Realtime CORBA

Alcatel

Hewlett-Packard Company

Lucent Technologies, Inc.

Object-Oriented Concepts, Inc.

Sun Microsystems, Inc.

Tri-Pacific

In collaboration with:

Deutsche Telekom AG

France Telecom

Humboldt-University

Mitre

Motorola, Inc.

Washington University

Version 1.0 - Initial RFP Submission

January 19, 1998

OMG Document Number orbos/98-01-08

Contacts

Zoely Canela
Alcatel Alsthom Recherche
Route de Nozay
91460 Marcoussis
France
phone +33 1 69 63 12 71
fax +33 1 69 63 17 89
E-mail: canela@aar.alcatel-alsthom.fr

Jishnu Mukerji
Hewlett-Packard Company
300 Campus Drive, MS 2E-62
Florham Park, NJ 07932
USA
phone +1 973 443 7528
fax +1 973 443 7422
E-mail: jis@fpk.hp.com

Judy McGoogan
Lucent Technologies
Room 5B-427
2000 N. Naperville Road
Naperville, IL 60566
USA
phone +1 630 713-7355
fax +1 630 979-9364
E-mail: jmcgoogan@lucent.com

Peter Kortmann
Tri-Pacific Software Consulting Corp.
1070 Marina Village Parkway Suite 202
Alameda, CA 94501
USA
phone +1 510 814 1775
fax +1 510 814 1788
E-mail: peter@tripac.com

Marc Laukien
Object-Oriented Concepts, Inc
44 Manning road
Billerica, MA 01821
USA
phone +1 978 439 92 85
fax +1 978 439 92 86
E-mail: ml@ooc.com

Michel Gien
Sun Microsystems
6 av Gustave Eiffel
78180 Montigny Le Bretonneux
France
phone +33 1 30 64 82 22
fax +33 1 30 57 00 66
E-mail: Michel.Gien@France.Sun.COM

Trademarks

Contents of Submission

Realtime CORBA

Initial Joint Submission

Alcatel/HP/Lucent/OOC/Sun/Tri-Pacific

Version 2
January 19, 1998

1 Introduction

Alcatel, Hewlett-Packard Company, Lucent Technologies, Object-Oriented Concepts, Sun Microsystems and Tri-Pacific in collaboration with Deutsche Telekom AG, France Telecom, Humboldt-University, Mitre, Motorola and Washington University, are pleased to provide this first submission in response to the OMG "Realtime CORBA" RFP.

2 References

In addition to documents referring to background information on OMG's Object Management Architecture, the following documents are referenced in this document:

[1] [CORBA 2.1] *The Common Object Request Broker: Architecture and Specification*, Revision 2.1, August 1997.

[2] [ORB Portability] *ORB Portability Joint Submission*, OMG Document orbos/97-05-15.

[3] [RFP Real Time] *Realtime Request For Proposal*, OMG Document orbos/97-09-31.

[4] [ODP Reference] *ODP Reference Model: Architecture*, ITU-T|ISO/IEC Recommendation X.903|International Standard 10746-3

[5] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[6] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), pp. 529–545, Reading, MA: Addison-Wesley, 1995.

[7] [RT White Paper] *Realtime CORBA White Paper,* December 1996

[8] [ReTINA] *Requirements for a Realtime ORB* ReTINA ACTS Project AC048 RT-TR-96-08.1, May 1996

3 Overall Design Rationale

3.1 Rationale for IDL

3.1.1 Realtime API

The Realtime API is specified in IDL so that it is independent of the programming language. While specifying the Realtime API, a strongly typed approch was used because strong typing is an important concept in object oriented programming paradigm. Strong typing also to reduce the risks of programming errors.

3.1.2 Interface Inheritance

The IDL which is defined uses interface inheritance to achieve genericity. Genericity brings openness and reusable APIs. For the second submission, we are investigating further the use of ObjectsByValue or the POA *local-only* on server side as an alternative to some interfaces.

3.1.3 Exceptions

The submitters are concerned with small memory footprint for a realtime ORB. As much as possible the IDL of the Realtime API tries to take advantages of existing system exceptions and the ability to specify a minor code.

3.1.4 Locality Constraints

A locality constrainted object is like a regular object except for the fact that it can only be accessed from within its capsule (see [4]) in which it is instantiated. Consequently, a reference to such objects cannot be externalized either through marshaling or through the **ORB::object_to_string** operation. Any attempt to do so should raise the **CORBA::MARSHAL** exception. Any attempt to use such an object through DII should raise the **CORBA::NO_IMPLEMENT** exception. Additionally, since they are not accessible from outside the capsule, they may not be registered with an Interface Repository.

A consequence of this restriction is that references to a locally constrained object may not be passed as a parameter of any operation of a normal object. However, it is OK to pass references to locally constrained objects as parameters in operations of another locality constrained object as long as the two objects reside in the same capsule.

3.2 Rationale for RT module

Realtime APIs are defined in a separate module called **RT**. The use of a separate module clarifies the organization of the APIs and most of all position it as an extension of the Realtime ORB compared to traditional ORBs.

3.3 Rationale for Realtime Portable Object Adapter module

The realtime ORB requires a realtime portable object adapter. A separate module which uses the Portable Object Adapter is defined to specify the realtime extensions to the POA. These extensions are separate so that traditional ORBs are not obliged to implement them.

4 Proof of Concept

This specification is based on the combined experience on prototypes and products of the submitters:

- ChorusORB r5
  http://www.chorus.com or http://www.sun.com
- HP
  http://www.hp.com/hpj/97feb/feb97a9.htm
- Fixed Priority Scheduling
  http://www.tripac.com
- Pluggable Protocols
  http://www.ooc.com
- TAO
  http://www.cs.wustl.edu/~schmidt/TAO.html

5 Issues Addressed in This Submission

5.1 Mandatory Requirements

This submission addresses the following mandatory requirements:

- **Extensions to adopted OMG specifications**: Introduces extensions to existing OMG specifications. The only overlap may be with OMG's Interceptor Facility, which is currently being revised. The final submission will be aligned with the revised OMG architecture for interceptors.
- **Schedulable entities**: At the ORB level, defines threads, requests, replies, messages and transport end-points as "schedulable entities". Specifies: 1) thread attributes, 2) a thread management API that allows creation and deletion of threads as well as modification of thread attributes, 3) thread specific storage, and 4) thread pools. Also specifies APIs for: 1) request queue and flow control, and 2) management of transport end-points.

  At the services level, defines a new Scheduling Common Object Service to facilitate plugging in various fixed-priority, realtime scheduling policies. The Scheduling Service is based on the notion of a global, uniform priority assignment to threads. It defines interfaces for client and server scheduling entities that allow management of priorities and priority ceilings.
- **Propagating client's priority**: At the ORB level, specifies "ORB flexibility enablers" that allow integration of specific realtime strategies such as QoS support, priority inheritance and others. Flexibility enablers: 1) allow applications to store arbitrary data in an object reference, and 2) support an interceptor mechanism to manage the semantics of method invocation. Interceptor APIs increase the flexibility of the realtime ORB and allow implementation of a variety of application-selected strategies. Interceptors are introduced for requests, portable object adapters, transport, and threads.

  Application programmers can choose to use a Scheduling Service or to set scheduling parameters using ORB primitives directly.
- **Avoiding/Bounding priority inversion:** The Scheduling Service interfaces coupled with the characteristics of the realtime ORB enable designers of analyzable realtime systems to bound priority inversion. Multiple transport end-points and buffer management API avoid or bound the priority inversion by directing client requests to the thread with correct priority.
- **Avoiding/Bounding blocking**: APIs of the realtime ORB and the Scheduling Service integrate several elements to avoid/bound blocking on method invocations. ORB APIs are specified for: 1) threads, 2) request queues, 3) transport management, 4) buffer management, 5) interceptors, and 6) synchronization facilities. The Scheduling Service obtains its input from interceptors and the request. It uses the request queue API to schedule requests that must be processed in priority order.
- **Resources**: Resources include: threads, buffers and memory, transport end-points, request queues and synchronization objects (e.g., mutexes, semaphores, condition variables, multiple readers/single writer, ...). APIs are defined for each.

- **Interfaces for selecting interaction and transport protocols*:*** The generic Transport API creates, deletes, and manages attributes for transport end points. Specific attributes for TCP/IP are defined. Additionally, a flexible mechanism is defined for integrating new protocols in the realtime ORB at both the transport and the interaction level.
- **Binding**:
  Server Binding: Enables the server to establish appropriate bindings so that realtime requests can be processed with no (or minimal bounded) priority inversion by integrating an appropriate use of transport resources, the appropriate setting of server threads, a well configured request queue, and a scheduler that drives the dispatching and execution of requests.
  Client Binding: Enables the client to choose and set appropriate bindings with a selection and configuration of transport resources; specification of Client QoS so that realtime invocations can be processed. (this is also related to the **ORB Flexibility Enablers).**
- **POA:** Uses the Portable Object Adapter not the Basic Object Adapter. Identifies POA activation policies that should not be used - in order to guarantee end-to-end predictability. Contains a placeholder for POA extensions for Realtime.

5.2 Optional Requirements

The optional requirements are dealt with in the submission as follows:

- **Time limits for replies**: One example of a control parameter that could be used with the generic request queue API is “the maximum time that a request can wait without being processed.” Application designers may also address this need by using the flexibility provided by: 1) the ability to store arbitrary data in an object reference, 2) the interceptor mechanism, and 3) the Scheduling Service.
- **Installation of user-provided transport protocols**: Pluggable protocols are defined to provide a flexible mechanism for integrating new transport level protocols in the realtime ORB.
- **RT interaction protocols**: Pluggable protocols are also defined to provide a flexible mechanism for integrating new interaction level protocols in the realtime ORB. However, the submission does not define a specific realtime interaction protocol, since other OMG Task Forces (e.g., the Telecom DTF) are currently addressing this via RFPs for a variety of ESIOPs (e.g., SS7, ATM, OSI, etc).
- **Run-time interfaces for “schedulable entities”**: The Thread API, Transport API, and Request Queue API address this.

5.3 Issues

This submission addresses the following issues:

- **Assumptions about the underlying operating system:** Defines as a minimum a specialization of the generic thread attributes to control POSIX threads. Also compatible with other threading APIs, such as Win32 Threads.

- **Relationship to POSIX:** See above.
- **Relationship to Concurrency Service, Time Service, Transaction Service, and Event Service:** Addressed in §16 “Relation with COS Specifications” on page 95.
- **Role of the Security Service:** TBD
- **Definition of binding:** Defines binding based on ODP. Proposes use of the Portable Object Adapter’s interfaces and operations for a server to bind its servants to object references. Specifies the addition of realtime parameters to the binding information recorded/retrieved by the object adapter.
- **Relationship to the Messaging Service:** Addressed in §7.5 “Asynchronous invocations” on page 18.
- **How to build realtime CORBA applications:** Examples of usage appear throughout the document, and this is also addressed in §15 “Example Scenario” on page 95.

6 Problem Statement

6.1 Conventional CORBA Architecture - Background

Figure 1 on page 10, shows the conventional structure of an ORB. The ORB is responsible for all of the mechanisms required to find the Servant for the request, to prepare the Servant to receive the request, and to communicate the data making up the request. To make a request, the Client can use the DII or an IDL stub. It can interact with the ORB for some functions. On the server side, the Servant receives a request as an up-call either through the IDL generated skeleton or through the DSI. The ORB core locates the servant, transmits parameters, and transfers the control to the Servant through an IDL skeleton or the DSI. The object adapter provides generation and interpretation of object references, method invocation, servant activation and deactivation, mapping of object references to servants. The ORB core provides the GIOP/IIOP interaction protocol.

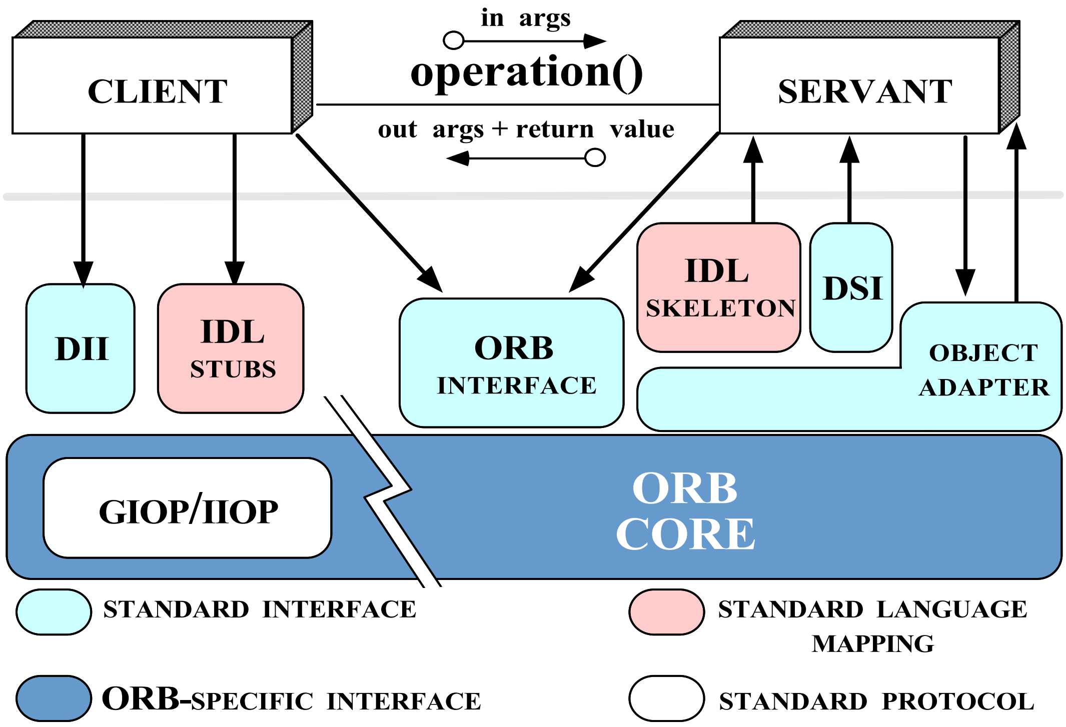

Figure 1 Components in the CORBA Reference Model

6.2 CORBA Problems for Realtime Applications

The conventional CORBA architecture does not meet the needs of many realtime applications:

- This architecture does not provide interfaces to specify end-to-end QoS requirements which are necessary for realtime applications. Furthermore, the opaqueness of the ORB core concerning resources and the full invocation path makes end-to-end QoS enforcement hard to meet.
- Lack of realtime programming features such as asynchronous invocations, timed operations and transport layer flow control notification.
- Lack of performance optimizations lead to significant throughput and latency overhead. Improvements in data copying, message buffering and de-multiplexing are required to allow many performance sensitive realtime applications to use a CORBA ORB.

7 Architecture Overview of a CORBA ORB for Realtime Applications

7.1 Introduction to Realtime and End-to-end predictability

Realtime systems need to specify timeliness and throughput quality of service (QoS) requirements to obtain guarantees about the fulfillment of these requirements. The nature of guarantees provided may vary from best-effort (where the system provides no quantitative guarantee of how well or how often it will meet application QoS requirements) to deterministic (where the system guarantees that application requirements will be strictly met throughout the lifetime of the application). In a distributed system, the end-to-end predictability qualifies the temporal behavior of the system from best-effort to deterministic. The distributed realtime system should take into account the following two important elements:

- End-to-end predictability requires establishing communication paths between client and server with defined characteristics.
- The network characteristics must be taken into account. Network latency, network error rate (such as packet loss or cell loss) have an impact on the resulting end-to-end predictability.

Establishment of the communication path between objects (e.g., client and server objects) is referred to as a binding process. The result of this process is also called a binding. This notion of binding is exactly that introduced in the Reference Model for ODP ([4]). It is an end-to-end notion, not just server adapter to server applications, and it is not limited to client-server configurations. For instance, a binding can support multimedia streams, group communication etc.

The notion of binding is not new to CORBA. Conventional ORBs establish a binding between the client and the server that the client wants to invoke. Most often this binding is made implicitly and no QoS requirement can be specified. The realtime ORB architecture requires that implicit bindings as well as explicit bindings be possible, and that mechanisms exist for realtime applications to specify their QoS and for the realtime ORB to enforce it.

Establishing a binding in a client-server configuration to guarantee the requested end-to-end predictability requires consideration at three lebels: client side, network, and server side.

- On the client side, the realtime ORB must establish the client part of a client-server binding (briefly, *client binding*) to guarantee that the QoS required by the client can be met. Depending on the QoS required, this may range from simple actions, such as opening or using an opened connection, up to complex actions, such as pre-allocating resources and pre-establishing dedicated transport or network connections.
- The network must provide protocols supporting the appropriate QoS. Client QoS information is used as an input for configuring and using such protocols.

- On the server side, the realtime ORB must establish the server side of a binding (briefly, *server binding*) so that realtime requests can be processed in a timely fashion with no (or minimal bounded) priority inversion, in order to meet the end-to-end QoS.

On both the client and server side, meeting the QoS constraints implies an appropriate use of transport resources, of supporting threads, and request queues. This in turn implies an integrate scheduling to take place, ensuring the proper dispatching and handling of realtime requests.

Establishing a binding may involve third party servers (on the client side as well as on the server side). The realtime ORB architecture does not mandate any policy on how, when and what is done to establish a binding. However, it provides:

- The architectural elements for bindings to be established (ie, the Realtime API)
- QoS guarantee for bindings which are successfully established (ie, realtime capability).

7.2 Key Architectural Elements

Threads are an important abstraction in realtime applications. Several threads of execution are commonly used to guarantee the predictability of realtime applications. With the Realtime ORB, a server must be ready to process several invocations coming from clients with different priorities. The server must guarantee, via the Realtime ORB, the end-to-end QoS that was requested and set for its different clients. This gurarantee is hard to meet in mono-thread environment. Therefore, the Realtime ORB executes *a priori* on a multi-threaded environment.

To enforce the end-to-end QoS required by clients, there exist many policies for a Realtime ORB. The specification of the QoS itself, also depends on the policy that is used to guarantee it. For example some applications will need priority inheritance; others will use more complex QoS information (e.g., for a Rate Monotonic scheduler) and so on. Defining only policies for the Realtime ORB will not be satisfactory for all realtime applications.

The Realtime ORB architecture which is presented here focuses on the definition of mechanisms instead of policies. Mechanisms are defined to enforce the end-to-end QoS at different levels. They do not specify how the enforcement is made but rather provide enablers for policies to be defined. It is probably the most important element of the Realtime ORB to specify these mechanisms for plugging in specific realtime policies.

Mechanisms are not useful if no policy is defined. The Realtime ORB architecture defines important realtime policies which are commonly used or should be suitable for many realtime applications.

The basic CORBA architecture is retained, but interfaces are added to allow realtime developers control over the end-to-end predictability of the system by providing them with a clear knowledge of the full path of method invocations.

The key elements of the realtime ORB are defined as follows:

- **ORB Resource Control**
  The Realtime CORBA architecture defines key elements and an API for controlling resources used by the ORB. Resources include the following: threads, buffers and memory, transport end-points, and request queues.
  The ORB Resource Control interface is the API that realtime applications will use to specify their end-to-end QoS requirements so that the ORB endsystem can attempt to enforce these requirements. This API is defined in §8 "Control of Resources" on page 20.

- **ORB Flexibility Enablers**
  Realtime distributed applications are not limited to a single priority or QoS model. They are not limited to a single failure model either. The specification of the QoS itself depends on the policies used to guarantee it. Applications may use different approaches to predictability (e.g., using Rate Monotonic theory or deadline-based scheduler). Thus, ORB Resource Control alone is not sufficient because it does not dictate the interaction between resources and a particular invocation under varying conditions.

  Componentized Object References and Interceptors are Realtime ORB Flexibility Enablers that support an integrated yet flexible architecture which provides realtime application developers the ability to influence or control ORB behavior to meet a variety of realtime system environments.

  This API is presented in §9 "ORB Flexibility Enablers" on page 42.

- **ORB Synchronization Facilities**
  Synchronization objects are used in multi-threaded applications to serialize access to data shared by several threads. Such objects are used by the ORB to protect internal ORB data. Applications can also use synchronization objects to protect their own data. Synchronization objects are defined in §10 "Synchronization Facilities" on page 59.

- **ORB Pluggable Schedulers**
  The scheduler is an important element of realtime applications. A realtime ORB cannot provide a single scheduler that will fit all realtime application needs. The ability for the ORB to plug-in and correctly integrate a variety of application schedulers is an important element of the architecture. Pluggable schedulers will operate on the ORB resources and will use ORB Flexibility Enablers to control method invocations. Since Fixed Priority Scheduling is one specific scheduler which is required in a major category of realtime applications, it will be the first ORB Pluggable Scheduler to be defined for Realtime CORBA. This scheduler is presented in §11 "Fixed Priority Scheduling Service" on page 65.

- **ORB Pluggable Protocols**
  A flexible mechanism for integrating new protocols in the realtime ORB is necessary. Pluggable protocols operate at two levels: 1) the interaction level 2) the transport level. The first level describes and controls the format of messages exchanged by client and servers (e.g., GIOP, DCE-CIOP). The second level represents the raw physical transport which is used to exchange those messages (e.g., TCP/IP, SS7, ...).

7.3 Schedulable Entities

An entity is schedulable when the resources or processing capabilities it provides must be explicitly managed to meet the QoS needs of tasks sharing the ORB endsystem. The Realtime ORB contains several entities which are schedulable. Basically, entities may be scheduled at two levels:

- within the operating system
- within the ORB

The first level is refered to as the OS scheduler and the second level is refered to as the ORB scheduler.

The following entities are schedulable:

- **Threads**
  Threads execute application and ORB code. They may be suspended by the OS scheduler. The OS scheduler assigns a processor to a thread and at a given time it decides which thread is best to have a processor.
- **Requests**
  Requests are an ORB schedulable entity. An ORB scheduler has the ability to decide whether the request must be processed now or later. Basically, the ORB scheduler can organize requests as they arrive according to their priority or importance (see §8.3 “Request Queue and Flow Control” on page 31).
- **Replies**
  Replies are an ORB schedulable entity. On the client side, a scheduling is necessary when concurrent threads are waiting for replies on the same client-side end-point (e.g., two threads waiting for a reply on the same TCP/IP connection). Deferred synchronous replies also need to be scheduled.
- **Messages**
  Messages are schedulable entities that are scheduled outside of the ORB, in most cases in the operating system or its drivers. Messages may be scheduled to handle a request or a reply produced by the ORB scheduler. The message can be scheduled inside the transport driver. For example with ATM networks, a message can be directed to one virtual circuit or to the other depending on its associated QoS parameter.
- **Transport End-Points**
  Transport end-points, for example TCP/IP socket connections, need to be scheduled. Connections may be multiplexed or not (that is shared by several client threads). Scheduling access to multiplexed and non-multiplexed connection end-points has a big impact on priority inversion and non-determinism. Transport end-points are scheduled by the ORB scheduler.

