Real-time CORBA Specification

January 2005
Version 1.2
formal/05-01-04

An Adopted Specification of the Object Management Group, Inc.

implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Contents

Contents

Contents

Preface

Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 600 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based. More information is available at http://www.omg.org/.

Intended Audience

The architecture and specifications described in this manual are aimed at software designers and developers who want to produce applications that comply with OMG standards for the Object Request Broker (ORB). The benefit of compliance is, in general, to be able to produce interoperable applications that are based on distributed, interoperating objects. As defined by the Object Management Group (OMG) in the *Object Management Architecture Guide*, the ORB provides the mechanisms by which objects transparently make requests and receive responses. Hence, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

Context of CORBA

The key to understanding the structure of the CORBA architecture is the Reference Model, which consists of the following components:

- **Object Request Broker**, which enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for interoperability between applications in hetero- and homogeneous environments. The architecture and specifications of the Object Request Broker are described in this manual.
- **Object Services**, a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Services are necessary to construct any distributed application and are always independent of application domains. For example, the Life Cycle Service defines conventions for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Specifications for Object Services are contained in *CORBAservices: Common Object Services Specification.*
- **Common Facilities**, a collection of services that many applications may share, but which are not as fundamental as the Object Services. For instance, a system management or electronic mail facility could be classified as a common facility. Information about Common Facilities will be contained in *CORBAfacilities: Common Facilities Architecture*.
- **Application Objects,** which are products of a single vendor on in-house development group that controls their interfaces. Application Objects correspond to the traditional notion of applications, so they are not standardized by OMG. Instead, Application Objects constitute the uppermost layer of the Reference Model.

The Object Request Broker, then, is the core of the Reference Model. It is like a telephone exchange, providing the basic mechanism for making and receiving calls. Combined with the Object Services, it ensures meaningful communication between CORBA-compliant applications.

Associated Documents

The CORBA documentation set includes the following books:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for the Object Services.
- *CORBAfacilities: Common Facilities Architecture* contains the architecture for Common Facilities.

OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote.

You can download the OMG formal documents free-of-charge from our web site in PostScript and PDF format. Please note the OMG address and telephone numbers below:

OMG Headquarters

250 First Avenue

Needham, MA 02494

USA

Tel: +1-781-444-0404

Fax: +1-781-444-0320

http://www.omg.org

Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren't required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding specified in the *C++ Language Mapping Specification.*

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *Common Object Request Broker Architecture (CORBA)* specification, "Compliance to COM/CORBA Interworking."

As described in the *OMA Guide*, the OMG's Core Object Model consists of a core and components. Likewise, the body of *CORBA* specifications is divided into core and component-like specifications.

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Helvetica bold - OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier bold` - Programming language elements.

Helvetica - Exceptions

Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Alcatel
- Ericsson
- France Telecom
- Hewlett-Packard Company
- Highlander Communications, L.C.
- Humboldt-University
- Inprise Corporation
- IONA Technologies, Plc.
- Lockheed Martin Federal Systems, Inc.
- Lucent Technologies, Inc.
- MITRE Corporation
- Motorola, Inc.
- Nortel Networks
- Objective Interface Systems, Inc.
- Object-Oriented Concepts, Inc.
- SPAWAR Systems Center
- Sun Microsystems, Inc.
- Tri-Pacific Software, Inc.
- University of Rhode Island
- Washington University

Real-time CORBA Base Architecture 1

Contents

This chapter contains the following topics.

Real-time CORBA is an optional set of extensions to CORBA tailored to equip ORBs to be used as a component of a real-time system.

1.1 Goals of the Specification

In any architecture, there is a tension between a general purpose solution and supporting specialist applications. Real-time developers have to pay strict attention to the allocation of resources and to the predictability of system execution. By providing the developer with handles on managing resources and on predictability, Real-time CORBA sacrifices some of the general purpose nature of CORBA in order support the development of real-time systems.

Real-time development has further specialist areas: “hard” real-time and “soft” real-time; different resource contention protocols and scheduling algorithms, etc. This specification provides a Real-time CORBA that is sufficiently general to span these

variations in the form of a single compliance point. The one restriction imposed by the specification is to fixed priority scheduling. Real-time CORBA does not currently address dynamic scheduling.

The prescriptions made by this specification are not essential for general purpose CORBA development. Furthermore, for some use-cases of CORBA; for example, Enterprise Distributed Computing, the features of Real-time CORBA would be irrelevant. EDC tends to focus on usability and developer productivity. Placing these goals way above predictability means that EDC CORBA developers would never do things like configure thread pools.

The goals of the specification are to support developers in meeting real-time requirements by facilitating the end-to-end predictability of activities in the system and by providing support for the management of resources.

Real-time CORBA brings to real-time system development the same benefits of implementation flexibility, portability, and interoperability that CORBA brought to client-server development.

There is one important non-goal for this specification. It is not a goal to provide a portability layer for the real-time operating system (RTOS) itself. The POSIX Real-time extensions already address this need. Real-time CORBA is compatible with the POSIX Real-time Extensions but by not wrapping the RTOS the specification facilitates the use of Real-time CORBA on operating systems that fall outside of the POSIX Real-time Extensions.

1.2 Extending CORBA

To provide specialist capabilities for specialist application without over constraining non real-time development, Real-time CORBA is positioned as a separate Extension to CORBA. The set of capabilities provided by Real-time CORBA constitute an optional, additional compliance point.

Real-time CORBA is defined as extensions to CORBA 2.2 (formal/98-12-01) and the Messaging Specification (orbos/98-05-05). It is necessary to look beyond CORBA 2.2 because the policy framework used in Real-time CORBA is that from the Messaging Specification. Secondly, deferred synchronous, asynchronous, and oneway invocations are important tools in developing real-time systems.

1.3 Approach to Real-time CORBA

1.3.1 The Nature of Real-time

Developers of CORBA-compliant distributed, object oriented systems rely on the CORBA Specification to support the functional aspects of those systems. However, there is a class of problems where some of the requirements relate the functionality of the system to Real-World time, be it measured in minutes or in microseconds. For these systems, timeliness is as important as functionality.

A parcel delivery service that commits to next day delivery across the country is relating the functional requirement of transporting a parcel from “A” to “B” to Real-world time; that is, “one day.” For the organization to meet this non-functional requirement, it must analyze the system, identify the activities, and bound the time taken to perform them. It must also decide what resources (people, planes, etc.) are allocated to the problem. The use of those resources in performing particular activities must be coordinated so that one activity doesn’t prejudice the Real-World time requirement of another activity. If the arrival rate of parcels and the isolation of resources from the outside world are known, then the organization can (ignoring component failures) guarantee “next day” delivery. If the arrival pattern of parcels is variable and the peak rate would suggest a large amount of resources (which would at other times be largely idle), then the organization could fall back to statistical predictability: offering “next day delivery or your money back.”

Relating functional requirements to real-world time may take several forms. A response time requirement might say that the occurrence of event “A” shall cause an event “B” within 24 hours. A throughput requirement might say that the system shall cope with 1000 occurrences of an event per hour. A statistical requirement might say that 95% of the occurrences of event “A” shall cause an event “B” within 24 hours. All these forms of requirement are real-time requirements. A system that meets real-time requirements is a real-time system.

1.3.2 Meeting Real-time Requirements

Deterministic behavior of the components of a real-time system promotes the predictability of the overall system. In order to decide *a priori* if a real-time requirement is met, the system must behave predictably. This can only happen if all the parts of the system behave deterministically and if they “combine” predictably.

The interfaces and mechanisms provided by Real-time CORBA facilitate a predictable combination of the ORB and the application. The application manages the resources by using the Real-time CORBA interfaces and the ORB’s mechanisms coordinate the activities that comprise the application. The real-time ORB relies upon the RTOS to schedule threads that represent activities being processed and to provide mutexes to handle resource contention.

1.3.3 Distributable Thread

This specification provides an abstraction for distributed real-time programming as an end-to-end schedulable entity termed *distributable thread*. Such a distributed execution model is essential to support:

- the statically-scheduled Real-time CORBA Base Architecture described in this and the following chapter; and
- as a foundation for dynamically scheduled real-time CORBA systems described in the Dynamic Scheduling chapter below.

Prior versions of this specification loosely described the term "*activity.*" This term was avoided to reduce conflict with prior usage in CORBA specifications such as Workflow Management (formal/00-05-02) and Additional Structuring Mechanisms for the OTS Specification (formal/02-09-03).

This specification does not attempt to address more advanced issues such as fault tolerance, propagation of system information, and control along the path of a distributable thread, etc. These facilities may be provided in a subsequent revision of this specification.

For further details on the term *distributable thread* refer to *Section 3.9, "Distributable Thread," on page 3-11*.

1.3.4 End-to-End Predictability

One goal of this specification is to provide a standard for CORBA ORB implementations that support end-to-end predictability. For the purposes of this specification, "end-to-end predictability" of timeliness in a fixed priority CORBA system is defined to mean:

- respecting thread priorities between client and server for resolving resource contention during the processing of CORBA invocations;
- bounding the duration of thread priority inversions during end-to-end processing;
- bounding the latencies of operation invocations.

A Real-time CORBA system will include the following four major components, each of which must be designed and implemented in such a way as to support end-to-end predictability, if end-to-end predictability is to be achieved in the system as a whole:

1. the scheduling mechanisms in the OS;
2. the real-time ORB;
3. the communication transport;
4. the application(s).

Real-time ORBs conformant to this specification are still reliant on the characteristics of the underlying operating system and on the application if the overall system is to exhibit end-to-end predictability.

Note – An OS that implements the IEEE POSIX 1003.1-1996 Real-time Extensions has the necessary features to facilitate end-to-end predictability. It is still possible for an OS that doesn't implement some or all of the POSIX Real-time Extensions specification to support end-to-end predictability. Real-time CORBA is not restricted to such OSs.

1.3.5 Management of Resources

Providing end-to-end predictability will entail explicit choices in how much resources are deployed in a system. Certain requirements will lead to static partitioning of these resources among activities.

For real-time requirements of the statistical kind and for some throughput requirements, the level of resources needed to make the system “schedulable” can be prohibitive. Real-time CORBA systems can still provide assurances that requirements are met due to the explicit control provided over resources.

Resources come in three categories: process, storage, and communication resources. Real-time CORBA offers control over threadpools, which objects the threads within them are used for, and what priorities they might run at. Real-time CORBA also appends some storage resources to threadpools for the specific capability of handling a number of concurrent requests above the number of threads provided. Real-time CORBA provides control over transport connections: which are shared and which are allocated for what priority of activity.

1.4 Compatibility

1.4.1 Interoperability

Real-time CORBA does not prescribe an RT-IOP as an ESIOP. There are a number of pragmatic reasons for this. There are many specialized scenarios in which Real-time CORBA can be deployed. These different scenarios do not exhibit enough common characteristics to allow a common interaction protocol to be defined. Secondly, each scenario will impose a different transport protocol. Without agreeing on a common transport, interoperability isn’t possible.

Instead of specifying an RT-IOP, this specification uses the “standard extension” mechanisms provided by IIOP. These mechanisms are GIOP ServiceContexts, IOR Profiles, and IOR Tagged Components. Using these it is possible for IIOP to provide protocol support for the mechanisms prescribed in Real-time CORBA.

The benefit is that two Real-time CORBA implementations will interoperate. Interoperability may not be as important for a Real-time CORBA system as for a CORBA system because real-time dictates a measure of system-wide design control to deliver predictability and therefore also some control over which ORB to deploy.

The second benefit is that the specified extensions define what features of a vendors own Real-time IOP can be mapped onto IIOP. This allows vendors to bridge between different Real-time CORBA implementations.

1.4.2 Portability

Providing real-time applications with portability across real-time ORBs is a goal of this specification, providing a portability layer for real-time operating systems is not a goal. Basing such an RTOS wrapper on say, POSIX Real-time Extension would constrain the range of operating systems to which Real-time CORBA can add value.

Any real-time system will be carefully configured to meet its real-time requirements. This includes taking account of the behavior and timings of the ORB itself. Porting an application to a different Real-time ORB will necessitate that the application be reconfigured. Portability cannot be “write once run everywhere” for Real-time CORBA. What it does do is reduce the risk to a development of having to port.

1.4.3 CORBA - Real-time CORBA Interworking

In many systems Real-time CORBA components will have to interwork with CORBA components. The interfaces (in particular IIOP extensions) are specified so that this is functionally possible. Clearly, in any given system, there will be timing and predictability implications that need to be considered if the real-time component is not to be compromised.

CORBA applications can be ported to Real-time ORBs. They simply will not make use of the extra functions provided. Porting a real-time application to a non-real-time ORB will sacrifice the predictability of that application but the two platforms are functionally equivalent.

1.5 Real-time CORBA Architectural Overview

Real-time CORBA defines a set of extensions to CORBA. The extensions to the CORBA Core are specified in the Real-time CORBA Extensions chapter. The extensions to the Real-time Base Architecture required to support dynamically scheduled systems is specified in the Dynamic Scheduling chapter.

Figure 1-1 shows the key Real-time CORBA entities that are specified. The features that these relate to are described in overview in the following sections.

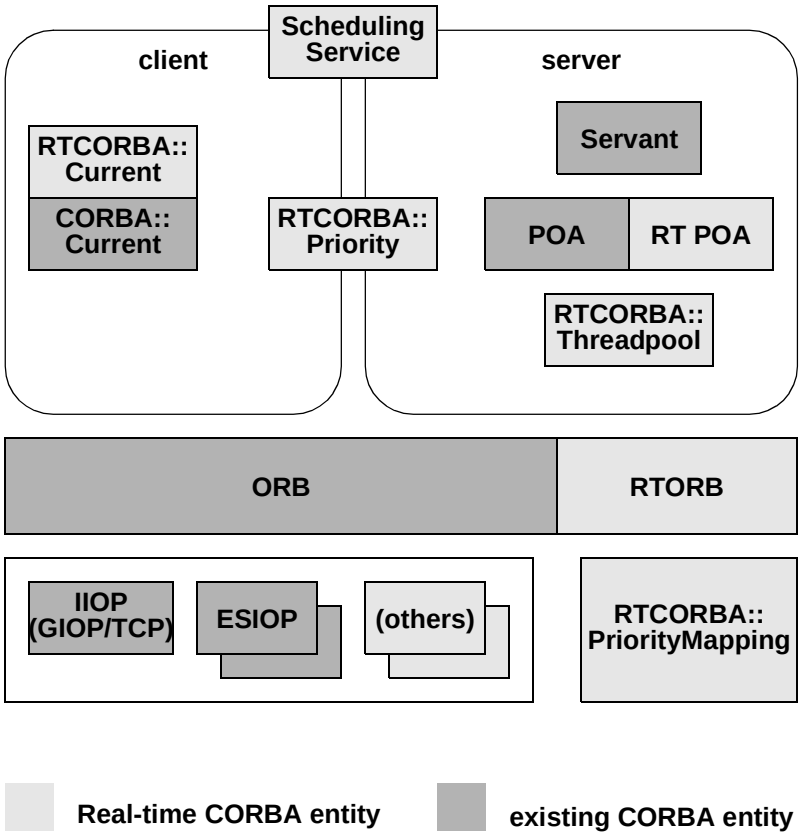

Figure 1-1 Real-time CORBA Extensions

1.5.1 Real-time CORBA Modules

All CORBA IDL specified by Real-time CORBA is contained in new modules RTCORBA and RTPortableServer (with the exception of new service contexts, which are additions to the IOP module.)

1.5.2 Real-time ORB

Real-time CORBA defines an extension of the ORB interface, **RTCORBA::RTORB**, which handles operations concerned with the configuration of the real-time ORB and manages the creation and destruction of instances of other Real-time CORBA IDL interfaces.

1.5.3 Thread Scheduling

Real-time CORBA uses threads as a schedulable entity. Generally, a thread represents a sequence of control flow within a single node. Threads for part of an activity. Activities are “scheduled” by coordination of the scheduling of their constituent threads. Real-time CORBA specifies interfaces through which the characteristics of a thread that are of interest can be manipulated. These interfaces are Threadpool creation and the Real-time CORBA Current interface.

Note – The Real-time CORBA view of a thread is compatible with the POSIX definition of a thread.

1.5.4 Real-time CORBA Priority

Real-time CORBA defines a universal, platform independent priority scheme called *Real-time CORBA Priority.* It is introduced to overcome the heterogeneity of different Operating System provided priority schemes, and allows Real-time CORBA applications to make prioritized CORBA invocations in a consistent fashion between nodes with different priority schemes.

