified real-time events. In case of a success, corresponding event data was returned to the consumers. In all other cases, as mentioned earlier, CORBA exceptions of type RT_Exception were raised and processed successfully by the consumers, indicating that the specified events have not been delivered yet.

Overhead and Latency. The Real-Time Event Channel's implementation was tested on a Sun Sparc Station 5 by periodically generating and sending a real-time event to the push consumer through the Event Channel. The purpose of this test was to determine the dispatch latency, which would be an essential part of the total event response time. Thus,

the time between accepting the real-time event from a supplier and forwarding it to a consumer must be measured and analyzed first.

Recall that the Real-Time Event Channel receives events from suppliers by listening and reading from a configurable IP Multicast socket. Next, the Event Channel creates a bounded thread (LWP) that will deposit data into the Event Buffer and forward the event to consumers. Thus, from 7 to 9 systems calls must be done by the Event Channel to dispatch a real-time event. Experimental tests show that it takes about 7.8 ms (on average) for the Real-Time Event Channel to process a real-time event.

To determine the event response time, push supplier and consumers implementations were used. The supplier was run on the Sun Sparc Station 5 and periodically generated a real-time event. Two consumers - one on the same node, the other on the Sun Sparc IPX - waited for delivery of that event via the Real-Time Event Channels, which were also running on both Sparc stations. The experimental tests show that for the consumer running on the Sun Sparc Station 5, the event response time (time between a supplier generating a real-time event and a consumer receiving that event) was about 90.6 ms. For the consumer running on the Sun IPX, the event response time was about 96.6 ms.

Analyzing these results and taking into consideration some imprecision (e.g. network delay, clocks offset, context switching, etc.) along with dispatch latency, we can see that 85-90% of the overhead and delays are due to the network communications via IP Multicasting. The implementation of IP Multicasting is based on the extended Unix sockets' facilities and special routers, which are responsible for resolving IP addresses of

corresponding multicast groups. As the result, there is a possibility to have some tangible delays that may or may not be tolerated in real-time systems.

The event response time and dispatch latency for a pull consumer were tested by periodically invoking the Real-Time Event Channel on the same node. Since the implementation of a pull consumer is based on the standard CORBA method invocations, the event response time, in general, depends on their efficiency. Executing a set of tests and evaluating the results, the dispatch latency for a pull consumer was determined to be about 40.1 ms. The event response time was determined to be about 117.6 ms excluding the bind() call to the Real-Time Event Channel, and about 161 ms including the bind() call. Recall that the ORB is responsible for finding the Real-Time Event Channel that will handle the request and binding the consumer to it. This involves returning an object reference for the Real-Time Event Channel.

It was determined that 70-75% of the latency is due to the fact that ORBIX inefficiently handles local CORBA calls that were initiated by the pull consumers. Moreover, ORBIX introduces a noticeable latency for handling method invocations that have data structures in their list of parameters. The dispatch latency is due to the overhead associated with getting access to the Event Buffer and searching through it for a specified real-time event.

Finally, the implementation of a pull consumer uses a special CORBA IDL type, *CORBA::any*, for passing the event data between a pull consumer and a Real-Time Event Channel. In a CORBA environment, this data type is used to pass a value of an arbitrary type as a parameter or a return value. Although ORBIX has implemented a number of ways of constructing and interpreting this data type the efficiency of those methods is

unknown. It is obvious that an extra data copying, moving, dereferencing and transmission must be done by the Stubs/Skeletons/ORB, which adds some delay and might significantly decrease the performance of the Real-Time Event Service.

Chapter 6

Conclusion

6.1 Contributions

This thesis has presented a set of extensions for expressing timing constraints in a CORBA environment. The basis for the work was the CORBA/RT white paper that specifies desired capabilities for extending CORBA for real-time. While these desired capabilities cover a wide range of the CORBA standard, the results of this thesis focus on

some of the features that are necessary for expression and enforcement of timing constraints. CORBA clients can now express their timing requirements, such as deadlines, importance and quality of service, on requests that they make to servers.

Once these requirements are specified, the new and/or extended object services provide their enforcement. The Global Time Service ensures that all clocks in the system are synchronized, and provides the consistent notion of a “global current time”. The Real-Time Event Service enforces the distribution of real-time events in priority order with real-time enforcement of event response time, and provides timestamp information associated with those events.