In the document, the term scheduler alone is commonly used to refer to the ORB scheduler.

7.4 End to End Predictability

End-to-end predictability can be studied globally at three levels: 1) on the client side; 2) on the network; and 3) on the server side.

- First, on the client side, client QoS information must be passed to the server, and the appropriate binding specified by the server must be used to perform the invocation.
- Second, the network must support appropriate protocols that avoid message priority inversions. Client QoS information is used as an input for configuring and using such protocols.
- Finally, the server must establish appropriate bindings so that realtime requests can be processed with no (or minimal bounded) priority inversion. This requires integrating an appropriate use of transport resources, the appropriate setting of server threads, a well configured request queue and last but not least a scheduler that drives the dispatching and execution of requests.

Figure 2, "Client side configuration for end-to-end predictability," on page 16 illustrates a possible configuration on the client side to ensure end-to-end predictability. The client defines several threads and assigns them appropriate QoS characteristics (e.g., priority). Bindings are established that logically comprise, on the client side, stubs and communication protocols (including at least a transport protocol that drives and interact with the node I/O subsystem). The role of a stub is to transform an operation into a request with QoS characteristics derived from that of the issuing thread, and to pass it to the supporting protocols. When the request is passed to the supporting protocols, ORB interceptor mechanisms can be activated to provide the ability for runtime scheduling to take place. The runtime scheduler can e.g., determine which transport connection to use (depending on the binding configuration), based on the request's QoS characteristics. The transfer of the request is then performed by using the chosen connection.

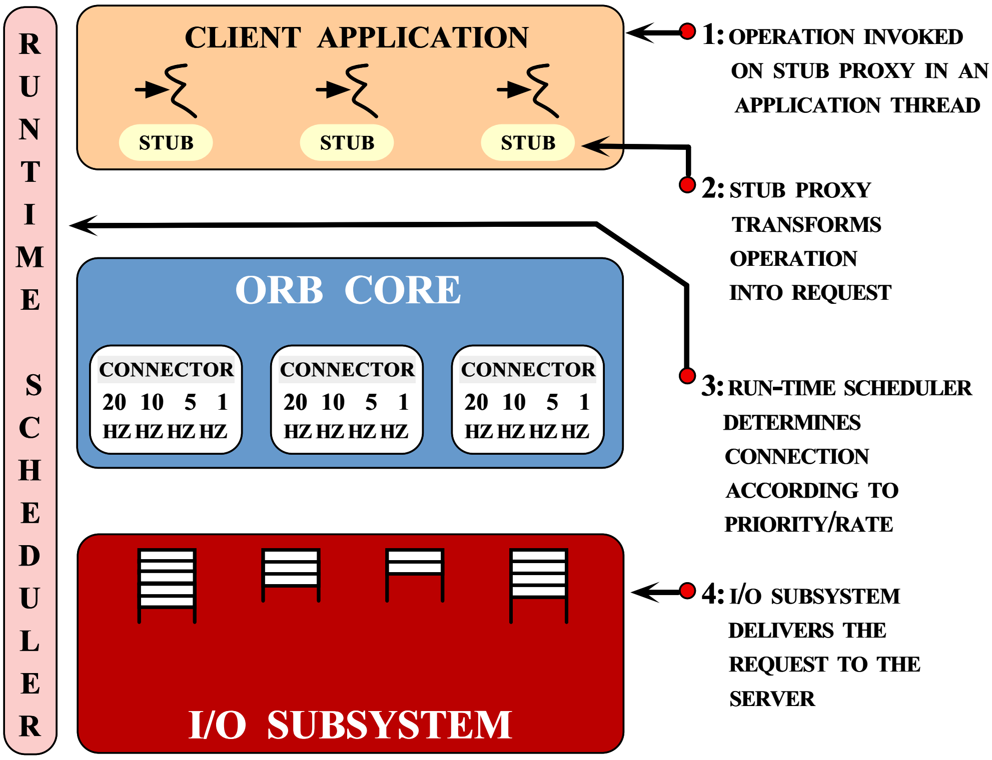

Figure 2 Client side configuration for end-to-end predictability

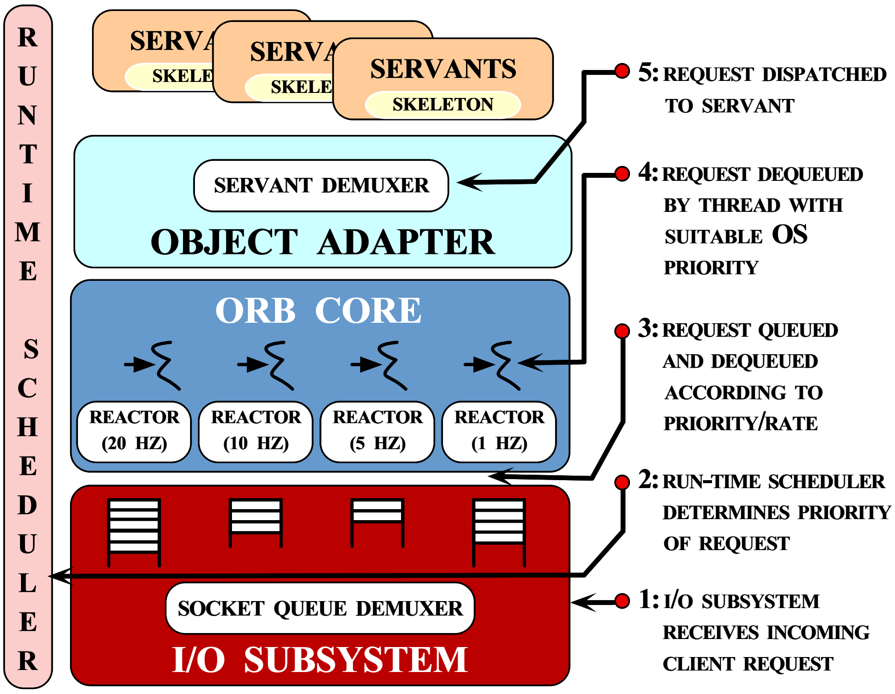

Figure 3 Server side configuration for end-to-end predictability

Figure 3, "Server side configuration for end-to-end predictability," on page 17 illustrates a possible configuration on the server side to ensure end-to-end predictability. Four transport end-points are shown. Each of them is associated with a separate set of threads and a separate request queue. Each of the threads have different priorities so that a priority distinction can be made on messages which arrive on the transport end-points. More generally, the realtime ORB integrates several elements to guarantee the end-to-end predictability on the server side:

- The Realtime Thread API provides the ability to allocate several threads (e.g., pool of threads) and to dedicate them to a particular transport end-point. By doing so, requests received on a given transport end-point are received by threads with appropriate priorities.
- The Request Queue API provides a flexible management of requests received by the server. Strategy for queuing requests and processing them is configurable. A flow control mechanism can be implemented. Depending on the realtime policy of the application, the flow control may be layer-to-layer within the ORB or end-to-end across the network. In the first case, the flow control mechanism is local while in the second case it is distributed.
- The Transport Management API allows fine-grained control of communication resources. A realtime server is able to create several transport end-points in order to differentiate client priorities within the transport media. This differentiation is

necessary to reduce the message priority inversion which may occur on the transport media. This is a key enabler for correctly supporting priority based transport protocols such as ATM.

- The Buffer Management API controls the allocation of memory buffers used for request and replies. Minimizing the amount of shared resources by threads in an ORB is important to improve predictability and avoid priority inversion. The Buffer Management API uses thread specific storage to avoid the use of locks for protecting memory buffers.
- The Interceptors API provides inputs for transparently and flexibly introducing specific elements of the Realtime ORB. The Realtime ORB activates the interceptors at different levels so that specific realtime policies may be defined. As an example, an interceptor may be defined to extract from the client request the information about its QoS (e.g., the global priority that is defined in §11 "Fixed Priority Scheduling Service" on page 65). Then, the interceptor can save this information in a thread specific data so that this may later be retrieved by the servant thread.
- The scheduler obtains its input from interceptors and the request. It uses the Realtime Thread API to schedule threads. It uses the Request Queue API to schedule requests that must be processed.

The example showed that the object adapter contained only one object map, shared by the four transport end-points. The object map is used to record bindings made by the application. With the use of the POA, two separate object adapters may be created and each of them maintains its own object map. The server is able to associate a specific transport end-point to a POA.

7.5 Asynchronous invocations

CORBA specifies oneway methods with asynchronous semantics. However, the invocation of these methods offers no visibility at all concerning the final outcome of the invocation. The reliability of the invocation delivery relies on the underlying transport mechanism, but there is no way to check for proper invocation delivery.

Similarly, the underlying transport protocols might exhibit a behavior incompatible with realtime when accessing sites that are stopped or crashed (TCP timeouts for example).

Since a specification for the CORBA Messaging Service has not yet been apdopted by OMG, the final submission to the Realtime RFP will address Asynchronous Messaging in more detail.

Current proposals in response to the CORBA Messaging Service RFP define a non-procedural way to address asynchronous invocations, but it is a significant departure from the CORBA programming model. It forces an un-natural programming model which greatly exposes the underlying mechanisms.

What is required is a way to query asynchronous request status after it has been sent. To address asynchronous client invocation needs, this submission adds a new keyword async to CORBA IDL language. This keyword is treated only in the client side language mapping. Server side implementations will not be required to change since in the server side programmers point of view, all invocations are treated identically regardless of their invocation model characteristics.

Three models appear interesting:

- A polling model, where a handle could be passed by the client in the invoked method and later be polled.
- Alternatively, a oneway (or asynchronous) method (currently returning void) could return a handle to access the requests status.
- An upcall model, where a callback object/function is registered when sending the invocation.

Thus, clients can perform either asynchronous or synchronous requests on the server. To perform an asynchronous invocation, the client passes a response handler (ResponseHandler, a client object that handles the response of the request) in addition to the normal parameters needed by the request. The response handler encapsulates all of the return values (including inout and out values) and its generated methods are implemented by the programmer. The ResponseHandler will support both the polling and future type (upcall) mechanisms.

The ResponseHandler is an interface defined in the CORBA module as given in the following:

```
module CORBA {
    ...
    interface ResponseHandler {
        NVList  get_response() raises(..);
        NVList  get_next_response() raises (..);
        void    wait();
        void    callback_invoked();
    };
};
```

The **CORBA::ResponseHandler** object is the base class for the IDL language mapping generated response handler objects. It provides the asynchronous response handling behavior that all response handler objects will inherit. The **CORBA::ResponseHandler** object is a local object with the following non exhaustive operations

NVList get_response and get_next_response;

The asynchronous request is blocked until the response comes (responses are encapsulated in NVList return value). Each of the return, inout, out and exceptions will be encapsulated in the NVList.

void wait

Will wait for the client callback (subclass of this object for upcall asynchronous response handling).

void callback_invoked

This function is called implicitly by the derived class when one of its callbacks is invoked. This function internally keep track of the status of the responses.

The asynchronous invocation mechanism which is defined and mandated by the Realtime ORB architecture is in line with the CORBA Messaging service requirements with some simplifications to take care of footprint constraints.

8 Control of Resources

8.1 Overview

Realtime applications need the ability to control resources used by the ORB on both the client side and the server side. The management of resources can be classified in two main categories:

- Some resources are allocated statically or basically at initialization of the ORB (e.g., threads, transport end-points, buffers). Once allocated, such resources will not be freed until the end of the process in which they are allocated. When the ORB uses static resources, there is no additional runtime overhead for allocation or deallocation. Furthermore, it is not necessary to deal with resource exhaustion: they are static and therefore they exist (unless the ORB initialization has failed).
- Other resources are allocated dynamically (e.g., request buffers, upcall parameters, ...). This means that the ORB will need to allocate those resources and also deallocate them while the application performs ORB requests. Dynamic allocation introduces two problems: 1) unpredictability, and 2) resource exhaustion. The first problem is more or less solved by providing realtime APIs for resource allocation and deallocation. The second problem is hard to solve and may result in catastrophic behavior for the realtime application.

Pre-allocation of resources is an important step for end-to-end predictability. As much as possible, the dynamic allocation of resources by the ORB should be avoided. First because this reduces or avoids the unpredictability which is a consequence of resource allocation. Second because this eliminates the resource exhaustion problem. Pre-allocation of resources is a first step to enforce end-to-end predictability. To control this, the realtime ORB defines the following resource APIs:

- Realtime Thread API
- Request Queue API
- Transport Management API
- Buffer Management API

The section "Control of Resources" focuses on the definition and the management of these resources. Resource management APIs are generic enough to only define mechanisms for controlling these resources. Some specific APIs are also defined for the Realtime ORB to provide a minimal policy for managing resources. Specific realtime policies for managing resources may also be defined by realtime applications (if that is necessary). The use of these resources and their interactions with other components is discussed later.

8.2 Threads

Threads are used at two levels in a CORBA application. First, they are used by the ORB to wait for incoming requests, do the dispatching and execute the servant code. Second, they may be used by applications for their own needs. For the purpose of controlling ORB resources, we are only interested in the first category, however the Realtime Thread API will address both. The realtime application must be able to specify the characteristics and the number of threads that the ORB will use for dispatching requests. The Realtime Thread API is organized in four parts:

- Thread attributes define the characteristics of the thread and specify what can be configured by realtime applications.
- A thread management API provides control of threads: including their creation and deletion as well as changing their attributes at run-time.
- Thread specific data is a mechanism which allows applications to associate specific data to a thread.
- Threads can be grouped in a thread pool concept to allow the control of ORB runtime threads by realtime applications.

The Realtime Thread API is generic and in general does not specify policies for threads. However, POSIX being a widely used standard, the API defines an interface for controlling POSIX threads. Other thread policies like Win32 threads may easily be defined.

8.2.1 Thread Attributes

8.2.1.1 Architectural Considerations

Thread attributes represent the characteristics of the thread. For example thread attributes may be defined to control the priority of threads, their scheduling class and so on. Attributes are defined as an interface so that the API is generic and can be extended for specific thread implementations. The top-level interface is empty.

8.2.1.2 Specification

```
// IDL
module RT {
    interface ThreadAttributes { // locality constrained
    };
};
```

The **ThreadAttributes** interface represents the abstract thread attributes. Specific thread attributes are defined by creating an interface that derives from this abstract interface.

8.2.1.3 Locality Constraints

A **ThreadAttributes** object must be local to the process.

8.2.2 POSIX Thread Attributes

8.2.2.1 Architectural Considerations

The POSIX 1003.1b standard being widely used, the realtime ORB defines, as a minimum of specialization of the generic thread attributes, a set of thread attributes to control POSIX threads.

8.2.2.2 Specification

```
module RT {
    native StackAddr;
    interface PosixThreadAttributes : ThreadAttributes {
                                        // locality constrained
        struct SchedParam {
            long priority;
        };
        enum SchedulerType {
            SCHED_OTHER,
            SCHED_FIFO,
            SCHED_RR
        };

        attribute SchedParam sched_attr;
        attribute SchedulerType sched_policy;
        attribute unsigned long stack_size;
        attribute StackAddr stack_addr;
    };
};
```

The **PosixThreadAttributes** interface defines the most important parameters present in POSIX.

- The **stack_addr** can be specified, and the type of this parameter is implementation defined. The definition of the stack address is necessary for some realtime applications.
- The **sched_attr** member represents the POSIX scheduling parameters.
- The **sched_policy** member indicates the scheduling class. POSIX only mandates the fifo and round robin policies. The fifo scheduling class is represented by the **SCHED_FIFO** constant value and the round robin scheduling class is represented by the **SCHED_RR** constant value.
- The **stack_size** member is used to configure the size of the stack.

8.2.2.3 Locality Constraints

A **PosixThreadAttributes** object must be local to the process.

8.2.2.4 Example

The example below indicates how to create the POSIX thread attributes for creating a thread which:

- uses the FIFO scheduling class,
- uses a priority of 61,
- has a 16Kb stack.

```
// C++
RT::PosixThreadAttributes posix_attributes;

posix_attributes.set_sched_policy(SCHED_FIFO);
posix_attributes.set_sched_attr(61);
posix_attributes.set_stack_size(16 * 1024);
```

8.2.3 Thread Management

8.2.3.1 Architectural Considerations

Applications need to create threads and control them. The Realtime Thread API is used to:

- Create new threads with specific thread attributes
- Delete an existing thread
- Get or set the thread attributes

The ability to suspend and resume a thread is not proposed in the generic API because this is non-portable and the semantic of such operations would be difficult to specify. However, specific policies for managing threads may provide such operations.

8.2.3.2 Specification

```
module RT {
    interface Thread { // locality constrained
        attribute ThreadAttributes attr;

        void join(out long status);
        void detach();
    };
    interface CurrentThread : Thread{ // locality constrained
        void exit(in long status);
        ...
    };
    native ThreadParam;
    interface ThreadHandler { // locality constrained
        long run(in ThreadParam param);
    };
    interface ThreadFactory { // locality constrained
        Thread create_thread(in ThreadAttributes attr,
                             in ThreadHandler entry,
                             in ThreadParam param);
        CurrentThread get_current_thread();
        ...
    };
};
```

The **Thread** interface defines the operations for controlling a thread created by the ORB. The control is always local: the object reference cannot be transmitted in remote invocations. A thread can change or obtain attributes from another thread but both threads will belong to the same process.

The **attr** attribute defines the current attributes of the thread. Depending on the thread policies supported by the interface, the returned attributes will represent a more derived type than **ThreadAttributes**.

The **attr** attribute is also used to change some thread attributes. If some attributes cannot be changed, the **CORBA::NO_PERMISSION** system exception is raised. If some attributes have a wrong value, the **CORBA::BAD_PARAM** system exception is raised. The attributes which can be changed while the thread is running depend on the threading attributes. With the **PosixThreadAttributes**, only the **sched_attr** and **sched_policy** attributes can be changed. Changing the stack size or the stack address is not permitted. The validity of values specified in the thread attributes also depends on the thread attributes specification.

The **join** operation waits for the termination of a thread. The caller is blocked until the thread to join has terminated. Only one thread is able to wait for the termination of another thread. The exit status of the thread is returned in the status parameter.

The **detach** operation detaches the thread. When a thread is detached, no **join** operation is possible on it. Detaching a thread must be made if the thread is not subject to be joined. By doing so, system resources are freed automatically upon termination of the thread.

The **CurrentThread** interface represents the current thread. This interface also defines other operations which are presented later in §8.2.4 "Thread Specific Storage" on page 25.

The **exit** operation stops the current thread with a status. This exit status can be retrieved by another thread by calling the **join** operation.

The object reference which identifies the current thread can be obtained by using the **RT::ThreadFactory::get_current_thread** operation.

A **ThreadFactory** is the interface which allows the creation of new threads. The thread execution flow (e.g., the method) is represented by the **ThreadHandler** interface: when a thread is created, it executes the **run** operation. Thread creation can only be made locally, this is why the interface is defined with locality constraints. How to obtain a ThreadFactory object reference is explained in §8.6 "Strategy Factory" on page 40. Other operations are defined on the **ThreadFactory** interface and are presented in §8.2.4 "Thread Specific Storage" on page 25 and in §8.2.5 "Thread Pools" on page 28.

The **create_thread** operation creates a new thread and returns an object reference to control it. The thread is created with the attributes which are passed in the **attr** parameter. Once created, the thread will execute the **run** operation with the parameter **param** on the **ThreadHandler** object passed in **entry**. When the thread returns from the **run** operation, it terminates and is deleted. The return value of the **run** operation can be retrieved by another thread by calling the **join** operation.

8.2.3.3 Locality Constraints

A **Thread, ThreadHandler** and **ThreadFactory** objects must be local to the process.

8.2.4 Thread Specific Storage

8.2.4.1 Architectural Considerations

Thread specific storage is a very important mechanism for multi-threaded applications: it allows the application to associate specific data to a particular thread. Thread specific storage is used in the following situations:

- A typical usage is the use of specific data for storing the value of the **errno** system error code: each thread has its own knowledge of system errors.
- Thread specific storage improves the performance and simplifies multi-threaded applications by reducing the overhead of acquiring and releasing locks: thread specific storage does not need to be protected by locks. This is of a primarily importance, specially for allocation of resources. For example, thread specific storage can be used to associate a set of memory buffers to a particular thread. By doing so, when the thread needs some memory it allocates it by using its thread specific information. No lock is necessary, thus eliminating the possible priority inversion of threads that can occur if thread specific storage is not used.

The realtime threading API defines a thread specific data abstraction and specifies some operations to manipulate this abstraction.

Thread specific storage is associated to a key. Keys are global to a process: they are common to all threads. A thread can define several thread specific storage by associating each of them a separate key.

8.2.4.2 Specification

```
module RT {
    typedef unsigned long ThreadSpecificKey;
    interface ThreadSpecific { // locality constrained
        void destroy();
    };
    interface CurrentThread : Thread { // locality constrained
        ...
        void set_thread_specific(in ThreadSpecificKey key,
                                 in ThreadSpecific data);
        ThreadSpecific get_thread_specific(in ThreadSpecificKey key);
        void remove_thread_specific(in ThreadSpecificKey key);
        boolean has_thread_specific(in ThreadSpecificKey key);
    };
    interface ThreadFactory { // locality constrained
        ...
        ThreadSpecificKey create_thread_specific_key();
        void destroy_thread_specific_key(in ThreadSpecificKey key);
    };
};
```

The **ThreadSpecific** interface represents the thread specific storage abstraction. Application thread specific storage must be defined as an interface which derives from that interface. The derived interface holds the specific data.

The **destroy** operation is called by the ORB to delete the thread specific storage. This operation is called when a thread exits and has thread specific storage. It is also called by the ORB when the thread specific storage is removed.

The **set_thread_specific** operation associates the thread specific storage to a particular key. Once set, the thread keeps ownership of the thread specific storage. Applications must not release the thread specific storage. The previous storage is overridden if there was one. In that case, the **destroy** operation is called on the previous storage. The **CORBA::BAD_PARAM** system exception is raised if the key is not valid.

The **get_thread_specific** operation retrieves the thread specific storage. The content of the storage can be read and also modified. When it is modified, it is not necessary to call **set_thread_specific** to update the thread specific storage. The caller must not free the thread specific storage.

The **remove_thread_specific** operation deletes the thread specific storage associated to a particular key. The **destroy** operation is called by the ORB on the calling thread specific storage. This is the only way for deleting explicitly the thread specific storage object.

The **has_thread_specific** operation returns the boolean value **CORBA::TRUE** if the calling thread has a specific storage for a given key. No exception is raised by this operation.

The **ThreadFactory** defines two operations for creating and deleting the thread specific keys. The **create_thread_specific_key** operation allocates a new thread specific key. If no more keys can be allocated due to implementation limits, the **CORBA::IMPL_LIMIT** system exception is raised.