For consistency, Real-time CORBA applications always should use CORBA Priority to express the priorities in the system, even if all nodes in a system use the same native thread priority scheme, or when using the server declared priority model.

1.5.5 Native Priority and PriorityMappings

Real-time CORBA defines a **NativePriority** type to represent the priority scheme that is ‘native’ to a particular Operating System.

Priority values specified in terms of the Real-time CORBA Priority scheme must be mapped into the native priority scheme of a given scheduler before they can be applied to the underlying schedulable entities. On occasion, it is necessary for the reverse mapping to be performed, to obtain a Real-time CORBA Priority to represent the present native priority of a thread. The latter can occur, for example, when priority inheritance is in use, or when wishing to introduce an already running thread into a Real-time CORBA system at its present (native) priority.

To allow the Real-time ORB and applications to do both of these things, Real-time CORBA defines a **PriorityMapping** interface.

1.5.6 Real-time CORBA Current

Real-time CORBA defines a Real-time CORBA **Current** interface to provide access to the CORBA priority of a thread.

1.5.7 Priority Models

One goal of Real-time CORBA is to bound and to minimize priority inversion in CORBA invocations. One mechanism that is employed to achieve this is propagation of the activity priority from the client to the server, with the requirement that the server side ORB make the up-call at this priority (subject to any priority inheritance protocols that are in use).

However, in some scenarios, it is sufficient to design the application system by setting the priority of servers, and having them handle all invocations at that priority. Hence, Real-time CORBA supports two models for the priority at which a server handles requests from clients:

- Client Propagated Priority Model: in which the server honors the priority of the invocation, set by the client. The invocation's Real-time CORBA Priority is propagated to the server ORB and the server-side ORB maps this Real-time CORBA Priority into its own native priority scheme using its **PriorityMapping**.

  Requests from non-Real-time CORBA ORBs; that is, ORBs that do not propagate a Real-time CORBA Priority with the invocation are handled at a priority specified by the server.

- Server Declared Priority Model: in which the server handles requests at a Real-time CORBA Priority assigned on the server side. This model is useful for setting a boundary where new activities are begun with a CORBA invocation.

1.5.8 Real-time CORBA Mutexes and Priority Inheritance

The **Mutex** interface provides the mechanism for coordinating contention for system resources. Real-time CORBA specifies an **RTCORBA::Mutex** locality constrained interface, so that applications can use the same mutex implementation as the ORB.

A conforming Real-time CORBA implementation must provide an implementation of **Mutex** that implements some form of priority inheritance protocol. This may include, but is not limited to, simple priority inheritance or a form of priority ceiling protocol. The mutexes that Real-time CORBA makes available to the application must have the same priority inheritance properties as those used by the ORB to protect resources. This allows a consistent priority inheritance scheme to be delivered across the whole system.

1.5.9 Threadpools

Real-time CORBA uses the Threadpool abstraction to manage threads of execution on the server-side of the ORB. Threadpool characteristics can only be set when the threadpool is created. Threadpools offer the following features:

- ***preallocation of threads*** - This helps reduce priority inversion, by allowing the application programmer to ensure that there are enough thread resources to satisfy a certain number of concurrent invocations, and helps reduce latency and increase predictability, by avoiding the destruction and recreation of threads between invocations.

- ***partitioning of threads*** - Having multiple thread pools associated with different POAs allows one part of the system to be isolated from the thread usage of another, possibly lower priority, part of the application system. This can again be used to reduce priority inversion.
- ***bounding of thread usage*** - A threadpool can be used to set a maximum limit on the number of threads that a POA or set of POAs may use. In systems where the total number of threads that may be used is constrained, this can be used in conjunction with threadpool partitioning to avoid priority inversion by thread starvation.
- ***buffering of additional requests*** beyond the number that can be dispatched concurrently by the assigned number of threads.

1.5.10 Priority Banded Connections

To reduce priority inversion due to use of a non-priority respecting transport protocol, RT CORBA provides the facility for a client to communicate with a server via multiple connections, with each connection handling invocations that are made at a different CORBA priority or range of CORBA priorities. The selection of the appropriate connection is transparent to the application, which uses a single object reference as normal.

1.5.11 Non-Multiplexed Connections

Real-time CORBA allows a client to obtain a private transport connection to a server, which will not be multiplexed (shared) with other client-server object connections.

1.5.12 Invocation Timeouts

Real-time CORBA applications may set a timeout on an invocation in order to bound the time that the client application is blocked waiting for a reply. This can be used to improve the predictability of the system.

1.5.13 Client and Server Protocol Configuration

Real-time CORBA provides interfaces that enable the selection and configuration of protocols on the server and client side of the ORB.

1.5.14 Real-time CORBA Configuration

New Policy types are defined to configure the following server-side RT CORBA features:

- server-side thread configuration (through Threadpools)
- priority model (propagated by client versus declared by server)
- protocol selection

- protocol configuration

Which of the CORBA policy application points (ORB, POA, Current) a given policy may be applied at is given along with the description of each policy.

Real-time CORBA defines a number of policies that may be applied on the client-side of CORBA applications. These policies allow:

- the creation of priority-banded sets of connections between clients and servers;
- the creation of a non-multiplexed connection to a server;
- client-side protocol selection and configuration.

In addition, Real-time CORBA uses an existing CORBA policy, to provide invocation timeouts.

Real-time CORBA Extensions 2

Contents

This chapter contains the following topics.

2.1 Real-time ORB

Real-time CORBA defines an extension to the **CORBA::ORB** interface, **RTCORBA::RTORB**. This interface is not derived from **CORBA::ORB** as the latter is expressed in pseudo IDL, for which inheritance is not defined. Nevertheless, **RTORB** is conceptually the extension of the ORB interface.

The **RTORB** interface provides operations to create and destroy other constituents of a Real-time ORB.

There is a single instance of **RTCORBA::RTORB** per instance of **CORBA::ORB**. The object reference for the **RTORB** is obtained by calling **ORB::resolve_initial_references** with the **ObjectId** "**RTORB**."

RTCORBA::RTORB is a local interface. The reference to the **RTORB** object may not be passed as a parameter of an IDL operation nor may it be stringified. Any attempt to do so shall result in a MARSHAL system exception (with a Standard Minor Exception Code of 4).

```
// IDL
module RTCORBA {

    local interface RTORB {

        ...

    };

};
```

2.1.1 Real-time ORB Initialization

Real-time ORB initialization occurs within the **CORBA::ORB_init** operation. That is a Real-time ORB's implementation of **CORBA::ORB_init** shall perform any actions necessary to initialize the real-time capability of the ORB.

In order to give the developer some control over a Real-time ORB's use of priorities the **ORB_init** operation shall be capable of handling an argv element of the form:

–ORBRTpriorityrange<optional-white-space><short>,<short>

Where **<short>** is a string encoding of an integer between 0 and 32767. The first integer should be smaller than the second. If the argv element string does not conform to these constraints, then a BAD_PARAM system exception shall be raised.

The two integers represent a range of CORBA Priorities available for use by ORB internal threads. Note that priority of Real-time CORBA application threads is controlled by other mechanisms. If the ORB cannot map these integers onto the native priority scheme, then it shall raise a DATA_CONVERSION system exception.

If the ORB deems the range of priorities to be too narrow for it to function properly, then it shall raise an INITIALIZE system exception (with a Standard Minor Exception Code of 1). For example, an implementation may not be able to function with less than, say, three distinct priorities without risking deadlock.

2.1.2 Real-time CORBA System Exceptions

Real-time CORBA provides a more constraining environment for an application than the environment provided by CORBA. This is reflected in the additional circumstances in which system exceptions can be generated. These circumstances need to be differentiated from the use of the same exception in CORBA.

Real-time CORBA uses many of the Standard System Exceptions with the same meaning as applies in CORBA. These uses need no differentiation. Where the use of a CORBA Standard System Exception has a meaning particular to Real-time CORBA, Standard Minor Exception Codes are assigned.

Table 2-1 Standard Minor Exception Codes used for Real-time CORBA

SYSTEM EXCEPTION	MINOR CODE	EXPLANATION
MARSHAL	4	Attempt to marshal local object.
DATA_CONVERSION	2	Failure of PriorityMapping object.
INITIALIZE	1	Priority range too restricted for ORB.
BAD_INV_ORDER	18	Attempt to reassign priority.
NO_RESOURCES	2	No connection for request's priority.

2.2 Real-time POA

Real-time CORBA defines an extension to the POA, in the form of the interface **RTPortableServer::POA**.

```
// IDL
module RTPortableServer {

    local interface POA : PortableServer::POA {

        ...

    };

};
```

Conformance to the Real-time CORBA Extensions also necessarily implies conformance to CORBA. In particular, a Real-time ORB will handle interfaces of type **PortableServer::POA** in accordance with the CORBA specification. For a Real-time ORB all such instances shall be of the subtype **RTPortableServer::POA**. That is it

shall always be possible to treat an instance of **PortableServer::POA** as an instance of **RTPortableServer::POA**; for example, successfully narrow in some language mappings.

A call to **ORB::resolve_initial_references("RootPOA")** shall return an interface of type **RTPortableServer::POA**. A Real-time POA will differ from a POA in two ways. Firstly, it shall provide additional operations to support object level priority settings (see Section 2.7.5, "Setting Server Priority on a per-Object Reference Basis," on page 2-13). Secondly, its implementation shall understand the Real-time Policies defined in this Extension. As the Real-time POA interface is derived from the POA interface, it shall support all the semantics prescribed for the POA.

2.3 Native Thread Priorities

A real-time operating system (RTOS) sufficient to use for implementing a Real-time ORB compliant with this specification, will have some discrete representation of a thread priority. This representation typically specifies a range of values and a direction for which values, higher or lower, represent the higher priority. The particular range and direction in this priority representation varies from RTOS to RTOS. This specification refers to the RTOS specific thread priority representation as a *native thread priority scheme*. The priority values of this scheme are referred to as *native thread priorities*.

Native thread priorities are used to designate the execution eligibility of threads. The ordering of native thread priorities is such that a thread with higher native priority is executed at the exclusion of any threads in the system with lower native priorities.

A native thread priority is an integer value that is the basis for resolving competing demands of threads for resources. Whenever threads compete for processors or ORB implementation-defined resources, the resources are allocated to the thread with the highest native thread priority value.

The *base native thread priority* of a thread is defined as the native priority with which it was created, or to which it was later set explicitly. The initial value of a thread's base native priority is dependent on the semantics of the specific operating environment. Hence it is implementation specific.

At all times, a thread also has an *active native thread priority*, which is the result of considering its base native thread priority together with any priorities it inherits from other sources; for example, threads or mutexes. An active native thread priority is set implicitly as a result of some other action. Its value is only temporary, at some point it will return to the base native thread priority.

Priority inheritance is the term used for the process by which the native thread priority of other threads is used in the evaluation of a thread's active native thread priority. A *priority inheritance protocol* must be used by a conforming Real-time CORBA ORB to implement the execution semantics of threads and mutexes. It is an implementation issue as to whether the Real-time ORB implements simple priority inheritance, immediate ceiling locking protocol, original ceiling locking protocol, or some other priority inheritance protocol.

Whichever priority inheritance protocol is used, the native thread priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists. At the point when a thread stops inheriting a native thread priority from another source, its active native thread priority is re-evaluated.

The thread's active native priority is used when the thread competes for processors. Similarly, the thread's active native priority is used to determine the thread's position in any queue; that is, dequeuing occurs in native thread priority order.

Native priorities have an IDL representation in Real-time CORBA, which is of type short:

```
// IDL
module RTCORBA {

    typedef short NativePriority;

};
```

This means that native priorities must be integer values in the range -32768 to +32767. However, for a particular RTOS, the valid range will be a sub-range of this range.

Real-time CORBA does not support the direct use of native priorities: instead, the application programmer uses CORBA Priorities, which are defined in the next section. However, applications will still use native priorities where they make direct use of RTOS features.

2.4 CORBA Priority

To overcome the heterogeneity of RTOSs, that is different RTOSs having different native thread priority schemes, Real-time CORBA defines a CORBA Priority that has a uniform representation system-wide. CORBA Priority is represented by the **RTCORBA::Priority type**:

```
//IDL
module RTCORBA {

   typedef short Priority;
   const Priority minPriority =     0;
   const Priority maxPriority = 32767;

};
```

A signed short is used in order to accommodate the Java language mapping. However, only values in the range 0 (minPriority) to 32767 (maxPriority) are valid. Numerically higher **RTCORBA::Priority** values are defined to be of higher priority.

For each RTOS in a system, CORBA priority is mapped to the native thread priority scheme. CORBA priority thus provides a common representation of priority across different RTOSs.

2.5 CORBA Priority Mappings

Real-time CORBA defines the concept of a **PriorityMapping** between CORBA priorities and native priorities. The concept is defined as an IDL native type so that the mechanism by which priorities are mapped is exposed to the user. Native is chosen rather than interface (even if locality constrained) because the full capability of the ORB; for example, POA policies and CORBA exceptions are too heavyweight for this use. Furthermore, a CORBA interface would entail the creation and registration of an object reference.

```
// IDL
module RTCORBA {

    native PriorityMapping;

};
```

Language mapping for this IDL native are defined for C, C++, Ada, and Java later in this section.

A Real-time ORB shall provide a default mapping for each platform; that is, RTOS that the ORB supports. Furthermore, a Real-time ORB shall provide a mechanism to allow users to override the default priority mapping with a priority mapping of their own.

The **PriorityMapping** is a programming language object rather than a CORBA Object and therefore the normal mechanism for coupling an implementation to the code that uses it (an object reference) doesn't apply. This specification does not prescribe a particular mechanism to achieve this coupling.

Note – Possible solutions include: recourse to build/link tools and provision of proprietary interfaces. Other solutions are not precluded.

2.5.1 C Language binding for PriorityMapping

```
/* C */
CORBA_boolean RTCORBA_PriorityMapping_to_native (
      RTCORBA_Priority           corba_priority,
      RTCORBA_NativePriority* native_priority );

CORBA_boolean RTCORBA_PriorityMapping_to_CORBA (
      RTCORBA_NativePriority native_priority,
      RTCORBA_Priority*         corba_priority );
```

2.5.2 C++ Language binding for PriorityMapping

```
// C++
namespace RTCORBA {

   class PriorityMapping {
      public:
         virtual CORBA::Boolean to_native (
               RTCORBA::Priority corba_priority,
               RTCORBA::NativePriority &native_priority );
         virtual CORBA::Boolean to_CORBA (
               RTCORBA::NativePriority native_priority,
               RTCORBA::Priority &corba_priority );
   };
};
```

2.5.3 Ada Language binding for PriorityMapping

```
-- Ada
package RTCORBA.PriorityMapping is

   type Object is tagged private;

   procedure To_Native (
      Self           : in Object ;
      CORBA_Priority : in RTCORBA.Priority ;
      Native_Priority: out RTCORBA.NativePriority ;
      Returns        : out CORBA.Boolean ) ;

   procedure To_CORBA (
      Self           : in Object ;
      Native_Priority: in RTCORBA.NativePriority ;
      CORBA_Priority : out RTCORBA.Priority ;
      Returns        : out CORBA.Boolean ) ;

end RTCORBA.PriorityMapping ;
```

2.5.4 Java Language binding for PriorityMapping

```
// Java
package org.omg.RTCORBA;
   public class PriorityMapping {

      boolean to_native (
         short corba_priority,
         org.omg.CORBA.ShortHolder native_priority
      );
      boolean to_CORBA (
         short native_priority,
         org.omg.CORBA.ShortHolder corba_priority
      );
}
```

2.5.5 Semantics

The priority mappings between native priority and CORBA priority are defined by the implementations of the **to_native** and **to_CORBA** operations of a PriorityMapping object (note, not a CORBA Object). The **to_native** operation accepts a CORBA Priority value as an in parameter and maps it to a native priority, which is given back as an out parameter. Conversely, **to_CORBA** accepts a **NativePriority** value as an in parameter and maps it to a CORBA Priority value, which is again given back as an out parameter.

As the mappings are used by the ORB, and may be used more than once in the normal execution of an invocation, their implementations should be as efficient as possible. For this reason, the mapping operations may not raise any CORBA exceptions, including system exceptions. The ORB is not restricted from making calls to the **to_native** and/or **to_CORBA** operations from multiple threads simultaneously. Thus, the implementations should be re-entrant.

Rather than raising a CORBA exception upon failure, a boolean return value is used to indicate mapping failure or success. If the priority passed in can be mapped to a priority in the target priority scheme, TRUE is returned and the value is returned as the out parameter. If it cannot be mapped, FALSE is returned and the value of the out parameter is undefined.

