Developing Standard Real-time CORBA Applications using the TAO Object Request Broker

A Proposal for a TAO Real-time CORBA Developers Guide

Douglas C. Schmidt
schmidt@uci.edu
Electrical and Computer Engineering Dept.
University of California, Irvine, 92697

David Levine
levine@cs.wustl.edu
Computer Science Dept.
Washington University, St. Louis, MO 63130

Abstract

To write, maintain, and optimize efficient, predictable, and scalable real-time systems, developers of real-time applications must know how to use the quality of service (QoS) features defined by CORBA specifications for real-time and asynchronous messaging. The proposed *Real-time CORBA Developers Guide* described in this document will explain these specifications in detail, as well as provide many concrete programming examples that illustrate how to apply the features of the Real-time CORBA and CORBA Messaging specifications effectively in practice. All the examples in this document will be implementable in the TAO real-time ORB. Thus, real-time application developers can use this developers guide to learn about advanced CORBA features, understand good design techniques and patterns for writing and optimizing real-time CORBA applications, as well as get hands-on practice writing such applications using TAO.

1 Introduction

A growing class of real-time systems require end-to-end support for various quality of service (QoS) aspects, such as bandwidth, latency, jitter, and dependability. These systems include command and control systems [1], manufacturing process control systems, videoconferencing [2], large-scale distributed interactive simulations [3], and testbeam data acquisition systems [4, 5]. In addition to requiring support for stringent QoS requirements, these systems have become *enabling technologies* for companies in markets where deregulation, global competition, and budget restrictions necessitate increased software productivity and quality.

Requirements for increased software productivity and quality motivate the use of *distributed object computing (DOC) middleware*, such as CORBA [6], which is an industry standard being defined by the Object Management Group (OMG). DOC middleware resides between applications and the underlying operating systems, protocol stacks, and hardware in complex real-time systems [7]. CORBA helps to decrease the cycle-time and effort required to develop high-quality systems by composing applications out of reusable software component services, rather than building them entirely from scratch.

Over the past two years, the use of CORBA middleware has increased significantly in domains, such as aerospace, telecommunications, medical systems, and distributed interactive simulations, that are characterized by stringent QoS requirements. The acceptance of CORBA in these domains stems from the following two factors:

1. Maturation of patterns and frameworks: Over the past decade, a substantial amount of R&D effort has focused on patterns and frameworks for high-performance and real-time applications and middleware. For instance, research conducted as part of the DARPA Quorum project [8], the QuO project at BBN [9], and the TAO [10] and TMO [11] projects at Washington University and UC Irvine, have identified key design patterns [12], optimization principle patterns [13], and frameworks [14, 15] that instantiate these patterns into high-quality, QoS-enabled DOC middleware components [16].

2. Maturation of standards: Over the past decade, the OMG's suite of standards has matured considerably, particularly with respect to high-performance and real-time systems. For instance, the OMG has recently adopted the Minimum CORBA [17], CORBA Messaging [18], and Real-time CORBA [19] specifications. Minimum CORBA removes features from the complete OMG CORBA specification that are not required by real-time and embedded systems. The Messaging specification defines several asynchronous method invocation models and exports QoS policies to applications. The Real-time CORBA specification includes features to manage CPU, network, and memory resources. This article describes the key features of the Real-time CORBA specification that are most relevant to researchers and developers of distributed real-time and embedded systems.

Unfortunately, programmers currently lack comprehensive and comprehensible sources of information on how to develop real-time applications using advanced CORBA specifications. Although the CORBA Real-time [19] and Messaging [18] specifications do an effective job of defining a standard set of APIs, they are woefully inadequate as a means for learning how to write, maintain, and optimize efficient, predictable, and scalable real-time systems. In particular, developers of real-time applications must understand how to master the patterns and principles associated with applying the quality of service (QoS) features defined by advanced CORBA specifications.

To address the lack of good documentation, we propose to write a *Real-time CORBA Developers Guide* that will demystify key advanced CORBA specifications related to programming real-time CORBA applications. This developers guide will provide many concrete examples that illustrate how to program key features of the Real-time CORBA and CORBA Messaging specifications. Therefore, real-time application developers can use this developers guide to learn more about advanced CORBA features, as well as gain a firm understanding of good design techniques and patterns for writing and optimizing real-time CORBA applications.

All examples in the developers guide will be distributed online along with the TAO real-time ORB [20]. Therefore, developers can interactively work through the many examples in the document in order to get hands-on practice writing, configuring, and optimizing real-time applications using TAO. However, since the examples in the book will be based on OMG standards, the knowledge and material will be transferable to any ORB that complies with advanced CORBA specifications.

The remainder of this document is organized as follows: Section 2 presents an overview of the Real-time CORBA specification in order to give a sense of the topics that will be covered in the *Real-time CORBA Developers Guide*. Naturally, the final version of the document will explore each of these topics in much greater detail, along with many concrete examples that can be programmed and tested using TAO. Section 3 presents a statement of work, which outlines the tasks, schedule, and budget for the proposed project.

