# Real-Time CORBA*

Victor Fay-Wolfe,
Lisa C. DiPippo and
Gregory Cooper
The University of Rhode Island
Kingston, RI USA 02881
lastname@cs.uri.edu

Russell Johnston
SPAWAR Systems Center
San Diego, CA USA
russ@spawar.navy.mil

Peter Kortmann
Tri-Pacific Software
Alameda, CA USA
peter@tripac.com

Bhavani Thuraisingham
MITRE Corporation
Bedford, MA
thura@mitre.org

## Abstract

This paper presents a survey of results in developing *Real-Time CORBA*, a standard for real-time management of distributed objects. The paper includes background on two areas that have been combined to realize Real-Time CORBA: the CORBA standards that have been produced by the international Object Management Group; and techniques for distributed real-time computing that have been produced in the research community. The survey describes major RT CORBA research efforts, commercial development efforts, and standardization efforts by the Object Management Group.

## *1* Introduction

*Middleware* is software that facilitates seamless client/server interaction in a distributed system. "Middle" refers to its place in a software abstraction hierarchy above transport protocols, but below clients and servers written in a high level programming language. Four prominent middleware interfaces are DCE by the Open Software Foundation [1], DIS/HLA used by the US military for massive distributed simulations, DCOM from Microsoft [2], and CORBA, a standard from the Object Management Group (OMG) [3]. With over 800 organizations participating in its development, CORBA has emerged as the most prominent among the middleware interfaces. The OMG has many Special Interest Groups (SIGS), Task Forces, and working groups developing specific aspects of CORBA. These aspects range from CORBA's interface to the Internet, to specific CORBA interfaces for industrial domains such as finance and medicine, to middleware protocols used in distributed systems.

In 1996 a SIG was formed within the OMG with the goal of developing support in the CORBA standard for *real-time* applications – applications in which there are end-to-end timing constraints across a distributed system. This interest in adding support for real-time applications into CORBA was the result of demand from application domains such as telecommunications, military command and control, manufacturing, and finance, all of which have timing constraints on client/server interaction in a distributed system. Furthermore, the feasibility of development of a real-time CORBA (RT CORBA) standard was indicated by several research efforts in RT CORBA that had been initiated in the early/mid 1990s. This paper presents a survey of research and development towards RT CORBA from three areas: research from academic and government groups, development from commercial CORBA vendors, and standardization from the OMG RT SIG.

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

This paper is structured as follows. Section 2 provides background on CORBA, its goals, architecture, and standardization process. Section 2 also presents background on techniques to support real-time requirements in distributed systems. Section 3 surveys academic and government RT CORBA research efforts as well as commercial RT CORBA development. Section 4 describes the RT CORBA standard draft that, at the time of this writing, is in the final stages of integration into the OMG's CORBA standard. Section 5 summarizes and outlines the likely future directions of RT CORBA.

# 2 Background

The study of RT CORBA combines two major areas of work: (1) the OMG's CORBA standard; and (2) real-time systems. In this section we present background on both of these areas.

## *2.1 CORBA*

The CORBA standard specifies interfaces that allow seamless interoperability among clients and servers under the object-oriented paradigm. It is produced by the Object Management Group, which has been meeting approximately every six to eight weeks since 1989. The CORBA specification process is evolutionary. Features of the standard are proposed, bid upon, debated and adopted piecemeal according to a roadmap established by the OMG. CORBA version 1.1 was released in 1992, version 1.2 in 1993, and version 2.0 in 1996. The V1.2 standard 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. Object services for naming, events, relationships, transactions, and concurrency control are addressed in Version 2.0 [4]. Also addressed in CORBA 2.0 is interoperability of CORBA middleware implementations from different vendors through its *Internet Inter-ORB Protocol* (IIOP). Services such as time synchronization and security are addressed in later revisions of CORBA 2.0. The OMG has been remarkably successful in agreeing upon increments to the standard and vendors have quickly made products available that meet the evolving standard.

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. That is, a client program should be able to invoke a method on a server object whether the object is in the client's address space or located on a remote node 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. A series of articles in [5] provides a technical overview of the CORBA standard, we very briefly summarize the standard here.

### 2.1.1 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. The IDL grammar includes a subset of ANSI C++, with additional constructs to support the method invocation mechanism.

Most common intrinsic C++ types are supported in CORBA IDL. The ORB handles differences in type representations among architectures (e.g. Big Endian, Little Endian). CORBA's IDL also specifies C++-like exceptions. IDL does not provide syntax for implementing methods. IDL bindings to various languages including C, C++, Java, Smalltalk, and Ada have been specified for this purpose.

As an example, consider an object that acts as a shared table for sensor data (represented as long integer values) for clients in a distributed system. A simple CORBA IDL for a `sensor_table` object is:

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

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 (the IDL compiler generates a remote method for reading each attribute). The IDL example also specifies two methods: *put* which stores a sensor value at a index into the table; and *get* which returns a sensor value given an index.

Client code in C++ to access a `sensor_table` object in a CORBA environment might look like the following:

```
long retval;
sensor_table *p;
p = bind("my_sensor_table");
retval = p->get(500);
```

Here, the client declares what appears to be a pointer, `p`, to a `sensor_table` object called `my_sensor_table`. The "pointer" is actually a CORBA type called an object reference, which provides the local representation of the CORBA object to the client. The client then makes a call to an ORB service to locate and bind the pointer to a reference to a remote server containing the `sensor_table` object. To retrieve a value from the `sensor_table` at index 500, the client issues the method invocation shown in

the last line: `p->get(500).` This method invocation assumes that a `sensor_table` server was previously implemented and registered with the CORBA ORB.

### 2.1.2 Implementing Clients and Servers

The process of implementing a client and server object in the C language is shown in Figure 1. The IDL specification is processed by an IDL compiler, which generates a header file for the CORBA object, stub code for linking into the client, and skeleton code for the server object. The client stub contains code that hides details of interaction with the server from the client code. Client stubs stand in for actual method calls by transparently directing method requests into the ORB. Server skeleton code is used by the ORB to forward method invocation requests to the server, and to return results to the client.

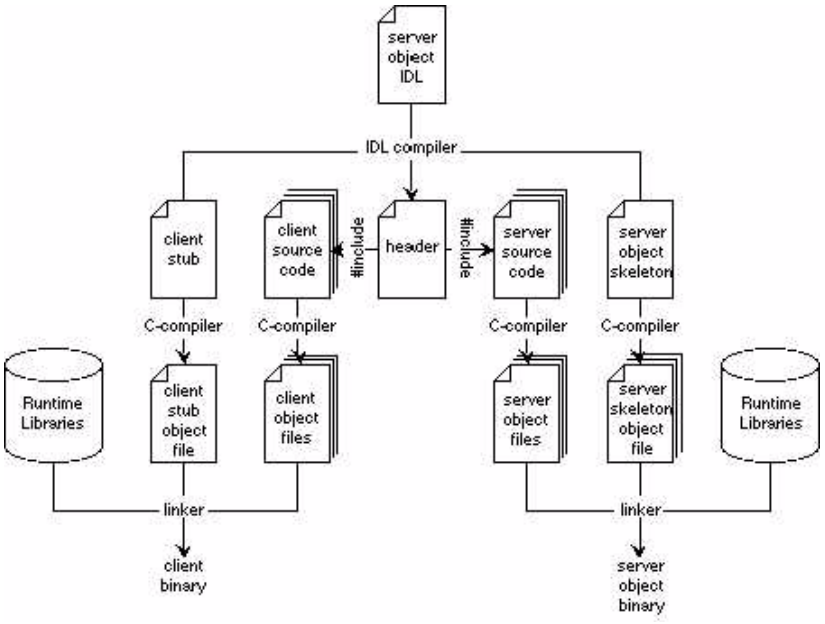


**Figure 1 - Process to Implement CORBA Clients and Servers**

### 2.1.3 The ORB

An ORB provides for:

1. *Locating a server object implementation for servicing a client's request*. The client makes a request by invoking a method on an object reference as described in the example of section 2.1.1. The client obtains that object reference either by a call to the CORBA *Naming Service*, which returns an object reference associated with a name; by a call to the CORBA *Trader Service*, which returns an object reference associated with specified functionality and quality of service; or by obtaining a Interoperable Object Reference using the IIOP protocol (the CORBA Naming and Trader Services are discussed in Section 2.1.4, IIOP is discussed in Section 2.1.5). Using the object reference, the ORB software locates the appropriate end point in the distributed system to which to send a message to initiate the

client's request. In a TCP/IP-based CORBA environment, locating the server usually involves identifying the server's IP address and port.

2. *Establishing a connection to the server.* After locating the server for the client's request, the ORB software establishes a network connection from the client to the server. In a TCP/IP-based system, this involves establishing a socket-based connection.

3. *Communicating the data making up the request.* The ORB software uses the connection to the server to send parameters of the method invocation to the server. This procedure involves packaging parameters so that they can be sent on the specific network connection. It also involves resolving differences in data representation between the client and server. In addition to parameters, other data about the client and its request is sent to the server.

4. *Activating and deactivating objects and their implementations.* Once the client's request arrives at the server, a part of the ORB called the *Object Adapter* accepts the incoming message and directs it to the active processing entity that will service the request. If the object requested is inaccessible, the ORB returns a CORBA exception to the client. The Object Adapter can also perform forms of concurrency control; for instance by queuing requests for serial access to the server. For instance, the Object Adapter in a TCP/IP-based ORB listens on the incoming port, accepts the client's request message, formats incoming parameters if necessary, and may direct an internal message to an operating system process, which in turn creates a thread to handle the request. The Object Adapter also routes a response message back to the client and deallocates appropriate resources when the service is complete.