It is important to note that this design does not change the current CORBA specifications, but rather extends them. The design described in this thesis extends CORBA standard by providing new features, such as the model of Timed Distributed Method Invocations, a RT_Environment structure to enable TDMIs, adding/extending services, such as the Global Time Service and the Real-Time Event Service.

The test results show that the TDMIs, Global Time and Real-Time Event Services perform correctly in the real-time system. The raw performance, overhead and latency of each of the implementations were measured and analyzed. The resulting performance numbers show that designed and implemented components can be used in a system with soft real-time requirements, and should be considered for the performance analysis of that system.

6.2 Comparison with Related Work

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 and scheduling parameters. The results presented in this thesis are important first steps towards achieving this goal.

Our design and implementation of TDMIs in CORBA does not have any direct analogies yet. The clients can now express timing constraints, importance and quality of service parameters on requests that they make to servers in the form of TDMIs. The Global Time Service was designed and implemented to support TDMIs in a distributed fashion, relying on a clock synchronization mechanism which ensures a consistent notion of a current global time. Thus, CORBA clients and servers can determine the current global time with minimal overhead by referring to their local system clocks. The Real-Time Event Service was designed and implemented to propagate, filter, and correlate real-time events in priority order. This Service also supports TDMIs with event driven deadlines, providing timestamp information for the real-time events. The design and implementation of TDMIs in our Real-Time CORBA uses a dynamic scheduling policy (EDF) with aging. Each CORBA call is scheduled on its local node by a Real-Time Daemon based on its scheduling policy. In order to meet real-time guarantees in our system we can allow method invocations to be made only in the form of a TDMI. This

forced us to implement our Real-Time Event Service without using the full set of CORBA Event Service interfaces.

This thesis has presented an important first step in providing support for real-time requirements in distributed computing environments such as CORBA. However, many other steps are still needed to produce a viable CORBA/RT specification and implementations.

6.3 Limitations and Future Work

The design and implementation of TDMIs, Global Time and Real-Time Event Services depend heavily on the underlying operating system, CORBA implementation, and network. For instance, our implementation actively uses the routines for assigning the real-time OS priorities; relies on OS's scheduling of the real-time tasks; relies on the network communications for sending and receiving CORBA requests and real-time events; uses signals, timers, and the other system resources. Thus, the efficiency and reliability of those facilities are required.

There is still significant work to be done to meet the many desired real-time capabilities in the CORBA/RT white paper. This includes hard guarantees of service times across the environment, guarantees of minimal inter-arrival time for server requests, and interface-level support for multi-threading.

It is worth pointing out that while this design is based on extending CORBA, many of the concepts can be applied to real-time distributed systems in general. For

instance, the Timed Distributed Method Invocation can be seen as an abstraction in which a client specifies the required timing behavior for a server and during transport.

List of References

[1] "CORBA/RT White paper". The Real-Time Platform Special Interest Group of the Object Management Group. December 5 1996. Editor: Judy McGoogan from Lucent Technologies.Currently available at FTP site: ftp://ftp.osaf.org/whitepaper/Tempa4.doc

[2] "CORBAservices: Common Object Service Specification". The Object Management Group. Revised Edition: March 31, 1995. Updated: March 28, 1996.
Currently available at HTTP site: http://www.omg.org/library/corbserv.htm

[3] "On Real-Time Extensions to the Common Object Request Broker Architecture:
A position paper" P. Krupp, A. Schafer, B. Thuraisingham, V. Fay-Wolfe.
Proceedings on the OOPSLA'94 workshop on Experiences with CORBA, October 1994
Currently available at HTTP site: http://www.cs.uri.edu/rtsorac/publications.html

[4] "Real-Time Method Invocations in Distributed Environments"
V. Fay-Wolfe, J. Black, B. Thuraisingham, P. Krupp. In Proceedings of the International High Performance Computing Conference, India, December 1995.
Currently available at HTTP site: http://www.cs.uri.edu/rtsorac/publications.html

[5] "Internet time synchronization: the Network Time Protocol"
Mills, D.L. IEEE Trans. Communications COM-39, October 10, 1991, 1482-1493.

[6] “Network Time Protocol (Version 3) specification, implementation and analysis”
Mills, D.L.. Network Working Group. Report Request For Comments(RFC)-1305, University of Delaware, March 1992. Revised from: Electrical Engineering Department Report 90-6-1, University of Delaware, June 1990.
Currently available at HTTP site: http://www.eecis.udel.edu/~mills/bib.html

[7] “On the accuracy and stability of clocks synchronized by the Network Time Protocol in the Internet system” .Mills, D.L. ACM Computer Communication Review 20, 1 (January 1990), 65-75.