2 Overview of Real-time CORBA

The Real-time CORBA (RT-CORBA) specification defines standard features that support end-to-end predictability for operations in *fixed-priority*[1] CORBA applications. This specification extends the existing CORBA standard [6] and the recently adopted OMG Messaging specification [18]. In particular, RT-CORBA leverages features from GIOP/IIOP version 1.1 and the Messaging specification's QoS policy framework. All these features and specifications are being integrated into the forthcoming CORBA 3.0 standard [24].

As shown in Figure 1 an ORB endsystem [10] consists

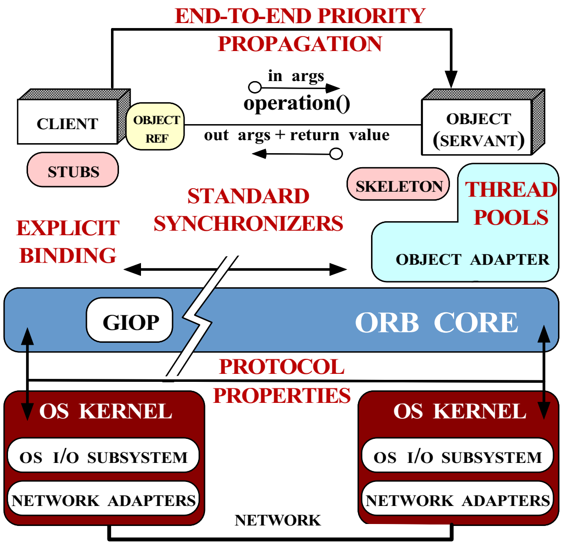

Figure 1: ORB Endsystem Features for Real-Time CORBA

of network interfaces, operating system I/O subsystems and communication protocols, and CORBA-compliant middleware components and services. The RT-CORBA specification identifies capabilities that must be *vertically* (*i.e.*, network interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, peer-to-peer) integrated and managed by ORB endsystems to ensure end-to-end predictable behavior for *activities*[2] that flow between CORBA clients and servers. Below, we outline these capabilities, starting from the lowest level of abstraction and building up to higher-level services and applications.

1. Communication infrastructure resource management: An RT-CORBA endsystem must leverage policies and mechanisms in the underlying communication infrastructure that support resource guarantees. This support can range from (1) managing the choice of connection for a particular invocation to (2) exploiting advanced QoS features, such as controlling the ATM virtual circuit cell pacing rate [25].

2. OS scheduling mechanisms: ORBs exploit OS mechanisms to schedule application-level activities end-to-end. Because RT-CORBA targets fixed-priority real-time systems, these mechanisms correspond to managing OS thread scheduling priorities. The RT-CORBA specification focuses on operating systems that allow applications to specify scheduling

[1] Subsequent OMG specifications are standardizing dynamic scheduling [21] techniques, such as deadline-based [22] or value-based [23] scheduling.

[2] An activity represents the end-to-end flow of information between a client and its server that includes the request when it is in memory, within the transport, as well as one or more threads.

priorities and policies. For example, the real-time extensions in IEEE POSIX 1003.1c [26] define a static priority FIFO scheduling policy that meets this requirement.

3. Real-Time ORB endsystem: ORBs are responsible for communicating requests between clients and servers transparently. A real-time ORB endsystem must provide standard interfaces that allow applications to specify their resource requirements to the ORB. The policy framework defined by the OMG Messaging specification [18] allows applications to configure ORB endsystem resources, such as thread priorities, buffers for message queueing, transport-level connections, and network signaling, in order to control ORB behavior.

4. Real-time services and applications: Having a real-time ORB manage endsystem and communication resources only provides a partial solution. Real-time CORBA ORBs must also preserve efficient, scalable, and predictable behavior end-to-end for higher-level services and application components. For example, a global scheduling service [10, 27] can be used to manage and schedule distributed resources. Such a scheduling service can interact with an ORB to provide mechanisms that support the specification and enforcement of end-to-end operation timing behavior. Application developers can then structure their programs to exploit the features exported by the real-time ORB and its associated higher-level services.

To manage these capabilities, RT-CORBA defines standard interfaces and QoS policies that allow applications to configure and control (1) *processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service, (2) *communication resources* via protocol properties and explicit bindings, and (3) *memory resources* via buffering requests in queues and bounding the size of thread pools. Applications typically specify these real-time QoS policies along with other policies when they call standard ORB operations, such as `create_POA` or `validate_connection`. For instance, when an object reference is created using a QoS-enabled POA, the POA ensures that any server-side policies that affect client-side requests are embedded within a *tagged component*[3] in the object reference. This enables clients who invoke operations on such object references to honor the policies required by the target object.