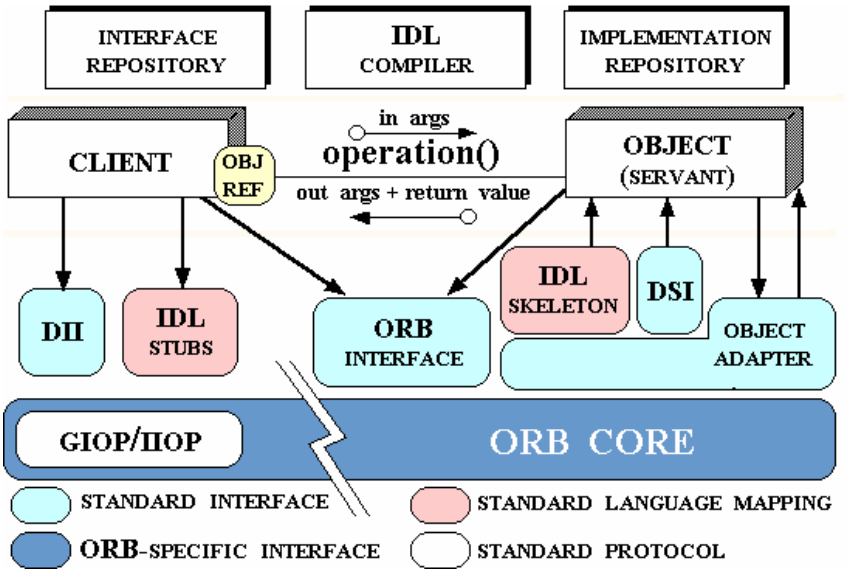


**Figure 2 - CORBA System Components (from [21])**

A client and its stubs, a server and its skeleton, and the interaction through the ORB are shown in Figure 2. The stubs and skeletons are part of the ORB and are produced by the IDL compiler as described in Section 2.1.2. Typically stubs are linked into client code and skeletons into server code. They interact with the ORB Core. The ORB Core is the message passing mechanism; in many current ORBS the ORB Core is a TCP/IP sockets-based component. As described above, the *Object Adapter* is the part of the ORB that connects the client's request to the stub code of the server object. The Object Adapter accepts the message from the ORB Core and initiates the processing of the service from the CORBA server object. CORBA 2.2 refined the specification of the Object Adapter by specifying the interface to a *Portable Object Adapter (POA)*. A POA is an object in the server that manages other CORBA objects in the server. The application programmer writing the server can specify POA *policies*, which are defined in the CORBA standard, to control how the POA manages objects. For instance, whether the server handles client requests by separate threads or a single thread that sequentially handles all client requests, can be specified as a POA policy. Other parts of the ORB: the Dynamic Invocation Interface (DII), Interface Repository, Implementation Repository, and Dynamic Skeleton Interface (DSI) are described in the CORBA specification, but are not directly relevant to RT CORBA, so we do not discuss them here.

### 2.1.4 Object Services

CORBA 2.0 contains the specifications for CORBA's Common Object Services. A full description of all of the services can be found in [4]. The following are some of the more widely used object services:

1. *Naming* – This service provides the ability to bind a name to an object relative to a naming context. It guarantees unique names for objects.

2. *Trader* – This service provides the ability for a client to selectively bind to an object based on certain criteria, such as quality of service provided by the object.

3. *Event* – This service provides basic capabilities for notification of named events. Suppliers can generate events without knowing the IDs of consumers. Consumers can use events without knowing the IDs of suppliers.

4. *Life Cycle* – This service allows for the creation and destruction of objects in the CORBA system.

5. *Persistence* – This service allows for making objects persistent on some storage medium.

6. *Transactions* – This service allows for construction of transactions, which are atomic collections of client calls to server objects.

7. *Concurrency Control* – This service allows objects to be locked by clients. The locking scheme is a version of database read/write locking.

8. *Externalization* – This service allows objects to be passed in a CORBA environment and among environments.

9. *Relationship* – This service allows the expression of semantic relationships among objects.

### 2.1.5 Internet Inter-ORB Protocol (IIOP)

CORBA 2.0 also provides capabilities for ORBs from different vendors to interact. The key component is an application layer protocol called the *General Inter-ORB Protocol* (GIOP), and its implementation on the TCP/IP transport layer protocol called the *Internet Inter-ORB Protocol* (IIOP). All CORBA 2.0 compliant ORBs must support their servers being accessed by clients using IIOP, even if the client is residing on a node controlled by a different CORBA system. A server that will accept IIOP invocations publishes its TCP/IP addresses as an *Interoperable Object Reference* (IOR). The client must obtain the IOR for the server some how, perhaps by using the CORBA Naming Service (similar to a browser needing to obtain a URL for a Web page some how before it can request the page). Once the IOR is obtained, a CORBA primitive translates the IOR into a CORBA object reference for use by the client.

### 2.1.6 CORBA's Future

The OMG is still growing in participation and in the scope of CORBA's capabilities. Vendors are producing software that meets the standards very soon after each revision to the standard comes out. New extensions are coming out of every meeting and many more are on the OMG's roadmap.

A very active part of the CORBA effort is the development of the OMG's Unified Modeling Language (UML). UML is a graphical language that depicts classes, their components, and their relationships. It also includes behavior diagrams to show how object-oriented programs execute. The UML working group started addressing real-time extensions to UML in 1999. These extensions include the ability to express timing constraints (see Section 2.2) on behavior diagrams.

CORBA 3.0, released in 1999, includes a more detailed description of the POA (see Section 2.1.3) including support for persistent objects that exist after they are created or activated. Also included in the POA specification in CORBA 3.0 is control of the threading policy used by the server. Options such as one-thread-per-request, one-thread-per-object, and thread-pools-per-object, are specified to indicate how the server will handle client requests. The real-time CORBA extensions discussed in Section 4 rely on the threading capability of CORBA 3.0 and extend this capability to address issues concerning the number of threads allowed and the priorities of the threads. CORBA 3.0 also contains support for asynchronous messaging. The previous CORBA specifications were built on the synchronous communication notion of remote procedure calls. The ability to asynchronously request services is essential, particularly for real-time applications. The CORBA 3.0 specification will allow for synchronous, asynchronous, and deferred synchronous (where the client can check for results of the request at any time after making the request.

A complete list of OMG working groups, their whitepapers, current standards, and draft standards, can be found on the OMG Web site at http://www.omg.org. Included is the work of the Real-Time Special Interest Group, which is producing the Real-Time CORBA standard described in Section 4. An overview of CORBA 3.0 and a discussion of CORBA's future are provided in [5].

## *2.2 Real-Time Systems*

In a real-time system there are timing constraints on execution. 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. Although speed/high-performance 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.

### 2.2.1 Expressing Timing Constraints

Most real-time systems specify a subset of the following constraints:

1. 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.

2. 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.

3. 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. Typically a *period frame* is established for each instance of the (repeated) task. As shown in Figure 3, period frame *i* specifies the default earliest start time and deadline for the $i^{th}$ instance of the task. When periodic execution is originally started, the first frame is established, at time *s* in Figure 3. For periodic execution with period *p*, the $i^{th}$ frame starts at time $s + (i-1)p$ and completes at time $s + (i)p$. As this indicates, the end of frame *i* is the beginning of frame *i+1*. Each instance of a task may execute anywhere within its period frame.

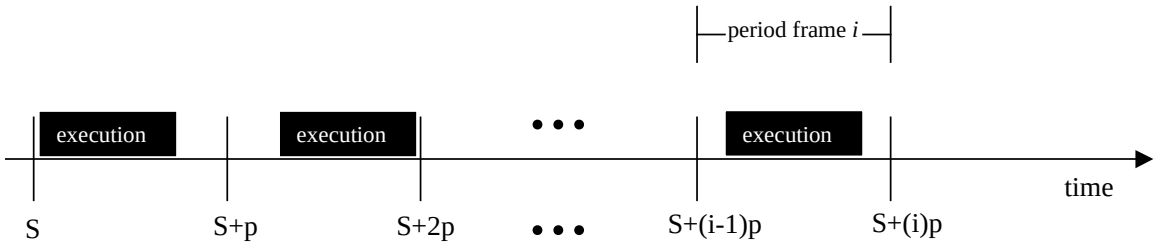


**Figure 3 - Example of Periodic Execution**

### 2.2.2 Modes of Real-time

Real-time constraints are classified as hard, firm, or soft, depending on the consequences of the constraint being violated. 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. Most real-time applications consist of predominantly soft real-time constraints. Graphic display updates are one of many examples of tasks with soft real-time constraints.

In some systems the mode of real-time is captured in a task's *importance* level. In systems such as the *Spring* real-time operation system from the University of Massachusetts [6], task importance is categorized according to the mode of its timing constraint (hard, firm, soft). In other systems, importance is more general and tasks can be assigned importance relative to each other over a wider granularity of levels. Note that importance is not the same as *priority*. Priority is a relative value used to make scheduling decisions. Often priority is a function of importance, but also can depend on timing constraints, or some combination of these, or other task traits.

### 2.2.3 Predictability

In order to meet timing constraints predictably, it must be possible to accurately analyze timing behavior. To analyze timing behavior, the scheduling algorithm for each resource and the amount of time that tasks require on each resource must be known. To fully guarantee this timing behavior, these resource utilizations should be worst case values, which tend to be pessimistic. Some soft real-time systems can tolerate average case values that offer no strong guarantee.

Assuming that worst-case resource utilizations are known, analyzing timing behavior for predictability depends on the scheduling algorithms used. In the next subsection we discuss several real-time scheduling techniques and the forms of analysis that these techniques facilitate.