Both **to_native** and **to_CORBA** must return FALSE when passed a priority that is outside of the valid priority range of the input priority scheme. For **to_native** this means when it is passed a short value outside of the CORBA Priority range, 0-32767; that is, a negative value. For **to_CORBA** this means when it is passed a short value outside of the native priority range used on that RTOS. This range will be platform specific.

Neither **to_native** nor **to_CORBA** is obliged to map all valid values of the input priority scheme (the CORBA Priority scheme or the native priority scheme, respectively.) This allows mappings to be produced that do not use all values of the native priority scheme of a particular scheduler and/or that do not use all values of the CORBA Priority scheme.

When the ORB receives a FALSE return value from a mapping operation that is called as part of the processing of a CORBA invocation, processing of the invocation proceeds no further. A DATA_CONVERSION system exception (with a Standard Minor Exception Code of 2) is raised to the application making the invocation. Note that it may not be possible to raise an exception to the application if the failure occurs on a call to a mapping operation made on the server side of a oneway invocation.

A Real-time ORB cannot assume that the priority mapping is idempotent. Users should be aware that a mapping that produces different results for the same inputs will make the goal of a schedulable system harder to obtain. Users may choose to implement a priority mapping that changes (through other, user specified interfaces). Users should however note that post-initialization changes to the mapping may well compromise the ORB's ability to deliver a consistently schedulable system.

2.6 Real-time Current

The **RTCORBA::Current** interface, derived from **CORBA::Current**, provides access to the CORBA Priority (and hence indirectly to the native priority also) of the current thread. The application can obtain an instance of Current by invoking the **CORBA::ORB::resolve_initial_references("RTCurrent")** operation.

A Real-time CORBA Priority may be associated with the current thread, by setting the priority attribute of the **RTCORBA::Current** object:

```
//IDL
module RTCORBA {

    local interface Current : CORBA::Current {
        attribute Priority base_priority;
    };

};
```

A BAD_PARAM system exception shall be thrown if an attempt is made to set the priority to a value outside the range 0 to 32767.

When the attribute is set to a valid Real-time CORBA Priority value, the value is immediately used to set the base native priority of the thread. The native priority value to use is determined by calling **PriorityMapping::to_native** on the installed PriorityMapping. The native thread priority shall be set before the set attribute call returns.

If the **to_native** call returns FALSE or if the returned native thread priority is illegal for the particular underlying RTOS, then a Real-time ORB shall raise a DATA_CONVERSION system exception (with a Standard Minor Exception Code of 2). In this case the priority attribute shall retain its value prior to the set attribute call.

Once a thread has a CORBA Priority value associated with it, the behavior when it makes an invocation upon a CORBA Object depends on the value of the **PriorityModelPolicy** of that target object.

Retrieving the value of this attribute returns the last value that was set from the current thread. If this attribute has not previously been set for the current thread, attempting to retrieve the value causes an INITIALIZE System Exception to be raised.

2.7 Real-time CORBA Priority Models

Real-time CORBA supports two models for the coordination of priorities across a system. These two models provide two, alternate answers to the question: Where does the priority at which the servant code executes come from? They are:

- Client Propagated Priority Model
- Server Declared Priority Model

These two models are described in Section 2.7.3, “Client Propagated Priority Model,” on page 2-12 and Section 2.7.4, “Server Declared Priority Model,” on page 2-13, respectively. The model to be used is selected by the **PriorityModelPolicy** described first.

2.7.1 PriorityModelPolicy

The Priority Model is selected and configured by use of the **PriorityModelPolicy**:

```
//IDL
module RTCORBA {

    // Priority Model Policy
    const CORBA::PolicyType
                PRIORITY_MODEL_POLICY_TYPE = 40;

    enum PriorityModel {
        CLIENT_PROPAGATED,
        SERVER_DECLARED
    };

    local interface PriorityModelPolicy : CORBA::Policy {
```

```
        readonly attribute PriorityModel priority_model;
        readonly attribute Priority server_priority;

    };

};
```

When the Server Declared Model is selected for a given POA, the **server_priority** attribute indicates the priority that will be assigned by default to CORBA Objects managed by that POA. This priority can be overridden on a per-Object Reference basis, as described in a sub-section below.

When the Client Propagated Model is selected, the **server_priority** attribute indicates the priority to be used for invocations from non-Real-time CORBA ORBs; that is, where there is no **RTCorbaPriority** ServiceContext on the request.

2.7.2 Scope of PriorityModelPolicy

The **PriorityModelPolicy** is applied to a Real-time POA at the time of POA creation. This is either through an ORB level default having previously been set or by including it in the policies parameter to **create_POA**. An instance of the **PriorityModelPolicy** is created with the **create_priority_model_policy** operation. The attributes of the policy are initialized with the parameters of the same name.

```
//IDL
module RTCORBA {

    local interface RTORB {
        ...
        PriorityModelPolicy create_priority_model_policy (
                        in PriorityModel priority_model,
                        in Priority server_priority
        );
    };

};
```

The **PriorityModelPolicy** is a client-exposed policy; that is, propagated from the server to the client in IORs. It is propagated in a **PolicyValue** in a **TAG_POLICIES** Profile Component, as specified by the CORBA QoS Policy Framework.

When an instance of **PriorityModelPolicy** is propagated, the **PolicyValue**'s ptype has the value **PRIORITY_MODEL_POLICY_TYPE** and the pvalue is a CDR encapsulation containing an **RTCORBA::PriorityModel** and an **RTCORBA::Priority**.

Note – Client-exposed policies and the mechanism for their propagation are defined in the *CORBA Messaging* specification (see the *Common Object Request Broker Architecture* (*CORBA)* specification, *Messaging* chapter).

The **PriorityModelPolicy** is propagated so that the client ORB knows which Priority Model the target object is using. This allows it to determine whether to send the Real-time CORBA priority with invocations on that object, and, in the case that the Server Declared model is used in combination with Priority Banded Connections, allows it to select the band connection to invoke over based on the declared priority in the tagged component.

The client may not override the **PriorityModelPolicy**.

2.7.3 Client Propagated Priority Model

If the target object supports the **CLIENT_PROPAGATED** value of the **PriorityModelPolicy**, the CORBA Priority is carried with the CORBA invocation and is used to ensure that all threads subsequently executing on behalf of the invocation run at the appropriate priority. The propagated CORBA Priority becomes the CORBA Priority of any such threads. These threads run at a native priority mapped from that CORBA Priority. The CORBA Priority is also passed back from server to client, so that it can be used to control the processing of the reply by the client ORB.

The CORBA Priority is propagated from client to server, and back again, in a CORBA Priority service context, which is passed in the invocation request and reply messages.

```
module IOP {

    const ServiceId       RTCorbaPriority = 10;

};
```

The **context_data** contains the **RTCORBA::Priority** value as a CDR encapsulation of an IDL short type.

Note – The **RTCorbaPriority** const should be added to a future version of GIOP.

The thread that runs the servant code, initially has the CORBA Priority of the invoking thread. Therefore if, as part of the processing of this request it makes CORBA invocations to other objects, these onward invocations will be made with the same CORBA Priority. If the CORBA Priority of the thread running the servant code is changed by the application, any subsequent onward invocations will be made with this new priority.

Note that priorities may be changed implicitly, by the platform (RT ORB + RTOS) whilst the servant code is executing due to priority inheritance.

2.7.4 Server Declared Priority Model

An object using the Server Declared Priority Model will have published its CORBA Priority in its object reference. When such an object is the target of an invocation the CORBA Priority at which the (remote) servant code will execute is available to the client-side ORB. The client-side ORB may use this knowledge internally. For example, in conjunction with priority banded connections.

Note – Client-side ORB execution to support an invocation should run at the priority of the client making the invocation. The extent to which this is achieved is a matter for implementation.

The client's Real-time CORBA Priority value is not passed with the invocation, in a service context, as it is in the Client Priority Propagation Model. A Real-time CORBA Priority is not passed in a reply message either.

Server-side threads running on behalf of the invocation run at a native priority mapped from the Real-time CORBA Priority associated with that CORBA Object, which is given in the **server_priority** attribute of the **PriorityModelPolicy** used at its creation.

Where an object, S1, using the Server Declared Priority Model makes invocations of its own upon another target object, S2, that uses the Client Propagated Priority Model, the priority propagated will be that of S1 and not that of S1's client. If the CORBA Priority of the thread executing S1's code is changed by the application, any subsequent onward invocations will be made with this new priority.

Note that priorities may be changed implicitly by the platform (RT ORB + RTOS) while the servant code is executing due to priority inheritance.

2.7.5 Setting Server Priority on a per-Object Reference Basis

The server priority assigned under the Server Declared Priority Model, by the **server_priority** attribute of the **PriorityModelPolicy**, can be overridden on a per-Object Reference basis. This is achieved by assigning the alternate server priority at the time of Object Reference creation or servant activation, using one of four additional operations, which are provided by the Real-time CORBA POA, **RTPortableServer::POA**. Thereafter, the ORB shall ensure that the servant code is run at a native thread priority corresponding to the CORBA priority supplied as input to these operations.

```
// IDL
module RTPortableServer {

    local interface POA : PortableServer::POA {
```

```
        Object create_reference_with_priority (
                        in CORBA::RepositoryId intf,
                        in RTCORBA::Priority priority )
            raises ( WrongPolicy );

        Object create_reference_with_id_and_priority (
                        in PortableServer::ObjectId oid,
                        in CORBA::RepositoryId intf,
                        in RTCORBA::Priority priority )
            raises ( WrongPolicy );

        PortableServer::ObjectId activate_object_with_priority (
                        in PortableServer::Servant p_servant,
                        in RTCORBA::Priority priority )
            raises ( ServantAlreadyActive, WrongPolicy );

        void activate_object_with_id_and_priority (
                        in PortableServer::ObjectId oid,
                        in PortableServer::Servant p_servant,
                        in RTCORBA::Priority priority )
            raises ( ServantAlreadyActive,
                        ObjectAlreadyActive, WrongPolicy );

    };

};
```

If the priority parameter of any of the above operations is not a valid CORBA priority or if it fails to match the priority configuration for resources assigned to the POA, then the ORB shall raise a BAD_PARAM system exception.

For each of the above operations, if the POA does not support the **SERVER_DECLARED** option for the **PriorityModelPolicy**, then the ORB shall raise a WrongPolicy user exception.

For each of the above operations, if the POA supports the **IMPLICIT_ACTIVATION** option for the **ImplicitActivationPolicy**, then the ORB shall raise a WrongPolicy user exception. This relieves an ORB implementation of the need to retrieve the target object's priority from "somewhere" when a request arrives for an inactive object.

If the **activate_object_with_id_and_priority** operation is invoked with a different priority to an earlier invocation of one of the create reference with priority operations, for the same object, then the ORB shall raise a BAD_INV_ORDER system exception (with a Standard Minor Exception Code of 18). If the priority value is the same, then the operation will be successful.

In all other respects the semantics of the corresponding; that is, without the name extensions "**_with_priority**" and "**_and_priority**" **PortableServer::POA** operations shall be observed.

2.8 Priority Transforms

Real-time CORBA supports the installation of user-defined Priority Transforms, to modify the CORBA Priority associated with an invocation during the processing of the invocation by a server. Use of these Priority Transforms allows application designers to implement Real-time CORBA systems using priority models different from either the Client Propagated or Server Declared priority models, described above.

There are two points at which a Priority Transform may affect the CORBA Priority associated with an invocation:

- During the invocation up call (after the invocation has been received at the server but before the servant code is invoked). This is referred to as an 'inbound' Priority Transform, and will occur before the first time the server-side ORB uses the **RTCORBA::Priority** value to obtain a native priority value, via a **to_native** operation on the Priority Mapping.
- At the time of making an 'onward' CORBA invocation, from servant application code. This is referred to as an 'outbound' Priority Transform.

Priority Transforms are user-provided functions that map one **RTCORBA::Priority** value to another **RTCORBA::Priority** value. In addition to the input priority value, the **ObjectId** of the target object is made available to the inbound transform while the **ObjectId** of the invoking object is made available to the outbound transform. For invocations not made from another CORBA Object; that is, made from an application thread, the outbound transform is still called, with a null value for the **ObjectId** parameter. The transform implementor has the option of leaving the priority unmodified in this case.

A pair of priority transforms, one at each of these two points, may be required to implement a particular priority protocol. For example, to implement a particular variety of distributed priority ceiling protocol, the inbound transform could add a constant offset to the CORBA Priority, and the outbound transform could subtract the same offset from the CORBA Priority, so that the onward invocation is made with the original CORBA Priority.

Priority Transforms are presented to the Real-time ORB as the implementation of the **transform_priority** operation for an instance of the locality constrained CORBA interface type **RTCORBA::PriorityTransform**:

```
// IDL
module RTCORBA {

    native PriorityTransform;

};
```

Language mappings for this IDL native are defined for C, C++, Ada, and Java later in this section.

The **PriorityTransform** is a programming language object rather than a CORBA Object and therefore the normal mechanism for coupling an implementation to the code that uses it (an object reference) doesn't apply. This specification does not prescribe a particular mechanism to achieve this coupling. A Real-time ORB shall provide a mechanism to allow users to install a priority transform.

Note – Possible solutions include: recourse to build/link tools and provision of proprietary interfaces. Other solutions are not precluded.

2.8.1 C Language Binding for PriorityTransform

The use of the **the_priority** parameter is that of an IDL inout parameter.

```
/* C */
CORBA_boolean RTCORBA_PriorityTransform_inbound (
      RTCORBA_Priority* the_priority,
      PortableServer_ObjectId oid );

CORBA_boolean RTCORBA_PriorityTransform_outbound (
      RTCORBA_Priority* the_priority,
      PortableServer_ObjectId oid );
```

2.8.2 C++ Language Binding for PriorityTransform

The use of the **the_priority** parameter is that of an IDL inout parameter.

```
// C++
namespace RTCORBA {

   class PriorityTransform {
      public:
         virtual CORBA::Boolean inbound (
               RTCORBA::Priority &the_priority,
               PortableServer::ObjectId oid );
         virtual CORBA::Boolean outbound (
               RTCORBA::Priority &the_priority,
               PortableServer::ObjectId oid );
   };
};
```

2.8.3 Ada Language binding for PriorityTransform

```
-- Ada
package RTCORBA.PriorityTransform is

   type Object is tagged private;

   procedure Inbound (
      Self           : in Object ;
      The_Priority   : in out RTCORBA.Priority ;
      Oid            : in PortableServer.ObjectId ;
      Returns        : out CORBA.Boolean ) ;

   procedure Outbound (
      Self           : in Object ;
      The_Priority   : in out RTCORBA.Priority ;
      Oid            : in PortableServer.ObjectId ;
      Returns        : out CORBA.Boolean ) ;

end RTCORBA.PriorityTransform ;
```

2.8.4 Java Language binding for PriorityTransform

The use of the **`the_priority`** parameter is that of an IDL inout parameter.

```
// Java
package org.omg.RTCORBA;
   public class PriorityTransform {

      boolean inbound (
         org.omg.CORBA.ShortHolder the_priority,
         org.omg.PortableServer.ObjectId oid
      );
      boolean outbound (
         org.omg.CORBA.ShortHolder the_priority,
         org.omg.PortableServer.ObjectId oid
      );
}
```

2.8.5 Semantics

Rather than raising a CORBA exception upon failure, a boolean return value is used to indicate Transform failure or success. If the priority passed in can be transformed, TRUE is returned and the value is returned as the out parameter. If it cannot be transformed, FALSE is returned and the value of the out parameter is undefined.

Both the inbound and outbound functions must return FALSE when passed a priority that is outside of the valid priority range for a CORBA Priority, 0-32767; that is, a negative value. If the transform doesn't recognize the ObjectId, then it should return FALSE.

Neither inbound nor outbound is obliged to transform all valid CORBA priority values. However, users should note that failure to do so will result in invocation at that priority failing.

When the ORB receives a FALSE return value from a Transform operation that is called as part of the processing of a CORBA invocation, processing of the invocation proceeds no further. An ORB that receives a FALSE return from a transform function shall, if possible, raise an UNKNOWN system exception on the application invocation. Note that it may not be possible to raise an exception to the application if the failure occurs on a call to a Transform operation made on the server side of a oneway invocation.

A Real-time ORB cannot assume that the priority Transform is idempotent. Users should be aware that a Transform that produces different results for the same inputs will make the goal of a schedulable system harder to obtain. Users may choose to implement a priority Transform that changes (through other, user specified interfaces). Users should however note that post-initialization changes to the Transform may well compromise the ORB's ability to deliver a consistently schedulable system.