Figure 1 illustrates how the various RT-CORBA features relate to the existing CORBA standard. Below, we describe how RT-CORBA features can be used to manage (1) processor resources and (2) inter-ORB communication. We also outline RT-CORBA features for managing memory resources, though the specification is less explicit on this topic, so we merge our memory management discussion with the two main topics.

[3]Tagged components are name/value pairs that can be used to export attributes, such as security or QoS values, from a server to its clients within object references [6].

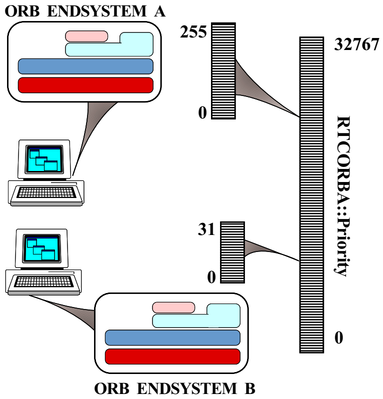

Figure 2: Mapping CORBA Priorities to Native Priorities

2.1 Managing Processor Resources

Strict control over the scheduling and execution of processor resources is essential for many fixed-priority real-time applications. Therefore, the RT-CORBA specification enables client and server applications to (1) determine the priority at which CORBA invocations will be processed, (2) allow servers to pre-define pools of threads, (3) bound the priority of ORB threads, and (4) ensure that intra-process thread synchronizers have consistent semantics in order to minimize priority inversion [28].

It is important to recognize that RT-CORBA's priority mechanisms cannot work miracles. In particular, ORB middleware cannot magically imbue a non-real-time OS or communication infrastructure with completely deterministic behavior [29]. When used in the appropriate environment, however, certain RT-CORBA features help application developers and integrators configure heterogeneous systems to preserve priorities end-to-end, as described below.

2.1.1 Priority Mechanisms

Conventional [6] CORBA ORBs provide no standard way for clients to indicate the relative priorities of their requests to ORB endsystems. This feature is necessary, however, to minimize end-to-end priority inversion, as well as to bound latency and jitter for applications with deterministic real-time QoS requirements. Therefore, the RT-CORBA specification defines the following platform-independent mechanisms to control the priority of operation invocations.

Priority type system: The RT-CORBA specification defines two types of priorities – *CORBA* and *native* – to handle OS heterogeneity. Each one-way or two-way CORBA operation can be assigned a CORBA priority, which ranges in value between 0 and 32767. Each ORB endsystem along an activity path can be customized to map CORBA priorities to native priorities, which may be unique on different endsystems. Figure 2 illustrates how CORBA priorities can be mapped onto two different native ORB endsystem priorities.

Priority models: The RT-CORBA specification defines a *PriorityModel* policy with two values, SERVER_DECLARED and CLIENT_PROPAGATED, as shown in Figure 3 and described below.

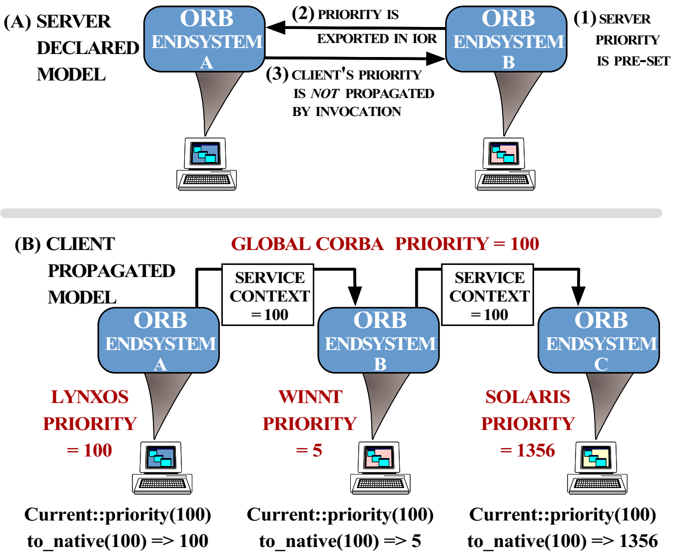

Figure 3: Real-time CORBA Priority Models

• **Server declared priorities:** This model allows a server to dictate the priority at which an invocation made on a particular object will execute. In the server declared model, the priority is designated *a priori* by a server based on the value of the *PriorityModel* policy in the POA where the object was activated. A single priority is encoded into the object reference, which is then published to the client as a tagged component in an object reference, as shown in Figure 3 (A).

Although the server declares the priority, the client ORB is aware of the selected priority model policy and can use this information internally. For example, priority-banded connections can be implemented on the client by matching invocation priorities and priority-bands with priorities advertised by a server. Thus, the ORB can guarantee that client invocations on a particular object are performed at the designated priority on the server.