The **destroy_thread_specific_key** operation deallocates the thread specific key. If the key is not valid, the **CORBA::BAD_PARAM** system exception is raised. Due to the error prone nature of this operation, an implementation may impose some restrictions about this operation including refusing it by raising the **CORBA::NO_PERMISSION** exception.

8.2.4.3 Locality Constraints

A **ThreadSpecific** object must be local to the process.

8.2.4.4 Example

The example below illustrates how an application can attach specific data to a thread and retrieve the data later. First of all, a specific data is defined by inheriting from the **ThreadSpecific** interface. The thread factory is used to create a new thread specific storage key. The creation of the key is made only once, at initialization time. The thread specific storage is allocated and attached to the thread.

```
// C++
class ThreadQoS : RT::ThreadSpecific {
   public:
      CORBA::ULong priority;
};

// C++, Initialization
RT::ThreadFactory_ptr thFactory = ...;
RT::ThreadSpecificKey qosKey =
               thFactory->create_thread_specific();

// Set specific storage
RT::CurrentThread_ptr th = ...;
         // Obtain the thread object reference
ThreadQoS* myQoS = ...;
         // Create the specific storage
myQoS->priority = 23;
th->set_thread_specific(qosKey, myQoS);
```

8.2.5 Thread Pools

8.2.5.1 Architectural Considerations

Threads may be grouped to represent a thread pool. The thread pool represents a set of threads which have, more or less the same characteristics (e.g., scheduling class, scheduling attributes, stack size,...) and are used to process invocations. Thread pools are used by the realtime ORB for various purposes:

- First, they are used by the transport layer to wait for incoming requests. The ability to control the number of threads as well as their attributes is a first step to guarantee the end-to-end predictability of the ORB.
- Second, they are used by the object adapter to dispatch the request. Here, thread pool characteristics allow control of the threads which will execute application code in the servant.
- Finally, they are used on the client side to receive replies and process deferred asynchronous replies.

The same thread pool may be used for both purposes: 1) to wait for incoming requests, and 2) to dispatch them. This avoids unnecessary thread context switches.

A thread pool is characterized by the following information:

- the number of threads in the pool,
- the attributes of the threads belonging to the pool: stack size, scheduling attributes,...
- a request queue which is attached to the pool (request queue management is discussed later),
- a policy for managing threads of the pool.

The thread pool API is generic and does not mandate any thread pool policy. Specific thread pool policies may be implemented by realtime applications if that is necessary. However, as a minimal behavior, it creates several threads with the same characteristics when a thread pool is created.

The Realtime Thread API is used to change thread attributes. Therefore, when a thread of the pool is used to process invocations, it is able to change its attributes. It is the responsibility of applications to restore the thread attributes, if this is necessary.

Figure 4, “Thread Pool Usage Example,” on page 29 illustrates a typical use of a thread pool. First, the ORB defines threads to wait for incoming requests. These threads are represented by the thread pool A. When an incoming request is received by one of these threads the request is queued on the request queue associated to the thread pool A. The realtime portable object adapter defines threads to process the requests. These threads are represented by the thread pool B. This second thread pool is also associated to the request queue. Threads of pool B extract from the request queue the requests that must be processed. Requests are transfered from threads of pool A to threads of pool B. In this example the two thread pools introduce an extra thread context switch. Obviously, the Realtime ORB allows to define other designs in which no extra thread context switch will occur.

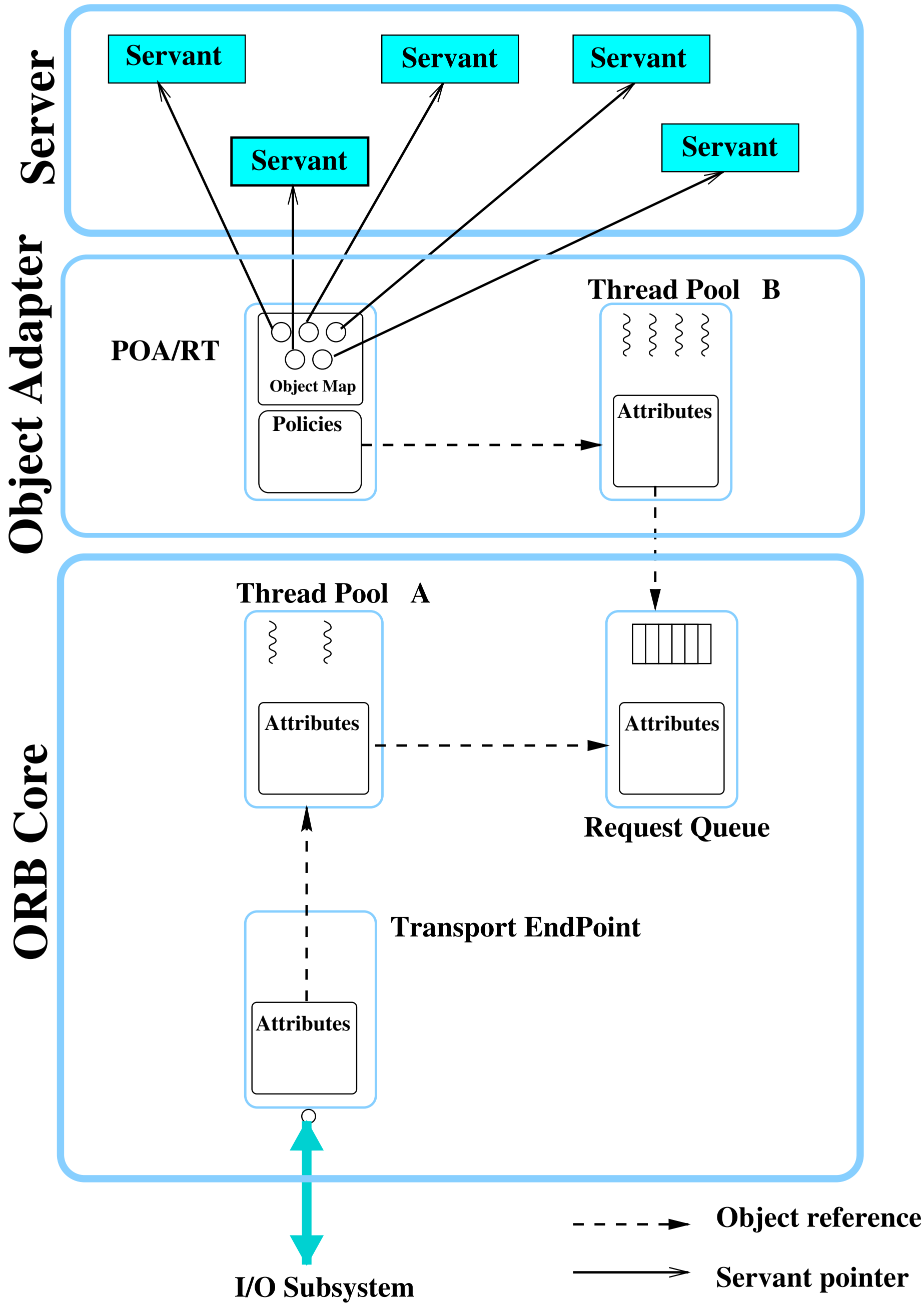

Figure 4 Thread Pool Usage Example

8.2.5.2 *Specification*

```
module RT {
    interface RequestQueue;
    interface ThreadPoolAttributes { // locality constrained
        attribute ThreadAttributes thread_attributes;
        attribute RequestQueue request_queue;
        attribute unsigned long number_of_threads;
    };
    interface ThreadPool { // locality constrained
        attribute ThreadPoolAttributes attr;

        void destroy();
    };
    interface ThreadFactory { // locality constrained
        ...
        ThreadPool create_thread_pool(in ThreadPoolAttributes attr);
    };
};
```

The **ThreadPoolAttributes** interface defines the global attributes of the thread pool. A specific implementation of a thread pool can provide new attributes by deriving from this interface. The global attributes are the following:

- **thread_attributes**
  This attribute specifies the characteristics of threads which are in the pool.
- **request_queue**
  This attribute specifies the request queue which is attached to the thread pool. Incoming requests received by threads of the pool are queued on this request queue. Threads of the pool which process requests obtain requests from this request queue. The **RequestQueue** interface is described in §8.3 “Request Queue and Flow Control” on page 31.
- **number_of_threads**
  This attribute indicates the current number of threads which are in the pool.

The **ThreadPool** interface defines the thread pool abstraction. Different thread pool semantics may be defined: first by setting different thread pool attributes and second by using different thread pool factories.

The **attr** attribute defines the attributes of the thread pool. In particular, it indicates in **number_of_threads** the actual number of threads that the pool contain. Depending on the thread pool semantic, a more derived interface than **ThreadPoolAttributes** may be returned.

The **attr** attribute is also used to change the attributes of the thread pool. An implementation may refuse to change the thread attributes of threads which already exist.

The **destroy** operation deletes the thread pool and the threads which are part of it.

The **create_thread_pool** operation creates a new thread pool with some initial characteristics. The factory object which is used determines the characteristics of the thread pool. Thread pool attributes which are passed may depend on the specific thread factory. The minimum semantic is to create **number_of_threads** threads with the attributes **thread_attributes**. If not all the requested number of threads could be created, the operation fails. This minimum semantic provides a static behavior for the thread pool.

8.2.5.3 Locality Constraints

A **ThreadPoolAttributes** and **ThreadPool** objects must be local to the process.

8.2.5.4 Example

The example below illustrates how to create a fixed size thread pool. First, thread attributes are defined (see the example in §8.2.2 “POSIX Thread Attributes” on page 22). Then, the thread pool attributes are defined to use the Posix thread attributes and it is configured to have 4 threads. The **create_thread_pool** operation is invoked on the thread factory. The factory creates 4 threads with the specified Posix thread attributes. It then returns the object reference for controlling and using the thread pool. Threads of the pool are waiting for jobs to be executed.

```
// C++
RT::PosixThreadPoolAttributes posix_attributes;
// ... see §8.2.2 "POSIX Thread Attributes" on page 22
RT::ThreadPoolAttributes pool_attributes;
RT::ThreadPool_var thread_pool;

pool_attributes.set_thread_attributes(posix_attributes);
pool_attributes.set_number_of_threads(4);
thread_pool =
      thFactory->create_thread_pool(pool_attributes);
```

8.3 Request Queue and Flow Control

8.3.1 Architectural Considerations

On the server side, the Realtime ORB receives client requests. To correctly process them a request queue is necessary:

- Basically, queuing is necessary for the scheduler to organize requests. Requests may be organized according to their priority (or QoS).
- Threads are scarce resources. In case of heavy load, it may be better to queue requests instead of just creating new threads.
- The request queue is a first element for a flow control mechanism.

The request queue is used to control:

- the number of requests in the queue,

- the organization of requests within the queue,
- the policy for keeping requests in the queue and for the flow control mechanism.

The request queue API is generic and does not define any particular policy. Specific policies may be implemented. For example:

- Some control parameters could include: the number of requests, the maximum size of request, the maximum time that a request can wait without being processed, the request priority and so on.
- Strategies for enqueuing requests could be defined: FIFO and priority order, then application-defined via a callback/interceptor API.
- A flow control can be introduced when the request queue is full, or when a request is waiting for too much time or whatever. The client can be informed of the server congestion and do whatever is appropriate (load balancing, wait+retry, abort, ...).
- The request queue can be attached to a thread pool so that when a thread is ready it can obtain a request from the queue and process it.

8.3.2 Specification

```
module RT {
    interface RequestQueueAttributes {
                              // locality constrained
    };
    interface RequestQueue { // locality constrained
        attribute RequestQueueAttributes attr;

        void put_request(in CORBA::ServerRequest req);
        CORBA::ServerRequest get_request();
        unsigned long pending_requests();
        void destroy(in long mode);
    };
    interface RequestQueueFactory {
                              // locality constrained
        RequestQueue create_request_queue(
                              in RequestQueueAttributes attr);
    };
};
```

The **RequestQueue** interface represents the top-level request queue and the **RequestQueueAttributes** interface represents attributes for controlling the request queue. No particular policy is defined in this request queue. This is why the **RequestQueueAttributes** interface is empty. Specific policies may be provided by creating interfaces which derive from these top-level interfaces.

The **attr** attribute defines the characteristics of the request queue. Depending on the request queue, a more derived interface than **RequestQueueAttributes** may be returned.

The **attr** attribute is also used to change the attributes for controlling the request queue. The semantics of the request queue attributes and whether they may be changed dynamically depends on the policy of the request queue.

The **put_request** operation inserts a new request in the request queue. The position of the new request within the queue depends on the policy of the queue.

The **get_request** operation extracts a request from the queue.

The **pending_requests** operation returns the number of requests which are present in the queue.

The **destroy** operation deletes the request queue. Specific policies may be implemented for destrying the request queue, including refusing the operation by raising the **CORBA::NO_PERMISSION** exception. The mode parameter specifies whether the queue must be deleted abruptly or gracefuly.

The **RequestQueueFactory** interface represents a factory for creating request queues. Several factories may exist and may create request queues with specific policies. How to obtain a **RequestQueueFactory** object reference is explained in §8.6 “Strategy Factory” on page 40.

The **create_request_queue** operation creates a new request queue. The initial characteristics of the request queue can be specified at creation time by passing appropriate request queue attributes.

8.3.2.1 Locality Constraints

A **RequestQueueAttributes**, **RequestQueue** and **RequestQueueFactory** objects must be local to the process.

8.4 Transport

Transport resources represent the resources which are managed by the transport layer of the ORB. A realtime application may need to create several transport end-points. An end-point is an object created and maintained by the transport layer. It is used by applications for sending and receiving data. An end-point identifies the source or destination of a message. The end-point has several characteristics:

- Localization attributes are used for physically sending or receiving data. With TCP/IP, localization attributes consist of the Internet address and the TCP/IP port. In most cases, localization attributes depend on the transport media.
- Threads which are waiting behind the end-point for messages.
- Other attributes and policies, like buffers, strategies for receiving or emitting messages and so on.

The support of multiple transport end-points and the ability to control these end-points is an important element of a realtime ORB. The Transport API is organized as follows:

- Transport attributes are identified. A generic representation is proposed and specific representation for TCP/IP is also given.

- The API for creating and managing transport end-points is defined
- A representation for transport connections is defined

8.4.1 Transport End-Point Management

The realtime ORB transport layer defines an API that allows the creation of new transport end-points and is also used to control these transport end-points. The Transport API gives an abstraction of the transport mechanism used by the ORB. This abstraction is independent of any transport media. Among the transport control facilities that it defines, the Transport API also increases considerably the portability of applications. The Transport API is generic and is suitable for all kinds of transports. It allows designers to specify some transport end-point parameters (e.g., the TCP/IP port number and so on). Some protocol-specific examples follow:

- TCP/IP
  With the TCP/IP transport layer, the creation of a new end-point will result in the allocation of a TCP/IP port and the ability for the ORB to accept connections on this port. The end-point may be associated with TCP/IP configuration parameters, in which case these parameters will be used to configure the connection when it is open.

The Transport API is used to:

- Create a new transport end-point
- Delete a transport end-point (abruptly or gracefully)
- Get or set some attributes related to the transport end-point
- Control the transport media by opening or closing the connection.

The Transport API also gives the ability to control the threads that the transport layer of the ORB will use to receive and send messages. A transport end-point is associated with a thread pool. The thread pool represents the threads that the transport layer uses for receiving and sending messages. Therefore, applications can control the number of threads as well as their characteristics.

Figure 5, “Transport End-Point Configuration Example,” on page 35 illustrates a possible configuration of a transport end-point. A thread pool is created and associated with the transport end-point. Threads of the pool are used by the transport layer of the ORB to receive incoming requests. The realtime portable object adapter is also configured to use the same thread pool. When a request is received by the transport layer of the ORB, the realtime portable object adapter will dispatch the request. This configuration does not create a thread context switch between the thread which receives the request and the thread which processes it: the same thread is used.

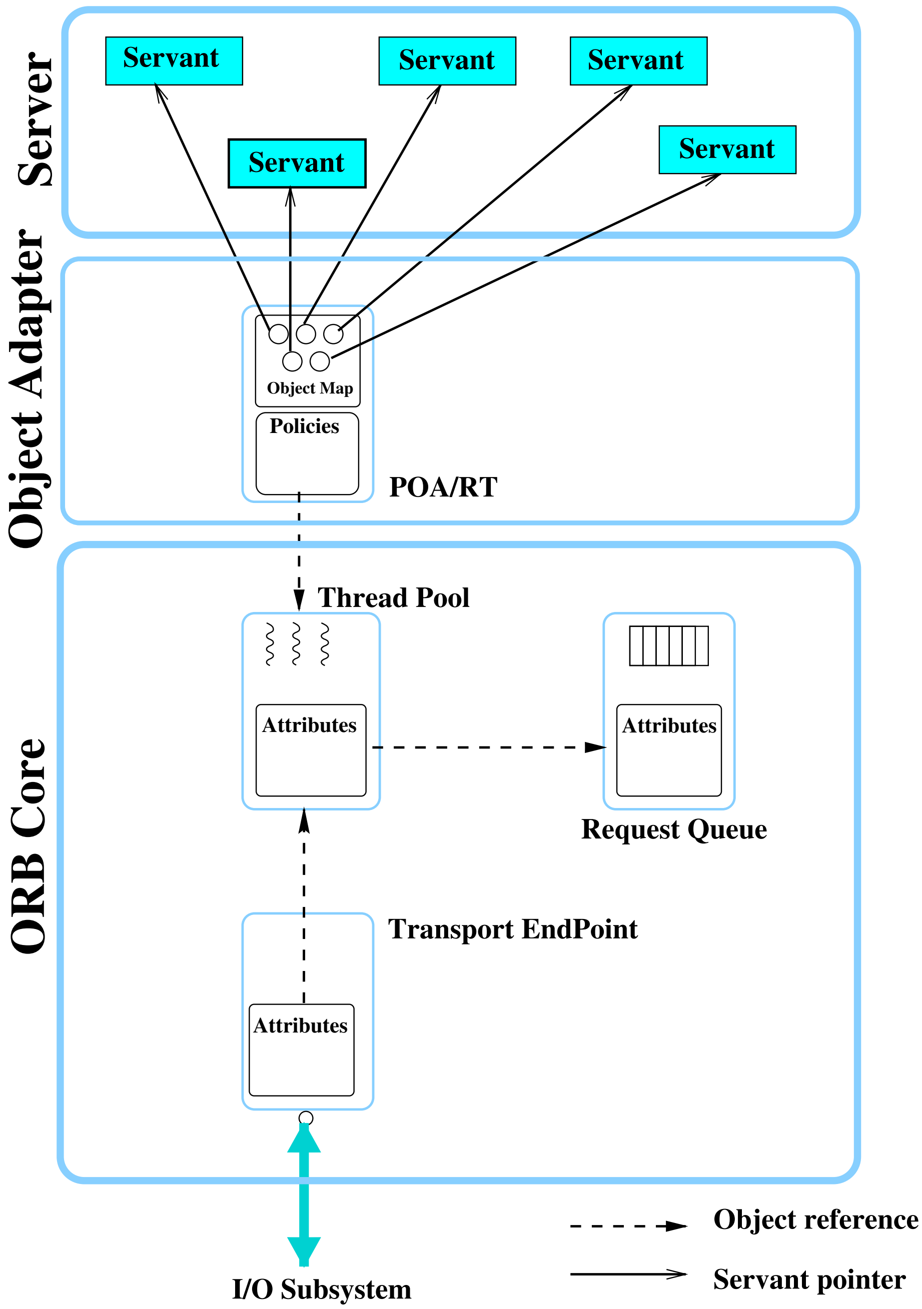

Figure 5 Transport End-Point Configuration Example

8.4.2 Transport Attributes

8.4.2.1 Architectural Considerations

Transport attributes depend on the characteristics of the transport media. The API is extensible so that it does not depend on the transport media. For this, transport attributes are represented by an interface with locality constraints. This is used to integrate new transports by providing new interfaces that derive from the top-level transport attributes. One common attribute is:

- the thread pool which is associated to the end-point for receiving requests

8.4.2.2 Generic Specification

```
module RT {
    interface TransportAttributes { // locality constrained
        attribute ThreadPool thread_pool;
    };
};
```

The **TransportAttributes** interface defines the top-level transport attributes. This only defines the **thread_pool** attribute which is used to specify the thread pool that defines the threads to wait for incoming invocations. Any kind of transport attributes must derive from this interface.

8.4.2.3 Locality Constraints

A **TransportAttributes** object must be local to the process.

8.4.3 TCP/IP Attributes

8.4.3.1 Architectural Considerations

TCP/IP is a commonly used transport mechanism. As a minimal policy for controlling a transport, the TCP/IP attributes are defined. Most of the parameters which may be controlled are those which are defined by the TCP/IP layer.

8.4.3.2 *Attributes Specification*

```
module RT {
    interface TcpTransportAttributes : TransportAttributes {
                                    // locality constrained
        attribute long tcp_send_size;
        attribute long tcp_recv_size;
        attribute boolean tcp_keep_alive;
        attribute boolean tcp_dont_route;
        attribute unsigned short tcp_port;
        attribute unsigned long tcp_addr;
    };
};
```

The **TcpTransportAttributes** interface defines the attributes for configuring the TCP/IP transport end-point. All the attributes except **tcp_port** and **tcp_addr** can be modified dynamically after the transport end-point is created. The TCP/IP attributes have the following meaning:

- **tcp_send_size**
  This attribute is used to control the size of the TCP/IP send buffers in the OS driver. It corresponds to the **SOL_SNDBUF** socket option.
- **tcp_recv_size**
  This attribute controls the size of the TCP/IP receive buffers in the OS driver. It corresponds to the **SOL_RECVBUF** socket option.
- **tcp_keep_alive**
  This attribute defines whether the TCP/IP stack must keep the connection alive by sending messages. This corresponds to the **SOL_KEEPALIVE** socket option.
- **tcp_dont_route**
  This attribute enables the routing bypass for outgoing messages. It corresponds to the **SOL_DONTROUTE** socket option.
- **tcp_port**
  This attribute defines the TCP/IP port number associated to the transport end-point. This attribute can be set only before the creation of the transport end-point. If no value is specified at the creation time, the TCP/IP transport will allocate one.
- **tcp_addr**
  This attribute specifies the Internet address of the transport end-point. It can be set only before the creation of the transport end-point.

8.4.4 *Locality Constraints*

A **TcpTransportAttributes** object must be local to the process.

8.4.5 Transport End-Point API

The Transport Management API consists of two parts. First, the transport end-point is controlled by the **TransportEndPoint** interface. Second, a given transport media provides a **TransportEndPointFactory** interface for creation and deletion of end-points. How to obtain a **TransportEndPointFactory** object reference is explained in §8.6 "Strategy Factory" on page 40.

```
module RT {
    interface TransportEndPoint { // locality constrained
        attribute TransportAttributes attr;

        boolean is_equal(in TransportEndPoint endPoint);
        string type();
        string to_url();
        void from_url(in string id);
        unsigned long hash(in unsigned long maximum);
        void open(in TransportAttributes a);
        void close();
        void destroy(in long mode);
    };
    interface TransportEndPointFactory {
                                        // locality constrained
        TransportEndPoint create_end_point(
                                        in TransportAttributes attr);
        unsigned long number_of_end_points();
        TransportEndPoint get_end_point(in unsigned long pos);
    };
};
```

The **attr** attribute defines the characteristics of the the transport end-point. The returned interface is an interface that derives from the **TransportAttributes** interface.