[8] “Request for Comments (RFC):1584”. J. Moy. Network Working Group. Proteon, Inc. March 1994.
Currently available at HTTP site: http://www.cis.ohio-state.edu/htbin/rfc/rfc1584.html

[9] "A high-performance endsystem architecture for real-time CORBA"
D. Schmidt, A. Gokhale, T. Harrison, G. Parulkar, Department of CS, Washington University. Will appear in the feature topic issue on Distributed Object Computing in the IEEE Communication magazine.

[10] "Operating System Support for a High-Performance, Real-Time CORBA"
T. Harrison, A.Gokhale, D. Schmidt, G.Parulkar, Department of Computer Science, Washington University. Presented at the International Workshop on Object-Orientation in Operating Systems: IWOOOS 1996 workshop, October 27-28, 1996, Seattle, WA

[11] “The Design and Performance of a Real-Time CORBA Object Event Service”.
T. Harrison, D. Levine, D. Schmidt, Department of Computer Science, Washington University. This paper has been submitted to OOPSLA’97, Atlanta, Georgia, October 1997.

[12] "Meeting the real-time synchronization requirements of multimedia in open distributed

processing" G. Coulson, G Blair, Distributed Multimedia Research Group, Department of Computing, Lancaster University, UK, 1992. Internal report number MPG-92-45.

[13] "Object Time Service. Request For Proposal (RFP)". The Object Management Group, December 1995. Internal document number 95-11-8.
Currently available at HTTP site: http://www.omg.org/corba/sectrans.htm#time

[14] "Object-oriented analysis and design with applications ". Second edition. G. Booch, Rational,Santa Clara, California. Published by The Benjamin/Cummings Publishing Company, Inc., 1994.

[15] Information available at HTTP site: http://www.iona.com

[16] Information available at HTTP site: http://www.chorus.com

[17] Information available at HTTP site: http://www.sun.com/solaris/solaris.html

[18] Information available at HTTP site: http://www.lynx.com

[19] Information available at HTTP site: http://www.wrs.com/vxwks52.html

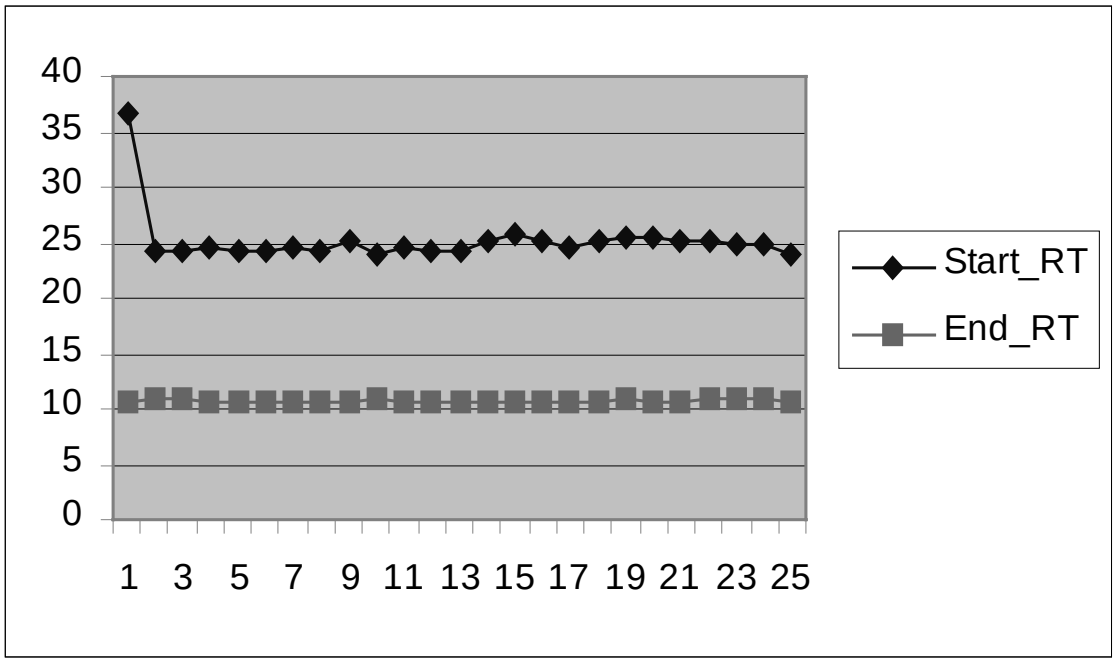
[20] "Scheduling algorithms for multiprogramming in a hard-real-time environment" C. Liu, J. Layland. Journal of the ACM, 20(1):46-61, 1973.