• **Client propagated priorities:** Although the server declared model is useful for certain real-time applications, it is not suited for all application use-cases. For instance, one way for a server to avoid priority inversions is to process incoming requests at a priority equivalent to the client thread that invoked the operation originally [28]. The RT-CORBA client propagated model allows clients to declare invocation priorities that must be honored by servers. In this model, each invocation carries the CORBA priority of the operation in the service context list that is tunneled with its GIOP request. Each ORB endsystem along the activity path between the client and server maps this end-to-end CORBA priority to a native OS priority and processes the request at this priority. Moreover, if the client invokes a two-way operation, its CORBA priority will determine the priority of the reply.

Figure 3 (B) depicts the case where an invocation from a client on ORB endsystem A to a server on ORB endsystem C results in an invocation on an intervening ORB endsystem B, each running operating systems with different native thread priority ranges. The CORBA priority of the client is propagated with the request. Each intervening server along the activity path maps the client's CORBA priority to a native priority that is appropriate for its host platform and end-to-end global priority. For example, on Windows NT the global CORBA priority can be mapped to a native OS priority of 26. Likewise, on Solaris, the same global CORBA priority can be mapped to a real-time thread with a priority of 135.

Priority transforms: The client propagated and server declared priority models described above are not sufficient for all applications. For instance, the server declared model only maps priorities to objects, which may be too coarse-grained for more dynamic use-cases. Likewise, although the client propagated model is more dynamic, there are use-cases where applications require additional control over the ultimate priority at which a given invocation is processed. For example, different priority ceiling protocols may be required in a server to handle *inbound invocations*, *i.e.*, before the upcall is performed, and *outbound invocations*, *i.e.*, before a client or servant performs a remote method invocation.

To support these uses-cases, therefore, the RT-CORBA specification permits a server application to define *priority transforms* that set the priority at which particular invocations are performed, *e.g.*, based on external factors, such as current server load, operation criticality [10], or the state of a global scheduling service [27]. Transforms are implemented as *hooks* that are applied as requests are received or sent. A transform hook is passed the current CORBA priority and target object id and can change the invocation priority, potentially by calling out to application-supplied code. The following two priority transform models are defined in RT-CORBA:

• **Inbound transforms:** These transforms are applied during the invocation upcall, *i.e.*, after reception by the ORB Core, but before the servant operation is dispatched in a server.

• **Outbound transforms:** These transforms are performed when an *onward* operation is invoked from a servant. An onward operation occurs whenever a servant invokes an operation on an object.

2.1.2 Thread Pools

Many embedded systems use multi-threading to (1) distinguish between different types of service, such as high-priority vs. low-priority tasks [1] and (2) support thread preemption to prevent unbounded priority inversion. Prior to the RT-CORBA specification, however, there was no standard API for programming multi-threaded CORBA servers. Thus, it was not possible to use CORBA to program multi-threaded real-time systems without using proprietary ORB features.[4]

One way to implement a server ORB without threads is to use a *reactive* concurrency model [30]. In this approach, a server ORB reads each request from the underlying communication mechanism, processes it to completion, and then retrieves the next request and so forth. If all requests require a fixed, relatively short amount of processing, a reactive concurrency model may be feasible. However, many distributed applications have complex object implementations that run for variable and/or long durations. Moreover, to avoid unbounded priority inversion and deadlock, real-time applications often require some form of pre-emptive multi-threading.

To address these concurrency issues, therefore, the RT-CORBA specification defines a standard *thread pool* model [30]. This model allows server developers to pre-allocate pools of threads and to set certain thread attributes, such as default priority levels. Thread pools are useful for real-time ORB endsystems and applications that want to leverage the benefits of multi-threading, while bounding the amount of memory resources, such as stack space, they consume. Moreover, thread pools can be optionally configured to buffer or not buffer requests, which provides further control over memory usage.

Thread pools can be defined and associated with POAs in an RT-CORBA server. Each POA must be associated with one thread pool, although a thread pool can be associated with multiple POAs. Figure 4 illustrates the creation and association of thread pools in a server.

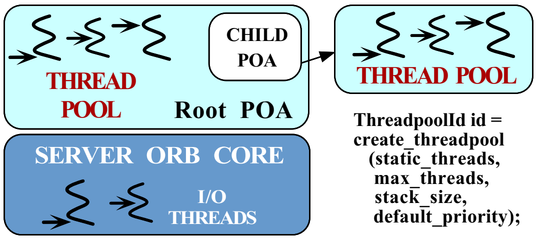

Figure 4: Server Thread Pools in RT-CORBA

The RT-CORBA specification defines two different thread pool styles, *with* and *without lanes*, as described below.

Thread pools: The simplest RT-CORBA thread pool model allows developers to control the overall concurrency level within server ORBs and applications. A thread pool is created with a fixed number of statically allocated threads that an ORB uses to process client messages. These pre-allocated threads will consume system resources even if they are not used, however. Therefore, RT-CORBA provides an interface that allows server developers to pre-allocate an initial number of so-called *static* threads, while allowing this pool to grow dynamically to handle bursts of client requests.