The **attr** attribute is also used to change the transport attributes. It can be used to specify the thread pool which is attached to the transport end-point. The transport attributes which are specified as parameter may be generic (**TransportAttributes**) or specific to a transport media (e.g., **TcpTransportAttributes**). The semantic of changing the transport attributes is specific to the transport media. In any case, the transport media should raise **CORBA::BAD_PARAM** if some transport attributes have a wrong value and it should raise **CORBA::NO_PERMISSION** if some attributes cannot be changed.

The **is_equal** operation is provided to compare two transport end-points and see whether they are identical. If the two transport end-points are not of the same transport media, they are not equal and the operation returns **CORBA::FALSE**.

The **type** operation returns a string which identifies the transport type.

The **to_url** and **from_url** operations convert the transport end-point to a URL string or a URL string to a transport end-point. The URL format is specific to the transport media.

The **hash** operation returns a hash number which corresponds to the transport end-point. Together with the **is_equal** operation, it may be used to store the transport end-points in a hash table and implement insertion and search primitives with a transport end-point as the key.

The **open** operation is used by clients to open the connection explicitly. Attributes for openning the connection are specified in **attr**.

The **close** operation is used by clients to explicitly close a connection.

The **destroy** operation destroys the transport end-point. The mode parameter controls the destruction of the end-point. Some transport media may support an abrupt destruction and also a graceful destruction.

The **create_end_point** operation creates a new transport end-point. The transport media that the new transport end-point uses depends on the **TransportEndPointFactory** object reference onto which the operation is made. Generic attributes or specific attributes can be passed at creation time. Specific attributes may be used to configure the end-point during the creation.

The **number_of_end_points** operation returns the number of transport end-point which have been created for the given transport.

The **get_end_point** operation returns a transport end-point. Together with **number_of_end_points**, it can be used to iterate over all the transport end-points which exist.

8.4.6 Locality Constraints

A **TransportEndPoint** and **TransportEndPointFactory** objects must be local to the process.

8.4.7 Example

The example below shows how to create a new TCP/IP transport end-point and configure it. First, the TCP/IP attributes are filled: the TCP/IP port is assigned the value 2002 and the TCP/IP send buffers are configured for 64Kb. Once created, the new transport end-point is identified by the **tcpEndPoint** object reference. After the transport end-point creation, the end-point is configured directly by using its attribute.

```
// C++
RT::Transport_var tcp = ...; // Obtain TCP/IP transport
object reference
RT::TransportEndPoint_ptr tcpEndPoint;
RT::TcpTransportAttributes tcpAttr;

// Create the TCP/IP end-point
tcpAttr.tcp_port = 2002;
tcpAttr.tcp_send_size = 64*1024;
tcp->create_end_point(attr, tcpEndPoint);

// Configure the end-point
RT::ThreadPool_ptr thPool = ...; // Obtain the thread pool
tcpAttr.thread_pool = thPool;
tcpEndPoint->attr(tcpAttr);
```

8.5 Buffers

The realtime ORB defines a Buffer API for controlling memory allocation. Pre-allocation of memory is necessary for realtime application: the realtime ORB must ensure that enough memory is available to process a request and it must not be blocked to wait for the availability of memory.

Note – The first submission does not define an API for this. The submitters are working on the subject and will propose a complete API for the second submission.

8.6 Strategy Factory

8.6.1 Architectural Considerations

Threads, requests and transport end-points represent important scheduling entities. Specific policies for managing these entities may be defined. To obtain the different scheduling entities that cooperate for a given scheduling policy, an interface is necessary. Such an interface acts as the entry point for obtaining the different factories. This interface, refered to as the realtime strategy factory interface, provides a uniform and coherent view of the realtime policies provided by the different resource factories.

8.6.2 Specification

```
module RT {
    interface StrategyFactory {
        ThreadFactory get_thread_factory();
        TransportEndPointFactory get_transport_factory(in string name);
        RequestQueueFactory get_request_queue_factory();
        ...
    };
};
```

The **StrategyFactory** interface represents the entry point to control ORB resources. It is used to obtain the Realtime Thread Factory, the Transport End-Point Factory and the Request Queue Factory. Depending on the realtime strategy object reference, the different factories which are returned may implement specific policies. The realtime strategy object reference is obtained by using the **ORB::resolve_initial_references** operation with the name "**RT_Strategy**".

The **get_thread_factory** operation returns the thread factory which allows creation of threads and thread pools.

The **get_transport_factory** operation returns the transport end-point factory for the transport identified by name.

The **get_request_queue_factory** operation returns the request queue factory which allows creation of request queues.

8.6.3 Locality Constraints

A **StrategyFactory** object is local to the process.

8.6.4 Example

The example below shows how to obtain the realtime strategy object reference.

```
CORBA::Object_var obj;
RT::StrategyFactory_var strategy;

obj = orb->resolve_initial_references("RT_Strategy");
strategy = RT::Strategy::_narrow(obj);
```

The example below indicates how the thread factory can be obtained by using the realtime strategy object.

```
RT::ThreadFactory_var thFactory;
thFactory = strategy->get_thread_factory();
```

9 ORB Flexibility Enablers

The flexible enablers presented in this chapter are important key elements for a flexible realtime ORB architecture. They allow an integration of specific realtime strategies such as QoS support, priority inheritance and others. The flexible enablers are composed of two parts:

- The first element is the ability for the ORB to let applications store arbitrary data in an object reference. This key element allows multiple forms of binding to coexist in the same ORB, and allows servers to store its realtime characteristics in the object reference.
- The second element is the support of an interceptor mechanism to be able to change or control the semantic of method invocation.

9.1 Componentized Object References

A componentized object reference is an object reference into which applications can store arbitrary data. When the object reference is passed in method invocation, data that was stored in it is also passed. The primary goal of storing data in the object reference is to allow applications to support the following:

- The servant can store some information in the object reference and retrieve this information when the client performs an invocation.
- Scalable compound objects can be defined easily by storing specific keys in the object reference and retrieving the key in the servant when the application performs an invocation.
- Characteristics of the server such as the protocols that it supports can be stored in the object reference. On the client side, such information could be used to optimize invocations by using the best protocol.
- If a server supports a specific protocol, information to establish the connection and/or to use that specific protocol can be stored in the object reference.
- Information to manage replicated servers and/or to provide a fault tolerant mechanism can be stored in the object reference and later can be fetched when needed.

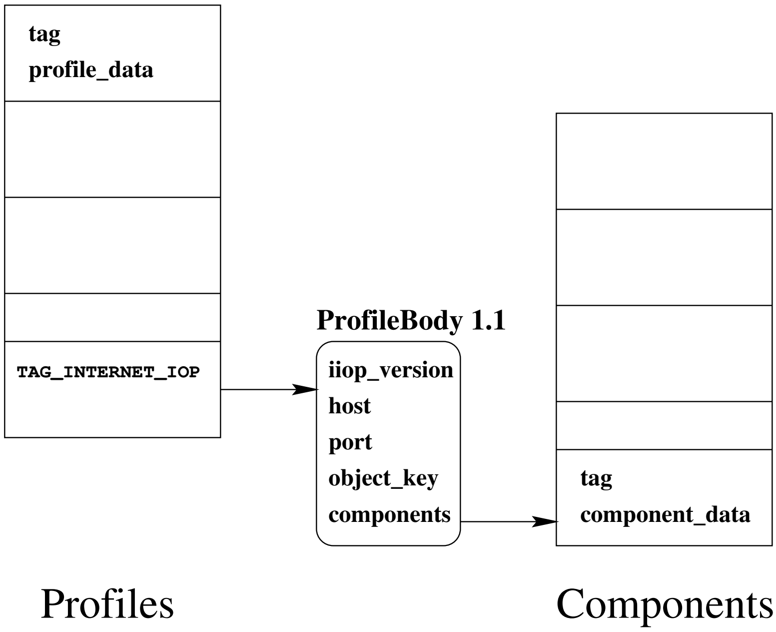

Figure 6 Object Reference Components

9.1.1 *Architectural Considerations*

Data which is inserted in the object reference is represented by a sequence of octets and is associated with a tag number. Data components which are stored in the object reference correspond to the IOP tagged profile structure **IOP::TaggedComponent**. The semantic is different: it is assumed that these data components are not removed from the IOR when it is passed to a bridge or another ORB.

The manipulation of data components can be made at three levels:

- **IOR_PROFILE_LEVEL**
  The data component is inserted, extracted or removed from the profile level of the IOR. The data component corresponds to a plain profile. According to GIOP specification, it can be removed by bridges or ORBs.
- **IIOP_PROFILE_LEVEL**
  The data component is inserted, extracted or removed from the **TAG_INTERNET_IOP** profile. In that case, because the data component is part of the **ProfileBody**, it can not be removed.
- **IIOP_KEY_LEVEL**
  The data component is inserted, extracted or removed in the key of the **TAG_INTERNET_IOP** profile.

Note – The submitters are working on the interface and will update it for the second submission.

9.1.2 Specification

The **CORBA::Object** definition is extended by defining four new operations which are used to access the data components.

```
#include <IOP.idl>
module CORBA {
    typedef IOP::ComponentId ComponentId;
    typedef sequence<octet> ComponentData;

    const long IOR_PROFILE_LEVEL = 1;
    const long IIOP_PROFILE_LEVEL = 2;
    const long IIOP_KEY_LEVEL = 4;
    const long EXCLUSIVE = 8;

    interface Object { // locality constrained with special operation mapping
        void set_component(in ComponentId id,
                           in long flags, in ComponentData d);
        void get_component(in ComponentId id,
                           in long flags, out ComponentData d);
        void remove_component(in ComponentId id, in long flags);
        boolean has_component(in ComponentId id, in long flags);
    };
};
```

For all these operations, the flags parameter indicates at which level the component must be inserted. Flag values are defined as constants instead of enumeration type because they may be or-ed together.

The **set_component** operation inserts or replaces a component in the object reference. If the flags parameter contains the **EXCLUSIVE** flag, the **CORBA::NO_PERMISSION** system exception is raised if the component exists already.

The **get_component** operation retrieves a component given its identifier. The operation raises **CORBA::BAD_PARAM** if the component is not defined.

The **remove_component** operation removes a component given its identifier.

The **has_component** operation determines whether a component is present or not. It returns the **CORBA::TRUE** value if it is present and **CORBA::FALSE** otherwise. No system exception is raised.

9.2 Interceptors

Note – The submitters believe that the Interceptor facilty is of critical importance to the realization of a Realtime CORBA standard. The current OMG definition of interceptor facility is inadequate to serve these needs. The submitters are aware of revision activities that are in progress in this area. This section contains the current thinking of the type of facilities that realtime needs in this area. The plan is to rationalize and align this with existing interceptor facility as appropriate in the final revised submission.

The realtime ORB must be flexible enough to allow modifications of or provide new characteristics to a method invocation. For example, some realtime applications will need to pass the priority of the client and use that priority in the server to implement the priority inheritance. Other applications will pass more complete QoS information such as time constraints, importance, dependencies and so on. The realtime ORB cannot implement all the existing realtime strategies. Instead, an extensible API is defined to increase the flexibility of the realtime ORB and to allow implementation of new strategies. This API is defined in the form of interceptors.

An interceptor is an object that has several of its operations which are called by the ORB at different levels of an invocation. Interceptor objects are provided by applications. They may be seen as a kind of specialized callback mechanism. Interceptors are chained together and their operations are executed sequentially.

9.2.1 Interceptor Categories

Several interceptor categories may be defined. The realtime ORB defines the following categories:

- Client Interceptors
  This category represents interceptors which are invoked by the ORB when a remote invocation is made. The interceptor defines several operations which are called at different steps of the method invocation. Client interceptors may be used to do some monitoring or logging of method invocation as well as to transparently pass the specific QoS information requested by the application.
- Server Interceptors
  These interceptors are called by the ORB when a remote invocation is received and must be processed. Several operations are defined to catch the processing of the remote invocation. This kind of interceptor can be used to monitor the server activity, to log the method invocations as well as to extract the QoS information passed by the client interceptor.
- Portable Object Adapter Interceptors
  These interceptors are called by the portable object adapter on the server side. They give the ability to be notified when the server queries the POA to create a new object reference. Such interceptors are used in cunjunction with the componentized object reference API to automatically store information in the object reference (e.g., QoS information supported by the server, ...).
- Transport Interceptors
  Transport interceptors are used by the transport layer of the ORB to notify the realtime application about transport events.
- Thread Interceptors
  This category represents interceptors which are called when the state of a thread is changed.

- Initialization Interceptors
  Initialization Interceptors are called by the ORB when it initializes. They are intended to simplify the integration and initialization of specific realtime policies.
- Message Interceptors
  Message interceptors operate at the message level. They are used for encryption or compression of messages.

Figure 7, “Client and Server Interceptors,” on page 46 illustrates the position of client and server interceptors within the CORBA architecture. Basically, the client and server interceptors are activated in the transition between the stubs/DII and the interaction protocol and also in the transition between the interaction and transport protocols.

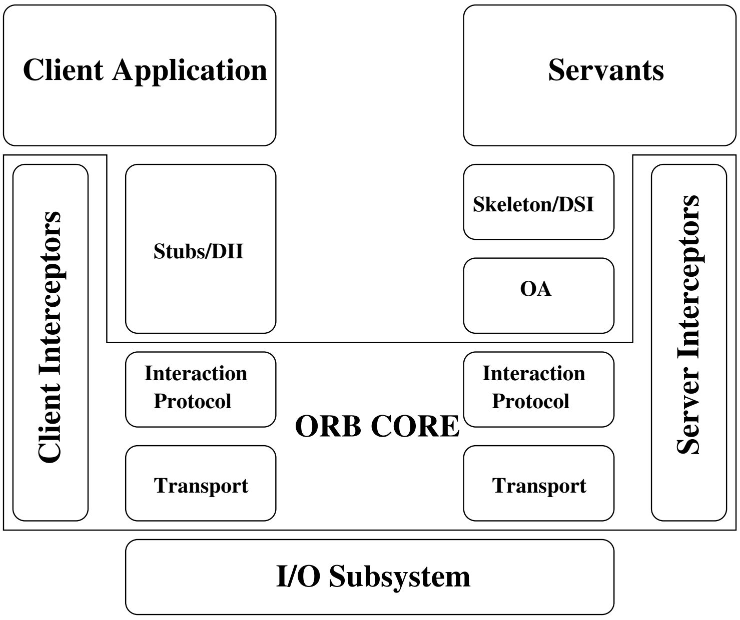

Figure 7 Client and Server Interceptors

9.2.2 General Rules

9.2.2.1 Architectural Considerations

Interceptors are objects provided by applications. Interceptors have several operations that are called by the ORB at different levels of a method invocation. The application can create several interceptor objects. In that case, interceptors of a given category are chained together and executed sequentially.

We define several kinds of interceptor objects, each of them providing several interceptor methods as stated above. The order of activation of these different methods is pre-defined by the ORB. As an example, when a client issues an invocation, request level interceptors are invoked before message interceptors on the way out. When the reply is received the message interceptors are invoked before the request interceptor.

When several interceptors are created, the interceptor methods of a given level (same method name of the same kind of interceptor object) are called sequentially. Therefore, it is necessary to specify the order in which they will be invoked. For this, each interceptor must be assigned a priority. This priority allows applications to define the order of execution of interceptors: interceptors are inserted in the chain according to their priority.

It is necessary to define two order of calls for interceptor method. The operation made by an interceptor on a message must be made at the same place in the state of the message. For example, if an interceptor encrypts a message and a second interceptor compresses the message, when the message is received we must first uncompress it and then decrypt the uncompressed message. This is possible only if we provide interceptor with the **Forward** and **Backward** semantics:

- **Forward**
  Some interceptor methods are called in the interceptor highest priority order. This means that the interceptor with the lowest priority has its method called last.
- **Backward**
  Opposite, some interceptor methods are called in the interceptor lowest priority order. The interceptor with the lowest priority has its method called first.

Interceptors follow the following rules:

- Creation
  Interceptor objects are created by applications. Once created, they are not yet active.
- Deletion
  Interceptor objects are released by applications.
- Activation and Deactivation
  The application must activate the interceptor explicitly. The priority of the interceptor object is specified when it is activated. Realtime ORBs may impose constraints on when activation or deactivation can occur (e.g., only at initialization time).
- Recursions
  An interceptor method can call the ORB as well as perform an invocation (remote or co-local). In those situations, and depending on what operation is made in the interceptor method, it may happen that the interceptor is called again by the ORB. It is the responsibility of applications to detect and break recursions if that is necessary.

9.2.2.2 Specification

```
module Interceptor {
    enum Status {
        INVOKE_CONTINUE,
        INVOKE_ABORT,
        INVOKE_RETRY
    };
    interface Root { // locality constrained
        typedef unsigned long Priority;

        readonly attribute Priority prio;

        const Priority LowestPriority = 0; // Priority'First
        const Priority HighestPriority = 0xffffffff;
                                            // Priority'Last;

        void activate(in Priority p);
        void deactivate();
        boolean is_active();
    };
};
```

The **Root** interface defines the interceptor abstraction. All categories of interceptors must derive from this interface to guarantee the general rules presented in previous section. The interface defines operations to activate, deactivate or query the state of the interceptor object. The priority of the interceptor object can be fetched by looking at the **prio** attribute.

The **Status** enum type defines the possible return values of interceptor operations. An enumeration is used to enforce the semantic of possible return values of interceptor operations. The semantic of enumerated values is the following:

- **INVOKE_CONTINUE**
  The invocation continues. If other interceptor objects exist in the chain, their methods are executed.
- **INVOKE_ABORT**
  The invocation is aborted. The call chain of interceptor objects is stopped. No other interceptor method will be called.
- **INVOKE_RETRY**
  The invocation is retried.

The **activate** operation activates the interceptor. When the interceptor is active, its operations are called by the ORB. The priority parameter defines the priority of the interceptor object. If the priority is equal to the constant **LowestPriority**, the interceptor object has the lowest priority at the moment of activation. Opposite, if the **HighestPriority** value is used, it has the highest priority at the moment of activation.

The **deactivate** operation deactivates the interceptor. Interceptor operations are not called by the ORB.

The **is_active** operation returns the state of the interceptor regardless of its activation.

9.2.2.3 Locality Constraints

A **Root** object must be local to the process.

9.2.3 Request Interceptors

9.2.3.1 Architectural Considerations

Request interceptors are invoked by the ORB when an invocation is made. A first request interceptor is involved on the client side and a second request interceptor operates on the server side. These two interceptors need to have enough information about the current request. The following list of actions is the minimal set of possible actions that request interceptors can do:

- Getting and setting the target object reference
  Obtaining the target object reference is important to be able to extract information from it. Changing the object reference is also an important need to be able to route the request to another server transparently. This allows replication, fault tolerance and other such mechanisms.
- Getting and setting the operation name
  Obtaining the operation name is used by monitoring or logging tools.
  Getting, setting and removing service context data
  Passing service context data is an important requirement for interceptors. This is a simple and extensible way for exchanging information transparently between client and servers. Service context data is presented later.
- Observing, clearing and raising exceptions
  Request interceptors have to deal with exceptions raised by the ORB at several levels. Logging and monitoring tools need to observe the exception and replication. Fault tolerant mechanisms need to analyze and hide the exceptions (in other words clear or override it).

The set of actions that a Realtime ORB permits can be larger but must not be smaller than the above list.

9.2.3.2 Specification

The Realtime ORB does not need a complete DII and DSI. In particular, operations to access or set parameters and return values are not required. In order to avoid to have light-weight implementations and also to avoid changing the existing DII/DSI interfaces, the interceptor module defines two interfaces for representing the request on the client and server sides.

```
module Interceptor {
    interface LWRootRequest { // locality constrained
        attribute Object target;
        attribute Identifier operation;
        ...
    };
    interface LWRequest : LWRootRequest { // locality constrained
        readonly attribute CORBA::Request request;
    };
    interface LWServerRequest : LWRootRequest { // locality constrained
        readonly attribute CORBA::Request request;
    };
};
```

The **LWRootRequest** interface represents a generic light-weight request. This interface is only defined for the purpose of simplification for defining the operations which are common to client and server requests. The **LWRequest** interface represents the light-weight request on the client side and the **LWServerRequest** interface represents the light-weight request on the server side.

The **request** attribute gives access to the DII/DSI request object if the Realtime ORB supports DII/DSI for interceptors. When DII/DSI is not available, accessing the attribute will raise the **CORBA::NO_IMPLEMENT** exception.

The **target** attribute gives access to the object reference used by the client to perform the invocation. The attribute can be queried and also set. Setting the attribute will redirect the invocation to the new target object reference. The Realtime ORB may impose restrictions on the places where the setting is made, regardless of the current state of the invocation. For example, setting the target object reference when the invocation is complete is useless and the Realtime ORB will raise the **CORBA::NO_PERMISSION** exception.

The **operation** attribute indicates the name of the operation which is invoked by the client.

9.2.3.3 Locality Constraints

The **LWRootRequest**, **LWRequest** and **LWServerRequest** objects are local to the process.

9.2.4 Service Context Data

9.2.4.1 Architectural Considerations

Client and server interceptors need to exchange data in a safe and transparent manner for realtime applications. For example, a client interceptor can pass the QoS information concerning the request, and the server interceptor can retrieve this QoS. IDL of applications may not be involved in this process, that is the QoS does not need to be specified at the IDL level. However, we do not preclude that this is the only way for QoS to be passed. It can also be passed explicitly.

The service context data represents arbitrary data which can be passed by client and server interceptors in method invocations. It is possible to specify several sets of service context data. Each set may be independent from the other and each may be uniquely identified. Service context data is identified by a service identifier which is represented by a number.

The IIOP protocol supports service context data in the request and reply messages. The service context data is identified by a service identifier followed by the data which appears as a sequence of octets. Figure 8 on page 51 illustrates the format of an IIOP request message. It is composed of a generic message header followed by the IIOP service context. The service context is a sequence of service context structures. Each structure is composed of a tag number and a sequence of octets. The content of the sequence of octets depends on the service. In the example, a priority number is stored in one of the service context data fields.