### 2.2.4 Real-Time Scheduling and Resource Access Control

Real-time scheduling essentially maps to a *bin-packing problem* where tasks with known resource utilizations are the boxes, and the timing constraints establish the size of the bin. That is, each task can be considered a "box" whose size is its utilization of the resource being scheduled. The start times and deadlines of the tasks establish the boundaries of a "bin", or collection of bins, in which the boxes must be packed. The bin-packing problem is NP-hard, so optimal real-time scheduling, in general, is an NP-hard problem. However, heuristics have been developed that yield optimal schedules under some strong assumptions, or near-optimal results under less-restrictive assumptions.

Typically, a scheduling algorithm assigns *priorities* to tasks. The priority assignment establishes a partial ordering among tasks. Whenever a scheduling decision is to be made, the scheduler selects the task(s) with highest priority to use the resource. There are several characteristics that differentiate scheduling algorithms. They are:

1. *Preemptive versus nonpreemptive*. If the algorithm is preemptive, the task currently using the resource can be replaced by another task (typically of higher priority).

2. *Hard versus soft real-time*. To be useful in systems with hard real-time constraints, the real-time scheduling technique should allow analysis of the hard timing constraints to determine if the constraints will be met predictably. For firm and soft real-time, predictability is desirable, but often a scheduling technique that can demonstrate best-effort or near-optimal performance is acceptable.

3. *Dynamic versus static*. In static scheduling algorithms, all tasks and their characteristics are known before scheduling decisions are made. Typically task priorities are assigned before run-time and are not changed. Dynamic scheduling algorithms allow task sets to change and usually allow for task priorities to change at run-time.

4. *Single versus multiple resources.* Single resource scheduling manages one resource in isolation. In many well-known scheduling algorithms, this resource is a single CPU. Multiple resource scheduling algorithms recognize that most tasks need multiple resources and schedule several resources. End-to-end schedulers schedule all resources required by the tasks.

**Real-Time Scheduling Algorithms**. For a known set of independent, periodic tasks with known execution times, Liu and Layland proved that *rate-monotonic* CPU scheduling is optimal [7]. Here optimal means that if any scheduling algorithm can cause all of the tasks in a set to meet their deadlines, then rate-monotonic can as well. Rate-monotonic scheduling is preemptive, static, single resource scheduling that can be used for hard real-time. Priority is assigned according to the rate at which a periodic task needs to execute. The higher the rate, which also means the shorter the period, the higher the priority. A supplemental result by Liu and Layland facilitates real-time analysis by proving that if the CPU utilization is less than approximately 69%, then the task set will always meet its deadlines [7]. Deadline monotonic

[8] scheduling is a variation of rate-monotonic in which a task may have a deadline prior to the end of its period. In this algorithm, highest priority is assigned to the task with the shortest deadline. The schedulability analysis for deadline monotonic scheduling is similar to that for rate-monotonic scheduling.

For *dynamic* scheduling that is also preemptive and single resource, Liu and Layland showed that earliest-deadline-first scheduling is optimal and that any task set using it with a utilization less than 100% will meet all deadlines.

**Real-Time Resource Access Control Algorithms**. Most real-time scheduling involves tasks that access shared resources. When a high priority task is forced to wait for a lower priority task to finish using a shared resource, a property known as *priority inversion* occurs. Real-time resource access control algorithms must account for this kind of blocking, and in some cases bound it. Rajkumar, Sha and others have shown that task sets where the tasks can coordinate via mechanisms such as semaphores, can be analyzed if they use *priority inheritance* protocols [9]. In these protocols, a lower-priority task that blocks a higher-priority task (e.g. by holding a semaphore), inherits the priority of the higher-priority task during the blocking. With priority inheritance techniques, priority inversion can be bounded and factored into the worst case execution time of each task.

The *priority ceiling protocol* (PCP) is a priority inheritance protocol that bounds priority inversion and prevents deadlock. The priority ceiling of a semaphore, or any other resource, is the priority of the highest priority task that will lock the semaphore. These priority ceilings are computed *a priori*. When a task requests a lock on a semaphore, the lock is granted only if the task's priority is strictly greater than the priority ceilings of all semaphores locked by other tasks.

The *distributed priority ceiling protocol* (DPCP) [9] extends the PCP by taking into account accesses to remote resources in a distributed system. In the DPCP, a resource that is accessed by tasks allocated to different processors than its own is called a *global resource*. All other resources (those only accessed by local tasks) are *local resources*. A critical section on a global resource is referred to as a *global critical section* (GCS). A *local critical section* (LCS) refers to a critical section on a local resource. The base priority (BP) of a system of tasks is a fixed priority, strictly higher than the priority of the highest priority task in the system. The DPCP assumes that higher numbers correspond to higher priorities. As in the single-node PCP, the priority ceiling of a local resource is the priority of the highest priority task that will ever access it. The priority ceiling of a global resource is the sum of the BP and the priority of the highest priority task that will ever access it. When a task executes a GCS, the task suspends itself on its local processor, and the GCS executes at a priority equal to the sum of the BP and the priority of the calling task on the host processor of the resource. Each processor in the system runs the PCP given the priorities and priority ceilings as described above.

The schedulability analysis of the DPCP is an extension of the schedulability analysis of the PCP. The main difference is that there are more forms of blocking due to access of remote resources. For instance,

the DPCP analysis must take into account blocking that occurs when a task requests a global resource on another node, but must wait for a lower priority task that currently holds the resource. An important difference between PCP and DPCP is that DPCP is usually harder to implement in actual systems because DPCP requires a common notion of priority across the distributed system. Also, analysis of DPCP requires knowledge of execution on all nodes in the system.

# 3 Research and Development in RT CORBA

In this section we present current work that has been done in the development of RT CORBA systems. We first describe in some depth three major RT CORBA research efforts that have been ongoing at: MITRE Corporation in Bedford, Massachusetts [11,12]; at the University of Rhode Island in conjunction with the US Navy SPAWAR Systems Center (SPAWARSYSCEN) in San Diego California and with Tri-Pacific Software Inc of Alameda, CA [14,15,16]; and Washington University in St. Louis[18,19,20,21]. We then discuss several other research projects that address certain specific features of RT CORBA systems. We end this section by describing several commercial ORBs that currently provide varying levels of real-time support. Most of the groups involved in the work described in this section have been part of the process of developing the RT CORBA 1.0 draft specification [10] described in Section 4. It is important to note that RT CORBA prototypes and products are increasing at a rapid rate. Discussing all of them is not possible. Therefore we discuss in this section some of the prevalent prototypes and products that exist at the time of this publication.

## *3.1 MITRE Corporation*

One of the first projects to incorporate expression and enforcement of end-to-end timing constraints into a CORBA system was designed by John Maurer and Bhavani Thuraisingham's group at MITRE in Bedford, MA [11]. This work identified requirements for the use of RT CORBA in command and control systems and prototyped the approach by porting the ILU ORB from Xerox to the Lynx real-time operating system. They then provided a distributed scheduling service supporting rate-monotonic and deadline-monotonic techniques. The resulting infrastructure, depicted in Figure 4, combines a POSIX-compliant real-time operating system, a real-time ORB, and an ODMG-compliant real-time ODBMS [12]. The MITRE system is an infrastructure for expressing and enforcing timing constraints in a CORBA system. It does not address real-time schedulability analysis. The MITRE prototype was designed for the US Airforce AWACS program, which in turn transferred it to Lockheed/Martin Corporation as an early basis for their HARDPACK commercial Real-Time CORBA system (see Section 3.5). MITRE has recently developed a mechanism for dynamic binding of clients to servers through a real-time version of the CORBA Trader Service [13].

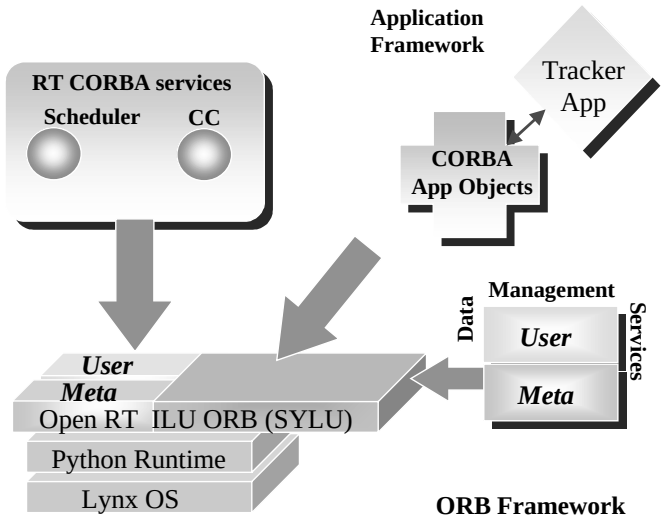


**Figure 4: MITRE's RT CORBA System**

## *3.2 University of Rhode Island & SPAWAR Systems Center & Tri-Pacific Software RT CORBA Research*

RT CORBA research at the University of Rhode Island and the SPAWAR Systems Center (SPAWARSYSCEN) includes both dynamic and static scheduling approaches. The dynamic RT CORBA research focuses on expression of many of the timing constraints discussed in Section 2.2.1 including deadlines, periods, and importance levels. It also focuses on best-effort enforcement of timing constraints in a RT CORBA system. URI & SPAWARSYSCEN's Real-Time CORBA research for static Real-time Systems has been developed by Tri-Pacific Software [14] of Alameda, CA into a product called *RapidSched*. This development is the basis of the current static scheduling service interface to the RT CORBA 1.0 draft specification [10]. RapidSched provides off-line schedulability analysis and efficient execution of the chosen scheduling policies. In this section we describe both of these research efforts.

### 3.2.1 Dynamic RT CORBA

URI & SPAWARSYSCEN's dynamic RT CORBA system [15] provides expression and best-effort end-to-end enforcement of soft real-time client method requests. The prototype implementation of this system extends Iona's Orbix ORB with real-time features.