Server applications can use the `create_threadpool` API to specify (1) the default number of static threads that are created initially, (2) the maximum number of threads that can be created dynamically, and (3) the default[5] priority of all these threads. If a request arrives and all existing threads are busy, a new thread may be created to handle the request. No additional thread will be created, however, if the maximum number of threads in the pool have been spawned.

A pool can be optionally pre-configured for a maximum buffer size or number of requests. If buffering is enabled for the pool, the request will be queued until a thread is available to process it. If no queue space is available or request buffering was not specified the ORB should raise a TRANSIENT exception, which indicates a temporary resource shortage. When the client receives this exception it can reissue the request at a later point.

Thread pools with lanes: Many real-time and embedded systems applications statically associate global CORBA priorities to pools of threads. For example, a telecommunications application may select three distinct priorities to represent low-latency, high-throughput, and best-effort request classes. Alternatively, a fixed set of rate-groups with corresponding global CORBA priorities are a convenient model for applications, such as avionics mission computing [1], with real-time periodic processing requirements. In these scenarios, it is desirable to partition the threads in a thread pool into different subsets, each with different priorities. Therefore, RT-CORBA defines a *thread pool with lanes* model, which enables developers to bound both the overall concurrency of a server and the amount of work performed at a given priority level.

For each lane in this thread pool model, the server specifies the CORBA priority, static thread count, and dynamic thread count. Dynamic threads are assigned the lane priority. Thread pools with lanes can be configured to allow lanes with higher priorities to borrow threads from lanes with lower priorities. If a thread is borrowed, its priority is temporarily raised to that

[4] Strictly speaking, the RT-CORBA specification is an optional part of the CORBA standard, though ORBs that implement it are obliged to adhere to its interfaces and policies.

[5] Threads within a pool may have their priorities changed dynamically in accordance with the priority models or priority transforms described in Section 2.1.1.

of the lane that borrows it. When the invocation processing is complete, its priority reverts to its previous value and the thread returns to its original lane. Thread pools with lanes also can be configured to support request buffering if no threads are available to process incoming requests.

2.1.3 Standard Synchronizers

As mentioned in Section 2.1.2, the CORBA specification [6] does not define a threading model. Thus, there is no standard, portable API that CORBA applications can use to ensure semantic consistency between their synchronization mechanisms and the internal synchronization mechanisms used by an ORB. Real-time applications, however, require this consistency to enforce priority inheritance and priority ceiling protocols [28].

To ensure semantic consistency, therefore, the RT-CORBA specification defines a standard set of *locality constrained* mutex operations. Figure 5 illustrates the mutex interface defined by RT-CORBA.

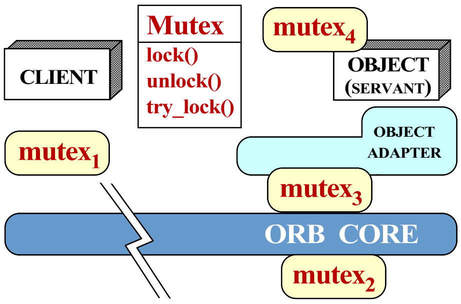

Figure 5: Standard Synchronizers

2.1.4 Global Scheduling Service

The scheduling abstractions defined by real-time operating systems such as VxWorks, LynxOS, and POSIX 1003.1c [26] implementations are relatively low-level. For instance, they require developers to map their high-level application QoS requirements into lower-level OS mechanisms, such as thread priorities and virtual circuit bandwidth/latency parameters. This manual mapping step is non-intuitive for many application developers, who prefer to design in terms of object interfaces and object operations.

To allow applications to specify their scheduling requirements in a higher-level, more intuitive manner, the RT-CORBA specification defines a global scheduling service [19]. This service is a CORBA object that is responsible for allocating system resources to meet the QoS needs of the applications that share the ORB endsystem. Applications can use the real-time scheduling service to specify the processing requirements of their operations in terms of various parameters, such as worst-case execution time or period, as shown in Figure 6.

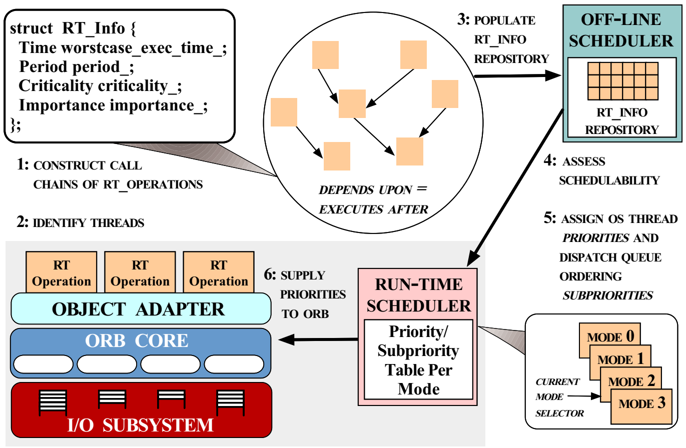

Figure 6: Real-Time CORBA Global Scheduling Service

2.2 Managing Inter-ORB Communication

Historically, the CORBA specification and conventional ORBs have supported *location transparency*, *i.e.*, applications cannot detect whether components are distributed or collocated in the same process [31]. Moreover, the features of the underlying OS, network, and/or bus are considered a black box. Although this encapsulation is useful for applications with best-effort QoS requirements, it is inadequate for applications with more stringent QoS requirements.