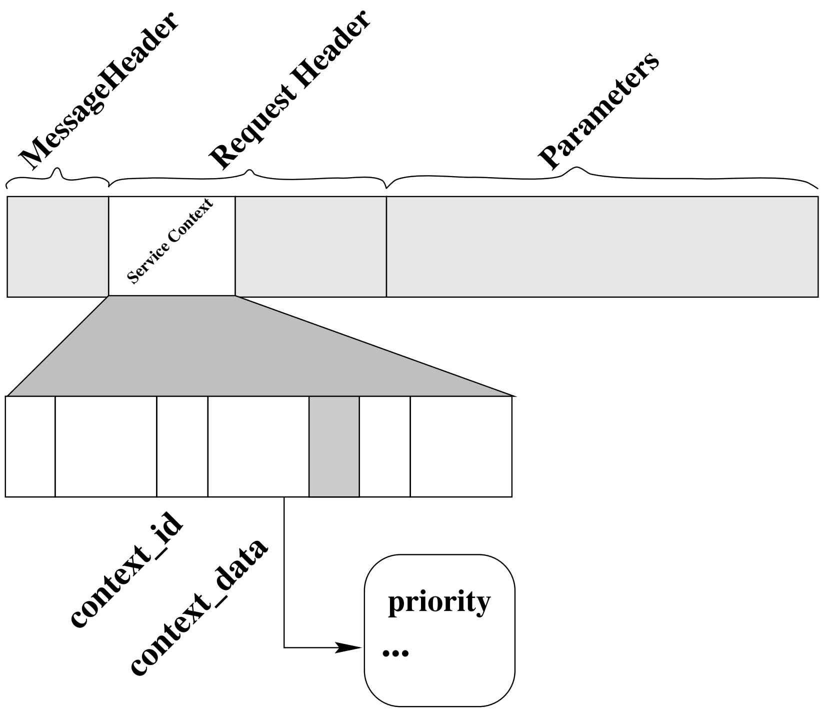

Figure 8 Service Contexts in GIOP Requests

The service context data of IIOP satisfies completely the transparent exchange of data between client and server. The interceptor defines only the operations for manipulating these service context data. Operations are defined to:

- Put in the request or reply message a service context data
- Extract from the request or reply message a service context data
- Remove a service context data and check whether a service context data is present or not.

9.2.4.2 Specifications

```
module Interceptor {
    typedef IOP::ServiceID ServiceID;
    typedef sequence<octet> ContextData;
    interface LWRootRequest { // locality constrained
        ...
        void set_service_context(in ServiceID id,
                                 in long flags, in ContextData d);
        ContextData get_service_context(in ServiceID id,
                                        in long flags);
        void remove_service_context(in ServiceID id);
        boolean has_service_context(in ServiceID id):
        ...
    };
};
```

Operations to create, get or remove service context data are defined on the **Interceptor::LWRootRequest** interface. They are available for both the client and server light-weight request interfaces. The only way to create, get or remove service context data is by using the light-weight request interfaces.

The **set_service_context** operation inserts or changes the service context data identified by a service id. If the flag **EXCLUSIVE** is defined and the service context data is already present in the request object, the **CORBA::NO_PERMISSION** system exception is raised.

The **get_service_context** operation retrieves a service context data given its service identifier. The operation raises the system exception **CORBA::BAD_PARAM** if the service context is not present in the request.

The **remove_service_context** operation removes the service context data given its service identifier.

The **has_service_context** operation checks whether a service context data is present. The operation returns **CORBA::TRUE** if it is and **CORBA::FALSE** otherwise. No system exception is raised.

9.2.5 Request Interceptor Context

9.2.5.1 Architectural Considerations

Client or server interceptors have several operations which are called by the ORB at different steps of a method invocation (this is necessary because the request is in a different state, regardless of its completion). In a given interceptor (client or server), the different methods of the same interceptor may need to exchange information. For example, an interceptor method may perform some actions and may need to save some of the results so that when the ORB invokes another method of the same interceptor, the results can be obtained. Request interceptor context objects are used for this. Within a client or server interceptor, they allow the interceptor to have a common data which is specific to the interceptor and to the request. The common data serves as a placeholder for interceptor methods to exchange information.

To use the mechanism, a client or server interceptor can create an interceptor context object. Such object is specific to the invocation and it can be retrieved only by the interceptor that created it, in any of its methods. The lifetime of the interceptor context object ends when the invocation is finished.

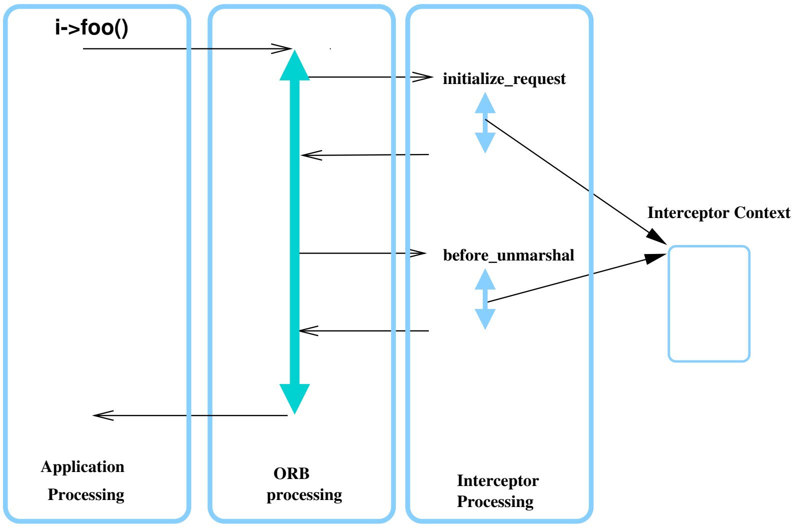

Figure 9 Interceptor Contexts

Figure 9, “Interceptor Contexts,” on page 53 gives an example of two interceptor methods which use an interceptor context object for sharing and storing specific request information. The two methods are defined for the same client interceptor object. In the example, the application invokes an operation on an object reference (i) denoting a remote server. While processing the request on the client-side, the ORB invokes the client interceptors. The **initialize_request** and **before_unmarshal** operations are two client interceptor operations (as described later in §9.2.6 “Client Interceptor” on page 54). These two operations share data concerning the current request. This data is saved in the interceptor context object.

9.2.5.2 Specification

```
module Interceptor {
    interface Context { // locality constrained
        void destroy();
    };
    interface LWRootRequest { // locality constrained
        void set_context(in Root interceptor, in Context ctx);
        Context get_context(in Root interceptor);
    };
};
```

The **Context** interface defines the root of interceptor contexts. Specific interceptor context objects must derive from this interface.

The **destroy** operation is called by the ORB when the current request is finished. This allows application to free the memory correctly.

The **set_context** operation attaches to the current request object and interceptor object a specific interceptor context. The request object has ownership of the interceptor context object. Applications must not delete it.

The **get_context** operation retrieves from the current request the interceptor context object which is associated with a particular interceptor.

9.2.5.3 Locality Constraints

A **Context** object is local to the process.

9.2.6 Client Interceptor

9.2.6.1 Architectural Considerations

The client interceptor has four interceptor methods. Interceptor methods are called at different steps in the invocation process. Each method gets as argument the **Interceptor::LWRequest** object which describes the current request. An implementation may impose some restrictions about what operations or what treatement can be made in a particular interceptor method.

Even though each method gets as argument the request object, it is not always possible to modify the request object in all interceptor methods. There are several places where the modification of the request description cannot be updated (easily) in the message. For example, once the request header is marshaled, it becomes difficult to change the operation name, the object reference, etc. The Realtime ORB architecture defines the places where the request object can be changed.

The client interceptor is called at four levels:

- At initialization of the request.
- Before sending the request to the server
- After reception of the reply or detection of server failure by the ORB

- When the request is finished

9.2.6.2 Specification

```
module Interceptor {
    interface ClientInterceptor : Root {
                          // locality constrained
        Status initialize_request(in LWRequest req,
                                  in CORBA::Environment env);
        Status after_marshal(in LWRequest req,
                             in CORBA::Environment env);
        Status before_unmarshal(in LWRequest req,
                                in CORBA::Environment env);
        Status finish_request(in LWRequest req,
                              in CORBA::Environment env);
    };
};
```

The **ClientInterceptor** interface represents the client interceptor abstraction. All its operations receive the request object which describes the client request and an environment object which contains the exception (if one exception is raised). The **initialize_request** and **after_marshal** operations follow the **Forward** call order and the before_unmarshal and finish_request operations follow the **Backward** call order.

The **initialize_request** operation is called after the request object reference is initialized and before creating the request header message in the interaction protocol. The operation is invoked by the client thread which performs the invocation. The request object provides the following minimal set of information:

- the target object reference onto which the invocation is made.
- the operation name which is invoked.
- the service contexts which are created by client interceptors.

In this operation, the interceptor is allowed at least to do the following:

- change the target object reference. Once changed, the invocation will be routed to the new target object reference installed by the interceptor method.
- change the operation name.
- insert or remove service contexts from the request object reference.
- raise an exception to abort the invocation.

The **after_marshal** operation is called after the interaction protocol has built the message and before the transport protocol is invoked. The operation is invoked by the client thread which performs the invocation. A realtime ORB may forbid any modification of the request, including change of target object reference, change of operation name and change of service contexts. The interceptor method can raise an exception to abort the invocation.

The **before_unmarshal** operation is called after the reply is received or if a communication failure is detected. For a synchronous operation, the operation is invoked by the client thread which performs the invocation. For an asynchronous invocation, the operation is invoked by a thread of the transport end-point thread pool.

The **finish_request** operation is called when the invocation is finished successfully or not.

9.2.6.3 Locality Constraints

A **ClientInterceptor** object must be local to the process.

9.2.7 Server Interceptor

9.2.7.1 Architectural Considerations

The server interceptor has four operations which are called at four levels:

- When a request is received
- When parameters are extracted and before calling the servant
- When the servant returns and before preparing the reply
- When the request is finished

9.2.7.2 Specification

```
module Interceptor {
    interface ServerInterceptor : Root {
        // locality constrained
        Status initialize_request(in LWServerRequest req,
                                  in CORBA::Environment env);
        Status after_unmarshal(in LWServerRequest req,
                                  in CORBA::Environment env);
        Status before_marshal(in LWServerRequest req,
                              in CORBA::Environment env);
        Status finish_request(in LWServerRequest req,
                              in CORBA;;Environment env);
    };
};
```

The **ServerInterceptor** interface represents the server interceptor abstraction. All its operations receive the request object which describes the request received by the ORB in the **req** parameter. The environment object passed in **env** contains the exception which is eventually raised by either the ORB or the servant. The **initialize_request** and **after_unmarshal** operations follow the **Forward** call order and the before_unmarshal and finish_request operations follow the **Backward** call order.

The **initialize_request** operation is called when a remote invocation is received by the server. The operation is called by a thread of the transport end-point thread pool. The request object provides the following minimal set of information:

- the target object reference onto which the invocation is made
- the operation name which is invoked

The **after_unmarshal** operation is called after the operation parameters have been unmarshaled from the request buffer. The operation is called by the thread that will execute the servant code.

The **before_marshal** operation is called after the servant method has been called or if an exception is raised and returned to the client. This is the only method which can be used to insert service context data in the reply. The operation is called by the thread that executes the servant code.

The **finish_request** operation is called after the invocation is finished; that is, after the reply was sent to the client.

9.2.7.3 Locality Constraints

A **ServerInterceptor** object must be local to the process.

9.2.8 POA Interceptor

The POA Interceptor is called by the POA or Realtime POA when object references are created. They are used to insert information in the object reference in a transparent manner for applications.

Note – The submitters are working on the subject and will propose an interface in the second revision. The next proposed submission will take into account the possible revisions and adoptions of interceptors by OMG.

9.2.9 Transport Interceptor

Note – The submitters are working on the subject and will propose an interface in the second revision. The next proposed submission will take into account the possible revisions and adoptions of interceptors by OMG.

9.2.10 Thread Interceptor

The thread pool currently defined will never fit to every application needs. A thread interceptor can inform the application when a thread starts to do some dispatching. It can also inform when the thread has nothing to do. This can increase the flexibility of the thread pool by being able to monitor the thread activities, create new threads, do some garbage collection of threads.

Note – The submitters are working on the interface and will provide a complete description in the second submission. The next proposed submission will take into account the possible revisions and adoptions of interceptors by OMG.

9.2.10.1 Locality Constraints

A **ThreadInterceptor** object must be local to the process.

9.2.11 Initialization Interceptor

9.2.11.1 Architectural Considerations

The introduction of specific realtime policies brings a non trivial challenge for their initialization. First, it is not possible to use the ORB before it is initialized. Specific realtime policies are therefore not allowed to instantiate their specific factories before ORB initialization. Once the ORB is initialized, it is often too late: the ORB has already settled some default behavior.

The role of the initialization interceptor is to simplify the integration of these specific realtime policies by letting them be aware of when the ORB is initialized. Initialization interceptors are activated when the ORB initialization operation **CORBA::ORB_init** is called.

9.2.11.2 Specification

```
module Interceptor {
    interface InitInterceptor : Root { // locality constrained
        Status initialize(in CORBA::ORB orb,
                          in CORBA::ORB id,
                          inout CORBA::arg_list);
    };
};
```

The **InitInterceptor** interface defines the initialization interceptor. This interceptor is only invoked in the initialization phase of the ORB, on the client as well as on the server side. Basically, the interceptor is triggered by invocations on the **CORBA::ORB_init** operation.

The **initialize** operation is called by the ORB as soon as the ORB has been initialized. The ORB object reference is passed in the **orb** parameter and the parameters that were used to initialize the orb are passed in the **id** and **arg_list** parameters. The **initialize** operation follows the **Forward** call order.

9.2.11.3 Locality Constraints

An **InitInterceptor** object must be local to the process.

9.2.12 Message Interceptors

Note – The submitters are working on the subject and will eventually propose an interface in the second revision. The next proposed submission will take into account the possible revisions and adoptions of interceptors by OMG.

---

10 Synchronization Facilities

10.1 Synchronization Objects

Synchronization objects are an important tool in multi-threaded applications. To increase the portability of applications the Realtime ORB defines an API for the following objects:

- **Mutexes**
  These objects are used to protect critical section of codes.
- **Semaphores**
  A semaphore is a non negative integer count and is generally used to coordinate access to resources.
- **Multiple readers/Single writer**
  Many threads can have simultaneous read-only access to data, while only one thread can have write access at any given time. Multiple read access with single write access is controlled by locks, which are generally used to protect data that is frequently searched.
- **Condition Variables**
  A condition variable enables threads to atomically block and test the condition under the protection of a mutual exclusion lock (mutex) until the condition is satisfied.

10.2 Mutexes

10.2.1 Architectural Considerations

Mutexes provide standard operations for enforcing mutual exclusion.

10.2.2 Specification

```
module RT {
    interface Mutex { // locality constrained
        void acquire();
        void release();
        boolean try_acquire();
        void destroy();
    };
};
```

The **Mutex** interface defines the operation for controlling and using a mutex. Policies for queuing and awaking threads are not specified. This allows the use of several kinds of mutexes with different policies. How a mutex object is created is explained in §10.6 "Synchronization Object Factory" on page 64.

The **acquire** operation acquires the target mutex. If the mutex is free, it becomes locked and the calling thread continues execution. Otherwise, the calling thread is blocked until the mutex is released.

The **release** operation releases the target mutex. If threads are blocked on the mutex, one of them is awakened.

The **try_acquire** operation attempts to acquire the mutex. It has the same effect as the **acquire** operation and returns **true** if the mutex is free. Otherwise, the calling thread is not blocked and **false** is returned.

The **destroy** operation destroys the mutex object.

10.2.3 Locality Constraints

A **Mutex** object is local to the process.

10.3 Semaphores

10.3.1 Architectural Considerations

Semaphores provide standard operations for controlling a semaphore counter.

10.3.2 Specification

```
module RT {
    interface Semaphore { // locality constrained
        void acquire();
        boolean try_acquire();
        void acquire();
        void destroy();
    };
};
```

The **Semaphore** interface defines the operation for controlling and using a semaphore. Policies for queuing and awaking threads are not specified. This allows the use of several kinds of semaphores with different policies. How a semaphore object is created is explained in §10.6 "Synchronization Object Factory" on page 64. The initial semaphore counter is specified at creation time.

The **acquire** operation decrements the semaphore counter. If the counter reaches a strictly negative value, the calling thread is blocked.

The **try_acquire** operation is identical to acquire except that it does not block. The operation returns **true** if the semaphore counter was decremented successfully and **false** if the counter is negative.

The **release** operation increments the semaphore counter. If the new value is negative or zero, a thread that has been blocked behind the semaphore is awakened.

The **destroy** operation destroys the semaphore object.

10.3.3 Locality Constraints

A **Semaphore** object is local to the process.

10.4 Multiple Readers Single Writer Lock

10.4.1 Architectural Considerations

Multiple readers and single writer lock serialize access to resources whose contents are searched more than they are changed.

10.4.2 Specification

```
module RT {
    interface RWLock { // locality constrained
        void acquire_write();
        boolean try_acquire_write();
        void acquire_read();
        boolean try_acquire_read();
        void release();
        void destroy();
    };
};
```

The **RWLock** interface defines the operation for controlling and using a multiple readers and single writer lock. Policies for queuing and awaking threads are not specified. This allows the use of several kinds of such locks with different policies. How a multiple readers and single writer lock object is created is explained in §10.6 "Synchronization Object Factory" on page 64.

The **acquire_write** operation acquires the lock if it is not locked for writing and there are no readers. If the lock is free, it becomes locked and the calling thread continues execution. Otherwise, the calling thread is blocked until there are no readers and no writers.

The **try_acquire_write** operation is identical to **acquire_write** except that it does not block if the lock cannot be acquired. It returns **true** if the lock for writting is acquired and **false** otherwise.

The **acquire_read** operation acquires the lock if it is not locked for writing. Otherwise, the calling thread is blocked until there are no writers.

The **try_acquire_read** operation is identical to **acquire_read** except that it does not block if the lock cannot be acquired. It returns **true** if the lock for reading is acquired and **false** otherwise.

The **release** operation releases the lock which was held by the thread. If threads are blocked on the lock, either for writing or for reading, one of them is awakened.

The **destroy** operation destroys the lock object.

10.4.3 Locality Constraints

A **RWLock** object is local to the process.

10.5 Condition Variable

10.5.1 Architectural Considerations

Condition variables provide standard operations for enforcing mutual exclusion, where threads sleep instead of spinning when contention occurs.

10.5.2 Specification

```
module RT {
    interface ConditionVariable { // locality constrained
        boolean wait(in Mutex m, in long delay);
        void signal();
        void broadcast();
        void destroy();
    };
};
```

The **ConditionVariable** interface defines the operation for controlling and using a condition variable. Policies for queuing and awaking threads are not specified. This allows the use of several kinds of mutexes with different policies. How a condition object is created is explained in §10.6 “Synchronization Object Factory” on page 64.

The **wait** operation atomically unlocks the mutex and blocks the calling thread on the target condition variable (for a maximum time of delay) until another thread calls **signal** or **broadcast**. The operation returns the value **true** if the condition variable was signal’ed and **false** if the maximum blocking time has expired.

The **signal** operation unblocks a thread waiting on the target condition variable, which thus becomes eligible to execute.

The **broadcast** operation unblocks all the threads waiting on the target condition variable, which thus become eligible to execute.

The **destroy** operation destroys the condition object.

10.5.3 Locality Constraints

A **ConditionVariable** object is local to the process.

10.6 Synchronization Object Factory

10.6.1 Architectural Considerations

A factory for the creation of synchronization objects is defined. The factory is used to create any kind of synchronization object: mutex, semaphore, multiple readers/single writer lock and condition variable. The factory defines the policies for queuing threads as well as for awaking threads.

The synchronization object factory defines policies for managing its locks. These policies must be inline and coherent with other realtime policies that are defined for threads, request queues and transport end-points. This is why the realtime strategy interface is the interface which returns the synchronization object factory (see §8.6 “Strategy Factory” on page 40).

10.6.2 Specification

```
module RT {
    interface SynchronizationFactory { // locality constrained
        Mutex create_mutex();
        Semaphore create_semaphore(in long count);
        RWLock create_RWLock();
        ConditionVariable create_condition();
    };
    interface StrategyFactory { // locality constrained
        ...
        SynchronizationFactory get_synchronization_factory();
    };
};
```

The **SynchronizationFactory** interface defines the operations for creating synchronization objects.

The **create_mutex** operation creates a new mutex object.

The **create_semaphore** operation creates a new semaphore object with a counter value set to **count**.

The **create_RWLock** operation creates a new multiple readers/single writer lock object.

The **create_condition** operation creates a new condition object.

The **SynchronizationFactory** object is obtained from the strategy factory presented in §8.6 “Strategy Factory” on page 40. This allows the strategy factory to return a synchronization factory which implements the policies which correspond to the realtime application needs.

The **get_synchronization_factory** operation returns the factory object for creating synchronization objects.

10.6.3 Locality Constraints

A **SynchronizationFactory** object is local to the process.

11 Fixed Priority Scheduling Service

Note – This section needs to be extensively reworked in the revised submission. While specification is headed in the right direction, examples shown in this section appeared to be not convincing enough to be usable.

This Section describes a new Scheduling Common Object Service to facilitate plugging in various fixed-priority realtime scheduling policies across the realtime CORBA system. The Scheduling Service uses the capabilities of the realtime ORB to facilitate enforcement of a uniform scheduling policy. Various implementations of the Scheduling Service may embody various scheduling policies. For instance, a Scheduling Service from one vendor may provide a policy that allows hard realtime analysis, such as global rate-monotonic scheduling with distributed priority ceiling resource management across the Realtime CORBA system.

The Scheduling Service is independent of the underlying ORB. That is, it is possible to implement the Scheduling Service without access to the internals of the ORB. However, a Scheduling Service built on the realtime ORB specified in the previous sections makes for better enforcement of end-to-end predictability.

Note that the Scheduling Service is not orthogonal to other capabilities in the realtime ORB. For instance, the realtime ORB allows applications to establish thread priorities and the Scheduling Service also sets thread priorities. The Scheduling Service is designed so that application programmers do not have to use the ORB capabilities directly. Instead, they can use the Scheduling Service to achieve a uniform scheduling policy across the CORBA system. Application programmers can choose to use a Scheduling Service or to set scheduling parameters using ORB primitives directly.

11.1 Global Priority Notion

The Scheduling Service is based on the notion of a global, uniform priority assignment to threads. Global priority is a total ordering of those threads in the system that are assigned a priority by the application programmer. Note that some threads may not be explicitly assigned a priority in application code (e.g. servant threads that inherit the priority of the client), but that eventually every thread will have a global priority. The programmer uses the Scheduling Service to assign each client thread one or more fixed, unique global priorities from 1 to N, with 1 being highest priority and N being lowest priority. A client may have more than one priority when parts of its execution have tighter timing constraints or higher importance than others. We anticipate that in most systems this priority assignment will be done a priori, perhaps with use of an off-line scheduling tool.