Note that Priority Transforms may be used with either the Client Propagated or the Server Declared Priority Models. If the Client Propagated model is used, the input priority to the inbound transform shall be the **RTCORBA::Priority** propagated from the client. If the Server Declared model is used, the input priority to the inbound transform will be the **RTCORBA::Priority** assigned to the target object. For the outbound transform, the input priority shall be the derived CORBA Priority.

2.9 Mutex Interface

Real-time CORBA defines the following **Mutex** interface:

```
//IDL
module RTCORBA {

    local interface Mutex    {

        void lock( );
        void unlock( );
        boolean try_lock(in TimeBase::TimeT max_wait);
            // if max_wait = 0 then return immediately
    };

    local interface RTORB {
```

```
        ...
        Mutex create_mutex();
        void destroy_mutex( in Mutex the_mutex );
        ...
    };
};
```

A new **RTCORBA::Mutex** object is obtained using the **create_mutex()** operation of **RTCORBA::RTORB**.

A Mutex object has two states: locked and unlocked. Mutex objects are created in the unlocked state. When the Mutex object is in the unlocked state the first thread to call the **lock()** operation will cause the Mutex object to change to the locked state. Subsequent threads that call the **lock()** operation while the Mutex object is still in the locked state will block until the owner thread unlocks it by calling the **unlock()** operation.

Note – If a Real-time ORB is to run on a shared memory multi-processor, then the Mutex implementation must ensure that the lock operations are atomic to all processors.

The **try_lock()** operation works like the **lock()** operation except that if it does not get the lock within **max_wait** time it returns FALSE. If the **try_lock()** operation does get the lock within the **max_wait** time period, it returns TRUE.

The mutex returned by **create_mutex** must have the same priority inheritance properties as those used by the ORB to protect resources. If a Real-time CORBA implementation offers a choice of priority inheritance protocols, or offers a protocol that requires configuration, the selection or configuration will be controlled through an implementation specific interface.

While a thread executes in a region protected by a mutex object, it can be preempted only by threads whose active native thread priorities are higher than either the ceiling or inherited priority of the mutex object.

Note – The protocol implemented by the Mutex (which priority inheritance or priority ceiling protocol) is not prescribed. Real-time CORBA is intended for a wide range of RTOSs and the choice of protocol will often be predicated on what the RTOS does.

2.10 Threadpools

Real-time CORBA Threadpools are managed using the following IDL types and operations of the Real-time CORBA RTORB interface:

```
//IDL
module RTCORBA {

    // Threadpool types
    typedef unsigned long ThreadpoolId;

    struct ThreadpoolLane {
        Priority        lane_priority;
        unsigned long   static_threads;
        unsigned long   dynamic_threads;
    };

    typedef sequence <ThreadpoolLane> ThreadpoolLanes;

    // Threadpool Policy
    const CORBA::PolicyType THREADPOOL_POLICY_TYPE = 41;

    local interface ThreadpoolPolicy : CORBA::Policy    {
        readonly attribute ThreadpoolId threadpool;
    };

    local interface RTORB {
        ...
        ThreadpoolPolicy create_threadpool_policy (
                        in ThreadpoolId threadpool
        );

        exception InvalidThreadpool {};

        ThreadpoolId create_threadpool (
                        in unsigned long    stacksize,
                        in unsigned long    static_threads,
                        in unsigned long    dynamic_threads,
                        in Priority         default_priority,
                        in boolean          allow_request_buffering,
                        in unsigned long    max_buffered_requests,
                        in unsigned long    max_request_buffer_size );

        ThreadpoolId create_threadpool_with_lanes (
                        in unsigned long        stacksize,
                        in ThreadpoolLanes      lanes,
                        in boolean              allow_borrowing
                        in boolean              allow_request_buffering,
                        in unsigned long        max_buffered_requests,
                        in unsigned long        max_request_buffer_size );

        void destroy_threadpool ( in ThreadpoolId threadpool )
            raises (InvalidThreadpool);

    };
};
```

The **create_threadpool** and **create_threadpool_with_lanes** operations allow two different styles of threadpool to be created: with or without ‘lanes,’ or division into sub-sets of threads at assigned different **RTCORBA::Priority** values. The two styles require some different parameters to be configured, as described in the two following sub-sections.

The configuration of stacksize and request buffering is common to both styles. The stacksize parameter is used to specify the stack size, in bytes, that each thread must have allocated. The configuration of request buffering is described in a sub-section below.

When a threadpool is successfully created, using either operation, a **ThreadpoolId** identifier is returned. This can later be passed to **destroy_threadpool** to destroy the threadpool. If a threadpool cannot be created because the parameters passed in do not specify a valid threadpool configuration, a BAD_PARAM system exception is raised. If a threadpool cannot be created because there are insufficient operating system resources, a NO_RESOURCES system exception is raised.

An instance of the **ThreadpoolPolicy** is created with the **create_threadpool_policy** operation. The attribute of the policy is initialized with the parameter of the same name.

The same threadpool may be associated with a number of different POAs, by using a **ThreadpoolPolicy** containing the same **ThreadpoolId** in each **POA_create**.

2.10.1 Creation of Threadpool without Lanes

To create a threadpool without lanes the following parameters must be configured:

- **static_threads** - specifies the number of threads that will be pre-created and assigned to that threadpool at the time of its creation. A NO_RESOURCES exception is raised if this number of threads cannot be created, in which case no threads are created and no threadpool is created.
- **dynamic_threads** - specifies the number of additional threads that may be created dynamically (individually and upon demand) when the static threads are all in use and an additional thread is required to service an invocation. Whether a dynamically created thread is destroyed as soon as it is not in use, or is retained forever or until some condition is met is an implementation issue.

  If **dynamic_threads** is zero, no additional threads may be dynamically created, and only the static threads are available. In either case, once the maximum number of threads (static plus any dynamic) has been reached, no additional threads will be added to the threadpool. Any additional invocations will wait for one of the existing threads to become available.
- **default_priority** - specifies the CORBA priority that the static threads will be created with. (Dynamic threads may be created directly at the priority they are required to run at to handle the invocation they were created for.)

2.10.2 Creation of Threadpool with Lanes

To create a threadpool with lanes, a lanes parameter must be configured, instead of the **static_threads**, **dynamic_threads**, and **default_priority** parameters. The lanes specify a number of **ThreadpoolLanes**, each of which must have the following parameters specified:

- **lane_priority** - specifies the CORBA Priority that all threads in this lane (both static, and dynamically allocated ones) will run at.
- **static_threads** - specifies the number of threads that will be pre-created, but in this case allocated to this specific lane, rather than the pool as a whole.
- **dynamic_threads** - specifies the number of dynamic threads that may be allocated to this lane. The relationship between static and dynamic threads is the same as in the case of threadpools without lanes: it determines whether and if so how many additional threads may be dynamically created. But in this case the dynamic threads are specific to this lane and are created with the CORBA Priority specified by **lane_priority**.

Additionally, to create a threadpool with lanes, the **allow_borrowing** boolean parameter must be configured to indicate whether the borrowing of threads by one lane from a lower priority lane is permitted or not.

If thread borrowing is permitted, when a lane of a given priority exhausts its maximum number of threads (both static and dynamic) and requires an additional thread to service an additional invocation, it may “borrow” a thread from a lane with a lower priority. The borrowed thread has its CORBA Priority raised to that of the lane that requires it. When the thread is no longer required, its priority is lowered once again to its previous value, and it is returned to the lower priority lane. The thread will be borrowed from the highest priority lane with threads available. If no lower priority lanes have threads available, the lane wishing to borrow a thread must wait until one becomes free (which may be one of its own.)

More generally, for both threadpools with and without lanes, if the priority of a thread is changed while dispatching an invocation, it is restored to its original priority before returning it to the threadpool.

2.10.3 Request Buffering

A Threadpool can be configured to buffer requests. That is when all of the available thread concurrency (static plus dynamic threads) is in use and when any capability to borrow threads has been exhausted then additional requests received are buffered.

If request buffering by the Threadpool is not required, the boolean parameter **allow_request_buffering** is set to FALSE, and the values of the **max_buffered_requests** and **max_request_buffer_size** parameters are disregarded. If request buffering is required, **allow_request_buffering** is set to TRUE, and the **max_buffered_requests** and **max_request_buffer_size** parameters are used as follows:

max_buffered_requests indicates the maximum number of requests that will be buffered by this Threadpool. **max_request_buffer_size** indicates the maximum amount of memory, in bytes, that the buffered requests may use. Both properties of a Threadpool are evaluated to determine the number of requests that will be buffered. An incoming request is not buffered by the Threadpool if either the number of buffered requests reaches **max_buffered_requests** or buffering the request would take the total amount of buffer memory used past **max_request_buffer_size**.

Either parameter may be set to zero, to indicate that that property is to be taken as unbounded. Hence, just the number of requests or just the maximum amount of buffer memory can be used to limit the buffering.

If, at the time of Threadpool creation, the ORB can determine that it does not have the resources to support the requested configuration, the Threadpool creation operation will fail with a NO_RESOURCES system exception.

2.10.4 Scope of ThreadpoolPolicy

The **ThreadpoolPolicy** may be applied at the POA and ORB level. A POA may only be associated with one threadpool, hence only one **ThreadpoolPolicy** should be included in the **PolicyList** specified at POA creation.

A **ThreadpoolPolicy** may be applied at the ORB level, by using the **set_policy_overrides** operation of the CORBA **PolicyManager** interface. When the policy is applied at the ORB level, it assigns the indicated threadpool as the default threadpool to use in the subsequent creation of POAs, until the default is again changed. The default is used if a **ThreadpoolPolicy** is not specified in the policies used at the time of POA creation.

2.11 Implicit and Explicit Binding

Real-time CORBA makes use of the **CORBA::Object::validate_connection** operation to allow client applications to control when a binding is made on an object reference.

Note – validate_connection and the definition of binding that it uses are defined in the *Common Object Request Broker Architecture (CORBA)* specification, CORBA Messaging chapter.

Using **validate_connection** on a currently unbound object reference causes binding to occur. Real-time CORBA refers to the use of **validate_connection** to force a binding to be made as ‘explicit binding.’ If an object reference is not explicitly bound, binding will occur at an ORB specific time, which may be as late as the time of the first invocation upon that object reference. This is referred to as ‘implicit binding,’ and is the default CORBA behavior unless an explicit bind is performed.

Real-time applications may wish to use explicit binding to force any binding related overhead (including the passing of messages between the client and server) to be incurred ahead of the first invocation on an object reference. This can improve the

performance and predictability of the first invocation, and hence the predictability of the application as a whole. The explicit bind may, for example, be performed during system initialization.

Once an explicit binding has been set up, via **validate_connection**, it is possible that the underlying transport connection (or other associated resources) may fail or may be reclaimed by the ORB. Rather than mandate that this shall not happen, it is left as a Quality of Implementation issue as to what guarantees of enduring availability an explicit binding provides.

The client-side applicable Real-time CORBA policies are applied to a binding in the same way as any other client-side applicable CORBA policies: using the **set_policy_overrides** operations at the ORB, Current, or Object scope (as defined in the CORBA QoS Policy Framework.)

The client-side applicable Real-time CORBA policies have the same effect whether they are applied to an implicit or explicit bind.

2.12 Priority Banded Connections

Priority banded connections are administered through the use of the Real-time CORBA **PriorityBandedConnectionsPolicy** Policy type:

```
// IDL
module RTCORBA {

    struct PriorityBand {
        Priority low;
        Priority high;
    };

    typedef sequence <PriorityBand> PriorityBands;

    // PriorityBandedConnectionPolicy
    const CORBA::PolicyType
        PRIORITY_BANDED_CONNECTION_POLICY_TYPE = 45;

    local interface PriorityBandedConnectionPolicy : CORBA::Policy {

        readonly attribute PriorityBands priority_bands;

    };
```

```
    local interface RTORB {
        ...
        PriorityBandedConnectionPolicy
                create_priority_banded_connection_policy (
                        in PriorityBands priority_bands
        );
    };

};
```

An instance of the **PriorityBandedConnectionPolicy** is created with the **create_priority_banded_connection_policy** operation. The attribute of the policy is initialized with the parameter of the same name.

The **PriorityBands** attribute of the policy may be assigned any number of **PriorityBands**. **PriorityBands** that cover a single priority (by having the same priority for their low and high values) may be mixed with those covering ranges of priorities. No priority may be covered more than once. The complete set of priorities covered by the bands do not have to form one contiguous range, nor do they have to cover all CORBA Priorities. If no bands are provided, then a single connection will be established.

Once the binding has been successfully made, an attempt to make an invocation with a Real-time CORBA Priority, which is not covered by one of the bands will fail. The ORB shall raise a NO_RESOURCES system exception (with a Standard Minor Exception Code of 2). Hence, a policy specifying only one band can be used to restrict a client's invocations to a range of priorities.

Note that the origin of the Real-time CORBA Priority value that is used to select which banded connection to use depends on the Priority Model of the target object. When invoking on an Object that is using the Client Propagated Priority Model, the client-set Real-time CORBA Priority is used to choose the band. Whereas, invoking on an Object that is using the Server Declared Priority Model, the server priority is used, as published in the IOR.

2.12.1 Scope of PriorityBandedConnectionPolicy

The **PriorityBandedConnectionPolicy** is applied on the client-side only, at the time of binding to a CORBA Object. However, the policy may be set from the client or server side. On the server, it may be applied at the time of POA creation, in which case the policy is client-exposed and is propagated from the server to the client in interoperable Object References. It is propagated in a **PolicyValue** in a TAG_POLICIES Profile Component, as specified by the CORBA QoS Policy Framework.

When an instance of **PriorityBandedConnectionPolicy** is propagated, the **PolicyValue**'s ptype has the value **PRIORITY_BANDED_CONNECTION_POLICY_TYPE** and the pvalue is a CDR encapsulation containing an **RTCORBA::PriorityBands** type, which is a sequence of

instances of **RTCORBA::PriorityBand**. Each **RTCORBA::PriorityBand** is in turn represented by a pair of **RTCORBA::Priority** values, which represent the low and high values for that band.

If the **PriorityBandedConnectionPolicy** is set on both the server and client-side, an attempt to bind will fail with an INV_POLICY system exception. The client application may use **validate_connection** to establish that this was the cause of binding failure and may set the value of its copy of the policy to an empty **PriorityBands** and attempt to rebind using just the configuration from the server-provided copy of the policy.

2.12.2 Binding of Priority Banded Connection

Whether bands are configured from the client or server-side, the banded connection is always initiated from the client-side.

In order to allow the server-side ORB to identify the priority band that each connection is associated with, information on that connection's band range is passed with first request on each banded connection. This is done by means of an **RTCorbaPriorityRange** service context:

```
// IDL
module IOP {

    const ServiceId RTCorbaPriorityRange = 11;

};
```

The **context_data** contains the CDR encapsulation of two **RTCORBA::Priority** values (two short types.) The first indicates the lowest priority and the second the highest priority in the priority band handled by the connection.

Once a priority band has been associated with a connection it cannot be reconfigured during the life-time of the connection. If an ORB receives a second, or subsequent, **RTCorbaPriorityRange** service context containing a different priority band definition, then it shall raise a BAD_INV_ORDER system exception (with a Standard Minor Exception Code of 18). If the priority band is the same as the connection's configuration, then processing shall proceed.

In case of an explicit bind (via **validate_connection**), this service context is passed on a request message for a '**_bind_priority_band**' implicit operation. This implicit operation is defined for Real-time CORBA only at this time. It is possible that non-Real-time ORB might receive such a request message. If so it is anticipated (but not prescribed) that it will reply with a BAD_OPERATION system exception with standard minor code 2. A future version of IIOP should formalize Real-time CORBA's use of the '**_bind_priority_band**' operation name in a GIOP Request message. Note that there is no API exposed for this implicit operation (unlike, for example, '**_is_a**').

When sending a '**_bind_priority_band**' request, a Real-time ORB shall marshall no parameters and the object key of the object being bound to shall be used as the request 'target.' The request shall be handled by the ORB, no servant implementation of this implicit operation will be invoked.

When a Real-time-ORB receives a **_bind_priority_band** Request it should allocate resources to the connection and configure those resources appropriately to the priority band indicated in the **ServiceContext**. Having done this the ORB shall send a "SUCCESS" Reply message. If the priority band passed is not well-formed; that is, it contains a negative number or the first value is higher than the second, then the ORB shall raise a BAD_PARAM system exception. If either of the priorities cannot be mapped onto native thread priorities; that is, to-native returns FALSE, then the ORB shall raise a DATA_CONVERSION system exception (with a Standard Minor Exception Code of 2). If the priority band is inconsistent with the ORB's priority configuration, then the ORB shall raise an INV_POLICY system exception. If the server-side ORB cannot configure resources to support a well-formed band specification, then a NO_RESOURCES exception shall be returned.