The expression of timing constraints is made through *Timed Distributed Method Invocations* (TDMIs) that include timing information such as deadline, and priority. These timing parameters are packed into a `RT_Env` ("Real-Time Environment") structure that is passed along with every client request so that the ORB and object services can enforce the timing constraints.

The best-effort enforcement of timing constraints is provided through the extension of, or addition to several CORBA 2.0 Common Object Services. When a client makes a TDMI on a server, the Global Priority Service provides a uniform global priority based on the constraints specified in the TDMI. It then translates the global priority to a priority that the server's local operating system can handle. For instance, on a node controlled by the VXWorks real-time operating system with 256 local priorities, the Priority Service translates the wide range of global priorities that CORBA entities on the node have, to the 256 local priorities. The local real-time operating system then enforces real-time priority-based scheduling. The current system uses a variation of Earliest Deadline First (EDF) scheduling, but can easily be changed to support other priority assignment schemes.

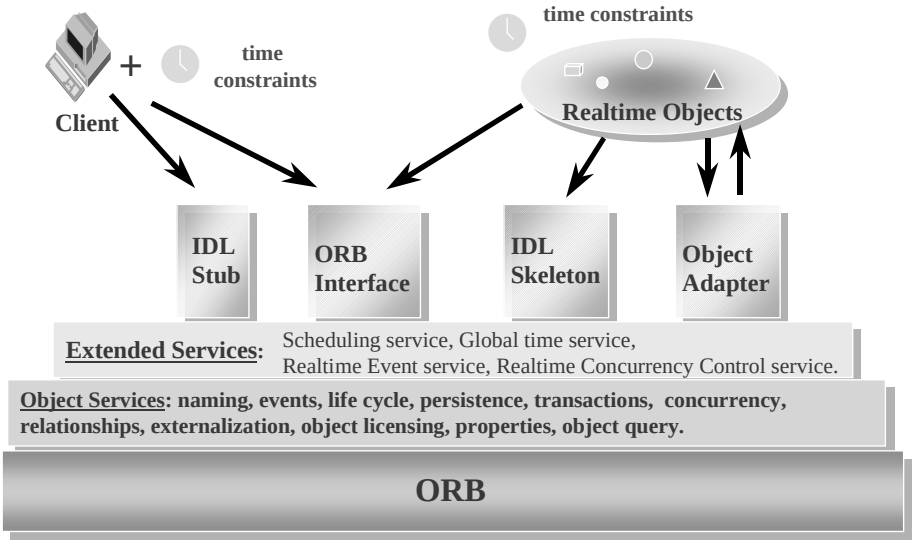


**Figure 5: URI & SPAWARSYSCEN RT CORBA**

The CORBA 2.0 Event Service has been extended for prioritizing delivery of real-time events. The real-time Event Service provides priority-based queues for its event channels. It also delivers the time that a particular event occurred so that a consumer can use the event to set relative timing constraints. Event priorities are based on the global priorities of the event's producer.

URI & SPAWARSYSCEN's dynamic RT CORBA system extends the CORBA 2.0 Concurrency Control Service to provide priority inheritance for requests that are queued on a server. When a TDMI requests a lock on a server's resource, the TDMI's execution priority is compared to those of all TDMI's holding conflicting locks on the resource. Conflicting TDMI's with lower priorities are raised to the requesting TDMI's priority, and the requesting TDMI is suspended.

The URI/SPAWARSYSCEN/TriPacific research group is currently developing a response to the Dynamic Scheduling RFP. The work described in this section will form the basis for the response.

### 3.2.2 Static RT CORBA

The work in RT CORBA static scheduling at URI & SPAWARSYSCEN & Tri-Pacific has produced four major results: (1) a graphic user interface for off-line analysis of RT CORBA systems; (2) a technique for mapping global priorities to local, system-specific priorities; (3) a scheduling service interface that has been adopted by the OMG for its RT CORBA 1.0 draft specification [10]; and (4) an implementation of the scheduling service interface. This product brings together all of the components discussed above to provide a Scheduling Service add-on for any commercial RT CORBA 1.0 compliant ORB.

**PERTS Front-End.** The RT CORBA research and development group at URI & SPAWARSYSCEN & Tri-Pacific has developed an extended version of the PERTS [144] real-time analysis tool that determines the schedulability of a RT CORBA system [16]. PERTS provides a graphical interface to allow users to enter real-time task information, such as deadline, execution time, and resource requirements. PERTS then computes a schedulability analysis on the given system using well-known techniques, such as rate-monotonic analysis [7]. PERTS was originally developed at the University of Illinois, Urbana-Champaign, and commercialized by Tri-Pacific Software. The group at URI & SPAWARSYSCEN & Tri-Pacific has developed a mapping from RT CORBA clients and servers to PERTS primitives – tasks and resources. A periodic client with *m* intermediate deadlines is mapped to *m* dependent tasks, each with the same period, and with deadlines corresponding to the intermediate deadlines of the client. Each server in the RT CORBA system is mapped to a PERTS resource. Network delay is treated as a single worst case parameter. This model of network delay allows for analysis, but is often quite pessimistic – Tri-Pacific is currently working on improved models of network delay. This allows users to enter RT CORBA constructs, and have PERTS automatically translate them into primitives that it can analyze. The extended PERTS analyzes the RT CORBA system using deadline monotonic scheduling and distributed priority ceiling protocol [9] for concurrency control. Given the real-time requirements of each client and server in the system, PERTS performs a multi-node scheduling analysis that takes into account execution time of clients and servers on all nodes, blocking times, network delay, and dependencies. If the system is found to be schedulable, the extended PERTS system produces priorities for each client task, and priority ceilings for each server resource in the system. If the system is found to be non-schedulable, PERTS produces graphs and other information for each client task to indicate what caused the system to be non-schedulable.

**Priority Mapping**. The output of the PERTS schedulability analyzer is a unique priority for each task in the system. It assumes an unlimited number of priorities. Unfortunately, in an actual distributed application, most operating systems do not have unlimited priorities. For instance, VXWorks [17] provides only 256 local priorities. The URI & SPAWARSYSCEN & Tri-Pacific research group has developed a technique for mapping the unlimited priorities produced by PERTS to the number of local priorities available on the systems involved. The algorithm, called the *Lowest Overlap First Priority Mapping Algorithm* [16], assigns multiple tasks to the same local priority, while ensuring the schedulability of the tasks in the system. The algorithm starts with a schedulable assignment of CORBA priorities in which a

certain number of CORBA priorities on one node must be mapped to the same local priority (due to more unique CORBA priorities required on the node than there are local priorities available). The algorithm tries combinations of overlapping CORBA priorities to the same local priority that would cause the system to remain schedulable. It accounts for the fact that mapping two CORBA priorities to the same local priority, which on most real-time operating systems are then scheduled first-come-first-serve, is a source of priority inversion. That is, a higher CORBA priority task may be forced to wait for a lower CORBA priority task due them both being mapped to the same local priority and the lower CORBA priority task arriving first. The algorithm attempts to overlap lower priority tasks on a node first. After each attempted overlap, the algorithm uses the PERTS schedulability analysis, enhanced with the ability to account for the additional priority inversion, to determine if the system with the overlap is schedulable. If it is not schedulable, the algorithm tries other overlap possibilities in increasing priority order. The Lowest Overlap First algorithm has been proven to be optimal under certain circumstances [16]. The extended PERTS system developed by the URI & SPAWARSYSCEN & Tri-Pacific research team has been augmented to implement this mapping algorithm, as well as several priority mapping heuristics [16] that are near-optimal and are computationally more tractable than the Lowest Overlap First algorithm. This enhanced version of PERTS now produces a specification of the local priorities at which each task will execute on its specific node.

**Scheduling Service**. The RT CORBA Scheduling service interface described later in Section 4.1.5 was developed by researchers at URI & SPAWARSYSCEN & Tri-Pacific and commercialized in Tri-Pacific's RapidSched product. RapidSched works with the extended PERTS system described above.

Recall that the extended PERTS produces a mapping of global priorities to local system priorities. PERTS also produces a second mapping of unique task names to global priorities and a third mapping of priority ceilings associated with unique names for each server in the system. These mappings are generated by PERTS as a set of configuration files that are read in by RapidSched when it is instantiated at system startup.

RapidSched currently implements deadline monotonic scheduling with DPCP for control of shared resources. All priorities and priority ceilings are computed *a priori* through PERTS, as described above. RapidSched uses interceptors to implement the PCP on each node. An interceptor is an ORB feature that provides an interface to allow application code to be executed in the internals of the ORB. RapidSched installs an interceptor that catches all calls to the object's methods. Before the method is executed and a result is passed back to the calling client, the interceptor executes the priority ceiling check; i.e. the priority of the client task is strictly higher than the highest priority ceiling of servers on the node that are locked by other tasks.

The objects of RapidSched are implemented as shared library code and are co-located with their respective clients and servers. Thus, there is no network delay for scheduling service calls, and inter-process communication on the same node is minimized. The scheduling objects communicate via shared memory,

mutexes, and condition variables to implement the concurrency control mechanism. Information about priority mapping is also stored in shared memory for fast run-time access.

The *RapidSched* techniques described here are useful for CORBA applications that have known execution characteristics and need a priori schedulability analysis, such as military command and control subsystems, and automated manufacturing control subsystems. Actual applications of this technology are described on Tri-Pacific's Web site [14].

## *3.3 Washington University – TAO*

Researchers at Washington University in St. Louis, have developed TAO (The ACE ORB) [18]. TAO is a high-performance, RT CORBA 2.0-compliant ORB that runs on a variety of operating system platforms with real-time features, such as VxWorks, Chorus, and Solaris. TAO's objective is to provide end-to-end Quality-of-Service (QoS) guarantees at multiple levels in the distributed system. The system consists of four major parts that carry out this objective: (1) the ORB; (2) the Scheduling Service; (3) the Event Service; and (4) the Real-Time I/O (RIO) subsystem.