Note that global priority is a conceptual ordering of threads; global priority is not necessarily the operating system priority, priority within the realtime ORB, nor other implementation-level priority at which the thread will actually execute. The Scheduling Service will map these global priorities to local priorities in the underlying system.

Also note that priority is not "importance". Importance is an application specification of the relative value of the thread or request/reply to the system. Importance is often used to determine the global priority ordering, but is not necessarily related.

Fixed priority scheduling entails, whenever possible, resolving scheduling conflicts by allowing the highest global priority thread to use a resource on which the conflict occurs.

When, for some reason such as consistency of a shared resource, the RT CORBA system does not resolve conflicts in priority order and causes a higher priority thread to wait for a lower priority thread, "priority inversion" is said to occur. Analyzable realtime systems require that priority inversion be bounded. This standard will provide interfaces that allow implementations that bound priority inversion.

11.2 Portability and Fixed Priority Scheduling

Portability is a fundamental requirement in the OMG. What does portability mean with respect to realtime? In the Scheduling Service model, the notion of global priority is meant to be portable. That is, threads ordered by the Scheduling Service on one CORBA system should have the same ordering (and the same ordinal global priority value) resulting from a Scheduling Service on another CORBA system. For instance, a thread with global priority 10 should be the 10th highest priority thread (there are threads with priority 1-9 that have higher priority) on one CORBA system and remain that way if the application is ported to another CORBA system.

However, the actual resulting schedule is not necessarily portable. That is, the order in which threads are actually executed on one CORBA system will not necessarily be the same on another CORBA system. Although all CORBA systems should seek to maintain the global priority ordering, most will allow some priority inversion (hopefully with good reason such as preserving the consistency of a shared resource). The amount and instances of priority inversion will differ among CORBA systems and thus will yield different schedules. This is important to note so that portable applications do not depend on scheduling decisions made by the CORBA system.

This definition of portability which does not mandate exact schedules be portable also allows for different Scheduling Service vendors to provide different scheduling mechanisms that meet different criteria.

11.3 Scheduling Service

The basic notion of the Scheduling Service is that threads have a SchedEntity interface associated with them to manipulate their priority and priority ceiling. The SchedEntity instances also provide the ability to pass the scheduling information to other parts of the CORBA system.

Here is the IDL For the Scheduling Service:

```
module CosScheduling {
    exception InvalidPriority {};
    exception InvalidPC {};
    exception PriorityNotSet {};
    exception ExistingSchedEntity {};

    typedef long Priority;

    const Priority NoScheduling = 0;
    const Priority NoCeiling = 0;

    interface ClientSchedEntity : RT::Scheduler {
        void  set_priority(in Priority global_priority)
            raises(InvalidPriority);

        Priority get_priority()
            raises (PriorityNotSet);
    };

    interface ServantSchedEntity : RT::Scheduler {
        void  set_priority(in Priority global_priority)
            raises(InvalidPriority);

        Priority get_priority()
            raises(PriorityNotSet);

        void set_priority_ceiling(in Priority priority_ceiling)
            raises(InvaildPC);

        Priority get_priority_ceiling();
            raises(PriorityNotSet);
    };

     interface SchedEntityFactory {
        ClientSchedEntity create_client_sched_entity(
            in Priority priority);
            raises(InvalidPriority, ExistingSchedEntity);

        ServantSchedEntity create_servant_sched_entity(
            in Priority priority,
            in Priority priority_ceiling);
            raises(InvalidPriority, ExistingSchedEntity);
    };
};
```

11.3.1 set_priority and get_priority

In both the **ClientSchedEntity** and **ServantSchedEntity** interfaces, the **set_priority** method changes the global priority value of the schedulable entity to the parameter global_priority, which must be a unique global priority from 1 to N. The scheduling service uses this value as the basis to perform implementation-specific setting of local priorities, if any, on the local ORB, operating system, and network. For instance, on realtime operating systems with a limited number of local priorities (e.g. 256 local priorities), the Scheduling Service **set_priority** method may need to map a large priority range to those local limited priorities. The details of this mapping are proprietary to each implementation of a Scheduling Service.

A global_priority parameter value of **CosScheduling::NoScheduling** indicates that priority-based scheduling is not used and that the default scheduling policy of the ORB, operating systems, and network is used.

An implementation raises **InvalidPriority** if global_priority's value is negative. An implementation may raise **InvalidPriority** if the specified global priority is over the maximum priority allowed by the CORBA system.

The method **get_priority** returns the global priority to which the schedulable entity has been set. If the global priority has not been set, then the method raises **PriorityNotSet**.

11.3.2 *set_priority_ceiling* and *get_priority_ceiling*

In the **ServantSchedEntity** interface, the **set_priority_ceiling** method causes the servant to use a priority inheritance protocol to bound priority inversion and to establish the priority of the servant while it executes on behalf of a client. The parameter priority_ceiling is the priority ceiling under which the thread is to execute. A value of **CosScheduling::NoCeiling** indicates that the servant will use the Basic Priority Inheritance protocol and not a Priority Ceiling protocol.

Note that for proper realtime scheduling, applications that set a priority ceiling for a schedulable entity must reset the ceiling using the **set_priority_ceiling(CosScheduling::NoCeiling)** when the entity terminates. The existence of a schedulable entity with a priority ceiling causes the Scheduling Service to assume that the schedulable entity is active. If the entity has terminated, this invalid assumption could cause other execution to be unnecessarily blocked. See the example in §11.4 “Example and Intended Use of The Scheduling Service” on page 69 for recommended usage of the Scheduling Service and in particular of resetting priority ceilings.

Exactly which priority inheritance protocol is used will depend on the implementation of the Scheduling Service. Many different priority inheritance protocols can be used with this interface. We assume that a Scheduling Service implementation will provide one of them. Recall that re-creating schedules is not a goal of realtime CORBA portability - this allows the flexibility of having different Scheduling Services with different priority inheritance approaches (including no priority inheritance).

An implementation raises **InvalidPC** if priority_ceiling's value is negative. An implementation may raise **InvalidPC** if the specified priority_ceiling is over the maximum priority allowed by the CORBA system.

The method **get_priority_ceiling** returns the global priority to which the schedulable entity has been set. If the global priority has not been set, then the method raises **PriorityNotSet**.

11.3.3 Factory

The **SchedEntityFactory** interface creates client and servant schedulable entity instances. A SchedEntity should be created only once for a client, once for the servant main program, and once for each servant method.

The client schedulable entity is initialized to the parameter **priority**. Note that after such an initialization, the **set_priority** method does not need to be called unless the client wants to eventually change the priority. Clients with a single priority throughout their execution will likely create a **ClientSchedEntity** with the factory and not have to invoke any of the **ClientSchedEntity** methods.

The servant schedulable entity is initialized to the parameter **priority** and to the parameter **priority_ceiling**. Note that the **set_priority** and **set_priority_ceiling** methods do not need to be called unless the servant wants to eventually change either the priority or ceiling. For most applications based on priority ceiling techniques, such a change is not recommended (the priority and the ceiling should be set once at the beginning of the servant execution) until the very end of execution. At the end of execution these servants will want to set the priority ceiling to **CosScheduling::NoCeiling** to indicate to the Scheduling Service that they are no longer executing.

An implementation raises **InvalidPriority** if specified priority value is negative. An implementation may raise **InvalidPriority** if the specified priority value is over the maximum priority allowed by the CORBA system.

An implementation raises **InvalidPC** if the specified priority_ceiling's value is negative. An implementation may raise **InvalidPC** if the specified priority_ceiling value is over the maximum priority allowed by the CORBA system.

An implementation raises **ExistingSchedEntity** if a SchedEntity entity already exists for the client/servant method.

11.4 Example and Intended Use of The Scheduling Service

Here is the way that we expect clients and servant methods to look.

Client

```
// C++
{
   CosScheduling::SchedEntityFactory_var sched_factory =
         ...;
   CosScheduling::ClientSchedEntity_var client_sched;

   client_sched =
      sched_factory->create_client_sched_entity(10);

   servant = bind(...);

   // some client code
   servant->method(params, 10);
   // rest of client code
}
```

```
Servant Main

// C++
{
   CosScheduling::SchedEntityFactory_var sched_factory;
   CosScheduling::ServantSchedEntity_var servant_sched;

   servant_sched =
      sched_factory->create_servant_sched_entity( 14,
                     CosScheduling::NoCeiling);
   main request loop:
   // accept request
   // create/attach a thread for request;
   end request loop
}
```

```
Servant Method (params, client_prio) //params to method
// C++
// note that client priority is passed explicitly as parame-
ter client_prio
{
   CosScheduling::ServantSchedEntity method_sched;

   method_sched =
      sched_factory->create_servant_sched_entity(
                     client_prio, 12);
 // method code

   method_sched->set_priority_ceiling(
                     CosScheduling::NoCeiling);
}
```

This example presumes that first, offline, a realtime scheduling analysis is done to establish a fixed priority ordering of client threads, and portions of client threads, in the entire CORBA system.

The sample client creates the schedulable entity instance and sets its global priority to 10. The Scheduling Service will map this to one of the realtime operating system's local priorities, and set a priority in the realtime ORB. The realtime ORB would then pass the request to the servant (enforcing priority as it goes).

The servant main program has been set to execute at the highest priority of any client that will access it (its priority ceiling, in this example is 14). We are assuming a high performance realtime POA so that launching/attaching to a thread to perform the method is a very fast operation as far as the servant's main request loop is concerned. This assumption justifies keeping the servant's main loop at the high priority (its priority ceiling) and not having it use a priority ceiling protocol. Strictly speaking, this is not a perfect fixed-priority technique, but the resulting priority inversion can be accounted for in adding the execution of the servant main loop and POA execution to the execution time of each method thread when the thread's execution is analyzed.

Once in the method code, the method creates a schedulable entity instance using the priority of the client, which is explicitly passed as a parameter to the servant method. It also uses the servant method's priority ceiling (12). Note that in some priority ceiling protocols, the priority ceiling of a method may be less than the priority ceiling of its servant's main loop (in our example, 12 < 14). The Scheduling Service will then perform a priority ceiling check and the factory call does not return until the priority ceiling check is satisfied (essentially making the servant method invocation wait). When the factory call does return, the servant method invocation is executing at the client's priority; possibly at a higher priority if the scheduling service determines that priority inheritance needs to occur. After executing the servant method code and completing, the servant method calls **set_priority_ceiling** with **CosScheduling::NoCeiling** to indicate that that the servant method is no longer active.

Note that use of a priority inheritance protocol is not necessary. In applications where the servant is to run at some fixed priority, the **set_priority** method can be used in the servants to set this priority without priority inheritance taking place.

12 Pluggable Protocols

Note – This section needs to be extensively reworked in the revised submission in order to introduce more flexibility. A higher level API for protocol plug-ins is being studied. The submitters will provide a complete interface for protocol plug-ins in the second revision.

12.1 Motivation

To deploy CORBA technology as a central building block of a Distributed Processing Environment in specific domains, the General Inter-ORB Protocol (GIOP) and the Internet Inter-ORB Protocol (IIOP) are in many cases not sufficient. For example, in the telecommunications domain the use of GIOP/IIOP instead of existing telecommunication protocols would result in a loss of efficiency and reliability. Such a use would pay no attention to proven networking technologies, like the reliable and efficient Signaling System No. 7 (SS7).

At present it is difficult and sometimes impossible to add new protocols to existing ORBs. In many cases a protocol implementor does not have access to the ORB's internal interfaces needed to add such new protocols. Moreover, even if an ORB implements interfaces to add new protocols, these interfaces are proprietary, meaning that a protocol plug-in can only be used with the particular ORB it was written for.

Defining common interfaces for such protocol plug-ins solves these problems. With common interfaces a protocol plug-in can be written once and then be reused with every ORB implementing these interfaces. This protects the often enormous investments for developing domain-specific protocol plug-ins. This approach also makes it possible to write protocol plug-ins for non-public, proprietary protocols, for example in domains where protocol specifications cannot be made public for security reasons (e.g. banking or military protocols).

12.2 The Open Communications Interface

The Open Communications Interface (OCI) defines common interfaces for pluggable protocols. It consists of two major parts: The Message Transfer Interface and the Remote Operation Interface.

The Message Transfer Interface supports connection-oriented, reliable protocols. TCP/IP makes one possible candidate for a Message Transfer Interface plug-in. If the underlying ORB uses GIOP, such a plug-in can implement the IIOP protocol. Other candidates are SCCP (Signaling Connection Control Part, part of SS7) or SAAL (Signaling ATM Adaptation Layer). Non-reliable or non-connection-oriented protocols can also be used if the protocol plug-in itself takes care of reliability and connection management. For example, UDP/IP can be used if the protocol plug-in provides for packet ordering and packet repetition in case of a packet loss.

The Remote Operation Interface supports protocols that directly implement the concept of remote procedure calls. Candidates for a Remote Operation Interface plug-in are DCE-RPC and TCAP (Transaction Capabilities Application Part, part of SS7). GIOP also falls into this category since GIOP basically implements remote procedure calls.

These two different levels of abstraction for protocol plug-ins allow the OCI to be used for a wide range of protocols. On the one hand it's easy to implement plug-ins for connection-oriented, reliabe protocols. These plug-ins can be light-weight since the ORB handles most of the protocol logic. On the other hand it is possible to make use of existing remote procedure call mechanisms in domains where this is desirable.

It is also possible to combine a plug-in for the Remote Operation Interface with a plug-in for the Message Transfer Interface, provided that the Remote Operation Interface plug-in supports the Message Transfer Interface. For example, a GIOP Remote Operation Interface plug-in supporting Message Transfer Interface plug-ins can be combined with a TCP/IP Message Transfer Interface plug-in. Together these two plug-ins implement the IIOP protocol.

12.3 Compliance Points

Pluggable Protocols are a separate compliance point. This means that an ORB can be compliant to the OCI without being compliant to the Realtime specification and vice versa. OCI compliance is defined as follows:

- An ORB implementing only the OCI Message Transfer Interface is “OCI Message Transfer Interface compliant”.
- An ORB implementing only the OCI Remote Operation Interface is “OCI Remote Operation Interface compliant”.
- An ORB implementing both the OCI Remote Operation Interface and the OCI Message Transfer Interface is “OCI compliant”.

Unless noted otherwise, no part of the Remote Operation Interface specification and Message Transfer Interface specification is optional.

12.4 The Message Transfer Interface

12.4.1 General

12.4.1.1 Design Patterns

The Message Transfer Interface is based on the commonly used design patterns Connector, Acceptor ([5]) and Reactor ([6]). These patterns already provide the basis for most ORB implementations, so in many cases this allows ORB vendors to switch to the Message Transfer Interface by simply “wrapping” existing code.

12.4.1.2 Exceptions

The Message Transfer Interface does not define any new exceptions. This would result in a severe performance penalty since such exceptions would have to be caught by the ORB and translated into standard CORBA system exceptions.

Instead, system exceptions (i.e. exceptions derived from SystemException) are used. That is, all operations are allowed to throw system exceptions but no other exceptions. A separate set of minor exceptions codes has to be defined for each protocol plug-in.

12.4.1.3 Thread Safety

Plug-ins for the Message Transfer Interface do not have to be thread safe. It’s the ORB‘s responsibility to ensure a serialized access to the interfaces. This allows using plug-ins for single-threaded and multi-threaded ORBs without performance loss.

12.4.1.4 Single-Threaded ORBs

The Message Transfer Interface supports single-threaded ORBs through the Reactor and the callback interfaces. This is an optional part, however, since using a Reactor and callback-based communication is not possible or inefficient in some environments. For example, in Java it's difficult to write Reactor classes since Java lacks system calls that directly support waiting for multiple events within a single thread.

12.4.1.5 Object References

For the representation of Object References the Message Transfer Interface uses Componentized object references described in §9.1 "Componentized Object References" on page 42. Using the Componentized object reference API, specific protofols can extract or insert their own profiles in the object reference.

12.4.1.6 Locality Constraints

All objects used by the Message Transfer Interface must be local to the process.

12.4.2 Interface Summary

12.4.2.1 Buffer

The Buffer interface can be viewed as an interface to an unbounded octet sequence with operations to set and get the sequence's length. However, the internal representation is implementation dependent and does not necessarily have to use unbounded sequences.

Besides the length attribute a sequence provides a position counter, which determines how many octets already have been read or sent.

12.4.2.2 Transport

The Transport interface is used to send and receive octet streams in the form of Buffer objects. There are blocking and non-blocking send/receive operations available, as well as operations that handle timeouts and detection of connection loss.

A Transport interface provides optional hooks for send and receive callbacks, which can be used by single-threaded ORBs for non-blocking input/output.

12.4.2.3 Acceptor and Connector

Acceptors and Connectors work as Factories for Transport objects. A Connector is used to connect clients to servers and an Acceptor is used by the server to accept incoming client connection requests.

Acceptors and Connectors also provide for operations to manage protocol-specific object reference profiles, like comparing profiles, adding profiles to an object reference or extracting object keys from profiles.

Acceptors provide optional hooks for accept callbacks, which can be used by single-threaded ORBs for non-blocking acceptance of connection requests.

12.4.2.4 Connector Factory

A Connector Factory is used by clients to create Connectors. No special Acceptor Factory is necessary, since an Acceptor is created just once at server startup then accepts incoming connection requests until it is destroyed on server shutdown. Connectors, however, need to be created by clients whenever a new object reference is received and a new connection for this object reference has to be established.

12.4.2.5 The Registries

The ORB provides a Connector Factory Registry and the Object Adapter an Acceptor Registry. These registries make it possible to plug in new protocols. Transport, Connector, Connector Factory and Acceptor must be written by the protocol implementors. The Connector Factory must then be registered with the ORB's Connector Factory Registry and the Acceptor must be registered with the Object Adapter's Acceptor Registry.

12.4.2.6 Reactor

Note – The Reactor interfaces are not defined yet. There are three ways how this can be done:

1. Define a complete set of Reactor interfaces and data types in IDL, including an event handler interface, operations for adding and removing event handlers, a system-specific "handle" type and operations for dispatching events. An instance of the Reactor is then provided by the ORB.

2. Don't define Reactor interfaces but add an operation to the Transport, Connector and Acceptor interfaces returning a system-specific "handle". For example, for Unix systems this handle can be a file descriptor.

3. Define only a minimal Reactor interface, which has operations for dispatching events only. An instance of the Reactor is provided by the protocol plug-in and not by the ORB. The protocol plug-in extends this Reactor interface internally by adding operations for event handler management.

Approach 1 and 2 have the drawback that all protocol plug-ins are required to use the same "handle" type, which is provided by the ORB's Reactor. Approach 3 doesn't require such a "handle" type to be defined, but it is not possible to plug in two different protocols using different Reactors.

12.4.3 Class Diagram

Figure 10, "Interface Diagram," on page 76 shows the classes and interfaces of the Message Transfer Interface. The ORB must provide abstract base classes for the interfaces Connector Factory, Connector, Transport and Acceptor. The protocol plug-in must inherit from these classes in order to provide concrete implementations for a specific protocol. The ORB must also provide concrete classes for the interfaces Buffer, Reactor, Connector Factory Registry and Acceptor Registry. An instance of the Connector Factory Registry and the Acceptor Registry is provided by the ORB and OA, respectively. Concrete implementations of the Connector Factory must be registered with the ORB's Connector Factory Registry and concrete implementations of the Acceptor must be registered with the Acceptor Registry.

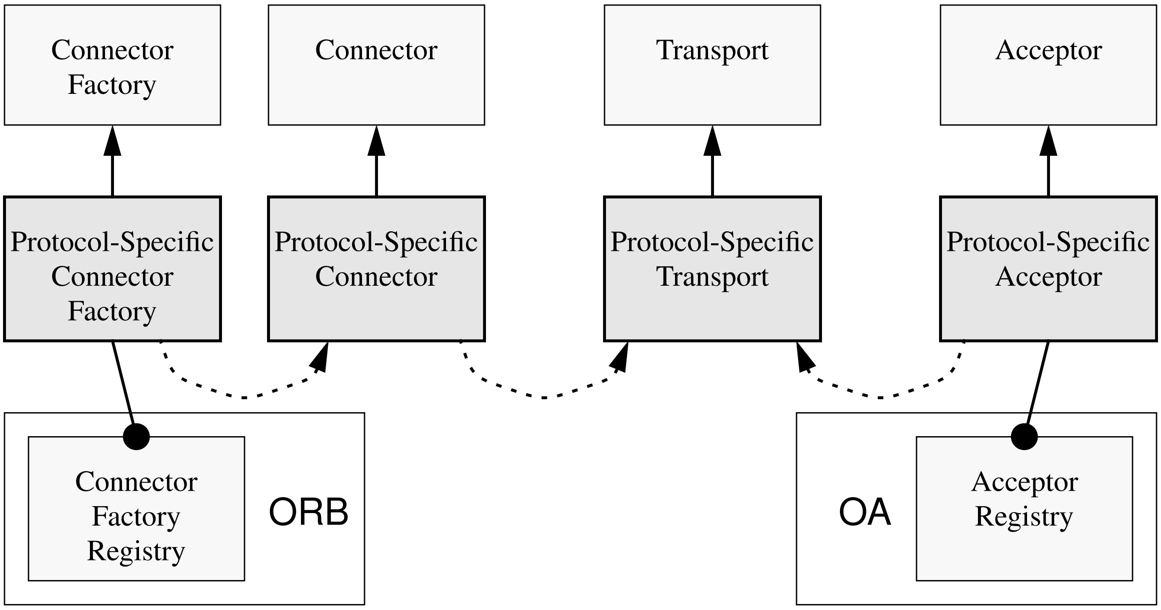

Figure 10 Interface Diagram

12.4.4 Specification

```
#pragma prefix "omg.org"

module OCI
{
    typedef sequence<octet> ObjectKey;

    interface Buffer { // locality constrainted
        attribute unsigned long length;
        attribute unsigned long pos;
        void advance(in unsigned long delta);
        unsigned long rest_length();
        boolean is_full();
    };

    interface ReceiveCB { // locality constrainted
        void receive_cb();
    };

    interface SendCB { // locality constrained
        void send_cb();
    };

    interface Transport { // locality constrained
        readonly attribute ProfileId tag;

        void receive(in Buffer buf, in boolean block);
        boolean receive_detect(in Buffer buf, in boolean block);
        void receive_timeout(in Buffer buf, in unsigned long timeout);

        void send(in Buffer buf, in boolean block);
        boolean send_detect(in Buffer buf, in boolean block);
        void send_timeout(in Buffer buf, in unsigned long timeout);

        void set_receive_cb(in ReceiveCB cb);
        void set_send_cb(in SendCB cb);
    };

    interface Connector { // locality constrained
        readonly attribute ProfileId tag;

        Transport connect();

        boolean compare(in Object object);
        ObjectKey extract_key(in Object object);
    };

    interface AcceptCB { // locality constrained
        void accept_cb();
    };

    interface Acceptor { // locality constrained
```

```
        readonly attribute ProfileId tag;

        Transport accept();

        void set_accept_cb(in AcceptCB cb);

        void add_profile(in ObjectKey key, inout Object object);
        boolean compare(in Object obj1, in Object obj2);
    };

    interface ConFactory { // locality constrained
        readonly attribute ProfileId tag;

        Connector create(in Object object);

        boolean compare(in Object obj1, in Object obj2);
    };

    interface ConFactoryRegistry { // locality constrained
        void add_factory(in ConFactory Factory);

        boolean compare(in Object obj1, in Object obj2);
    };

    interface AccRegistry { // locality constrained
        void add_acceptor(in Acceptor Acceptor);

        void add_profiles(in ObjectKey key, in Object object);
        boolean compare(in Object obj1, in Object obj2);
    };

}; // end module OCI

module CORBA
{
    interface ORB
    {
        // Other definitions

        attribute OCI::ConFactoryRegistry registry;
    };

    interface BOA // Or POA
    {
        // Other definitions

        attribute OCI::AccRegistry registry;
    };

}; // end module CORBA
```

12.4.4.1 OCI::Buffer

Buffer is an interface for an input/output buffer. An input/output buffer (or just "buffer") can be regarded as an object holding an unbounded sequence of octets. Note, however, that using octet sequences for the implementation is only one of many possibilities. It is not mandatory to use octet sequences internally. In addition to the sequence of octets, a buffer contains a position counter of type **unsigned long** which determines how many octets have already been sent or received. The IDL interface definition for buffer is incomplete and must be extended by the specific language mappings. For example, the C++ mapping defines the following additional operations:

- **`Octet* data()`**: This operation returns a C++ pointer to the first element of the array of octets, which represents the buffer's contents.
- **`Octet* rest()`**: Similar to **`data()`** this operation returns a C++ pointer, but to the n-th element of the array of octets with n being the value of the position counter.