To allow applications to control the underlying communication protocols and endsystem resources, therefore, the RT-CORBA specification defines standard interfaces that can be used to select and configure certain *protocol properties*. In addition, client applications can *explicitly bind* to server objects using priority-bands and private connections, as described below.

2.2.1 Selecting and Configuring Protocol Properties

CORBA uses inter-ORB communication mechanisms [32] to exchange requests between clients and servers. These mechanisms are built upon lower level protocols that provide various types of QoS. Inter-ORB protocol (IOP) instances are composed of both an ORB protocol and a mapping to a specific underlying transport protocol. For example, the Internet Inter-ORB Protocol (IIOP) is a mapping of the General Inter-ORB Protocol (GIOP) onto TCP/IP. Thus, an IOP contains two protocol layers – *ORB* and *transport* – each having its own set of protocol properties.

RT-CORBA defines an interface that permits applications to specify ORB- and transport-specific protocol properties that control various communication protocol features, such as ATM virtual circuits or Internet RSVP [33] traffic specification. Each ORB/transport protocol properties tuple is defined by a `Protocol struct` that ultimately resides in a sequence structs called a `ProtocolList`, as defined by the following CORBA IDL:

```
interface ProtocolProperties {};

typedef struct {
  IOP::ProfileId protocol_type;
  ProtocolProperties
    orb_protocol_properties;
  ProtocolProperties
    transport_protocol_properties;
} Protocol;
typedef sequence <Protocol> ProtocolList;
```

The order in which protocol properties appear in the `ProtocolList` is significant – it allows applications to indicate the order of their protocol preferences. For example, a client may specify that IIOP is more preferable than other protocol combinations.

To allow applications to select and configure their desired ORB/transport protocol properties, RT-CORBA defines the following pair of QoS policies, *ClientProtocol* and *ServerProtocol*.

Server-side protocol properties: CORBA servers can use the *ServerProtocol* policy to select which protocols to configure into an object reference. This policy can be passed with other POA policies when the `create_POA` operation is invoked on the `PortableServer::POA` interface. The *ServerProtocol* policy has two purposes: it (1) publishes a list of available protocols to clients and (2) defines protocol configuration attributes for server connections.

The POA ensures that the ordering of profiles in object references conforms to the ordering of protocols specified in the *ServerProtocol* policy. Thus, a server can export its protocol preferences to clients by passing them in object references whose profiles are arranged in a particular order. When a client receives the object reference, it can either accept the server's preference or use different selection criteria.

Client-side protocol properties: Client applications can use the *ClientProtocol* policy to select which protocols to use when they connect to objects. This policy is applied when a client obtains a binding to an object. The *ClientProtocol* policy indicates the protocol properties a client is interested in, as well as the ordering of its preferences.

The *ClientProtocol* policy can be set either by a client or server, but not both for the same object reference. Servers can publish particular protocol requirements and preferences on a per-object basis. In contrast, clients can use this policy to change protocol policies on a per-invocation basis. If set on the server, the `ClientProtocol` policy is propagated to the client in the object reference, as shown in Figure 7. This figure

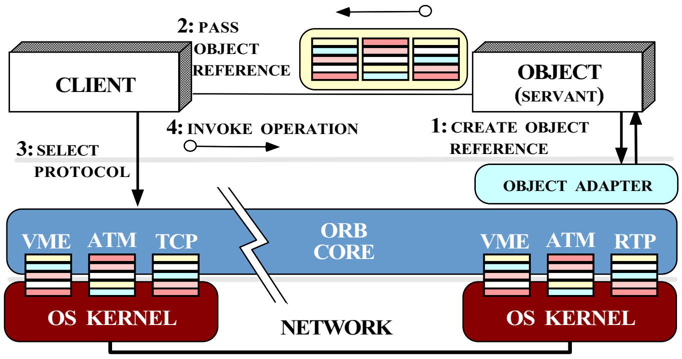

Figure 7: Configuring and Selecting Protocol Properties

illustrates how a server can designate the protocols available to the client. The server publishes the VME, ATM, or RTP protocols, in that order, in a tagged component in the object reference. The client then must abide by the *ClientProtocol* policy propagated by the server and select from one of these three protocols. This feature allows a server to enforce specific inter-ORB protocol requirements on clients.