### 3.3.1 TAO's ORB

TAO's ORB supports real-time by minimizing the necessary features required, and by optimizing features such as memory management, network protocols, and code generation. TAO's ORB Core is based on the ACE framework [18], which is a portable object-oriented middleware framework also developed at the Washington University. TAO uses ACE components to provide an efficient ORB Core that can be extended to adapt to new system environments and application requirements.

TAO's ORB Core supports a range of transport protocols, including a Real-Time Inter-ORB Protocol (RIOP) [18], that extends GIOP/IIOP with QoS attributes. RIOP is a mapping of GIOP that allows applications to transfer their QoS parameters end-to-end from clients to servants. Such attributes include priority, execution period, and communication class. For optimality, TAO's mapping can selectively omit transport layer functionality and run directly on top of ATM virtual circuits.

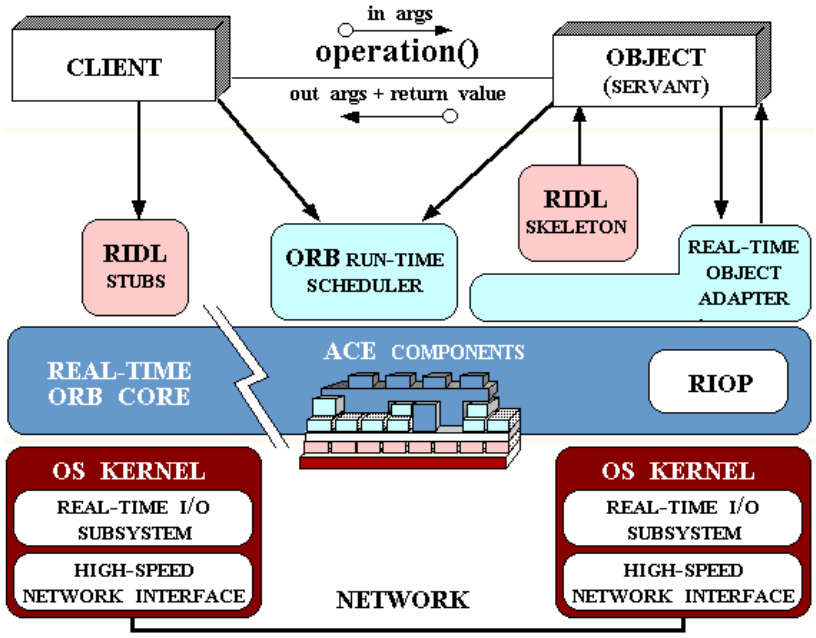


**Figure 6: The TAO System (from [21])**

TAO provides a Real-time Object Adapter (ROA) that can be configured to implement custom mechanisms that dispatch client requests according to application-specific real-time scheduling policies [18]. For instance, one of the strategies provided by the ROA is a variant of rate monotonic scheduling with real-time threads. TAO's ROA contains an object reference to the run-time scheduler, which dispatches client requests in accordance with a system-wide real-time scheduling policy. The run-time scheduler maps client requests to real-time thread priorities.

One of the key features of TAO's ORB is performance optimization. TAO optimizes the marshalling and demarshalling of operation parameters through Flick, an optimized IDL compiler [18]. TAO uses Flick to generate optimized client-side stubs and server-side skeletons from the IDL. Another optimization that TAO provides is memory management. Because dynamic memory management can be problematic for deterministic real-time systems, TAO is designed to minimize and eliminate data copying at multiple layers of the system.

### 3.3.2 TAO's Scheduling Service

TAO provides a Scheduling Service that guarantees the hard real-time QoS specifications of client requests [19]. The Scheduling Service supports both static scheduling, through off-line schedulability analysis, and dynamic scheduling, through policies such as admission control. There are two main components of the scheduling service: (1) the off-line schedulability analyzer; and (2) the run-time scheduler, which dispatches client requests through the ROA.

A TAO client expresses real-time attributes through an `RT_Info` structure for each of its schedulable operations and submits them to the scheduling service. The scheduling service examines the attributes of all registered operations, and performs schedulability analysis. If the system is found to be schedulable, the scheduling service assigns static priorities to the operations. If a dynamic scheduling policy is being used, static priority may be assigned initially according to an attribute such as operation criticality.

Once the priorities are assigned, the scheduling service determines the number and types of required dispatching queues, based on the number of required static priorities and the chosen scheduling strategy. At run-time, the ORB uses the run-time scheduler to retrieve the thread priority at which each queue dispatches operations, and the type of dispatching prioritization used by each queue. Each queue is associated with a static priority. When an operation request arrives from a client, the scheduling service dispatches it to the appropriate queue. If a dynamic scheduling policy is used, the scheduling service also determines the dynamic sub-priority at which the operation will run, within the already established static priority. TAO's scheduling service supports a variety of scheduling policies, including rate monotonic, and maximum urgency first [19].

### 3.3.3 TAO's Event Service

TAO's Event Service extends the Common Object Event Service specification to satisfy the QoS need of real-time applications [20]. Specifically, the event service uses real-time scheduling of CORBA events, instead of typical First-Come-First-Served scheduling to further support best-effort real-time scheduling. Consumers and suppliers specify their execution requirements and characteristics using QoS parameters. These parameters are integrated with the system-wide scheduling policy to determine priorities and preemption strategies.

TAO's event service provides filtering and correlation mechanisms that allow consumers to be more selective about which events they receive. Consumers are allowed to subscribe for a particular subset of events. The event service uses these subscriptions to filter supplier events, only forwarding them to interested consumers. Consumers can also specify AND and OR dependencies among the events that it will receive. For instance, a consumer can specify that it be notified only when all of the specified events have occurred. The event service also allows consumers to specify event dependency timeouts. A consumer can request to receive a timeout event if its dependencies are not met within some time period.

When a set of federated event channels are used in a system instead of a central event channel, TAO's event service provides a gateway servant to connect them. The gateway on a particular node subscribes to all of the events (and only the events) that are of interest to its consuming event channel.

### 3.3.4 TAO's Real-Time I/O Subsystem

TAO's real-time I/O subsystem (RIO) runs in the OS kernel. At the Core of the RIO subsystem is a "daisy-chained" network interface consisting of one or more ATM Port Interconnect Controller (APIC) chips. This is effective at optimizing it for ATM systems, but is limiting in its applicability to other systems. The RIO subsystem uses a thread pool to send and receive requests to and from clients across high-speed networks or I/O backplanes [21]. The RIO is built as an extension of the STREAMS communication I/O subsystem on Solaris. It focuses on alleviating key sources of priority inversion that are present in Solaris' I/O subsystem. For instance, in Solaris, thread-based priority inversion can occur when real-time threads depend on kernel processing that is performed a lower priority levels.

The RIO subsystem exploits the early demultiplexing feature of ATM to help alleviate packet-based priority inversion. It uses a packet classifier to place packets into priority-based queues. The RIO subsystem reduces thread-based priority inversion by vertically integrating packets received at the network interface with the corresponding thread priorities in the ORB Core. TAO schedules all protocol processing through kernel threads that are scheduled at the appropriate real-time priorities. Rather than asynchronously processing packets without regard to priority, the RIO subsystem provides a dedicated stream connection path that allocates buffers in the ATM driver and associates kernel threads with real-time priorities for protocol processing.

The TAO project represents a major effort towards meeting the requirements of the RT CORBA specification. The high-performance ORB and I/O subsystem provide for fast, efficient processing of client requests. The scheduling service provides a mechanism for expressing and enforcing QoS requirements of clients, in both static and dynamic scheduling environments. The event service provides mechanisms to streamline the delivery of events to interested consumers, and to allow consumers to specify and enforce timing requirements

## *3.4 Other Academic Work*

Several other ongoing research projects address selected features of RT CORBA systems. The Realize project at University of California, Santa Barbara [22] is a middleware system for use in standard operating systems, that can be used in conjunction with a CORBA ORB. Client requests are intercepted by Realize, encapsulated in multicast messages, and communicated to other processors. Realize is responsible for real-time scheduling, synchronization, distribution and replication of information, consistency of replicated information and fault recovery. An aim of Realize is to reduce the difficulty of developing real-time systems, and to allow the programming of distributed real-time programs as simple as single-node real-time programs.

At the University of Illinois, Urbana-Champaign, a project has been initiated that extends the scheduling service of TAO to allow for dynamic schedulability analysis through admission control [23]. For each

server in the system, a scheduling broker is instantiated. When a dynamically created client makes a request on the server, the scheduling broker performs schedulability analysis to ensure that admission of this request will not adversely affect the schedulability of any admitted requests. If the schedulability test fails, the client's request is rejected.

## *3.5 Commercial ORBS*

Several commercial CORBA implementations provide features that make them suitable for use in real-time applications. They include ChorusORB [24] from Sun Microsystems, ORBexpress [25] from OIS, and HARDPACK [26] from Lockheed Martin. Other CORBA products have been ported to real-time operating systems, although the ORB's themselves are not specifically engineered for real-time. They include Orbix from Iona Technologies [27] and Visibroker from Inprise [28].