A **_bind_priority_band** request message is sent on the connection for each band and must complete successfully; that is, yield a SUCCESS Reply message for all connections, before **validate_connection** returns success. If any one **_bind_priority_band** fails, then the entire banded connection binding fails. In this way, **validate_connection** sets up all the banded connections at time of binding.

If the service context is omitted on a **_bind_priority_band** request message, then the ORB shall raise a BAD_PARAM system exception.

A **bind_priority_band** is not performed in the case of an implicit bind, as it occurs at a time when a request is about to be sent on the connection representing the priority band that covers the current invocation priority. There is no point delaying the application request. Instead, the '**RTCorbaPriorityRange**' service context is passed on this first invocation request.

Thus, the implicit binding of a banded connection has the behavior that each band connection is only set up the first time an invocation is made from the client with an invocation priority in that band. This behavior offers consistency: the first invocation made on each band incurs any latency and predictability cost associated with binding. If no invocations are ever made in the priority range of a given band, its connection will never be established.

2.13 PrivateConnectionPolicy

This policy allows a client to obtain a private transport connection, which will not be multiplexed (shared) with other client-server object connections.

```
// IDL
module RTCORBA {

    // Private Connection Policy

    const CORBA::PolicyType
            PRIVATE_CONNECTION_POLICY_TYPE = 44;

    local interface PrivateConnectionPolicy : CORBA::Policy {};

    local interface RTORB {
        ...
        PrivateConnectionPolicy create_private_connection_policy (
        );
    };

};
```

An instance of the **PrivateConnectionPolicy** is created with the **create_private_connection_policy** operation. The policy has no attributes.

Note that it is not possible to explicitly request a multiplexed connection. Whether multiplexing is appropriate or not is a protocol specific issue, and hence an ORB implementation issue. By not requesting a private connection the application indicates to the ORB that a multiplexed connection would be acceptable. It is up to the ORB implementation to make use of this indication.

2.14 Invocation Timeout

Real-time CORBA uses the existing CORBA timeout policy, **Messaging::RelativeRoundtripTimeoutPolicy**, to allow a timeout to be set for the receipt of a reply to an invocation. The policy is used where it is set, to set a timeout in the client ORB. If a timeout expires, the server is not informed. Real-time CORBA does not require the policy to be propagated with the invocation, which the **RelativeRoundtripTimeoutPolicy** specification allows in support of message routers.

Note – The **RelativeRoundtripTimeoutPolicy** is specified in the *Common Object Request Broker Architecture (CORBA)* specification, CORBA Messaging chapter.

2.15 Protocol Configuration

Real-time CORBA uses two Policy types, based on a common protocol configuration framework, to enable the selection and configuration of protocols on the server and client side of the ORB.

2.15.1 ServerProtocolPolicy

The **ServerProtocolPolicy** policy type is used to select and configure communication protocols on the server-side of Real-time CORBA ORBs.

```
// IDL
module RTCORBA {

    local interface ProtocolProperties {};

    struct Protocol {
        IOP::ProfileId          protocol_type;
        ProtocolProperties      orb_protocol_properties;
        ProtocolProperties      transport_protocol_properties;
    };

    typedef sequence <Protocol> ProtocolList;

    // Server Protocol Policy
    const CORBA::PolicyType SERVER_PROTOCOL_POLICY_TYPE = 42;

    local interface ServerProtocolPolicy : CORBA::Policy {
        readonly attribute ProtocolList protocols;

    };

    local interface RTORB {
        ...
        ServerProtocolPolicy create_server_protocol_policy (
                            in ProtocolList protocols
        );
    };

};
```

An instance of the **ServerProtocolPolicy** is created with the **create_server_protocol_policy** operation. The attribute of the policy is initialized with the parameter of the same name.

A **ServerProtocolPolicy** allows any number of protocols to be specified and, optionally, configured at the same time. The order of the Protocols in the **ProtocolList** indicates the order of preference for the use of the different protocols. Information regarding the protocols must be placed into IORs in that order, and the client should take that order as the default order of preference for choice of protocol to bind to the object via.

The type of protocol is indicated by an **IOP::ProfileId** (from the specification of the IOR), which is an unsigned long. This means that a protocol is defined as a specific pairing of an ORB protocol (such as GIOP) and a transport protocol (such as TCP.)

Hence IIOP would be selected, rather than GIOP plus TCP being selected separately. IIOP in particular is represented by the value **TAG_INTERNET_IIOP** (or the value 0, that it is defined as.)

A Protocol type contains a **ProfileId** plus two **ProtocolProperties**, one each for the ORB protocol and the transport protocol.

The properties are provided to allow the configuration of protocol specific configurable parameters. Specific protocols have their own protocol configuration interface that inherits from the **RTCORBA::ProtocolProperties** interface. A nil reference for either **ProtocolProperties** indicates that the default configuration for that protocol should be used. (Each protocol will have an implementation specific default configuration, that may be overridden by applying the **ServerProtocolPolicy** at ORB scope. See the Policy Scope sub-section, below.)

```
//IDL
module RTCORBA {

    local interface TCPProtocolProperties : ProtocolProperties {
        attribute long    send_buffer_size;
        attribute long    recv_buffer_size;
        attribute boolean  keep_alive;
        attribute boolean  dont_route;
        attribute boolean  no_delay;
    };
    local interface RTORB {
        ...
        TCPProtocolProperties create_tcp_protocol_properties (
                            in long send_buffer_size,
                            in long recv_buffer_size,
                            in boolean keep_alive,
                            in boolean dont_route,
                            in boolean no_delay );
    };

};
```

TCP is the only protocol that RT CORBA specifies a **ProtocolProperties** interface for. Instances of **TCPProtocolProperties** may be created by using the **create_tcp_protocol_properties** of RTORB. A **ProtocolProperties** interface is not specified for GIOP, as GIOP currently has no configurable properties. A **GIOPProtocolProperties** type will be defined in the future, if any configurable properties are added to GIOP.

ProtocolProperties should be defined for any other protocols usable with an RT CORBA implementation, but unless they are standardized in an OMG specification their name and contents will be implementation specific. **ProtocolProperties** for other protocols may be standardized in the future, and a **ProtocolProperties** interface should be specified in the standardization of any other protocol, if it is to be usable in a portable way with RT CORBA.

2.15.2 Scope of ServerProtocolPolicy

Applying a **ServerProtocolPolicy** to the creation of a POA controls the protocols that references created by that POA will support (and their configuration if non- nil **ProtocolProperties** are given.) If no **ServerProtocolPolicy** is given at POA creation, the POA will support the default protocols associated with the ORB that created it. (Note that supplying a **ServerProtocolPolicy** overrides, rather than supplementing or sub-setting, the default selection of protocols associated with the ORB.)

The ORB's default protocols, and their order of preference, are implementation specific. The default may be overridden by applying a **ServerProtocolPolicy** at the ORB level. As a consequence, portable applications must override this Policy (and all other defaults) to ensure the same behavior between ORB implementations.

Only one **ServerProtocolPolicy** should be included in a given **PolicyList**, and including more than one will result in an INV_POLICY system exception being raised.

2.15.3 ClientProtocolPolicy

The **ClientProtocolPolicy** policy type is used to configure the selection and configuration of communication protocols on the client-side of Real-time CORBA ORBs. It is defined in terms of the same **RTCORBA::ProtocolProperties** type as the **ServerProtocolPolicy**:

```
// IDL
module RTCORBA {

    // Client Protocol Policy
    const CORBA::PolicyType CLIENT_PROTOCOL_POLICY_TYPE = 43;

    local interface ClientProtocolPolicy : CORBA::Policy {

        readonly attribute ProtocolList protocols;

    };

    local interface RTORB {
        ...
        ClientProtocolPolicy create_client_protocol_policy (
                        in ProtocolList protocols
        );
    };

};
```

An instance of the **ClientProtocolPolicy** is created with the **create_client_protocol_policy** operation. The attribute of the policy is initialized with the parameter of the same name.

When applied to a bind (implicit or explicit), the **ClientProtocolPolicy** indicates the protocols that may be used to make a connection to the specified object, in order of preference. If the ORB fails to make a connection because none of the protocols is available on the client ORB, an INV_POLICY system exception is raised. If one or more of the protocols is available but the ORB still fails to make a connection, a COMM_FAILURE system exception is raised. In both cases no binding is made.

If it is necessary to know which protocol a binding was successfully made via, a single protocol should be passed into each of a succession of explicit binds until one of them is successful.

If no **ClientProtocolPolicy** is provided, then the protocol selection is made by the ORB based on the target object's available protocols, as described in its IOR, and the protocols supported by the client ORB.

2.15.4 Scope of ClientProtocolPolicy

The **ClientProtocolPolicy** is applied on the client-side, at the time of binding to an Object Reference. However, the policy may be set on either the client or server-side. On the server-side, it may be applied at the time of POA creation, in which case the policy is client-exposed and is propagated from the server to the client in interoperable Object References. It is propagated in a **PolicyValue** in a **TAG_POLICIES** Profile Component, as specified by the CORBA QoS Policy Framework.

When an instance of **ClientProtocolPolicy** is propagated, the **PolicyValue**'s ptype has the value **CLIENT_PROTOCOL_POLICY_TYPE** and the pvalue is a CDR encapsulation containing an **RTCORBA::ProtocolList**, which is a sequence of instances of **RTCORBA::Protocol**. Each **RTCORBA::Protocol** is in turn represented by an **IOP::ProfileId** and two **RTCORBA::ProtocolProperties** representing the ORB and transport **ProtocolProperties**.

The on the wire representation of each **ProtocolProperties** type is protocol specific. The representation of the **TCPProtocolProperties** type is the CDR encoding of two longs followed by three booleans, to represent the **send_buffer_size**, **recv_buffer_size**, **keep_alive**, **dont_route**, and **no_delay** attributes respectively.

If the **ClientProtocolPolicy** is set on both the server and client-side, an attempt to bind will fail with an INV_POLICY system exception. The client application may use **validate_connection** to establish that this was the cause of binding failure and may set the value of its copy of the policy to an empty **ProtocolList** and attempt to re-bind using just the configuration from the server-provided copy of the policy.

2.15.5 Protocol Configuration Semantics

Note that the above API only allows policies to be set at POA creation time on the server-side, and object bind time on the client-side. No API is defined to allow (re)configuration of any policy after these times.

The protocol configuration selected at the time of POA creation is used to determine the server-side configuration that is to be used by the protocol in question for all connections from clients to objects that have references created by that POA.

However, as the configuration semantics of a protocol (such as whether a particular property can be configured on a per-connection basis or only globally for that instance of the protocol) are protocol specific, the exact semantics of protocol configuration via **ProtocolProperties** are not specified by Real-time CORBA, and must be specified on a per-protocol basis.

If a protocol offers a configurable property that can only be configured at some scope wider than that of the individual POA (say at the scope of the ORB instance), it can choose either to:

- Change that property at the wider scope when a different value is requested for the creation of a new POA. This will ensure that the new POA gets the configuration requested, but will also affect the configuration of new and possibly existing connections made to other CORBA Objects via the same protocol. The exact scope and semantics of the property change must be given as part of the documentation of the **ProtocolProperties** interface for that protocol.
- Not change the property, but instead raise an INV_POLICY exception and fail to create the new POA. In this way, the original value of the property is preserved for the existing references that use it. Once again, this behavior must be covered in the documentation of the **ProtocolProperties** interface for that protocol.

Which of the two strategies a protocol uses is an implementation issue.

2.16 Consolidated IDL

```
// IDL
module IOP {
    const ServiceId      RTCorbaPriority = 10;
    const ServiceId      RTCorbaPriorityRange = 11;
};


//File: RTCORBA.idl
#ifndef _RT_CORBA_IDL_
#define _RT_CORBA_IDL_
#ifdef _PRE_3_0_COMPILER_
#pragma prefix "omg.org"

#include <orb.idl>
#include <iop.idl>
#include <TimeBase.idl>
#else
import ::CORBA;
import ::IOP;
import ::TimeBase;
#endif
```

```
// IDL
module RTCORBA {
    typedef short NativePriority;

    typedef short Priority;

    const Priority minPriority = 0;
    const Priority maxPriority = 32767;

    native PriorityMapping;
    native PriorityTransform;

    // Threadpool types
    typedef unsigned long ThreadpoolId;

    struct ThreadpoolLane {
        Priority        lane_priority;
        unsigned long   static_threads;
        unsigned long   dynamic_threads;
    };

    typedef sequence <ThreadpoolLane> ThreadpoolLanes;

    // Priority Model Policy
    const CORBA::PolicyType PRIORITY_MODEL_POLICY_TYPE = 40;

    enum PriorityModel {
        CLIENT_PROPAGATED,
        SERVER_DECLARED
    };

    local interface PriorityModelPolicy : CORBA::Policy {

        readonly attribute PriorityModel priority_model;
        readonly attribute Priority server_priority;

    };

    // Threadpool Policy
    const CORBA::PolicyType THREADPOOL_POLICY_TYPE = 41;

    local interface ThreadpoolPolicy : CORBA::Policy    {
        readonly attribute ThreadpoolId threadpool;
    };

    local interface ProtocolProperties {};
```

```
struct Protocol {
    IOP::ProfileId        protocol_type;
    ProtocolProperties orb_protocol_properties;
    ProtocolProperties transport_protocol_properties;
};

typedef sequence <Protocol> ProtocolList;

// Server Protocol Policy
const CORBA::PolicyType SERVER_PROTOCOL_POLICY_TYPE = 42;

local interface ServerProtocolPolicy : CORBA::Policy    {
    readonly attribute ProtocolList protocols;
};

// Client Protocol Policy
const CORBA::PolicyType CLIENT_PROTOCOL_POLICY_TYPE = 43;

local interface ClientProtocolPolicy : CORBA::Policy {
    readonly attribute ProtocolList protocols;
};

// Private Connection Policy
const CORBA::PolicyType PRIVATE_CONNECTION_POLICY_TYPE = 44;

local interface PrivateConnectionPolicy : CORBA::Policy {};

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

struct PriorityBand {
    Priority low;
    Priority high;
};

typedef sequence <PriorityBand> PriorityBands;

// PriorityBandedConnectionPolicy
const CORBA::PolicyType
    PRIORITY_BANDED_CONNECTION_POLICY_TYPE = 45;

local interface PriorityBandedConnectionPolicy : CORBA::Policy {
    readonly attribute PriorityBands priority_bands;
};
```

```
local interface Current : CORBA::Current {
    attribute Priority the_priority;
};

local interface Mutex    {

    void lock( );
    void unlock( );
    boolean try_lock ( in TimeBase::TimeT max_wait );
    // if max_wait = 0 then return immediately
};


local interface RTORB {

    Mutex create_mutex( );
    void destroy_mutex( in Mutex the_mutex );

    exception InvalidThreadpool {};

    ThreadpoolId create_threadpool (
                    in unsigned long   stacksize,
                    in unsigned long   static_threads,
                    in unsigned long   dynamic_threads,
                    in Priority        default_priority,
                    in boolean         allow_request_buffering,
                    in unsigned long   max_buffered_requests,
                    in unsigned long   max_request_buffer_size );

    ThreadpoolId create_threadpool_with_lanes (
                    in unsigned long       stacksize,
                    in ThreadpoolLanes     lanes,
                    in boolean             allow_borrowing
                    in boolean             allow_request_buffering,
                    in unsigned long       max_buffered_requests,
                    in unsigned long       max_request_buffer_size );

    void destroy_threadpool ( in ThreadpoolId threadpool )
        raises (InvalidThreadpool);

    PriorityModelPolicy create_priority_model_policy (
                    in PriorityModel priority_model,
                    in Priority server_priority
    );
    ThreadpoolPolicy create_threadpool_policy (
                    in ThreadpoolId threadpool
    );
```

```
            PriorityBandedConnectionPolicy
                    create_priority_banded_connection_policy (
                            in PriorityBands priority_bands
            );
            ServerProtocolPolicy create_server_protocol_policy (
                            in ProtocolList protocols
            );
            ClientProtocolPolicy create_client_protocol_policy (
                            in ProtocolList protocols
            );
            PrivateConnectionPolicy create_private_connection_policy (
            );

            TCPProtocolProperties create_tcp_protocol_properties (
                            in long send_buffer_size,
                            in long recv_buffer_size,
                            in boolean keep_alive,
                            in boolean dont_route,
                            in boolean no_delay );
        };

        }; // End interface RTORB

};  // End module RTCORBA
#endif // _RT_CORBA_IDL_
```

```
//File: RTPortableServer.idl
#ifndef _RT_PORTABLE_SERVER_IDL_
#define _RT_PORTABLE_SERVER_IDL_
#ifdef _PRE_3_0_COMPILER_
#pragma prefix "omg.org"

#include <PortableServer.idl>
#include <RTCORBA.idl>
#else
import ::PortableServer;
import ::RTCORBA;
#endif

// IDL
module RTPortableServer {

#ifndef _PRE_3_0_COMPILER_
    typeprefix RTCORBA "omg.org";
#endif

    local interface POA : PortableServer::POA {
```

```
            Object create_reference_with_priority (
                                in CORBA::RepositoryId intf,
                                in RTCORBA::Priority priority )
                raises ( WrongPolicy );

            Object create_reference_with_id_and_priority (
                                in PortableServer::ObjectId oid,
                                in CORBA::RepositoryId intf,
                                in RTCORBA::Priority priority )
                raises ( WrongPolicy );

            PortableServer::ObjectId activate_object_with_priority (
                                in PortableServer::Servant p_servant,
                                in RTCORBA::Priority priority )
                raises ( ServantAlreadyActive, WrongPolicy );

            void activate_object_with_id_and_priority (
                                in PortableServer::ObjectId oid,
                                in PortableServer::Servant p_servant,
                                in RTCORBA::Priority priority )
                raises ( ServantAlreadyActive,
                                ObjectAlreadyActive, WrongPolicy );
        };

    };
    #endif // _RT_PORTABLE_SERVER_IDL_
```

Dynamic Scheduling 3

Contents

This chapter contains the following topics.