The particular properties for specific protocols can be defined via interface inheritance. For example, the standard TCP protocol properties are shown below:

```
interface TCPProtocolProperties
  : ProtocolProperties
{
  attribute long send_buffer_size;
  attribute long recv_buffer_size;
  attribute boolean keep_alive;
  attribute boolean dont_route;
  attribute boolean no_delay;
};
```

This protocol property interface permits applications to set common attributes of TCP endpoints. For example, the send and receive buffer size attributes can set the size of endpoint socket queues. Many TCP implementations use these values to determine the TCP window size, which in turn affects end-to-end throughput. If the `keep_alive` attribute is enabled TCP will send a probe on inactive connections to verify that they are still valid. Finally, the `no_delay` attribute disables TCP's Nagle algorithm so that small requests can be sent even if earlier requests have not yet been acknowledged.

2.2.2 Explicit Binding

The original CORBA specification only supported *implicit binding* [34]. In this model, resources along the activity path between a client and its server object are established *on-demand*, *e.g.*, after a client's first invocation on the server. Implicit binding helps preserve location transparency by allowing clients to access remote objects or collocated objects using a common programming model. In addition, it helps conserve OS and networking resources, such as socket handles and ATM virtual circuits, by (1) deferring network connections until they are actually used and (2) allowing multiple client threads in a process to be multiplexed through shared network connections to their corresponding servers.

Unfortunately, implicit binding is inadequate for real-time applications with deterministic QoS requirements. In particular, deferring object/server activation and resource allocation until run-time can increase latency and jitter significantly. Moreover, the use of connection multiplexing can yield substantial priority inversion [30] due to head-of-line blocking associated with connection queues that are processed in FIFO order.

To avoid these problems, the RT-CORBA specification defines an *explicit binding* mechanism that uses the `validate_connection` operation defined on the `CORBA::Object` interface in the CORBA Messaging specification. This mechanism enables clients to (1) pre-establish connections to servers and (2) control how client requests are sent over these connections. The following two policies – *priority-banded* and *private connections* – are defined to support explicit binding in RT-CORBA.

Priority-banded connections: Priority-banded connections allow clients to (1) specify explicit priorities for each network connection and (2) select the appropriate connection at run-time based on the CORBA priority of the thread that invoked an operation. Clients are responsible for specifying policies that define one or more priority-bands when they establish connections explicitly.

Priority-band information is exported to the server within the service context of the first invocation sent across the connection. For instance, explicit binding information is passed in a request for `_bind_priority_band`, which is an *implicit operation*.[6] When a server receives a `_bind_priority_band` request, which includes the requested priority in the service context, it allocates resources to the connection. Subsequent requests on this connection are then processed at the requested priority.

In the absence of an `_bind_priority_band` operation, an implicit bind is performed when the first invocation is sent over the connection. The service context of this request must contain the CORBA priority range, *i.e.*, minimum and maximum values, for the banded connection. The server then allocates any necessary resources to ensure subsequent requests arriving on this connection will be processed at the desired priority.

[6]Implicit operations are implemented by an ORB, not by an application object, and are typically used for internal inter-ORB communication and configuration.

Private connections: Many ORBs support *multiplexed* connections, which yield better utilization of connections and other limited OS resources [30]. However, real-time applications often require private, *i.e.*, *non-multiplexed*, connections, which are well-suited for applications that possess deterministic QoS requirements. In this case, a connection cannot be reused for another two-way request until the reply for the previous request is received. To support this feature, RT-CORBA provides a policy, *PrivateConnection*, that allows clients to select private connections that minimize the duration of any end-to-end priority inversions. Oddly, there is no API in RT-CORBA to explicitly request a multiplexed connection, *i.e.*, this is considered an ORB implementation detail.

Figure 8 illustrates the use of priority-banded, private connections between a client and server. In Figure 8 private con-

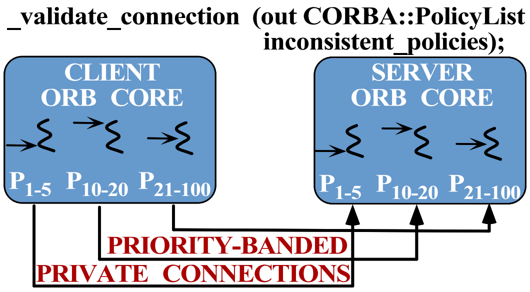

Figure 8: Explicit Binding

nections are combined with priority banding. Thus, each client operation is sent to the server over a pre-allocated connection that is assigned to a fixed priority range. The server ORB then processes the servant upcall at the specified priority and sends the reply across the same non-multiplexed connection. This combination of features ensures that end-to-end priorities are maintained and that key sources of priority inversion are eliminated.

3 Statement of Work

The scope of this effort is to create a *Real-time CORBA Developers Guide* that describes the key advanced features of the Real-time CORBA [19] and CORBA Messaging [18] that are relevant to programmers writing and optimizing real-time applications. This section describes the specific tasks to be performed and the deliverables to be produced during the 6 months of the proposed project, as well as the planned schedule and budget for the activities.