**ChorusORB**. ChorusORB has many features that make it appropriate for real-time systems. It runs on Chorus/ClassiX, Sun's embedded real-time operating system, which provides deterministic scheduling and efficient inter-process communication. ChorusORB supports both IIOP and a fast, lightweight, proprietary protocol for use between ChorusORB clients and servers. ChorusORB daemon consumes only about 140Kb of memory, achieves very low message latencies, and is highly optimized; for example, it automatically maintains thread pools for every server in order to make dispatch more efficient. It also provides a set of "Interceptor" API's that allow programmers to install routines that are called at certain stages of every invocation. This enables, for instance, the implementation of a transparent scheduling and concurrency-control policy, which is easily maintained and kept consistent throughout the system. In addition to Chorus/ClassiX, ChorusORB runs on a number of POSIX-compliant operating systems and on a wide range of hardware. It has language bindings to C++ and Java and is used primarily in applications related to telecommunications.

**ORBExpress**. Objective Interface Systems (OIS) produces a CORBA implementation called ORBexpress Real-Time that offers many promising features. It runs on WindRiver's VxWorks, one of the most widely used embedded real-time operating systems. ORBexpress Real-Time provides a distributed priority inheritance mechanism based on the priority ceiling protocol, which bounds priority inversion and eliminates deadlock. It also comes with a memory allocation mechanism optimized for real-time, which OIS claims runs four to six times as fast as conventional memory managers. One major problem that ORBexpress addresses is the unpredictability of transport protocols such as TCP/IP; its "pluggable protocol" interface allows other vendors and application programmers to install their own communications protocols, which may be based on shared memory, ATM networks, VME buses, or other deterministic transports. ORBexpress has language bindings to Ada-95 and C++ and is used in a variety of applications, including vehicle navigation and Command and Control systems.

**HARDPACK**. Lockheed Martin's HARDPACK has been designed to run in hard real-time defense-related applications. It provides priority-based scheduling and concurrency control with priority inheritance. It offers an interface to set periods and deadlines for tasks, including a notification mechanism for overruns. It uses the notion of a "priority transform" to allow a global ordering of tasks that maintains consistency across nodes in spite of limited-priority networks and operating system schedulers. It also has a "flexible bind" interface, which allows applications to specify when and how connections are made and broken. Like ORBexpress, it allows various communication protocols to be employed through a common interface. HARDPACK has language bindings to C, C++, Ada-95, and Ada-83. HARDPACK is being used in the US Air Force and NATO-AWACS platforms. It is being transitioned to a separate company for commercial release.

**Other ORBS**. In addition to the CORBA products that are specifically engineered for use in real-time systems, several other ORB's have been ported to run on real-time operating systems. For example, Highlander Communications has ported Inprise's Visibroker to both VxWorks and Integrated Systems' pSOSystem. Visibroker is a popular, general-purpose CORBA product that has been implemented with some efficiency considerations, such as multi-threading and support for load-balanced binding. Iona's has also ported its Orbix ORB to real-time operating systems, most notably VxWorks. Like Visibroker, Orbix is a general-purpose CORBA implementation that is used in a number of commercial applications. Orbix also has multi-threading support and a "Filter" interface similar to ChorusORB's "Interceptor" mechanism. By running on efficient real-time operating systems like VxWorks, these ORB's can reduce memory consumption, message latencies, and priority inversion, which makes them more useful in time-critical applications.

Clearly, there are a significant number of commercial CORBA products that are suitable to some degree for use in real-time systems. While some are more appropriate solutions than others, none offers an overall end-to-end guarantee of performance. Furthermore, until a real-time CORBA specification is adopted, any real-time features will be non-standard, non-portable, and hence not truly in the spirit of CORBA.

# 4 Real-Time CORBA Standard

The research and development described in Section 3 indicated that there was sufficient interest and results in RT CORBA to incorporate real-time support into the CORBA standard. The OMG Real-time SIG was formed in 1996. It is comprised of a diverse group from CORBA vendors, end users, government research labs and academia. It has forged close working relationships with other groups in the OMG including the Telecommunication Domain Task Force, the Command and Control working group, and the Security SIG. The RT SIG first produced a whitepaper [29] in 1996 outlining the concepts behind RT CORBA. Based on

these concepts, the RT SIG produced a series of Requests For Proposals (RFPs) in three areas of RT CORBA:

1. *Minimal CORBA* – this RFP requested the specification of a minimal set of CORBA features to allow ORBs that are efficient in execution time and in resource requirements. The RFP was issued in 1996 and responses to the RFP were received in 1997 [30].

2. *Fixed Priority Scheduling* – this RFP requested the specification of the main RT CORBA functionality under the assumption that priorities in the system are fixed. The RFP was issued in late 1997 and the responses were received in 1998, and synthesized into a standard in 1999 [10].

3. *Dynamic Scheduling* – this RFP requests the specification of RT CORBA that can be integrated with the Fixed Priority RT CORBA while including support for dynamic distributed real-time scheduling. The RFP was issued in February 1999, and responses are due in October 1999.

The SIG has issued, or plans to issue, RFPs in other areas such as fault-tolerance and high-performance CORBA as well.

In this section we review the draft RT CORBA standard for Fixed Priority Scheduling in detail since it is the basis for the major parts of the RT CORBA standard. We also describe the requirements for the extension to Dynamic Scheduling in RT CORBA. It is important to note that the standards are evolving. In this section we describe the standard as it exists at the time of this writing.

## *4.1 Fixed Priority RT CORBA*

The RT CORBA RFP for Fixed Priority RT CORBA required that vendors submit proposals that included extensions to the CORBA standard in at least the following areas:

1. *Priority* – define what priority means in a RT CORBA system and define an interface that allows these priorities to be expressed and enforced.

2. *Bounding Priority Inversion* – define interfaces that support mechanisms that bound priority inversion in a CORBA environment.

3. *Protocol Selection* – define interfaces that allow the selection of protocols and protocol properties for client/server communication.

The vendors were to demonstrate that these interfaces supported Fixed Priority Scheduling for enforcement of end-to-end timing constraints in a RT CORBA system. They were also to describe the assumptions and dependencies on the underlying distributed system, such as operating systems and networks. The commercial efforts described in Section 3.5 were among the major respondents to the RFP in January 1998. In the remainder of 1998 these companies, and others including Tri-Pacific Software, the University of

Rhode Island and Washington University academic groups, SPAWAR Systems Center and MITRE research labs, collaborated to create a single RT CORBA Fixed Priority Scheduling draft standard, which we describe here.

### 4.1.1 Fixed Priority RT CORBA Model

The RT CORBA model considers five components:

1. The Real-Time Operating System, which is assumed to use priority-based scheduling, such as that specified in the POSIX real-time operating system standard.
2. The Real-Time ORB, which provides real-time primitives for client/server interaction.
3. An optional Scheduling Service, which uses the primitives of the RT ORB to achieve a uniform scheduling policy in the CORBA system.
4. The network.
5. The application clients and servers.

The RT CORBA standard specifies real-time support only in the ORB and Scheduling Service parts of the model, with appropriate interfaces provided to the application code. The RT CORBA standard assumes priority-based scheduling capabilities in the operating systems on nodes in the system and does not assume any real-time capabilities in the network. That is, the RT CORBA specification only addresses real-time issues in the scope of the CORBA software. While support in the CORBA software is necessary for a complete distributed real-time solution, it is not sufficient because all parts of the system must be designed for real-time to get sufficient real-time support.

RT CORBA defines a *thread* as its schedulable entity. The RT CORBA notion of thread is consistent with the POSIX definition of threads [31]. At an instant of time, a thread has two priorities associated with it: a *CORBA Priority* and a *Native Priority*, similar to the notion of global and local priorities described in the University of Rhode Island & SPAWARSYSCEN work described in Section 3.2.

The CORBA Priority of a thread comes from a universal node-independent priority ordering of threads used throughout the CORBA system. This is done so that CORBA Priority is a meaningful priority order that spans nodes in the system. All priorities expressed in client and server application code, including those specified in RT CORBA Scheduling Service calls, are CORBA Priorities.

The Native Priority of a thread is the priority used by the underlying system (operating systems and network). For instance, a CORBA thread with a CORBA Priority 300 that needs to execute on a node managed by the VXWorks [17] real-time operating system must be assigned one of the operating system's 256 native priorities. RT CORBA uses the notion of a *Priority Mapping* to map the CORBA Priority of

thread to the Native Priority it needs to execute on the underlying system. RT ORBs come with a default priority mapping algorithm, but RT CORBA also specifies a `install_priority_mapping` method to allow application code, or the Scheduling Service (see Section 4.1.5), to install its own mapping. The ability to install a known priority mapping is important since the mapping of several CORBA Priorities to the same Native Priority is a source of priority inversion that often must be accounted for in real-time analysis. See Section 3.2.2 and [16] for a discussion of an optimal priority mapping algorithm, its resulting priority inversion, and its affect on schedulability analysis.

### 4.1.2 RT CORBA in Clients

Clients express CORBA priority in their threads by creating a local object called a `CORBA::Current`. When created using the RT CORBA interface, the Current object contains, in addition to other attributes, a *Priority* attribute, which will hold the CORBA Priority of the thread that created the Current object. A thread sets its CORBA Priority by writing the *Priority* attribute of its Current object. The RT ORB will map this CORBA Priority to the Native Priority on the local real-time operating system to execute the thread. The RT ORB also has access to the CORBA Priority in the Current object so that it can do things such as propagate the CORBA Priority to any CORBA servers that the thread calls.

RT CORBA also specifies a mechanism for clients to do *explicit binding* to servers. *Binding* refers to the client obtaining a reference to a CORBA object (similar to a pointer to a local object), with which it can invoke methods on the CORBA object. CORBA has several interfaces to allow clients to bind to CORBA servers. RT CORBA adds the explicit bind mechanism to allow the clients to specify the network protocols they wish to use, most importantly to allow the use of real-time protocols. Currently no real-time protocols are available in the RT CORBA standard, but protocols such as TAO's RIOP [18] are expected to be included in later drafts. The explicit bind capability in RT CORBA also allows clients to establish a dedicated connection to the server to alleviate blocking at the connection point. Explicit binding can also establish *priority bands* for shared connections to the server. Priority bands come from the TAO work described in Section 3.3. They are used to reduce the priority inversion due to non-priority respecting protocols. Essentially, priority bands allows the client to establish multiple connections to a server each with different CORBA priorities.