Section I - Overview and Rationale

3.1 Overview

3.1.1 Dynamic Scheduling

In real-time, we distinguish between two types of distributed systems based on how the system is used, and its impact on the underlying infrastructure. There are *static* and *dynamic* distributed systems.

Static distributed systems are those where the processing load on the system is within known bounds such that *a priori* analysis can be performed. This means that the set of applications that the system could be running is known in advance, and that the workload that will be imposed on each application can be predicted within a known bound. Such systems often have a limited number of application configurations that can be executed, sometimes referred to as system modes. In these systems, a schedule for the execution of applications can be worked out in advance for each system mode, with a bounded amount of variation. As a result, the underlying infrastructure (operating system and middleware) need only be able to support executing that schedule.

One common approach to static systems is the use of operating system priorities to manage deadlines. Offline analysis is performed to map different application temporal requirements (such as frequency of execution) onto the available priorities. If the underlying infrastructure always respects these priorities, including preempting low priority threads when higher priority threads become eligible to run, and providing priority inheritance, this is sufficient.

Dynamic distributed systems, on the other hand, do not have a sufficiently predictable workload to allow this approach. It may be that the set of applications is either too large or not known in advance, the processing requirements for an application is too variable to be pre-planned, the arrival time of the inputs is too variable, or some other source of variability. For these types of systems, the underlying infrastructure must be able to satisfy real-time requirements in a dynamically changing environment.

This chapter is focused on systems in which the discipline for scheduling CORBA ORB and application threads (e.g., highest priority first, or earliest deadline first, or least laxity first) may be chosen by the application or system designers; and the scheduling input values needed by a scheduler to control execution (called *scheduling parameter elements* in this document) for that scheduling discipline (e.g., priority deadline, expected execution time) may be changed by the application dynamically (i.e., at any time). In contrast, Real-time CORBA 1.0 (ORBOS/99-02-12 with ORBOS/99-03-29) is focused on fixed priority systems.

3.1.2 Distributable Thread

This specification replaces the term and concept of an *activity* that appeared as a design and analysis suggestion in Real-time CORBA 1.0 with a specification for an end-to-end schedulable entity termed *distributable thread*. Additionally, this specification's introduction of the entity termed *scheduling segment* completes the replacement of the *activity* concept from Real-time CORBA 1.0.

3.2 Rationale

Dynamic scheduling is widely employed in real-time and distributed real-time computing systems. This specification extends Real-time CORBA 1.0 to encompass these dynamic systems as well as static systems.

In most real-time systems, especially distributed systems, cost-effectiveness demands that the computing system employ as much application-specific knowledge about the application and its execution environment as feasible. Much of this knowledge can be best captured in the scheduling discipline. This specification allows such application-specific scheduling disciplines to be implemented by a pluggable scheduler.

An end-to-end execution model is essential to achieving end-to-end predictability of timeliness in a distributed real-time computing system. This is especially important in dynamically scheduled systems. The end-to-end execution model may be provided according to a formal standard specification (as herein), or as an ad hoc, custom-made creation by multiple different application programmers.

3.3 Notional Scheduling Service Architecture

This specification is based on a scheduling service architecture for a hypothetical or notional scheduling service plug-in. The specification does not require that scheduling service implementations conform to this notional architecture. It is presented only to provide an aid to understanding and describing the specification of a generic scheduling service framework.

The application interacts with the scheduler, passing information about its (the application's) scheduling needs and its predicted use of system resources. The scheduler is responsible for determining how best to meet the schedule given that resource usage. The scheduler will use one or more scheduling disciplines, such as Earliest Deadline First or Maximum Urgency First, to achieve this goal.

The application must also interact with the scheduler whenever there is a significant change in its scheduling needs or its predicted resource usage. In addition, the application must ensure that the scheduler is able to run as often as it needs to in order to maintain the schedule – the application should not, for example, disable interrupts or pre-emption for long periods of time, or create high priority threads that the scheduler does not know about. If there are insufficient naturally occurring interaction points, the application must include some additional interactions with the scheduler just to guarantee that overruns and other errors can be detected in a timely way.

In a CORBA environment, the application can take an action (for example, making a CORBA request) that could impact the schedule. In the notional architecture, the ORB is responsible for interacting with the scheduler at these points, so that the scheduler can take into account the transitioning of control from one processing node to another. Thus, a set of specific ORB-scheduler interfaces are defined.

3.4 Goals of this Specification

Dynamic Scheduling generalizes the Real-time CORBA Base Architecture to meet the requirements of a much greater segment of the real-time computing field. There are three major generalizations:

- Any scheduling discipline may be employed.
- The scheduling parameter elements associated with the chosen discipline may be changed at any time during execution.
- The schedulable entity is a *distributable thread* that may span node boundaries, carrying its scheduling context among scheduler instances on those nodes.

While the Real-time CORBA 1.0 Scheduling Service interfaces have been replaced, this specification is backward compatible with the semantics of the Scheduling Service defined in the Real-time CORBA 1.0 specification. Compatible implementations of both specifications may be used in different ORB instances within the same system. Not all features of this specification can be used in such mixed systems.

This specification imposes no requirements on base real-time operating systems, other than the conventional ability to dispatch threads in a pre-emptive fashion. This specification imposes no additional constraints on the real-time operating system beyond those in Real-time CORBA 1.0.

3.5 Scope

This specification adds interfaces for a small set of well known scheduling disciplines to CORBA as optional compliance points. This specification does not attempt to provide all the interfaces necessary for interoperability of dynamically scheduled applications and schedulers in heterogeneous systems. Rather, this specification provides a framework upon which schedulers can be built and lays the foundation for future full interoperability, and provides sufficient interfaces for applications to be built using the set of included scheduler disciplines.

This specification defines a set of ORB/scheduler interfaces that will allow the development of portable (i.e., ORB implementation independent) schedulers. For the defined disciplines, this specification also specifies interfaces that will allow the development of portable (i.e., ORB and scheduler implementation independent) applications. The specification of portable application interfaces for other scheduling disciplines is left to future revisions.

Note that most scheduler implementations will extensively utilize features of the underlying operating system, and in some cases the networking software. This aspect of scheduler implementation is outside of the scope of this specification. Therefore, this specification does not provide the portability of schedulers except with respect to ORB interactions.

This specification does not provide interoperability between scheduling disciplines and thus not between different scheduler implementations. A scheduling framework is provided and the mechanism used for passing information between scheduler instances is provided via GIOP service contexts. However, the format and content of the information passed in the GIOP service contexts are not specified. On-the-wire interoperability between scheduling disciplines and the corresponding scheduler implementations is left to future specifications.

This specification provides an abstraction for distributed real-time programming (the *distributable thread*). This specification does not attempt to address more advanced issues such as fault tolerance, propagation of system information and control along the path of a distributable thread, etc. These facilities may be provided in a subsequent revision of this specification.

Section II - Concepts

3.6 Sequencing: Scheduling and Dispatching

Usually multiple execution entities (hereafter referred to as “threads”) contend for one or more exclusively accessed resources – notably processor cycles, but also others, both physical (e.g., communication paths) and logical (e.g., synchronizers). This contention must be resolved into a sequence of resource accesses (e.g., thread executions). In general, contention for all shared resources should be resolved in a consistent manner, although this is not yet common practice. For example, processors may be allocated by priority, networks by first come first served, locks by serializability, disks by head movement distance, etc. All resource contention can be resolved by one of two sequencing means: either scheduling or dispatching.

Thread *scheduling* is deciding in what order they all will execute. Each time thread scheduling is performed, a sequence is established – a schedule – for all threads ready at that time. Scheduling is performed *statically* (prior to execution time), by a person or a program, or *dynamically* (at execution time) by a user or the system software.

Thread *dispatching* is granting resource access (e.g., running the currently most eligible thread). When scheduling is employed, dispatching occurs in schedule order.

Thread scheduling is not always necessary nor computationally feasible – dispatching alone may be sufficient.

Moreover, some actions are never threads and thus not schedulable – most commonly, interrupt service routines and certain OS services, which execute either when invoked or automatically as needed (other OS services are scheduled in concert with application entities).

Dispatching, when scheduling is not employed, establishes a thread resource access – e.g., execution sequence – one thread or non-schedulable action at a time.

The execution sequence may change at a sequencing point (either a *scheduling point* or a *dispatching point*), such as when a thread becomes ready or blocked, or a thread contends for a resource, or a thread time constraint is violated.

Contention for execution (and all other sequentially shared physical and logical resources) generally should be resolved according to an application-specific *sequencing optimality criterion* that seeks maximal usefulness to the system. In real-time systems, that usefulness is based primarily on (but not limited to) timeliness and predictability of timeliness. (Other factors, not related to real-time, commonly found in sequencing criteria include relative importance, precedence constraints, resource ownership, etc.).

Sequencing optimality criteria are what define *timeliness* for a given system or application. Consequently, they also distinguish hard and soft real-time. *Hard real-time* has a single timeliness factor in its sequencing optimality criterion: always meet all hard deadlines. *Soft real-time* includes all other possible timeliness factors in sequencing (usually scheduling) optimality criteria – very common examples are "minimize mean weighted tardiness," "minimize the number of missed deadlines according to importance," and "minimize maximum tardiness."

Informally, a property is *predictable* to the degree that it is known in advance. One end point of the predictability scale is *determinism*, in the sense that the property is known exactly in advance. The other end point of the predictability scale can be characterized as maximum entropy, in the sense that nothing at all is known in advance about the property. In stochastic real-time systems (which include hard real-time systems as a special case), one well-defined way to measure predictability is coefficient of variation C_v, which is defined as variance/mean^2. The deterministic distribution, $C_v = 0$, and the extreme mixture of exponentials distribution is an example of a maximally non-deterministic property whose $C_v = \infty$.

In every real-time system, timeliness of each application and system action is somewhere on this predictability scale. Hard real-time systems have deterministic timeliness in the sense that they always meet all of their hard deadlines. Soft real-time systems have non-deterministic timeliness – e.g., characterized stochastically, such as minimizing either mean or maximum tardiness.

Given a sequencing optimality criterion, a *sequencing discipline* is selected or devised to satisfy it. There are a great many widely used sequencing disciplines; common examples in real-time computing systems include highest priority first (or just "priority"), earliest deadline first (EDF), and least laxity first (LLF). There may be more than one discipline that satisfies a given criterion – e.g., the hard real-time

criterion is satisfied by: the EDF and LLF disciplines (under specific conditions), among others; or appropriate assignment and manipulation of priorities. Conversely, a specific discipline may be suitable for different criteria: EDF satisfies the hard real-time criterion, and also satisfies the soft real-time criterion “minimize maximum tardiness” (among others); priorities can be used to satisfy either the hard or various soft real-time criteria.

When scheduling is employed, the sequencing discipline is usually called a *scheduling discipline*, and when only dispatching is employed, the discipline is usually called a *dispatching rule*.

A *sequencing algorithm* implements a sequencing discipline. In general, a discipline can be implemented by many different possible algorithms.

This specification uses the term *scheduling* to include the case when scheduling (and thus dispatching in schedule order) is employed, and the case when only dispatching is employed, because both of those cases involve selecting a sequencing optimality criterion and a corresponding discipline and algorithm.

3.7 Well Known Scheduling Disciplines

There are many widely used scheduling disciplines, but real-time computing theory and practice are focused on a small number of them, some of which are summarized below. The constructs in the IDL will become clear later in this specification.

3.7.1 Fixed Priority Scheduling

The fixed priority scheduling discipline provides for pre-emptive scheduling or dispatching of threads based on a simple numeric priority. When a higher priority thread is created or becomes unblocked, it preempts a lower priority executing thread and executes immediately.

```
module FP_Scheduling
{
    struct SegmentSchedulingParameter
    {
        RTCORBA::Priority base_priority;
    };

    local interface SegmentSchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute SegmentSchedulingParameter value;
    };

    struct ResourceSchedulingParameter
    {
        RTCORBA::Priority resource_priority_ceiling;
    };
```

```
    local interface ResourceSchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute ResourceSchedulingParameter value;
    };

    local interface Scheduler
        : RTScheduling::Scheduler
    {
        SegmentSchedulingParameterPolicy
            create_segment_scheduling_parameter
                (in SegmentSchedulingParameter value);

        ResourceSchedulingParameterPolicy
            create_resource_scheduling_parameter
                (in ResourceSchedulingParameter value);
    };
};
```

Note that an analysis technique for scheduling fixed priority systems is Rate Monotonic Analysis (RMA). The rate monotonic analysis assigns fixed priorities to periodic threads based on their execution rates or periods – the thread having the highest rate (shortest period) is assigned the highest priority. Normally, the characteristics of all threads and their execution environment are known in advance, and rate monotonic scheduling is statically performed off-line. In this case, the Real-time CORBA 1.0 Fixed Priority discipline can be employed. Often thread behavior or execution environment characteristics such as system loading vary with some dynamic parameter, time or date, operational status of supporting systems, etc.

3.7.2 Earliest Deadline First (EDF)

The earliest deadline first discipline uses the execution completion deadline of the threads as the basis for their execution eligibility – a thread that has a shorter (closer) deadline is more eligible than one with a longer (later) deadline. In some cases, a thread's deadline is constant during the thread's lifetime, and in other cases it changes (for example, a thread's deadlines may be nested). When EDF is used to meet deadlines (i.e., for hard real-time), it requires that all deadlines can be met, in which case it is most often employed statically. Other factors can be used in conjunction with deadlines to create enhanced EDF-like disciplines that always meet all deadlines if possible, and that shed or defer load when overloaded. When EDF is used to minimize maximum tardiness (i.e., for soft real-time), it may be employed either statically or dynamically. EDF can be employed either as a scheduling discipline or as a dispatching rule.

```
module EDF_Scheduling
{

    struct SchedulingParameter
    {
        TimeBase::TimeT deadline;
```

```
        long        importance;
    };

    local interface SchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute SchedulingParameter value;
    };

    local interface Scheduler : RTScheduling::Scheduler
    {
        SchedulingParameterPolicy
            create_scheduling_parameter
                (in SchedulingParameter value);
    };
};
```

3.7.3 Least Laxity First (LLF)

A least laxity (or “time to go”) first discipline assigns execution eligibility based on laxity value, where

laxity = deadline - current time - estimated remaining computation time.