The **length** attribute is used to set and get the buffer's length, i.e. the total number of octets the buffer can hold.

The **pos** attribute is the position counter. Note that the buffer's length and the position counter don't depend on each other. There are no restrictions on the values permitted for the counter. This implies that it's even legal to set the counter to values beyond the buffer's length.

advance adds a value to the position counter.

is_full checks whether the buffer is full. The buffer is considered full if its length is equal to the position counter's value.

rest_length returns the length of the rest of the buffer. The **rest_length** is the length minus the position counter's value. If the value of the position counter exceeds the buffer's length, the return value is undefined.

12.4.4.2 OCI::Transport

Transport is an interface for a transport object. Transport objects provide operations for sending and receiving octet streams. In addition, it is possible to register callbacks with the transport object, which are invoked whenever data can be sent or received without blocking.

The readonly attribute **tag** returns the profile id tag for the specific protocol.

The **receive** and **send** operations receive or send a buffer's contents. If the **block** parameter is set to TRUE, the operation blocks until the buffer has been fully received or sent. If **block** is set to FALSE, these operations are non-blocking.

receive_detect and **send_detect** work like **receive** and **send**, but they signal a connection loss by returning FALSE instead of throwing a system exception.

The **receive_timeout** and **send_timeout** operations work like **receive** and **send**, but it is possible to specify a time-out value (in milliseconds). On return the caller can test whether there was a time-out by checking if the buffer has completely been received or sent, respectively. A zero time-out value is equivalent to calling receive or send with the block parameter set to FALSE.

set_receive_cb and **set_send_cb** are used to set callbacks, which are called whenever it is possible to receive or send a buffer (partially or completely) without blocking. Only one receive and one send callback can be set. Subsequent calls to these operations overwrite the old settings. A nil argument removes the current send or receive callback.

12.4.4.3 OCI::Connector

The **Connector** interface is used by CORBA clients to initiate a connection to a server. It also provides operations for the management of object references specific profiles.

The readonly attribute **tag** returns the profile id tag for the specific protocol.

The **connect** operation is used by CORBA clients to establish a connection to a CORBA server. It returns a transport object, which can be used for sending and receiving octet streams to and from the server.

compare checks whether an object reference contains at least one profile that matches this Connector. Only the connection specific profile data is taken into account, not the object key. This operation allows to reuse connections.

extract_key extracts an object key from a profile of the object reference matching this Connector. If more than one profile matches, the particular key returned from these profiles is implementation specific.

12.4.4.4 OCI::Acceptor

An **Acceptor** is used by CORBA servers to accept client connection requests. It also provides operations for the management of specific profiles in the object reference.

The readonly attribute **tag** returns the profile id tag for the specific protocol.

accept is used by CORBA servers to accept client connection requests. It returns a transport object which can be used for sending and receiving octet streams to and from the client.

The **add_profile** operation adds a new profile that reflects this Acceptor to an object reference.

The **compare** operation checks two object references for equality. Two object references are considered equal if at least one profile from the first object reference is equal to at least a profile from the second object reference.

set_accept_cb is used to set a callback that is called if a connection request can be accepted without blocking. Only one accept callback can be set. Subsequent calls to this operation overwrite the old setting. A nil argument removes the current callback.

12.4.4.5 OCI::ConFactory

The **ConFactory** interface serves as a Factory for Connector objects.

The readonly attribute **tag** returns the profile id tag for the specific protocol.

create creates a new Connector. All connection specific data is taken from a profile of the object reference that matches this Factory. If more than one profile matches, the particular profile that is used is implementation specific. A nil object reference is returned if the object reference does not contain a profile matching this Factory.

The **compare** operation checks two object references for equality. Two object references are considered equal if at least one profile from the first object reference is equal to at least one profile from the second object reference.

12.4.4.6 OCI::ConFactoryRegistry

A **ConFactoryRegistry** serves as a Registry for ConFactory objects. An instance of a ConFactoryRegistry is provided by the ORB.

add_factory adds a Connector Factory to the Registry.

compare checks two object references for equality. It calls the **compare** operation of the registered Connector factories. If at least one of these calls returns TRUE, the object references are considered equal and TRUE is returned. If none of the calls return TRUE, the object references are considered different and FALSE is returned.

12.4.4.7 OCI::AccRegistry

An **AccRegistry** serves as a Registry for Acceptor objects. An instance of an AccRegistry is provided by the ORB.

add_acceptor adds an Acceptor to the Registry.

add_profiles adds new profiles to an object reference. For each registered Acceptor a new profile is added by calling the Acceptor's **add_profile** operation.

compare checks two object references for equality by calling the **compare** operation of the registered Connector factories. If at least one of these calls returns TRUE the object references are considered equal and TRUE is returned. If none of the calls return TRUE the IORs are considered different and FALSE is returned.

12.5 The Remote Operation Interface

Note – The submitters are working on the subject and they will propose a complete interface for the second revision of the submission.

13 CORBA API

13.1 Object References and Transport End-Points

13.1.1 Architectural Considerations

The object reference contains various information that is used by the ORB to perform the method invocation. One type of information concerns the transport end-pointend-point. With a server supporting several transport end-points, at least two strategies exist. The first strategy provides fixed object references with no choice for the client about the transport end-point. The second strategy provides flexible object references and the ability for the client to choose the transport end-point.

- In strategy 1, the object reference only contains one transport end-point. The client which must perform an invocation has no choice for the transport end-point. This strategy forces the server to specify the transport end-point that the client will use. This strategy is useful to make sure that the client will not use a wrong transport end-point (for example one with higher priority) and to ensure that object references remain small entities. The drawback is probably an increased complexity of the distributed application: it is the responsibility of the server or third party factory to build an object reference which is suitable for the client.
- With strategy 2, the object reference contains several transport end-points. The client chooses the transport end-point that it must use for an invocation. The choice can be made according to various parameters (QoS, client priority,...).

The first strategy is probably nice to reduce the size of object references and also increase the performance of invocations. However, this can lead to some problems if for example client A passes the object reference to client B being more prioritized. In that case, client B might invoke the server using a wrong transport end-point. In such a situation, the second strategy gives the ability for client B to use the appropriate transport end-point.

The architecture does not mandate either solution. The choice of strategy is left to the realtime application.

A simple management of transport end-points can be made available on an object reference. The idea is to allow a server (or a third-party-tool) to update or query an existing object reference about the transport end-points that it knows. Here are examples where this may be useful:

- On the server side when the server must return an object reference to a client. The server has the ability to decide what transport end-points the client can use.
- In a third-party-tool dedicated to QoS negotiation, the negotiator can remove from the object reference all the transport end-points except the one that results from negotiation. By doing so, the client object reference remains small, the client is forced to use the specified transport end-point and there is no overhead on the client side to choose a suitable transport end-point (no choice).

- The object reference contains several transport end-points. The client chooses the transport end-point that it must use for an invocation. The choice can be made according to various parameters (QoS, client priority, ...).

Note – The submitters are working on the subject to provide interfaces that give the same semantics and functionalities as the proposed interface. However, being conscient that the **CORBA::Object** interface is absolutely not the correct place for such operations, they will propose new separate interfaces in the second revision of this submission.

13.1.2 Specification

```
module CORBA {
    interface Object { // locality constrained
        void insert_transport_end_point(
                    in RT::TransportEndPoint endPoint,
                    in long pos);
        void remove_transport_end_point(
                    in RT::TransportEndPoint endPoint);
        void remove_transport_end_point_at(in long pos);
        RT::TransportEndPoint get_transport_end_point(in long pos);
        long number_of_transport_end_points();
        void select_transport_end_point(
                    in RT::TransportEndPoint endPoint);
        void select_transport_end_point_at(in long pos);
    };
};
```

The **insert_transport_end_point** method allows insertion of a new transport end-point. No verification about the validity of the transport end-point can be made. One restriction that the ORB may provide is that this operation is only possible by the server which *owns* the object reference. The operation verifies that the transport end-point is not already present. The transport end-point is inserted at the position specified by `pos` within the list.

The **remove_transport_end_point** method removes a transport end-point. The transport end-point is searched within the list and then removed. The second form for removing the transport end-point is **remove_transport_end_point_at** which allows removal of a transport end-point at a given position within the list.

The **get_transport_end_point** method returns a transport end-point given its index. Together with **number_of_transport_end_points** it can be used to iterate over the supported transport end-points.

The **select_transport_end_point** method allows to select, before an invocation, the transport end-point which must be used. If the object reference contains only one end-point, this is not useful. The selected transport end-point is passed as parameter of the method. It must be present in the object reference. Any invocation with the object reference will be made with the selected transport end-point. If the object reference is passed to a method invocation, the selected transport end-point will remain. A second form of selection is available with the **select_transport_end_point_at** operation which selects the transport end-point at a given position within the list.

13.1.3 GIOP Transport End-Points

The GIOP 1.1 protocol allows specification of several components in the IOR profile. It is proposed to allocate one component for specifying a list of transport end-points.

13.2 Client Binding and QoS

13.2.1 Architectural Considerations

Note – The submitters are working on the client binding model and the QoS specification. The complete description of the client binding model will be described in the final submission.

In the same way that there is a server-side binding to tie an object to an OA, there is a need for a client-side binding to perform client-side actions when exploiting an object reference.

Classically, binding in computer systems denotes a set of actions or process for associating or interconnecting different objects of a computing system. Binding implies setting up an access path between two objects, which in turn typically comprises locating the target of the access path, checking access rights, setting up appropriate data structures in support of the access path to enable communication between objects following this path.

Take an object which is accessible through two different protocols, one is a traditional heavyweight WAN protocol (such as TCP/IP), the other a lightweight shared memory protocol (such as LRPC). There is a need for a policy to choose between the two, depending on where the client is. On the same machine, LRPC will be a good choice, and TCP/IP elsewhere.

Elements such as QoS, or transport protocols can be chosen through such policies. We will detail several policy examples later.

The policy can be a result of two different influences: that of the client, deciding which policy he prefers, or that of someone else (likely the server side), that will encapsulate a policy choice within the object references.

To lighten the burden placed on the programmer, these policy choices must be encapsulated in well-defined components. We will call these components Binding Policies. They are managed by a Binding Policy Manager. Their role is to prepare invocations at binding time, following policies that are defined either by the servant or by the client. The main result of a client binding operation can be to define the transport end-point that will be used for invocations, but arbitrary side-effects can also be performed.

The CORBA Object References can contain several transport end-points, along with additional components. Within these components is defined a Binding Policy component, composed of a Binding Policy Identifier and associated Binding Policy Data (an opaque container). The Policy Data is interpreted by the Binding Policy object itself in a specific way.

The client can perform binding in two ways :

- requesting binding from the Binding Manager using either a default policy or the policy specified by the server.
- requesting binding from a explicitly specified specialized Binding Policy with specific arguments. The Binding Policy is free to dispatch the request to another policy if required.

Anybody can define new binding policies, although it would be desirable to start with a few basic types. We outline a few example policies:

- **Closest**: the closest (in communication time) target is selected.
- **Verify**: the targets are tested for validity and accessibility before being selected.
- **QoS**: a channel with the selected Qos is used.
- **Encrypted**: an encrypted protocol is used depending on where the client is. The binding data contains a private key for encryption. Interceptors are used to encrypt the invocation data.
- **Interpreted**: the Binding Data contains data that can be dynamically interpreted to implement the binding policy.

The possibilities are endless and might be outside the scope of this proposal.

Typically, binding policies will be identified by Strings (as exemplified in the following sections).

13.2.2 Specification

```
module ClientBinding {
    interface Policy {
        void bind(inout Object object);
    };
    interface Manager {
        void bind(inout CORBA::Object object);
        void add(in Policy policy);
        void remove(in Policy policy);
        Policy find(in String type);
    };
    interface Server {
        CORBA::Object create (in String t, in PolicyData,
                                          in RepositoryID id);
    };
};
```

13.2.3 Locality Constraints

The **Policy**, **Manager** and **Server** objects are local to the process.

13.2.4 Example

The extract below gives an example of how the client can use the Binding Manager and how it specifies its policy.

```
...
CORBA::ORB_ptr orb = CORBA::ORB_init (argv, orb_identifier);
CORBA::Object_ptr bMgr =
   orb->resolve_inital_reference("BindingManager");
...
Binding::Manager_ptr bindingManager =
   Binding::Manager::_narrow (bMgr);
CORBA::Object_ptr gRef =  NS->resolve ("myService");


...// simple case using default or server-assigned policy

bindingManager->bind (gRef);
MyService mRef = MyService::_narrow (gRef);
mRef->foo();

...// elaborate case requesting Qos handling
Qos qos(...);
BindingPolicy pol = bindingManager->find ("QOS");
(BindingPolicyWithQOS)(pol).bind (gRef, qos);
MyService mRef = MyService::_narrow (gRef);
mRef->foo();
```

The extract below indicates how the server side uses the Binding Server.

```
...
CORBA::ORB_ptr orb =
   CORBA::ORB_init (argv, orb_identifier);
CORBA::Object_ptr bServ =
   orb->resolve_inital_reference("BindingServer");
Binding::Server_ptr bindingServer =
   Binding::Server::_narrow (bServ);
…..
RepositoryID repID = ...;
```

14 Realtime POA

The goal of this section is twofold:

- to extract from the POA specification [ORB Portability] the definitions appropriate to realtime applications. There is a generally accepted statement that a simplified, dedicated version of the POA specification is the right approach for a specific CORBA profile.
- to extend the POA specification to fulfill realtime requirements introduced by this proposal.

As a consequence of locality constraints to the POA specification, in this section the term binding refers to the binding on the server side.

14.1 Applying POA Specification to the Realtime CORBA Profile

14.1.1 Architectural Considerations

Guaranteeing end-to-end predictability requires the POA to avoid calling external, unpredictable events such as calling a servant locator for an incoming request.

The root POA is a distinguished CORBA object provided to the applications through the initial list of object references. Existing OMG default policies that apply to the root POA are discussed below with respect to realtime requirements:

- *Request Processing Policy:* **USE_ACTIVE_OBJECT_MAP_ONLY**

  The POA maintains an Active Object Map for servants. Alternatives are:

  - **USE_DEFAULT_SERVANT**. A default servant is invoked.

  - **USER_SERVANT_MANAGER**. A servant manager is called to locate the servant.

  The latter policy requires an external service that cannot guarantee end-to-end predictability. Although a default servant could be used, this proposal favors the default policy.

- *Servant Retention Policy*: **RETAIN**

  This policy is required when the Active Object Map is used. Alternative is **NON_RETAIN**, which requires a default servant or a servant manager and so, is not appropriate for the proposal.

- *Implicit Activation Policy*: **NO_IMPLICIT_ACTIVATION**

  Applications must activate all servants explicitly using the **activate_object** operation. The **RETAIN** and **USE_ACTIVE_OBJECT_MAP** combination requires a **NO_IMPLICIT_ACTIVATION** policy as described in [ORB Portability §3.3.7.6.1].

- *Thread Policy*: **ORB_CTRL_MODEL**

The ORB is responsible for assigning multiple threads to concurrent requests. Alternative is **SINGLE_THREAD_MODEL** whose semantics will be part of the extensions discussed in §14.2 “Extending the POA Specification” on page 90.

- *Lifespan Policy*: **TRANSIENT**

  Servants registered in the POA cannot outlive the process in which they are first created. Alternative is **PERSISTENT**. This latter policy cannot guarantee end-to-end predictibility since there is no guarantee to associate an active and even an existing POA to the servant invoked by an incoming request.

- *Id Assignment Policy*: **SYSTEM_ID**

  The POA assigns Object Ids to servants. Alternative is **USER_ID**: applications assign Object Ids to servants. Only the former allows use of the Object Id as part of a hash key for the Active Object Map, which is of prime interest for supporting high-performance lookup algorithms.

- *Object Id Uniqueness Policy*: **UNIQUE_ID**

  Exactly one Object Id is associated to one servant. Alternative is **MULTIPLE_ID**: multiple Object Ids may be associated to one servant. This latter policy is of no interest when Object Ids are assigned by the POA, as described above.

As a conclusion, only the default policies are of interest for a realtime CORBA profile. Child POAs will be created using implicitly the same policies. There is no need to support operations for creating alternative policies.

In a similar way:

- no POA Manager is required. The only operation that requires a reference to some POA Manager is **create_POA**. This operation is not appropriate for the proposal and is replaced by an operation creating a POA with an associated thread pool (see §8.2.5 “Thread Pools” on page 28).

- no Servant Manager is required. There is no need for functionality to locate servants.

- no Adapter Activator is required. As specified in [ORB Portability, §3.3.3], an application server that creates all its needed POAs at the beginning of execution does not need to use or to provide an adapter activator. Aternative is to create POAs during request processing, which cannot guaranteee end-to-end-predictibility.

The following operations should create object references and are not supported: **create_reference**, **create_reference_with_id**, **servant_to_reference**, **id_to_reference**. They do not allow to create reference associated with binding data.

14.1.2 Specification

The Realtime CORBA specification requires only operations relevant to the default policies to be supported. The corresponding **PortableServer** module is defined in §14.2 “Extending the POA Specification” on page 90. Definitions are briefly described below. Full specifications are defined in [ORB Portability].

14.1.2.1 The Servant IDL Type

native Servant;

Values of the type **Servant** are programming-language-specific implementations of CORBA interfaces.

14.1.2.2 The ObjectId Type

typedef sequence<octet> ObjectId;

The **ObjectId** type is defined as an unbound sequence of octets.

14.1.2.3 POA Interface

- *find_POA*

  POA find_POA (

  in string name,

  in boolean wait_for_completion);

  This operation finds the POA object of **name** argument from the root POA object. **FALSE** is expected as second argument for the realtime CORBA profile. Support for an adapter activator should be required otherwise.

- *destroy*

  void destroy (

  in boolean etherealize_objects,

  in boolean wait_for_completion);

  This operation destroys the POA and all descendant POAs. **TRUE** is required as first parameter for the **RETAIN** policy. If **TRUE** is passed as second argument, the **destroy** operation will return only after all requests in process have completed.

- *the_name*

  readonly attribute string the_name;

  This attribute identifies the POA.

- *the_parent*

  readonly attribute POA the_parent;

  This attribute identifies the parent of the POA.

- *activate_object*

  ObjectId activate_object (in Servant p_servant);

This operation registers the **servant** argument in the Active Object Map and returns the associated Object Id.

- *deactivate_object*

  void deactivate_object (in ObjectId oid);

  This operation removes the servant associated with the **ObjectId** argument from the Active Object Map.

- *servant_to_id*

  ObjectId servant_to_id (in Servant p_servant);

  The Object Id associated with the **servant** argument is returned.

- *reference_to_servant*

  Servant reference_to_servant (in Object reference);

  This operation returns the servant associated with the **reference** argument in the Active Object Map.

- *reference_to_id*

  ObjectId reference_to_id (in Object reference);

  This operation returns the Object Id value encapsulated by the **reference** argument.

- *id_to_servant*

  Servant id_to_servant (in ObjectId oid);

  This operation returns the active servant associated with the **ObjectId** argument.

14.2 Extending the POA Specification

14.2.1 Architectural Considerations

In addition to the functionality described in the previous section, a realtime POA profile must provide support for the POA to control thread management.

Extensions are designed using the following criteria:

- only one thread pool is associated with a POA. The thread pool and an invocation control policy are specified when a POA is created. Policies may not be changed on an existing POA. Policies are not inherited from the parent POA.
- binding data are specified to the POA when an object reference is created. Binding data include a thread policy appropriate to the servant.

IDL specifications for these extensions are encapsulated in a **RT** module embedded in the **PortableServer** module. Specifications are described in the next section and the corresponding IDL is recapitulated in §A.2 “POA” on page 100. Note that the Realtime Object Adapter is defined as the **ROA** interface deriving from the **POA** interface.

14.2.2 Specification

14.2.2.1 Thread Pool Policy

Objects with the **ThreadPoolPolicy** interface are obtained using the **ROA::create_thread_pool_policy** operation and passed to the **ROA::create_POA_with_thread_pool** operation to specify the threading pool model used with the created POA. The value attribute of **ThreadPoolPolicy** contains the value supplied to the **ROA::create_POA_with_thread_pool** operation from which it was obtained. The following values can be supplied:

- **THREAD_POOL** - A thread from the pool is used to process the request.
- **THREAD_PER_REQUEST** - A new thread is created to process the request. The pool must have been configured with an unlimited maximum of threads.

  The thread pool policy must be defined by the implementer of the pool in accordance with the thread pool attributes as specified in §8.2.5 "Thread Pools" on page 28. **::RT::RequestQueue** operations are responsible for defining the behavior when no thread is available from the pool (see in §8.3 "Request Queue and Flow Control" on page 31). For example, an exception may be raised, or the request may be put in a queue.

14.2.2.2 Servant Policy

Values for the servant policy are obtained from the **ServantPolicyValue** enumeration and supplied to the **ROA::create_binding_data** operation. The following values can be supplied:

- **THREAD_NONE** - No new thread needs to be created on the reception of a request. Invocation is expected to be short-time and the ORB does not need to create a new thread.
- **THREAD_SAFE** - Only one thread can execute an operation on the invoked servant. This model is particularly appropriate for mono-threaded servants used in multithreaded environments. It is strictly equivalent to the **SINGLE_THREAD_MODEL** thread policy as specified in [ORB Portability, §3.3.7.1].
- **THREAD_NOT_SAFE** - Multiple threads can execute concurrently on a same servant whatever the invoked operation is. There is no concurrency control performed by the ORB.