### 4.1.3 RT CORBA Mutexes

RT CORBA provides a *Mutex* as a means to coordinate access to resources in the CORBA system. The Mutex interface looks much like an operating system Mutex from the POSIX standard [31]. There are methods to `create_mutex, lock, unlock,` and `try_lock.` The `try_lock` method includes a parameter to indicate a maximum wait time. A thread calling `lock` on an unlocked Mutex will obtain the lock. All subsequent threads that request the locked Mutex will be blocked by queuing them in CORBA Priority order. When the thread holding the lock calls `unlock,` the highest priority blocked thread

obtains the lock. Note that Mutexes are CORBA-wide entities allowing threads on different nodes to coordinate access to system resources.

The RT CORBA Mutex requires priority inheritance behavior (see Section 2.2.4). Exactly what form of priority inheritance is used is implementation-dependent, but the application can be assured that at least basic priority inheritance of CORBA priority will take place among block threads. Other forms of priority inheritance protocols, like a form of Priority Ceiling Protocol require interfaces that are specific to the RT CORBA implementation.

### 4.1.4 RT CORBA For Servers

CORBA is designed to allow servers to be written independently of the clients that will access them. A CORBA *server* can be thought of as a process in which reside the CORBA objects that clients use. That is, a CORBA object provides methods that perform the service for the client; the server is the process that contains the CORBA objects and assigns threads to invoke the methods of the CORBA objects on the client's behalf. Although not mandated in the standard, most real-time CORBA implementations will keep persistent servers so that servers are always active. Otherwise, having to activate a server could adversely affect timing constraint enforcement. When the server creates/assigns a thread to invoke a method on a CORBA server on the client's behalf, the server is said to *dispatch* a thread. RT CORBA provides primitives to control the dispatching of server threads; it does not provide primitives to use in the application code of CORBA objects themselves.

A CORBA server process uses one or more objects called *Portable Object Adapters* (POAs) to manage the creation/deletion of CORBA objects and the dispatch of threads. When a request comes to a CORBA server process, the server uses information in the request to find the appropriate POA to dispatch the servant to handle the request. A CORBA POA object is constructed with a set of *policies* specified in the CORBA standard. These policies specify how the POA creates/deletes objects and dispatches threads for the objects that it manages. RT CORBA server primitives are written entirely in terms of additional POA policies for:

1. *Thread configuration* - specifies the creation of *thread pools* from which to dispatch threads;
2. *Server priority* - specifies how to determine at what CORBA priority the server thread should execute;
3. *Communication protocols* - used by the server to accept clients calls to the CORBA objects;

The POA policies are parameterized and also can make use of system information to allow the POA to make specific dispatching decisions. For instance, POA objects have access to the priority of the client when a method request arrives. This information can be used by the POA to establish priorities of the servant threads and to make concurrency control decisions about when to dispatch a request.

**Thread Configuration RT CORBA POA Policy.** Recall that for each invocation of a CORBA object by a client, a thread on the server is required to process the request on the client's behalf. Non-threaded CORBA systems typically have a single process (one thread) that handles invocation requests serially. Serial handling of requests has obvious drawbacks for real-time due to lack of concurrency that causes increased blocking. RT CORBA assumes a multi-threaded server. Each POA has a *thread pool* associated with it, similar to the Washington University TAO RT CORBA design [18]. The POA thread pool characteristics are defined by a RT CORBA POA "Thread Configuration Policy". This policy specifies that a thread pool consists of *statically allocated threads*, which are threads created when the POA is instantiated at the time the server starts; and *dynamically allocated threads*, which are threads that are created at the time the client's request is received. Note that in general, dynamically allocated threads can cause unpredictable real-time performance. The thread pool POA policy specifies the number of static and dynamic threads through two parameters to the policy: *static_threads***,** which specify the number of statically allocated threads, and *max_threads*, which specifies the maximum number of threads that can be active in the POA (statically allocated threads plus dynamically allocated threads). If the server sets *static_threads* to be equal to *max_threads* in the Thread Configuration POA policy, then no threads will be created dynamically. Note that the ability to limit the maximum number of threads created and to control how many are pre-allocated are both important for bounding resource use, which is important for achieving predictability in real-time systems.

A POA's thread pool is created in *priority lanes***.** Priority lanes are groups of threads within the POA's thread pool that are created at the same priority. Each priority lane has its own *static_threads* and *max_threads* parameters to indicate how many threads of that priority a POA will have. This scheme allows the original priority of threads within the server to be set. The "Server Priority Model" POA policy, described next, allows this original default priority of a thread to change to account for the priority of the client on whose behalf it will execute and to account for priority inheritance.

**Server Priority RT CORBA POA Policy.** The thread pool priority lanes establish the original priority of threads before they are dispatched to service a client's request. The Server Priority POA policy specifies what priority the thread will execute at after it has been dispatched. There are two models currently specified in RT CORBA: "Client Priority Propagation", where the server thread executes at the CORBA priority of the client that requested it; and "Server-Set Priority", where the server thread executes at a priority set as a parameter to the policy.

**Communication Protocols RT CORBA POA Policy.** The POA policy for communication protocols allows the server to specify a prioritized list of protocols it wishes to use to communicate with clients. This list is placed in the object reference that is returned to a client when the client binds to a server. The client's configurable protocol capability, described in Section 4.1.2, can then use this list to choose a protocol with which to communicate with the server. The RT CORBA standard specifies the definition of TCP as a protocol selection by providing an interface for the server to set TCP's configurable properties

such as the send buffer size and receive buffer size. Currently, TCP is the only property specified. However, the POA policy for protocol selection was put into RT CORBA so that real-time protocols, that for instance support priority, can be configured and used once standard ones become available.

RT CORBA 1.0 also provides a RT POA policy for *Priority Banded Connections.* This policy allows multiple clients to have multiple connections at different priorities to the same server. A server's POA establishes ranges of priorities, each range called a *band.* A client is connects to the server in the band in which the client's CORBA priority falls. If the client's CORBA priority is not in a band specified by the POA, the client's connection is refused by the server. Priority banding is provided to reduce priority inversion that occurs in non-real-time ORBs where all clients connect at the same priority allowing higher priority clients to be queued behind lower priority clients waiting on the single connection point (such as a TCP/IP socket).

### 4.1.5 Fixed Priority Scheduling Service

RT CORBA also specifies a *Scheduling Service* that uses the RT CORBA primitives to facilitate enforcing various fixed-priority real-time scheduling policies across the RT CORBA system. The Scheduling Service abstracts away from the application some of the complication of using low-level RT CORBA constructs, such as the POA policies. For applications to ensure that their execution is scheduled according to a uniform policy, such as global Rate Monotonic Scheduling, RT ORB primitives must be used properly and their parameters must be set properly in all parts of the RT CORBA system. A Scheduling Service implementation will choose CORBA Priorities, POA policies, and priority mappings in such a way as to realize a uniform real-time scheduling policy. Different implementations of the Scheduling Service can provide different real-time scheduling policies.

The Scheduling Service uses "names" (strings) to provide abstraction of scheduling parameters (such as CORBA Priorities). The application code uses these names to specify CORBA Activities and CORBA objects. The Scheduling Service internally associates these names with actual scheduling parameters and policies. This abstraction improves portability with regard to real-time features, eases use of the real-time features, and reduces the chance for errors.

The Scheduling Service provides a `schedule_activity` method that accepts a name and then internally looks up a pre-configured CORBA priority for that name. The Scheduling Service also provides a `create_POA` method to create a POA and set the POA's RT CORBA thread pool, server priority, and communication policies to support the uniform scheduling policy that the Scheduling Service is enforcing. For instance, if the Scheduling Service were enforcing a scheduling policy with priority ceiling semantics, it might create thread pools with priority lanes at the priority ceiling of the objects it manages to ensure that threads start at a high enough priority before dispatch. The Scheduling Service provides a third method,

```
0       install_priority_mapping(. . .);

Client
C1    sched = create scheduling service object;
C2    obj = bind to server object
C3    sched->schedule_activity ("activity1");
C4    obj->method1( params );   // invoke the object
C5    sched->schedule_activity ("activity2");
C6    obj->method2(params );

Server Main
S1    sched = create scheduling service object;
S3    poa1 = sched->create_POA(. . .);
S4    obj = poa1->creat_object ( params );   // create object
S5    sched->schedule_object(obj, "Object1" );
      ...
```

**Figure 7: Example of RT CORBA Static Scheduling Service**

called `schedule_object`, that accepts a name for the object and internally looks up scheduling parameters for that object. For instance, it could set its priority ceiling so that it can do a priority ceiling check at dispatch time.

The example in Figure 8 illustrates how the Scheduling Service could be used and also illuminates some of the issues in creating RT CORBA clients and servers. Assume that a CORBA object has two methods: `method1` and `method2`. A client wishes to call `method1` under one deadline and `method2` under a different deadline.

In Step 0, the Scheduling Service installs a priority mapping that is consistent with the policy enforced by the Scheduling Service implementation. For instance, a priority mapping for an analyzable Deadline Monotonic policy might be different than the priority mapping for an analyzable Rate Monotonic policy.

The `schedule_activity` calls on lines `C3` and `C5` specify names for CORBA Activities. The Scheduling Service internally associates these names with their respective CORBA priorities. These priorities are specified when the Scheduling Service is instantiated at system startup. For instance, the URI & SPAWARSYSCEN & Tri-Pacific RapidSched Scheduling Service implementation [16] specifies deadline monotonic priorities through a configuration file.