A thread with lower laxity is more eligible than one with higher laxity. An LLF discipline is sometimes used for environments where thread execution time requirements vary significantly. In such environments, a thread with a long execution time may be released prior to threads with less laxity becoming ready-to-run. The laxity estimate is updated as the thread execution duration estimate is updated at run time. An LLF discipline may specify that a thread with negative laxity should not (continue to) execute. Thus, LLF is primarily a dynamic discipline. LLF may be used either to meet deadlines (i.e., for hard real-time) or to maximize minimum lateness (or tardiness) (i.e., for soft real-time). LLF can be employed either as a scheduling discipline or a dispatching rule.

```
module LLF_Scheduling
{
    struct SchedulingParameter
    {
        TimeBase::TimeT deadline;
        TimeBase::TimeT estimated_initial_execution_time;
        long        importance;
    };
    // laxity = deadline
    //        - {current time}
    //        - (estimated_initial_execution_time –
    //                  {time executed thus far})

    local interface SchedulingParameterPolicy
        : CORBA::Policy
```

```
    {
        attribute SchedulingParameter value;
    };

    local interface Scheduler : RTScheduling::Scheduler
    {
        SchedulingParameterPolicy
            create_scheduling_parameter
                (in SchedulingParameter value);
    };
};
```

3.7.4 Maximize Accrued Utility (MAU)

A maximize accrued utility discipline uses a utility function associated with each thread to establish a thread schedule; MAU cannot be used as a dispatching rule. Each function provides a mapping from that thread's completion time to a utility value. For example, completing very close to but prior to the deadline may be most useful, while completing much earlier than the deadline may have less utility, and completing after the deadline may have zero or negative utility. Conventional deadlines are a special case of utility functions. MAU disciplines seek schedules that result in maximal accrued (e.g., summed) utility. Thus, MAU disciplines are intended for dynamic systems.

```
module Max_Utility_Scheduling
{
    struct SchedulingParameter
    {
        TimeBase::TimeT deadline;
        long        importance;
    };

    local interface SchedulingParameterPolicy
        : CORBA::Policy
    {
        attribute SchedulingParameter value;
    };

    local interface Scheduler : RTScheduling::Scheduler
    {
        SchedulingParameterPolicy
            create_scheduling_parameter
                (in SchedulingParameter value);
    };
};
```

3.8 Distributed System Scheduling

Scheduling in a distributed system can be divided into four cases. Cases 1 through 3 are the single-level scheduling cases.

Case 1 is that scheduling occurs completely independently on each node, and application behaviors that involve more than a single node do not have trans-node end-to-end timeliness requirements that are used by the node schedulers. That is the common non-real-time case.

Case 2 is that scheduling occurs independently on each node, but an application behavior that involves more than one node propagates its end-to-end timeliness context, which is then used by each node's scheduler while the behavior is active at that node. At each node, the distributable thread competes with the other threads at that node, according to its end-to-end needs. System-wide scheduling is coherent but not generally globally optimal. That is the distributed real-time case this specification explicitly addresses.

Case 3 is that scheduling on each node is global in the sense that there is a logically singular system-wide scheduling algorithm instantiated on all nodes, and node instances of this algorithm interact to cooperatively schedule all nodes in a globally optimal way. This specification does not explicitly support case 3 because such scheduling is very difficult (intractable in general) from both the conceptual and implementation standpoints, but desires not to preclude it.

Case 4 is all the multi-level scheduling cases: there is at least one level of "meta-scheduling" above the case 1 or 2 node schedulers that seeks to improve global optimality by adaptively adjusting some combination of scheduling parameter elements, schedulable entity, scheduling contexts, scheduling algorithms, scheduling disciplines, and node load balancing. Case 4 includes sub-cases corresponding to cases 2 and 3. This specification does not explicitly support case 4, but again, desires not to preclude it.

3.9 Distributable Thread

The defining characteristic of any real-time distributed computing system, whatever its programming model, is that the end-to-end timeliness (optimality and predictability of optimality, as defined in Section 3.6, "Sequencing: Scheduling and Dispatching," on page 3-5) of trans-node application behaviors is acceptable to the application.

In most cases, the fundamental requirement for achieving acceptable end-to-end timeliness is that a trans-node application behavior's timeliness properties and parameters – time constraints, expected execution time, execution time received thus far, etc. – be explicitly employed for resource management (scheduling, etc.) consistently on each node involved in that application trans-node behavior. As stated in Section 3.8, "Distributed System Scheduling," on page 3-11, "consistently" refers to distributed scheduling case 3 in this specification.

In some static real-time distributed systems, these properties and parameters can be instantiated *a priori*; but in general, and in all dynamic real-time distributed systems, these properties must be propagated among corresponding computing node resource managers (e.g., in operating systems, Java virtual machines, and middleware).

In real-time distributed CORBA systems, a trans-node application behavior's end-to-end timeliness (and potentially other) properties and parameters must be acquired from the client's ORB, propagated with the invocation, and deposited in the servant's ORB, when operation invocations occur; and similarly for any associated returns.

For (at least) dynamic real-time CORBA systems, the fixed priority propagation mechanism in the Real-time CORBA 1.0 specification is insufficient – a trans-node application behavior abstraction is needed. A natural abstraction is suggested by CORBA's native control flow programming model – a thread that can execute operations in objects without regard for physical node boundaries. In this specification, that programming model abstraction is termed a *distributable thread* (see Figure 3-1). In this figure, Objects A, B, and C may be running on either different processing nodes or within the same address space.

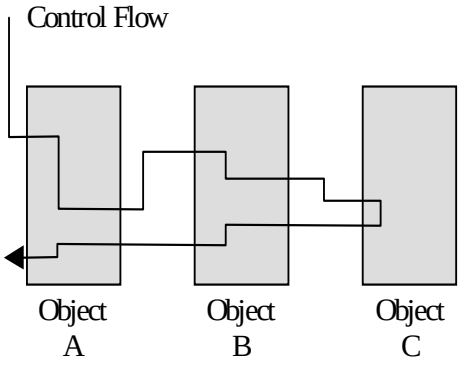

Figure 3-1 Control Flow in a Distributed Processing System

The distributable thread is the schedulable entity. Each distributable thread has a unique system-wide id. Each distributable thread may have one or more execution scheduling parameter elements – e.g., priority, time constraints such as deadlines or utility functions, importance – that specify the acceptable end-to-end timeliness for completing the sequential execution of operations in object instances that may reside on multiple physical nodes. The semantics of acceptability with respect to these end-to-end timeliness parameters is defined by the application, in the context of the scheduling discipline being used. Execution of the distributable thread is governed by the scheduling parameter elements, on each node it visits (see Figure 3-2).

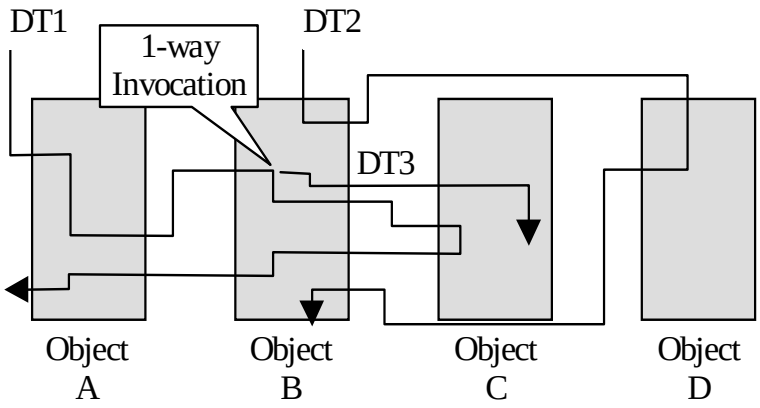

Figure 3-2 Distributed Threads

A distributable thread can extend and retract its locus of execution points among operations in object instances across physical computing nodes by location-independent invocations and (optionally) returns. Within each node, the flow of control is equivalent to normal local thread execution.

The synchrony of a conventional two-way operation invocation (or RPC) programming model is often cited as a concurrency limitation. But that criticism does not apply to the distributable thread model. A distributable thread is a sequential abstraction, like a local thread. A distributable thread is always executing somewhere, while it is the most eligible there – it is not doing send/wait's as with conventional operation invocations.

Remote invocations and returns are scheduling events at both client and servant nodes. Each node's processor is always executing the most eligible distributable thread while the others wait.

A distributable thread always has exactly one execution point (*head*) in the whole system. New distributable threads may be created, or sleeping ones awakened, when needed. An application or system may have multiple distributable threads. Multiple distributable threads execute concurrently and asynchronously, by default. Distributable threads synchronize through operation execution; the writers of each object control distributable thread concurrency in that object.

An exception that occurs anywhere along a distributable thread's locus of execution can be forwarded to and raised at the head of that distributable thread. Subsequently, the exception propagates from the head back up the distributable thread to the nearest enclosing exception handler.

Distributable thread-based programming models imply the need for a number of supporting facilities; these programming models can be differentiated by the facilities that they provide, and the approaches employed to provide them. Not all, or any, of these facilities are included in this specification. These supporting facilities include (but are not limited to) the following, which are not addressed in this specification:

- Some asynchronous “happenings” (i.e., changes in system state) of interest to a distributable thread may have to be coordinated with current distributable thread execution. For example, a violated time constraint, or the failure of a node or network path over which a distributable thread is extended, might require notification of the distributable thread’s head – as soon as possible, if the distributable thread is currently executing, and otherwise as soon as the distributable thread becomes the most eligible to execute.

  Certain other events that occur at the distributable thread’s head – e.g., synchronous exceptions (e.g., traps) and asynchronous exceptions (e.g., time constraint expirations) – may require the distributable thread to execute a local exception handler and then return back up the invocation chain to execute one or more appropriate exception handlers at those places. After such an exception, the programming model could allow the distributable thread to either continue execution where the exception was initially delivered (a *continuation* model) or terminate, or the model could require that the distributable thread always terminate (a *termination* model).

- Distributable thread control actions (e.g., suspend, resume, abort, time constraint change, etc.) may have to be propagated to, and carried out at, the distributed thread’s head.

- Mechanisms may have to be provided to support maintaining correctness of distributed execution, and consistency of distributed data – in both cases, as defined by the application – for concurrent activities of one or more applications.

- The code that is responsible for detecting/suspecting failure for an appropriate set of nodes may require visibility to failures locally perceived by a distributable thread.

All of these facilities generally would be required to be timely (e.g., subject to completion time constraints).

Section III - Overview of the Programming Model

This section presents an overview of the application programming model that is being provided. Since the specification defines a scheduling framework, as well as a limited set of scheduling disciplines, this section deals with the concepts that apply across schedulers and scheduling disciplines.

3.10 Scheduler

In this specification a scheduler is realized as an extension to Real-time CORBA that utilizes the scheduling needs and resource requirements of one or more applications to manage the order of execution of those applications on the distributed nodes of a CORBA system. A scheduler provides operations for applications to announce their requirements, which the scheduler takes into consideration when it affects the order in which threads are dispatched by the operating system.

A scheduler will be run in response to specific application requests, such as defining new scheduling parameter elements, and in response to specific application actions, such as CORBA invocations. The latter will be implemented using the CORBA Portable Interceptor interfaces. The scheduler utilizes the information provided in these interfaces to manipulate which threads are most eligible for execution by the underlying operating system. This control is via whatever interfaces the operating system provides, which are outside of the scope of CORBA. Thus, although scheduler implementations could be independent of any particular ORB implementation, as long as the ORB conforms to this specification, the scheduler will be closely tied to the operating system.

The scheduler architecture is based on the premise that a distributed application can be considered to be a set of distributable threads (see Section Distributable Thread), which may interact in a number of ways, sharing resources via mutexes, sharing transports, parent/offspring relationships, etc. The mechanisms of interaction are irrelevant to this specification.

The scheduler architecture assumes that the problem of satisfying scheduling needs can be addressed by managing the allocation of resources to distributable threads. The distributable thread provides a vehicle for carrying scheduling information across the distributed system.

Distributable threads interact with the scheduler at specific scheduling- points, including application calls, locks and releases of resources, and at pre-defined locations within CORBA invocations. The latter are required because CORBA invocations are points at which the distributable thread may transition to another processor, and the scheduling information must be reinterpreted on the new processor.

3.10.1 Scheduler Characteristics

This specification does not assume a single scheduling discipline for Real-time CORBA. Schedulers are developed to implement a particular scheduling discipline or disciplines. Both available products and technical literature abound with examples of schedulers implementing various scheduling disciplines. This specification defines only the interface between the ORB/application and the scheduler, and is intended to foster the development of schedulers that are not dependent on any particular ORB (although a particular scheduler implementation may choose to take advantage of the features of a particular ORB). Note that schedulers will likely be dependent on the underlying operating system, and this specification does not address these operating system interfaces, since they are outside of the scope of CORBA.

This specification addresses schedulers that will optimize execution for the application scheduling needs on a processor-by-processor basis (see Section 3.8, “Distributed System Scheduling,” on page 3-11, case 2). That is, as the execution of an application distributable thread moves from processor to processor, its scheduling needs are carried along and honored by the scheduler on each processor. This does not preclude the development of schedulers that perform global optimization, but this specification does not specifically address that type of scheduler.

The schedulers considered in relation to this specification will have in common processing stages where they acquire information about the demand for resources, an optional processing stage where they plan the processing schedule (when scheduling, as opposed to dispatching alone, is used), and a processing phase where they affect how threads are dispatched by the operating system. This specification does not impose any requirement on how the scheduler developer defines these processing stages. The specification does define the minimum set of scheduling points (points in time or code when the scheduler will execute).

This specification also provides the scheduler APIs for a small set of scheduling disciplines, including fixed priority, as defined in Real-time CORBA 1.0. This supports application portability for these disciplines.

The current specification does not address full interoperability across scheduler vendor implementations. To achieve this, one would have to define the scheduling discipline, the scheduling parameter elements, and the service context that is used to propagate the scheduling characteristics of the application. The submitters believe that more implementation experience is needed before full interoperability is possible. Therefore, this specification only provides a complete API definition for a limited set of well-understood scheduling disciplines and does not define a standard service context for any scheduling disciplines. Future specifications will define standardized service contexts and the APIs for additional disciplines.

3.10.2 Scheduling Parameter Elements

This specification defines a *scheduling parameter* as a container of potentially multiple values called *scheduling parameter elements*. The scheduling parameter elements are the values needed by a scheduling discipline in order to make scheduling decisions for an application. A scheduling discipline may have no scheduling parameter elements, only one, or several; the number and meaning of the scheduling parameter elements is scheduling discipline specific. A single scheduling parameter, which may contain several scheduling parameter elements, is associated with an executing thread via the **begin_scheduling_segment** operation. A thread executing outside the context of a scheduling segment has no scheduling parameter associated with it and is scheduled by the native scheduling of the operating system, typically priority based.

Some scheduling disciplines will acquire the information about application resource and scheduling requirements at system/application design time (static scheduling); these schedulers typically would load the resulting scheduling information into a data structure that is accessed at run time. Other schedulers are intended to react to dynamic runtime system demands (dynamic scheduling). These cases represent different scheduler “interaction styles.” The interaction style will depend on the scheduler implementation and, possibly, on the particular scheduling discipline. This specification provides a general scheduler interface that can be used by either style of scheduler interactions.

This specification also allows various types of interactions for static scheduling. The specific approach to be used will be discipline-specific. For example, the application may provide its scheduling parameter elements, and the associated names, in advance

so that the scheduler can store them internally; this could be done during some form of application initialization. Alternatively, the application can provide scheduling parameter elements each time it invokes scheduler operations.

The specific information needed by a scheduler will depend on which discipline(s) it implements. For example, simple deadline scheduling may need only the thread's deadline and the amount of CPU time that the thread will consume. Another discipline might utilize relative importance as one of its inputs. This specification has defined a standard interface for passing a set of scheduling discipline-specific information to a scheduler via the elements of a scheduling parameter. The definition of the structure, types, and the handling of these scheduling parameter elements is scheduling discipline-specific. The elements are only defined for the subset of scheduling disciplines provided in this specification.

3.10.3 Pluggable Scheduler and Interoperability

This specification provides a "pluggable" scheduler. A particular ORB in the system may have any scheduler installed, or may have no scheduler. If an ORB has a scheduler installed, all applications run on that ORB are "under the purview" of that scheduler.

Application components may interoperate, in the context of a particular scheduling discipline, as long as their ORBs have compatible schedulers installed (meaning that the schedulers implement the same discipline, and follow a CORBA standard for that discipline) and the scheduler implementations use a compatible service context. As noted above, the current specification does not define any standard service contexts for scheduler interoperability, although future revisions are anticipated in this area.

A scheduler may choose to support multiple disciplines, but this specification does not address how different scheduling disciplines might interact. This may also be addressed in future revisions.

3.10.4 Distributable Threads

A distributable thread (see Section 3.9, "Distributable Thread," on page 3-11) is the fundamental abstraction of application execution in this specification. A distributable thread incorporates the sequence of actions associated with a user-defined portion of the application that may span multiple processing nodes, but that represents a single logical thread of control. Distributed applications will typically be constructed as several distributable threads that execute logically concurrently.