14.2.2.3 Binding Data Interface

attribute servantPolicyValue servant_policy;

Objects with the **BindingData** interface are obtained using the **ROA::create_binding_data** operation and passed to the **ROA::create_reference_with_data** operation. The value attribute of **BindingData** contains the servant policy value supplied to the **ROA::create_binding_data** operation from which it was obtained.

14.2.2.4 Creating a POA with a Thread Pool

```
ROA create_POA_with_thread_pool (
        in string adapter_name,
        in ::RT::ThreadPool pool,
        in ThreadPoolPolicy policy)
    raises (AdapterAlreadyExists, InvalidThreadPoolPolicy);
```

This operation creates a new POA of type **ROA** as a child of the target POA. The specified name identifies the new POA with respect to other POAs with the same parent POA. If the target POA already has a child POA with the specified name, the **AdapterAlreadyExists** exception is raised. The **::RT::ThreadPool** parameter defines the pool associated with the POA. The **ThreadPoolPolicy** parameter defines the policy associated with the pool. If this policy is not one of the expected policies, the **InvalidThreadPoolPolicy** exception is raised.

14.2.2.5 Policy Creation Operations

```
ThreadPoolPolicy create_thread_pool_policy (
        in ThreadPoolPolicyValue value);
```

This operation returns a reference to a policy object with the specified value. The application is responsible for calling the inherited destroy operation on the returned reference when it is no longer needed.

14.2.2.6 Binding Data Creation Operation

```
BindingData create_binding_data (in ServantPolicyValue value);
```

This operation returns a reference to a **BindingData** object containing the specified servant policy value. The application is responsible for calling the inherited destroy operation on the returned reference when it is no longer needed.

14.2.2.7 Creating a Reference with (Binding) Data

```
Object create_reference_with_data (

        in CORBA::RepositoryId intf,

        in BindingData data);
```

This operation creates an object reference that encapsulates a POA-generated ObjectId value and the specified binding data. This operation does not cause an activation to take place. The **activate_object** operation must be called after the object creation and prior to any request received by the ORB and applied to the servant referenced by the object.

14.2.3 IDL for Extensions to POA

```
module PortableServer
{
    module RT
    {
        // Policy interfaces
        enum ThreadPoolPolicyValue {
            THREAD_POOL,
            THREAD_PER_REQUEST
        };
        interface ThreadPoolPolicy
        {
            readonly attribute ThreadPoolPolicyValue value;
        };

        enum ServantPolicyValue {
            THREAD_NONE,
            THREAD_SAFE,
            THREAD_NOT_SAFE
        };

        // Binding Data
        interface BindingData {
            attribute servantPolicyValue servant_policy;
        };

        // Realtime Object Adapter Interface
        interface ROA : POA
        {
            // POA creation
            ROA create_POA_with_thread_pool (
                in string adapter_name,
                in ::RT::ThreadPool pool,
                in ThreadPoolPolicy policy);

            // Factories for Policy objects and Binding Data
            ThreadPoolPolicy create_thread_pool_policy (
                in ThreadPoolPolicyValue value);
            BindingData create_binding_data (
                in ServantPolicyValue value);


            // reference creation operation
            Object create_reference_with_data (
                in CORBA::RepositoryId intf,
                in BindingData data);
        };
    };
};
```

14.3 Usage Scenario

Applications get an object reference to the realtime root POA through the initial list of references and the **_narrow** operation:

```
// C++
CORBA::ORB_ptr orb =
CORBA::ORB_init (argc, argv, orb_identifier);
CORBA::Object_ptr obj =
orb->resolve_initial_references ("RootPOA");
PortableServer::RT::ROA_ptr rootPOA =
PortableServer::RT::ROA::_narrow (obj);
if (CORBA::is_nil (rootPOA) == CORBA::TRUE)
{
      error ("not a realtime root POA");
}
```

A nil object reference returns by the _narrow operation means that the POA registered through the list of initial references was not of real type **PortableServer::RT::ROA** as expected. This test allows an early check against a bad usage of the root POA.

Applications create a child POA as follows:

```
// C++
RT::ThreadFactory_ptr threadFactory = ...;
RT::ThreadPoolAttributes_ptr poolAttributes = ...;
RT::TheadPool_ptr pool =
threadFactory->create_thread_pool (poolAttributes);

PortableServer::RT::ThreadPoolPolicyValue poolPolicyValue =
...;
PortableServer::RT::ThreadPoolPolicy_ptr poolPolicy =
rootPOA->create_thread_policy_pool (poolPolicyValue);
PortableServer::RT::ROA_ptr rtPOA =
rootPOA->create_POA_with_thread_pool
("rtPOA", pool, poolPolicy);
```

Applications register servants with the **rtPOA** as follows:

```
// C++
PortableServer_Servant servant = ....;
PortableServer::ObjectId oid =
rtPOA->activate_object (servant);
```

Applications get an object reference for this servant as follows:

```
// C++
CORBA::RepositoryId intf = ...;
PortableServer::RT::ServantPolicyValue servantPolicyValue =
...;
PortableServer::RT::BindingData_ptr binding_data =
rootPOA->create_binding_data (servantPolicyValue);

CORBA::Object_ptr obj =
rtPOA->create_reference_with_data(intf, binding_data);
```

15 Example Scenario

This chapter presents some client/server scenario to illustrate the flexibility of the realtime ORB architecture and to show how the realtime end-to-end predictability can be guaranteed.

Note – The submitters will provide several examples in the second submission.

16 Relation with COS Specifications

The realtime CORBA RFP recommends addressing the following CORBA Comon Object Services: Concurrency, Time, Transaction and Event Service. This chapter gives a quick overview of the meaning and impact of realtime in CORBA services, and more specifically those mentioned above.

16.1 The COS Specifications and realtime

The Design Principles document explicitly states that quality of service is a characteristic of the implementation. This implies a number of things :

- CORBAservices don’t specify any constraint pertaining to the realtime behavior of the service implementations. They don’t specify interfaces to give access to realtime information either.
- The specification of the CORBAservices leaves to the implementation the placement of the components implementing the services. The implementation can thus be centralized, local or distributed, or even replicated. These design choices can have a dramatic impact on the response time of the services. This problem is inherent to the current service specification model.
- The service implementations could favor some operations against others depending on their structure and design choices. This is typically the case when caching requests. Nothing is provided to take advantage nor to detect this situation.

Because of all these points, the CORBAservices as specified cannot offer realtime guarantees.

To coexist in a realtime environment, CORBAservices should provide information about their operational realtime characteristics, such as best execution time (i.e. cached), typical execution time, worst execution time, dependencies with other services.

Additionally, the services should favor implementations offering quick response time and possibly allow fixed-time service semantics (with abortion when time is expired).

16.2 Service Classification

There are many CORBAservices. We make a very synthetic classification in order to outline a number of needs. The CORBAservices are not equally involved in the writing of realtime applications. Some are mainly used during the application's initialization phase while others are used along the whole execution.

16.2.1 Services Used in Initialization

Services are not always uniformly used during application execution. A number of them are mostly used during the application initialization phase. This doesn't mean that these services are unusable outside of this phase, but that they can affect adversely the realtime behavior when they are used in sensitive situations.

The following services are considered important in the OMG architecture:

- Naming Service
- Trading Service
- Property Service
- Relationship Service
- Licensing Service

Each of those services have an impact on the initialization phase:

- Naming Service : there is no strong realtime requirement for the naming service if one considers that it is only used in the initialization phase.
- Trader Service : this service might have a part to play as it can allow an application in initialization phase to access references for implementations offering services conforming to the realtime requirements.
- Property Service : during the initialization phase, this service might be used to provide realtime information. By exporting a PropertyService interface, an implementation might provide its operating realtime characteristics.
- Relationship Service : this service might be useful in finding out relationship between services, for example to determine realtime properties at run-time.
- The Licensing Service : its impact should be marginal and specific to the initialization phase.

16.2.2 *Run-Time Services*

The following services are likely to have an impact on the execution phase of the application.

- Time Service
- Concurrency Service
- Event Management Service
- Query Service
- Transaction Service
- Query Service
- Transaction Service
- Lifecycle Service
- Persistent Object Service
- Security Service
- Externalization Service

16.2.3 *Independent Services*

The CORBA design principles specify that services should be as orthogonal as possible. Even though those services can cooperate, some of them are independent.

- Concurrency Control Service
- Time Service
- Event Management Service
- Security Service
- Property Service
- Query Service

16.3 *Realtime Required Services*

The following services are mandated by the RealTime RFP :

- Time Service : this service offers an interface with a timing resolution close to a nanosecond. While most implementations will never come that far, the limits of the clock's resolution are largely outside of the CORBA domain. This service also offers timers, however there is no mention of the system's behavior when two timers spring up at the same time, particularly when they run in activities with different priorities.

- Concurrency Service : this service might have an important impact on the realtime behavior as standalone locks (outside of transactions) appear to be blocking, without delays. However, the concurrency service is crucial to resource management and could be used in implementing the MPCP realtime resource scheduling protocol.
- Event Management Service : This service is involved whenever there is a requirement for secure asynchronous calls, as the oneway mechanism provides no guarantees nor information about the completion of a request. This requirement appears in a large number of realtime systems, hence this service needs to provide adequate support for realtime operation. However, this mechanism is both costly and unnatural notably because of its non-procedural appearance. Deferred synchronous calls as the base model for oneway invocations would provide a better solution.
- Transaction Service : the semantics of transactions is somewhat unsuited to realtime systems as it usually requires locking strategies involving several participants and possible rollback to a previous state. However, the option of time-limited transactions helps having a behavior compatible with some realtime requirements.

Appendix A Consolidated IDL

A.1 CORBA Modules Extension

The CORBA module is extended to provide new operations:

```
#include "iop.idl"

module CORBA {
    typedef IOP::ComponentId ComponentId;
    typedef sequence<octet> ComponentData;

    const long IOR_PROFILE_LEVEL = 1;
    const long IIOP_PROFILE_LEVEL = 2;
    const long IIOP_KEY_LEVEL = 4;
    const long EXCLUSIVE = 8;

    interface Object { // locality constrained with special operation mapping
        void set_component(in ComponentId id,
                                    in long flags, in ComponentData d);
        void get_component(in ComponentId id,
                                    in long flags, out ComponentData d);
        void remove_component(in ComponentId id, in long flags);
        boolean has_component(in ComponentId id, in long flags);


        // The following operations are subject to be moved in
        // another interfaces to avoid modification of CORBA Object
        // and also dependencies to Realtime module.
        void insert_transport_end_point(
                        in RT::TransportEndPoint endPoint,
                        in long pos);
        void remove_transport_end_point(
                        in RT::TransportEndPoint endPoint);
        void remove_transport_end_point_at(in long pos);
        RT::TransportEndPoint get_transport_end_point(in long pos);
        long number_of_transport_end_points();
        void select_transport_end_point(
                        in RT::TransportEndPoint endPoint);
        void select_transport_end_point_at(in long pos);
    };
};
```

A.2 POA

```
module PortableServer
{
    module RT
    {
        // Policy interfaces
        enum ThreadPoolPolicyValue {
            THREAD_POOL,
            THREAD_PER_REQUEST
        };
        interface ThreadPoolPolicy
        {
            readonly attribute ThreadPoolPolicyValue value;
        };

        enum ServantPolicyValue {
            THREAD_NONE,
            THREAD_SAFE,
            THREAD_NOT_SAFE
        };

        // Binding Data
        interface BindingData {
            attribute servantPolicyValue servant_policy;
        };

        // Realtime Object Adapter Interface
        interface ROA : POA
        {
            // POA creation
            ROA create_POA_with_thread_pool (
                in string adapter_name,
                in ::RT::ThreadPool pool,
                in ThreadPoolPolicy policy);

            // Factories for Policy objects and Binding Data
            ThreadPoolPolicy create_thread_pool_policy (
                in ThreadPoolPolicyValue value);
            BindingData create_binding_data (
                in ServantPolicyValue value);


            // reference creation operation
            Object create_reference_with_data (
                in CORBA::RepositoryId intf,
                in BindingData data);
        };
    };
};
```

A.3 Interceptor Module

```
#pragma prefix "omg.org"
module Interceptor {
    enum Status {
        INVOKE_CONTINUE,
        INVOKE_ABORT,
        INVOKE_RETRY
    };

    interface Root { // locality constrained
        typedef unsigned long Priority;

        readonly attribute Priority prio;

        const Priority LowestPriority = 0; // Priority'First
        const Priority HighestPriority = 0xffffffff;
                                            // Priority'Last;

        void activate(in Priority p);
        void deactivate();
        boolean is_active();
    };

    interface Context { // locality constrained
        void destroy();
    };

    typedef IOP::ServiceID ServiceID;
    typedef sequence<octet> ContextData;

    interface LWRootRequest { // locality constrained
        attribute Object target;
        attribute Identifier operation;

        void set_service_context(in ServiceID id,
                                 in long flags, in ContextData d);
        ContextData get_service_context(in ServiceID id,
                                        in long flags);
        void remove_service_context(in ServiceID id);
        boolean has_service_context(in ServiceID id):

        void set_context(in Root interceptor, in Context ctx);
        Context get_context(in Root interceptor);
    };

    interface LWRequest : LWRootRequest { // locality constrained
        readonly attribute CORBA::Request request;
    };
```

```
        interface LWServerRequest : LWRootRequest { // locality constrained
            readonly attribute CORBA::Request request;
        };

        interface ClientInterceptor : Root { // locality constrained
            Status initialize_request(in LWRequest req,
                                      in CORBA::Environment env);
            Status after_marshal(in LWRequest req,
                                 in CORBA::Environment env);
            Status before_unmarshal(in LWRequest req,
                                    in CORBA::Environment env);
            Status finish_request(in LWRequest req,
                                  in CORBA::Environment env);
        };

        interface ServerInterceptor : Root { // locality constrained
            Status initialize_request(in LWServerRequest req,
                                      in CORBA::Environment env);
            Status after_unmarshal(in LWServerRequest req,
                                   in CORBA::Environment env);
            Status before_marshal(in LWServerRequest req,
                                  in CORBA::Environment env);
            Status finish_request(in LWServerRequest req,
                                  in CORBA;;Environment env);
        };

        interface InitInterceptor : Root { // locality constrained
            Status initialize(in CORBA::ORB orb,
                              in CORBA::ORB id,
                              inout CORBA::arg_list);
        };
    };
```

A.4 Realtime Modules

The following modules are new to the realtime ORB:

```
// IDL
#include <corba.idl>
#pragma prefix "omg.org"

module RT {
    // Realtime Thread API
    interface ThreadAttributes { // locality constrained
    };

    typedef unsigned long ThreadSpecificKey;
    interface ThreadSpecific { // locality constrained
        void destroy();
    };

    interface Thread { // locality constrained
        attribute ThreadAttributes attr;

        void join(out long status);
        void detach();
    };
    interface CurrentThread : Thread{ // locality constrained
        void exit(in long status);

        void set_thread_specific(in ThreadSpecificKey key,
                                 in ThreadSpecific data);
        ThreadSpecific get_thread_specific(in ThreadSpecificKey key);
        void remove_thread_specific(in ThreadSpecificKey key);
        boolean has_thread_specific(in ThreadSpecificKey key);
    };
    native ThreadParam;
    interface ThreadHandler { // locality constrained
        long run(in ThreadParam param);
    };

        // Thread Pools
    interface ThreadPoolAttributes { // locality constrained
        attribute ThreadAttributes thread_attributes;
        attribute RequestQueue request_queue;
        attribute unsigned long number_of_threads;
    };
    interface ThreadPool { // locality constrained
        attribute ThreadPoolAttributes attr;

        void destroy();
    };
    interface ThreadFactory { // locality constrained
        Thread create_thread(in ThreadAttributes attr,
                             in ThreadHandler entry,
                             in ThreadParam param);
        CurrentThread get_current_thread();
        ThreadPool create_thread_pool(in ThreadPoolAttributes attr);
```

```
};

// Request Queue API
interface RequestQueueAttributes { // locality constrained
};
interface RequestQueue { // locality constrained
    attribute RequestQueueAttributes attr;

    void put_request(in CORBA::ServerRequest req);
    CORBA::ServerRequest get_request();
    unsigned long pending_requests();
    void destroy(in long mode);
};
interface RequestQueueFactory {
                              // locality constrained
    RequestQueue create_request_queue(
                              in RequestQueueAttributes attr);
};

// Transport API
interface TransportAttributes { // locality constrained
    attribute ThreadPool thread_pool;
};
interface TransportEndPoint { // locality constrained
    attribute TransportAttributes attr;

    boolean is_equal(in TransportEndPoint endPoint);
    string type();
    string to_url();
    void from_url(in string id);
    unsigned long hash(in unsigned long maximum);
    void open(in TransportAttributes a);
    void close();
    void destroy(in long mode);
};
interface TransportEndPointFactory {
                              // locality constrained
    TransportEndPoint create_end_point(
                                  in TransportAttributes attr);
    unsigned long number_of_end_points();
    TransportEndPoint get_end_point(in unsigned long pos);
};

// Synchronization Facilities
interface Mutex { // locality constrained
    void acquire();
    void release();
```

```
    boolean try_acquire();
    void destroy();
};
interface Semaphore { // locality constrained
    void acquire();
    boolean try_acquire();
    void acquire();
    void destroy();
};

interface RWLock { // locality constrained
    void acquire_write();
    boolean try_acquire_write();
    void acquire_read();
    boolean try_acquire_read();
    void release();
    void destroy();
};
interface ConditionVariable { // locality constrained
    boolean wait(in Mutex m, in long delay);
    void signal();
    void broadcast();
    void destroy();
};
interface SynchronizationFactory { // locality constrained
    Mutex create_mutex();
    Semaphore create_semaphore(in long count);
    RWLock create_RWLock();
    ConditionVariable create_condition();
};

// Entry point for factories
interface StrategyFactory {
    ThreadFactory get_thread_factory();
    TransportEndPointFactory get_transport_factory(in string name);
    RequestQueueFactory get_request_queue_factory();
    SynchronizationFactory get_synchronization_factory();
};


// Realtime Policies: POSIX and TCP/IP specifications
// POSIX Threads
native StackAddr;
interface PosixThreadAttributes : ThreadAttributes {
                                    // locality constrained
    struct SchedParam {
        long priority;
    };
    enum SchedulerType {
        SCHED_OTHER,
        SCHED_FIFO,
```

```
            SCHED_RR
        };

        attribute SchedParam sched_attr;
        attribute SchedulerType sched_policy;
        attribute unsigned long stack_size;
        attribute StackAddr stack_addr;
    };

    // TCP/IP Specific Attributes
    interface TcpTransportAttributes : TransportAttributes {
                        // locality constrained
        attribute long tcp_send_size;
        attribute long tcp_recv_size;
        attribute boolean tcp_keep_alive;
        attribute boolean tcp_dont_route;
        attribute unsigned short tcp_port;
        attribute unsigned long tcp_addr;
    };
};
```

A.5 Client Binding

```
#pragma prefix "omg.org"

module ClientBinding {
    interface Policy {
        void bind(inout Object object);
    };
    interface Manager {
        void bind(inout CORBA::Object object);
        void add(in Policy policy);
        void remove(in Policy policy);
        Policy find(in String type);
    };
    interface Server {
        CORBA::Object create (in String t, in PolicyData, in RepositoryID id);
    };
};
```

A.6 Scheduling Service

```
#pragma prefix “omg.org”

module CosScheduling {
    exception InvalidPriority {};
    exception InvalidPC {};
    exception PriorityNotSet {};
    exception ExistingSchedEntity {};

    typedef long Priority;

    const Priority NoScheduling = 0;
    const Priority NoCeiling = 0;

    interface ClientSchedEntity {
        void  set_priority(in Priority global_priority)
            raises(InvalidPriority);

        Priority get_priority()
            raises (PriorityNotSet);
    };

    interface ServantSchedEntity {
        void  set_priority(in Priority global_priority)
            raises(InvalidPriority);

        Priority get_priority()
            raises(PriorityNotSet);

        void set_priority_ceiling(in Priority priority_ceiling)
            raises(InvaildPC);

        Priority get_priority_ceiling();
            raises(PriorityNotSet);
    };

     interface SchedEntityFactory {
        ClientSchedEntity create_client_sched_entity(
            in Priority priority);
            raises(InvalidPriority, ExistingSchedEntity);

        ServantSchedEntity create_servant_sched_entity(
            in Priority priority,
            in Priority priority_ceiling);
            raises(InvalidPriority, ExistingSchedEntity);
    };
};
```

A.7 Pluggable Protocol

```
#pragma prefix "omg.org"
module OCI
{
    typedef sequence<octet> ObjectKey;

    interface Buffer { // locality constrainted
        attribute unsigned long length;
        attribute unsigned long pos;
        void advance(in unsigned long delta);
        unsigned long rest_length();
        boolean is_full();
    };

    interface ReceiveCB { // locality constrainted
        void receive_cb();
    };

    interface SendCB { // locality constrained
        void send_cb();
    };

    interface Transport { // locality constrained
        readonly attribute ProfileId tag;

        void receive(in Buffer buf, in boolean block);
        boolean receive_detect(in Buffer buf, in boolean block);
        void receive_timeout(in Buffer buf, in unsigned long timeout);

        void send(in Buffer buf, in boolean block);
        boolean send_detect(in Buffer buf, in boolean block);
        void send_timeout(in Buffer buf, in unsigned long timeout);

        void set_receive_cb(in ReceiveCB cb);
        void set_send_cb(in SendCB cb);
    };

    interface Connector { // locality constrained
        readonly attribute ProfileId tag;

        Transport connect();

        boolean compare(in Object object);
        ObjectKey extract_key(in Object object);
    };

    interface AcceptCB { // locality constrained
        void accept_cb();
    };
```

```
    interface Acceptor { // locality constrained
        readonly attribute ProfileId tag;

        Transport accept();

        void set_accept_cb(in AcceptCB cb);

        void add_profile(in ObjectKey key, inout Object object);
        boolean compare(in Object obj1, in Object obj2);
    };

    interface ConFactory { // locality constrained
        readonly attribute ProfileId tag;

        Connector create(in Object object);

        boolean compare(in Object obj1, in Object obj2);
    };

    interface ConFactoryRegistry { // locality constrained
        void add_factory(in ConFactory Factory);

        boolean compare(in Object obj1, in Object obj2);
    };

    interface AccRegistry { // locality constrained
        void add_acceptor(in Acceptor Acceptor);

        void add_profiles(in ObjectKey key, in Object object);
        boolean compare(in Object obj1, in Object obj2);
    };

}; // end module OCI
```