Task 1: Motivate, Describe, and Illustrate Key Real-time CORBA Features

The features of Real-time CORBA will include the following:

1. Portable priorities
2. End-to-end priority propagation
3. Protocol properties
4. Thread pools
5. Explicit binding
6. Standard synchronizers

The deliverables for Task 1 will be contained in a comprehensive document that will be released for review 2 months after the start of the project.

Task 2: Motivate, Describe, and Illustrate Key CORBA Messaging Features

These features will include the following:

1. *QoS Policies* for the ORB, *e.g.*:
   - Timeouts
   - Priority
   - Reliable one-ways
2. Asynchronous Method Invocation (AMI) models
   - Poller model
   - Callback model

The deliverables for Task 2 will be contained in a comprehensive document that will be released for review 4 months after the start of the project.

Task 3: Develop Online Tutorial Examples

These examples will illustrate concretely how to program real-time applications using the advanced features of Real-time CORBA and CORBA Messaging documented in Tasks 1 and 2. The deliverables for Task 3 will be distributed online in HTML format along with the TAO real-time ORB [20]. Therefore, developers can interactively work through the many examples in the document in order to get hands-on practice writing, configuring, and optimizing real-time applications using TAO. The tutorial examples will be released incrementally as they are completed. All examples will be finished by the end of the 6 month period of performance.

Budget

The participants in this effort include the following personnel:

1. Faculty members (*i.e.*, Dr. Douglas C. Schmidt – Ph.D., Associate Professor, UC Irvine and Dr. David Levine – Ph.D., Director of the Center for Distributed Object Computing, Washington University) at 10% during the 6 month period of performance.
2. Graduate student (*i.e.*, Carlos O'Ryan, Ph.D. candidate UC Irvine) at 100% during the period of performance.

The total cost of the proposed 12 month effort is **$52,500**. The following table provides a cost breakdown for this project.

Description	Amount
2 faculty members at 10%	20,000
1 graduate research assistant at 100%	15,000
University overhead at 50%	17,500
Total budget	52,500

References

[1] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[2] S. Mungee, N. Surendran, and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the Hawaiian International Conference on System Sciences*, Jan. 1999.

[3] C. O'Ryan, D. C. Schmidt, and D. Levine, "Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations," in *Proceedings of the* 5^{th} *Workshop on Object-oriented Real-time Dependable Systems*, (Montery, CA), IEEE, Nov. 1999.

[4] SLAC, "BaBar Collaboration Home Page." http://www.slac.stanford.edu/BFROOT/.

[5] A. Kruse, "CMS Online Event Filtering," in *Computing in High-energy Physics (CHEP 97)*, (Berlin, Germany), Apr. 1997.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[7] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[8] DARPA, "The Quorum Program." http://www.darpa.mil/ito/research/quorum/index.html, 1999.

[9] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[10] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[11] K. Kim and E. Shokri, "Two CORBA Services Enabling TMO Network Programming," in *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*, IEEE, January 1999.

[12] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.

[13] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[14] M. Fayad, R. Johnson, and D. C. Schmidt, eds., *Object-Oriented Application Frameworks: Problems & Perspectives*. New York, NY: Wiley & Sons, 1999.

[15] M. Fayad, R. Johnson, and D. C. Schmidt, eds., *Object-Oriented Application Frameworks: Applications & Experiences*. New York, NY: Wiley & Sons, 1999.

[16] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[17] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., August 1998.

[18] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 ed., May 1998.

[19] Object Management Group, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

[20] Center for Distributed Object Computing, "TAO: A High-performance, Real-time Object Request Broker (ORB)." www.cs.wustl.edu/$\sim$schmidt/TAO.html, Washington University.

[21] Object Management Group, *Dynamic Scheduling*, OMG Document orbos/99-03-32 ed., March 1999.

[22] S. Wang, Y.-C. Wang, and K.-J. Lin, "A General Scheduling Framework for Real-Time Systems," in *IEEE Real-Time Technology and Applications Symposium*, IEEE, June 1999.

[23] E. D. Jensen, "Eliminating the Hard/Soft Real-Time Dichotomy," *Embedded Systems Programming*, vol. 7, Oct. 1994.

[24] S. Vinoski, "New Features for CORBA 3.0," *Communications of the ACM*, vol. 41, pp. 44–52, October 1998.

[25] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[26] "Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]," 1995.

[27] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 2000.

[28] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Proceedings of the Real-Time Systems Symposium*, (Huntsville, Alabama), December 1988.

[29] D. L. Levine, D. C. Schmidt, and S. Flores-Gaitan, "An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers," in *Proceedings of the Multimedia Computing and Networking 2000 (MMCN00) conference*, (San Jose, CA), ACM, Jan. 2000.

[30] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems*, To appear 2000.

[31] N. Wang, D. C. Schmidt, and S. Vinoski, "Collocation Optimizations for CORBA," *C++ Report*, vol. 11, October 1999.

[32] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[33] R. Braden et al, "Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification." Internet Draft, May 1997. ftp://ietf.org/internet-drafts/draft-ietf-rsvp-spec-15.txt.

[34] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.