[21] "Expressing and Enforcing Timing Constraints in a Real-Time CORBA System" L.C DiPippo, R. Ginis, M. Squadrito, S. Wohlever, V. Fay-Wolfe, I. Zykh. Computer Science Department, University of Rhode Island. This paper has been submitted to the RTAS' 97. Montreal, Canada. June 1997

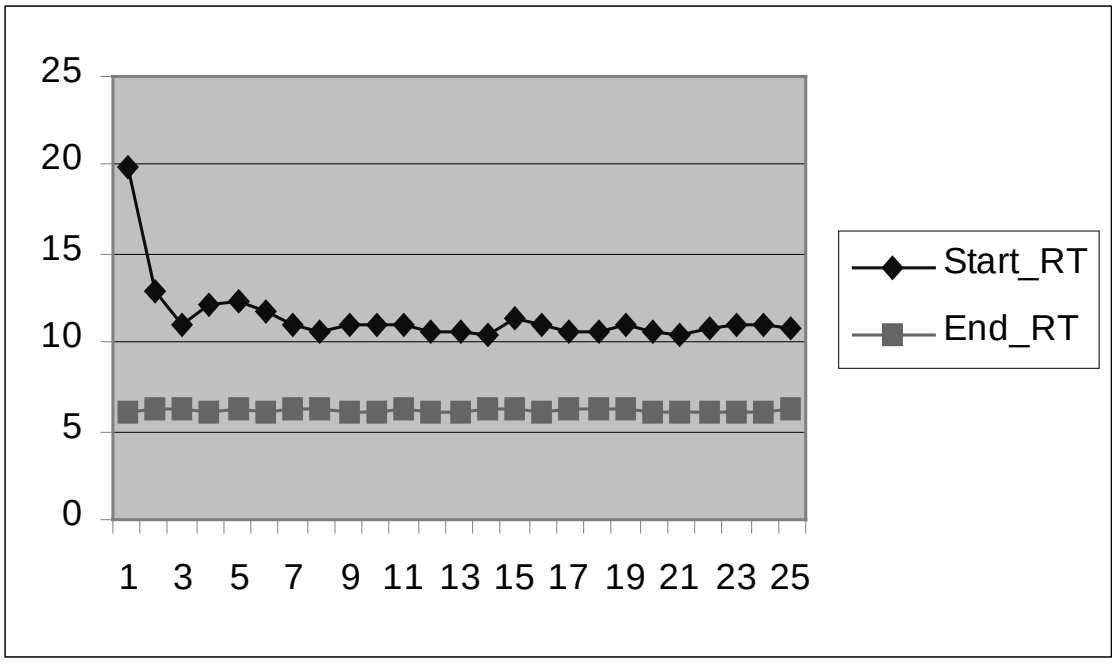
Appendix

This appendix contains graphs based on the raw performance data that was accumulated during testing. Some of the graphs have peaks that can be explained as follows:

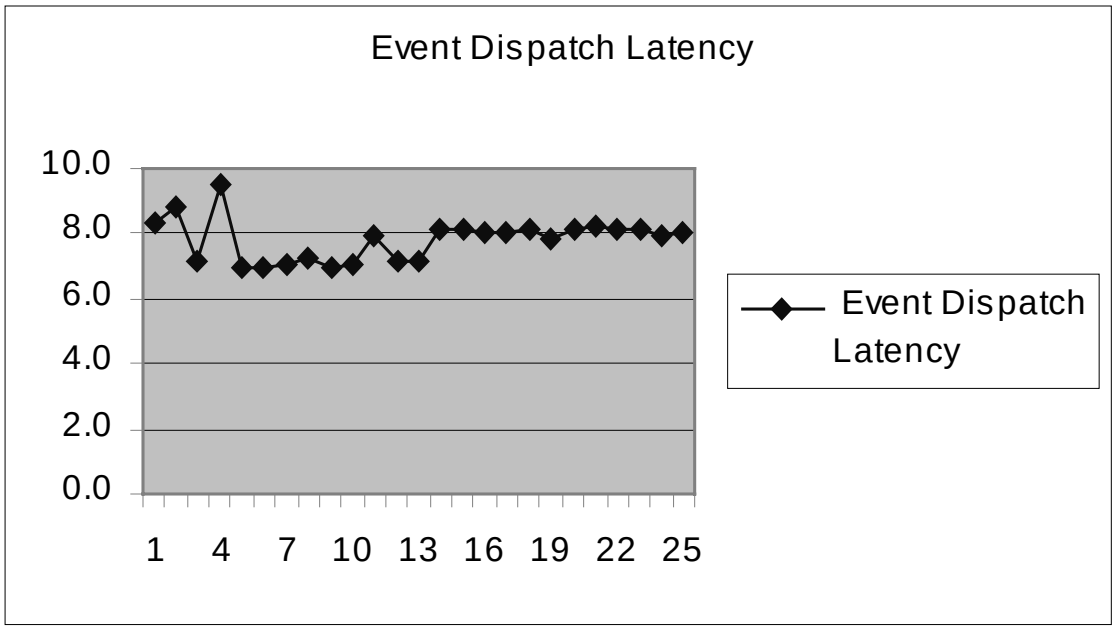
- **Graph 1** and **Graph 2**. The first peak is due to the delay that Orbix introduces when it processes the first CORBA call from a client/server to a Real-Time Daemon. The actual object reference to a Real-Time Daemon is generated and interpreted at that point.
- **Graph 3**. There are some peaks that are due to the delays in the operating system and availability of system resources. For instance, the Real-Time Event Dispatcher creates threads to dispatch incoming real-time events in priority order. Thus, the operating system response time for the corresponding system calls may be different, and depends on its load.
- **Graph 4**. There are some peaks that are due to network delays and availability of system resources. For instance, during the performance tests, two Sun Sparc stations were sharing some file directories via NFS that might produce additional network traffic.