The server in the example has two Scheduling Service calls. The call to `create_POA` allows the application programmer to set the non-real-time policies, and internally sets the real-time policies to enforce the scheduling algorithm of the Scheduling Service. The resulting POA is used in line `S4` to create the object. The second Scheduling Service call in the server is the `schedule_object` call in line `S5`. This call allows the Scheduling Service to associate a name with the object. Any RT scheduling

parameters for this object, such as the priority ceiling, are assumed to be internally associated with the object's name by the Scheduling Service implementation.

### *4.2 Dynamic RT CORBA*

The current RT CORBA draft standard was designed for supporting fixed priority scheduling and some general real-time features. It was not designed to support *dynamic scheduling*, where clients and servers dynamically enter and exit the system and priorities may change over time. The OMG RT SIG has issued a request for proposals (RFP) in February 1999 to extend the RT CORBA standard to support dynamic scheduling. The RFP specifies requirements for what the eventual RT CORBA standard must contain to support dynamic scheduling.

The Dynamic RT CORBA standard will support the specification of *policy* and *parameters* to be used to schedule each task. That is, instead of each task carrying with it a CORBA Priority, as is required in the current RT CORBA standard, tasks will carry a richer description that includes parameters, such as deadline, "importance", delay, and period. The task will also carry a policy, such as Earliest-Deadline-First (EDF), or EDF weighted by importance, that is to be used to schedule it. The exact policies supported in the Dynamic RT CORBA standard will be determined from the policies suggested in responses from CORBA vendors to the RFP. The Dynamic RT CORBA standard will also support handling occurrences when the policy is violated (this is called "policy inversion", which is similar to the fixed priority "priority inversion" notion). Also, handling of violations of scheduling parameters, such as missing deadlines, will be supported. Finally, the Dynamic RT CORBA standard will state its relation to the current RT CORBA standard's support for fixed priority scheduling. A draft Dynamic RT CORBA standard expected in early 2000.

Although the Fixed Priority Real-Time CORBA 1.0 and the Dynamic additions that will be in CORBA 2.0 will constitute the main part of what will be Real-Time CORBA, there are other extensions on the OMG RT SIG roadmap. These include: relation to fault tolerance, real-time transactions, and high-performance CORBA.

## 5 Conclusions

This paper has described the early stages of research and development in Real-Time CORBA. The original work done at MITRE, the University of Rhode Island & SPAWAR Systems Center, and Washington University laid the groundwork to incorporate results from the real-time research community into the CORBA standard. These results, along with parallel development efforts by companies such as Sun, Lockheed/Martin, and Objective Interface Systems, demonstrated the feasibility of RT CORBA software. This validation, along with significant demand from application domains such as military command and control, telecommunications, manufacturing, and finance, caused the Object Management Group to initiate

a standardization process in 1996. In 1998 the first draft RT CORBA standard was issued. Vendors such as Sun, Lockheed/Martin, Objective Interface Systems, Iona, and Visigenics have all committed to producing middleware software that meets the RT CORBA standard. Other vendors such as Tri-Pacific Software Inc have committed to making tools to support RT CORBA developers. The flexibility and sound structure of the CORBA standardization process, along with the willingness of the OMG's RT SIG to incorporate promising research results into the process have created an effective, useful combination of standard, middleware implementations, techniques, and tools to address real-time distributed object computing in a portable, commercially available way under the Real-Time CORBA name.

**Acknowledgments.** Many people have contributed indirectly to this paper due to their ground-breaking work in developing Real-Time CORBA; we mention some of them here. Peter Krupp of Iona Technologies was one the originators of the concepts for RT CORBA while at MITRE Corporation. He also served as the first co-chair of the OMG's RT SIG. Dock Allen of Computing Device's International has been co-chair of the RT SIG since its inception and has been primarily responsible for the fast progress of the standard effort. Douglass Locke of Lockheed/Martin and E. Douglas Jensen of MITRE were instrumental in drafting both RT CORBA RFP's. Douglas Schmidt and David Levine and their group at Washington University in St Louis produced the TAO system from which many of the RT CORBA techniques were derived. John Black, Michael Squadrito, Roman Ginis, Steve Wohlever, Igor Zykh, Levon Esibov, Rama Bethmangalkar, and Sean White all did significant work on the RT CORBA system at the University of Rhode Island, with the co-authors of this paper, to help produce important results. Ben Watson of Tri-Pacific Software led the original development of RapidSched. The following drafters of the RT CORBA draft standard did excellent work in bringing the requirements and techniques together: Jonathan Currey and Ken Black of Highlander Communications, Michel Ruffin of Alcatel, Jishnu Mukherji of Hewlett-Packard, Oisin Hurely of Iona Technologies, Tom Barker of Lockheed/Martin, Judy McGoogan of Lucent Technologies (who has also served as co-chair of the RT SIG), Dave Stringer of Nortel Networks, Bill Beckwith of Object Interface Systems, and Michel Gien of Sun Microsystems.

## References

[1] The Open Group, *Distributed Computing Environment*. Electronic document: http://www.camb.opengroup.org/tech/dec.

[2] P.E. Chung, Y. Huang, S. Yajnik, D. Liang, J. Shih, C. Wang, and Y. Wang, *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer*. web document: http://www.bell-labs.com/~emerald/dcom_corba/Paper.html.

[3] Object Management Group, http://www.omg.org.

[4] OMG, *CORBAServices: Common Object Services Specification.* OMG, Inc., 1996.

[5] K. Seetharaman, editor. "The CORBA Connection" (series of articles). *Communications of the ACM* 41(10) pp. 34-73; Oct. 1998.

[6] John Stankovic and Krithi Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," *ACM Operating Systems Review*, no. 3, vol. 23, pp. 54-71, July 1989.

[7] C. Liu and J. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 30, pp. 46-61, January 1973.

[8] J. W.-S. Liu, *Real-Time Systems*. To be published by Prentice-Hall, March 2000.

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

[10] OMG, *Realtime CORBA*. Electronic document at http://www.omg.org/docs/orbos/98-10-05.pdf.

[11] P. Krupp, A. Schafer, B. Thuraisingham, and V.F. Wolfe, "On Real-Time Extensions to the Common Object Request Broker Architecture," In *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) '94 Workshop on Experiences with CORBA*, Sept. 1994

[12] E. Bensley, et. al., "Object-Oriented Approach for Designing Evolvable Real-Time Command and Control Systems," In *Proceedings of the Workshop on Real-Time Dependable Systems*. Feb. 1996.

[13] S. Wohlever et al., "Adaptive Distributed Real-Time Object Management For Command and Control Systems: Volume II," MITRE Technical Report MTR 98B0000067; Sept 1998. MITRE Corporation, Bedford, Ma.

[14] Tri-Pacific Software Inc., at http://www.tripac.com.

[15] L. DiPippo, V.F. Wolfe, R. Johnston, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, "Expressing and Enforcing Timing Constraints in a Dynamic Real-Time CORBA System," *Real-Time Systems*, vol. 16, issue 2/3, May 1999.

[16] L. DiPippo, V.F. Wolfe, L. Esibov, G. Cooper, R. Johnston, B. Thuraisingham, J. Mauer, "Scheduling and Priority Mapping for Static Real-Time Middleware," *Real-Time Systems* special issue on real-time middleware, to appear.

[17] WindRiver Systems, http://www.wrs.com/.

[18] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications Journal*. Summer 1997.

[19] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems* special issue on real-time middleware, to appear.

[20] Timothy H. Harrison, Carlos O'Ryan, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," Submitted to *IEEE Journal on Selected Areas in Communications*. available at http://www.cs.wustl.edu/~schmidt/TAO.html.

[21] Douglas C. Schmidt, Fred Kuhns, Rajeev Bector and David L. Levine, "The Design and Performance of an I/O Subsystem for Real-time ORB Endsystem Middleware," Submitted to *Real-Time Systems*. available at http://www.cs.wustl.edu/~schmidt/TAO.html.

[22] V. Kalogeraki, P.M. Melliar-Smith, L.E. Moser, "Soft Real-Time Resource Management in CORBA Distributed Systems," In *Proceedings of the 1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, December 1997.

[23] W. Feng, U. Syyid and J. W.-S. Liu, "Providing for an Open, Real-Time CORBA," In *Proceedings of the 1997 IEEE Workshop on Middleware for Distributed Real-Time Systems and Services*, San Francisco, CA, December 1997.

[24] Chorus Systems, *Chorus/COOL-ORB R3 Product Description*. Technical Report CS/TR-95-157.3. June 1996.

[25] Objective Interface Systems, Inc., *Realtime CORBA (response to OMG RFP).* OMG TC Document orbos/98-01-15. January 1998.

[26] Lockheed Martin Federal Systems, Inc., *Realtime CORBA (response to OMG RFP).* OMG TC Document orbos/98-01-13. January 1998.

[27] Iona Technologies, Northern Telecom., *Realtime CORBA Extensions (response to OMG RFP).* OMG TC Document orbos/98-01-09. January 1998.

[28] Visigenic Software, Inc. and Highlander Communications, L.C., *Realtime CORBA (response to OMG RFP)* OMG TC Document orbos/98-01-14. January 1998.

[29] The Realtime Platform Special Interest Group of the OMG, *CORBA/RT Whitepaper*. Electronic document: http://www.omg.org/docs/realtime/96-12-01.doc. December 1996.

[30] OMG, *Real-Time Special Interest Group's Request For Proposals*. Electronic document at http://www.omg.org/docs/realtime/97-05-03.txt.

[31] IEEE, *IEEE Standard Portable Operating System Interface for Computer Environments (POSIX) 1003.1*, IEEE, New York, 1990.