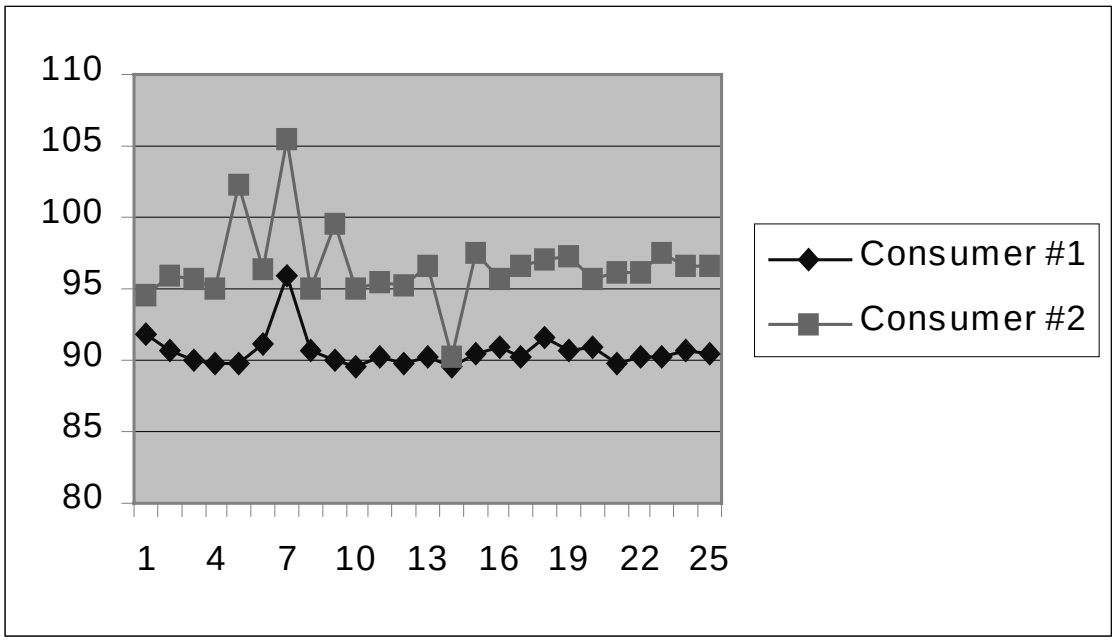
Graph 1. Client side: TDMI performance



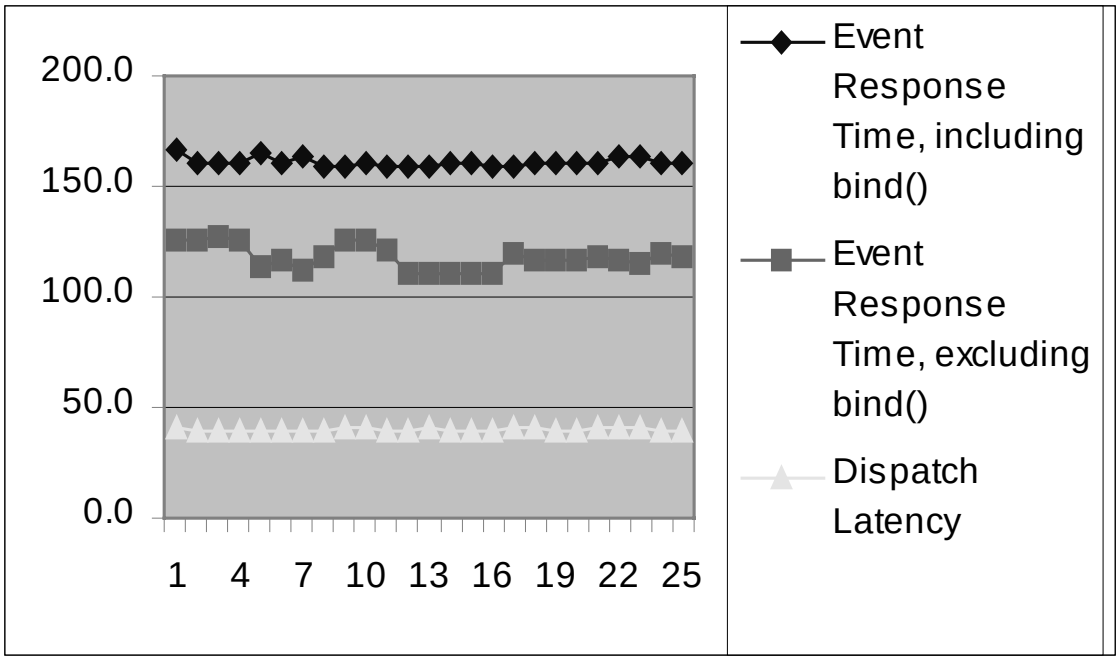
Graph 2. Server side: TDMI performance



Graph 3. Real-Time Event Channel: Event Dispatch Latency



Graph 4. Push Consumers: Event Response Time
(Consumer #1 on Sun Sparc Station 5, Consumer #2 on Sun Sparc IPX)



Graph 5. Pull Consumer: Event Response Time and Dispatch Latency

Bibliography

Coplien, J. *Advanced C++ Programming Style and Idioms*. Addison-Wesley Publishing Company, Reading, MA, 1992,

Coulouris, G., Dollimore, J. *Distributed Systems. Concepts and Design*. Second Edition. Addison-Wesley Publishing Company, Reading, MA. 1994

Donohoe, P., Shapiro, R., Weiderman, N. *Hartstone Benchmark User's Guide. Version 1.0*. Carnegie Mellon University, Software Engineering Institute, March 1990.

Gallmeister, B. *POSIX .4. Programming for the real world*. O'Reilly & Associates. Inc. , Sebastopol, CA. 1995.

Guttman, M. & Matthews, J. *The Object Technology Revolution*. John Wiley & Sons, Inc. New York, NY, 1995.

IEEE Standard for Information Technology. *Portable Operating System Interface (POSIX). Part1: System Application Program Interface. Amendment 1: Realtime Extension*. Institute of Electrical and Electronics Engineers (IEEE), Inc. New York, NY, 1994.

Levine, J., Mason, T., Brown, D. *Lex&Yacc*. O'Reilly & Associates. Inc. , Sebastopol, CA. 1992

Lippman, S. *C++ Primer*. Addison Wesley, Reading, MA, 1993

Mullender, S. *Distributed Systems*. Second Edition, reprinted in 1994. Addison-Wesley Publishing Company. Reading, MA.

Silberschartz, A., Peterson, J., Galvin, P. *Operating System Concepts*. Addison-Wesley Publishing Company, Reading, MA. 1992

Stroustrup, B. *The Design and Evaluation of C++*. Addison-Wesley Publishing Company, Reading, MA, 1994

Tanenbaum, A. *Distributed Operating Systems*. Prentice Hall. Englewood Cliffs, NJ. 1995.