More precisely, a distributable thread is the locus of execution between points in the application that are significant to the application developer, and it carries the scheduling context of the application from node to node as control passes through the system via CORBA requests and replies. It might encompass part of the execution of a local (or native) thread or multiple threads executing in sequence on one or more processors. If it encompasses multiple threads, then it also encompasses the various phases; that is, "in-transit," "static," "active," etc., which might occur as the locus of execution moves among threads.

A distributable thread may have a scheduling parameter containing multiple element values associated with it. These scheduling parameter elements become the scheduling control factors for the distributable thread and are carried with the distributable thread via CORBA requests and replies. Scheduling parameter elements can be associated with a thread by the application invoking the **begin_scheduling_segment** or **update_scheduling_segment** operations (see Section 3.10.6, "Scheduling Segments, Parameter Elements, and Schedulable Entities," on page 3-19). The application may call the **spawn** operation to create a distributable thread and a corresponding native thread in the current processor and associate scheduling parameter elements with it.

A distributable thread has at most one head (execution point) at any moment in time. If there is a branch of control, as occurs with a CORBA oneway invocation, the originating distributable thread remains at the client and continues execution (as long as it remains the most eligible). A new distributable thread is implicitly created to process each oneway invocations.

Each distributable thread has a globally unique id within the system, which can be accessed via the **get_current_id** operation. The distributable thread id can be used to obtain a reference to a distributable thread, via the **lookup** operation. This reference can then be used to cancel that distributable thread, via the **cancel** operation. The **cancel** operation results in a CORBA::THREAD_CANCEL system exception being raised in the cancelled distributable thread.

3.10.5 Implicit Forking and Joining

Typically, an intrinsic part of any concurrency model is the semantics for the creation of new execution contexts, or *forking*, and the synchronization of multiple execution contexts, or *joining*.

Explicit forking is provided for in this specification by the **spawn** operation. Due to time constraints explicit joining was not provided by this specification. Future finalizations and revision task forces are encouraged to provide for this capability.

Certain aspects of the core CORBA programming model and the programming model of various CORBA services introduce the implicit forking of distributable threads. One example in the core CORBA specification is *oneway* invocations if made with a synchronization scope of **SYNC_NONE** or **SYNC_WITH_TRANSPORT**. This occurs because the distributable thread making the invocation is unblocked before the operation on the servant executes. Applications may optionally associate an "implicit scheduling parameter" for a distributable thread that is associated with any implicitly created distributable threads created from that distributable thread.

When a distributable thread executing a scheduling segment implicitly forks another distributable thread, the forked distributable thread's scheduling parameter is determined as follows:

- If the implicit scheduling parameter is set for the innermost scheduling segment of the forking distributable thread, then the ORB must use this value in implicitly forking any distributable threads.

- Otherwise, the ORB must use the operative scheduling parameter of the innermost scheduling segment for the implicit forking of any distributable threads.

As with forking, there are certain aspects of the core CORBA programming model and the programming model of various CORBA services that introduce the implicit joining of distributable threads. An example of an implicit join is the polling mode introduced by asynchronous messaging. This occurs because the distributable thread calling the poll operation can wait to "join up with" the distributable thread that ran the operation on the servant to get the results of the asynchronous invocation. Note that the initial asynchronous invocation call is an implicit fork that results in the distributable thread used to run the operation on the servant.

When a distributable thread executing a scheduling segment implicitly joins another distributable thread, there is neither inheritance nor propagation of either distributable thread's scheduling parameter to the other distributable thread.

3.10.6 Scheduling Segments, Parameter Elements, and Schedulable Entities

In this specification, distributable threads consist of one or more (potentially nested) *scheduling segments*. Within a distributable thread, scheduling segments can be sequential and/or nested. Nesting creates *scheduling scopes*.

Each scheduling segment represents a sequence of control flow with which a particular set of scheduling parameter elements is associated. A scheduling segment is delineated by **begin_scheduling_segment** and **end_scheduling_segment** statements in the code. The application may use the segment name on the end statement, as an error check. The scheduling parameter associated with a distributable thread may be updated with a call to **update_scheduling_segment**.

At runtime, a scheduling segment has a single starting point, and a single ending point (although it could be coded with multiple possible ending points, during execution only one ending point can be invoked). Segments may span processor boundaries. This specification places no restrictions on the placement of **begin_scheduling_segment**'s and **end_scheduling_segment**'s; an **end_scheduling_segment** may occur on a different processor than the **begin_scheduling_segment**, and may even occur somewhere up the chain of CORBA requests.

As a distributable thread moves from object instance to object instance through CORBA invocations, it may extend (and possibly retract) itself through one or more processes or processors. When this happens, the distributable thread may be contending with a new set of distributable threads for resources.

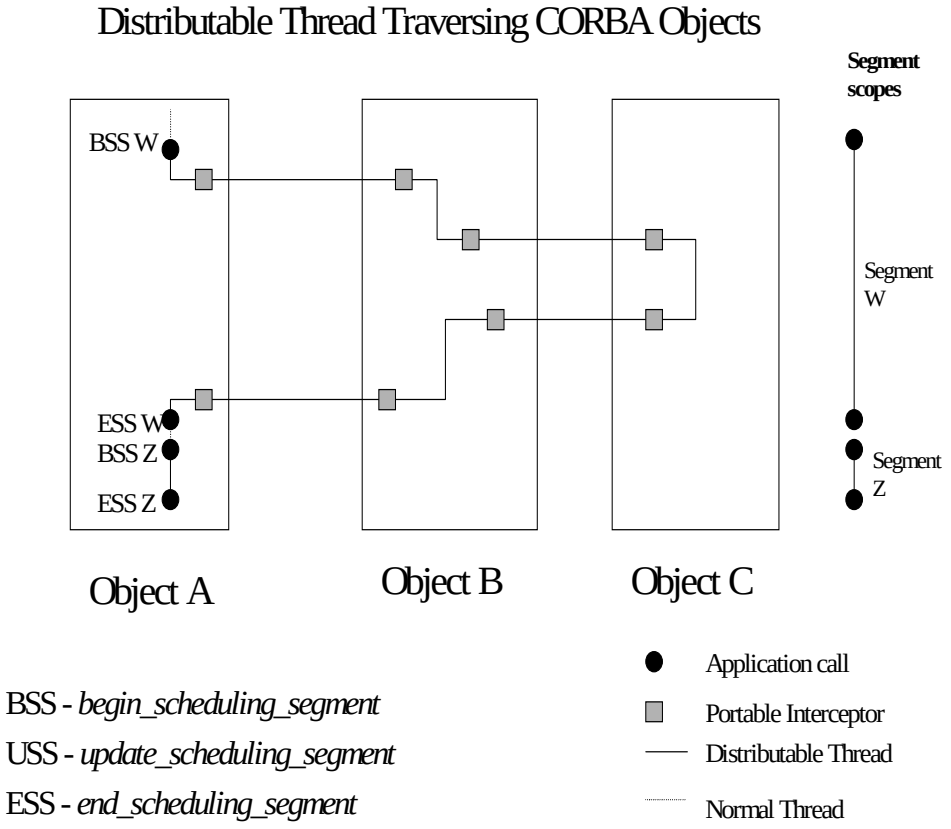

Figure 3-3 A Distributable Thread with Two Sequential Segments

Figure 3-3 illustrates a simple distributable thread which contains two sequential segments. The distributable thread begins in object instance A, with segment W, and traverses object instances B and C before returning to A, where the first segment ends and a new segment (Z) begins. Portable interceptors are invoked each time the distributable thread transitions to another object instance via a CORBA request (on both the client and servant side) and again as the distributable thread returns. Note that these object instances could be on different processors.

Suppose the scheduling discipline is Earliest Deadline First, which implies that the illustrated distributed thread must (implicitly) carry its deadline along as it progresses through the various processor environments. Further, assume that the scheduling discipline calls for scheduling segments that have missed their deadline to be terminated. This last condition implies that the scheduler must be maintaining a list of deadlines. The **begin_scheduling_segment**, **update_scheduling_segment**, and **end_scheduling_segment** operations serve to enter, update or remove deadlines, but the scheduler must also address what happens when a set deadline expires.

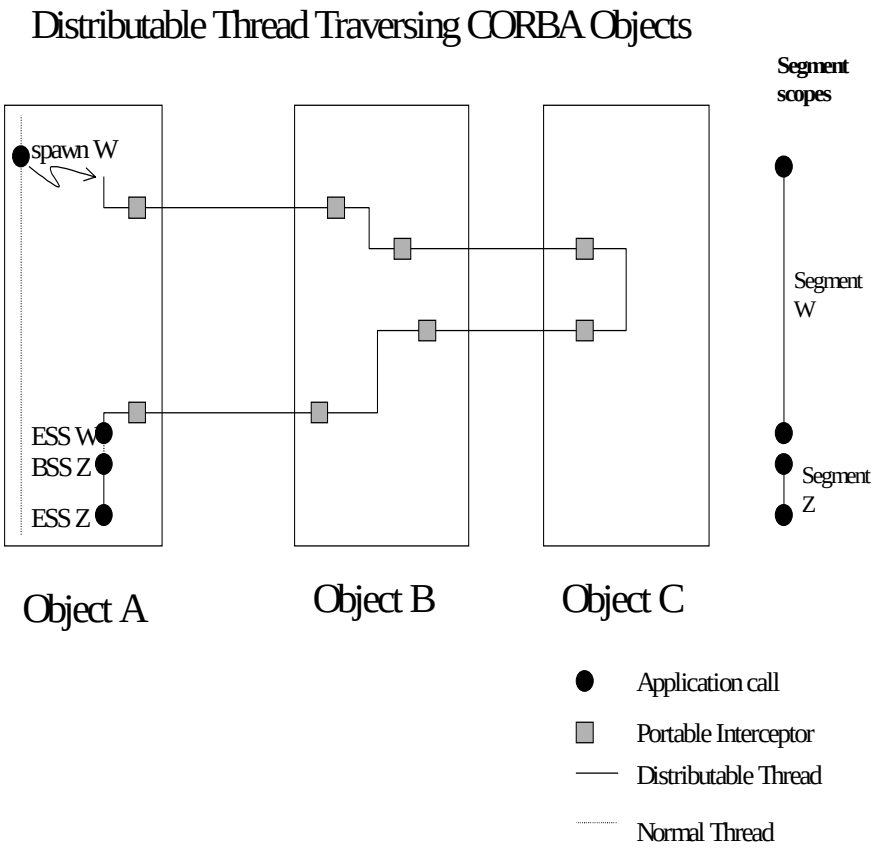

Figure 3-4 A Distributable Thread Created by a Spawn Operation

Figure 3-4 illustrates the use of a *spawn* to create a distributable thread. Note that the spawn also serves as the beginning for the initial segment (W) of the distributable thread.

Some scheduling disciplines may support the nesting of scheduling segments, which permits independently developed software components to define their own scheduling segments. The component would create an additional scheduling segment by embedding one or more pair of calls to **begin_scheduling_segment** and **end_scheduling_segment**. The handling of unspecified parameter elements (defaulting) is discipline-specific. In some cases, unspecified elements will use the values from the next outer segment (if any). In other cases, predefined or application defined default values might be used.

Each **begin_scheduling_segment** provides a new set of scheduling parameter elements for the distributable thread. If the distributable thread is already in a segment, these new parameter elements will replace the current set until a matching **end_scheduling_segment** occurs. An **end_scheduling_segment** statement

causes the distributable thread to return to the previous scheduling parameter (if any). Thus, a distributable thread may contain multiple scheduling segments that are executed sequentially, each of which may contain nested segments. This specification does not place any limits on the level of nesting that a scheduling discipline will support.

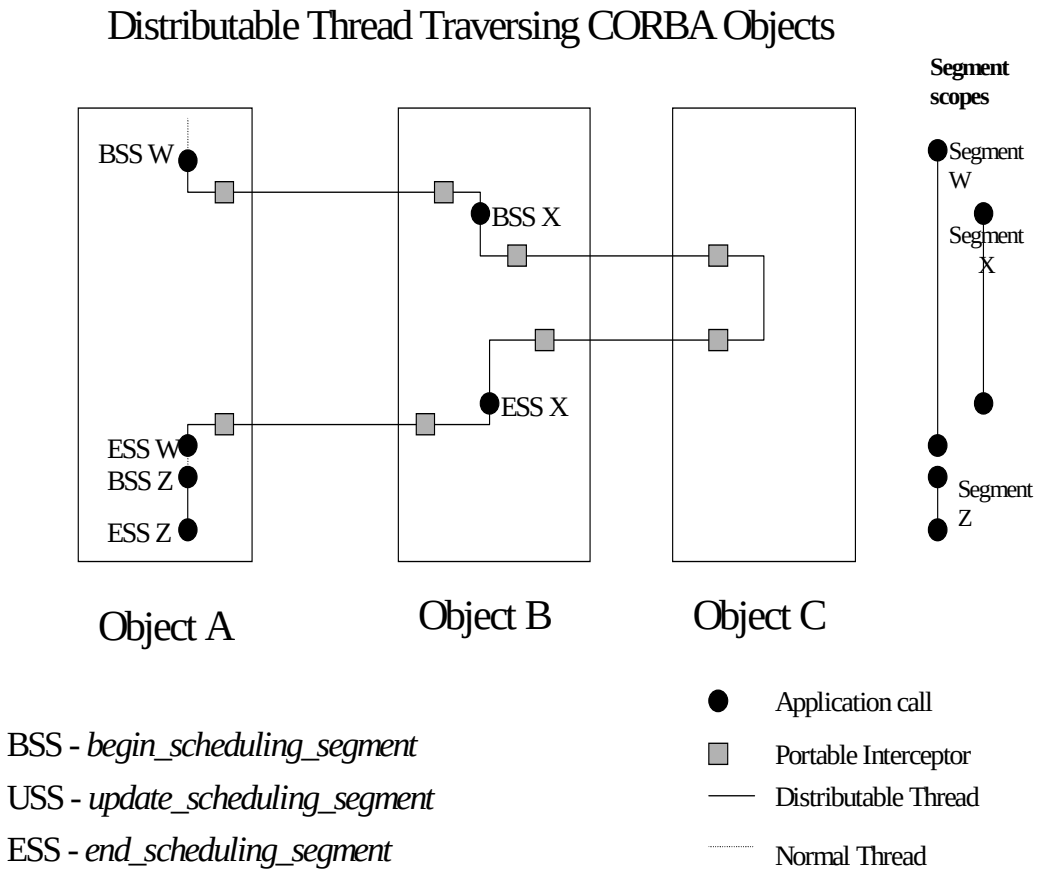

Figure 3-5 Distributable Thread with Nested Segments

Figure 3-5 illustrates segment nesting. In this case, segment X is nested within segment W. At the point where segment X begins, the scheduling context of segment W is logically pushed onto a stack, and segment W's scheduling parameter elements are used for the distributable thread. When segment X ends, the distributable thread returns to the scheduling parameter elements for segment W.

In the case of EDF, all of these segments involve the requirement that they complete by some deadline, but they would probably be different deadlines. In the case of nested segments (W, X, and Y) the tightest deadline may come from any of the segments.

A distributed thread executing in a single object instance may, at different times, have different deadlines. Note that where the distributed thread first executes in object instance B its deadline will be the deadline for segment W. However, as soon as segment X begins, the deadline must be selected from the tighter of the outer (W) or inner (X) scheduling segment.

It is expected that each instance of the scheduler must monitor the time constraints of every distributed thread that is currently traversing its node.

A scheduling parameter element that is created in one object instance must be considered in other object instances as the distributed thread passes through them. In the illustration, the deadline established in object instance A must be considered with respect to all other deadlines that exist in the domain of object B, and similarly as the distributed thread extends to object C.

How a scheduler addresses distributed dynamic scheduling is implementation dependent, but it is likely that the features of the portable interceptor will be required. By requiring use of an interceptor that targets the scheduler for the outgoing and incoming sides of the connection at both the client and server sides, the scheduler can address these characteristics. A client-side outgoing interceptor can address moving the deadline compliance monitoring while the associated server side incoming interceptor can address the continuing deadline compliance monitoring and distributed thread scheduling with respect to the server side workload.

The application may also invoke the scheduler within a segment, either to allow the scheduler to notify the application if it has had a scheduling failure (such as a missed deadline), or to modify the current segment's scheduling parameter elements. This is done via the **update_scheduling_segment** operation. The update operation allows the application to occasionally check in with the scheduler, and can also be used to change scheduling parameter elements dynamically, without creating a new segment.

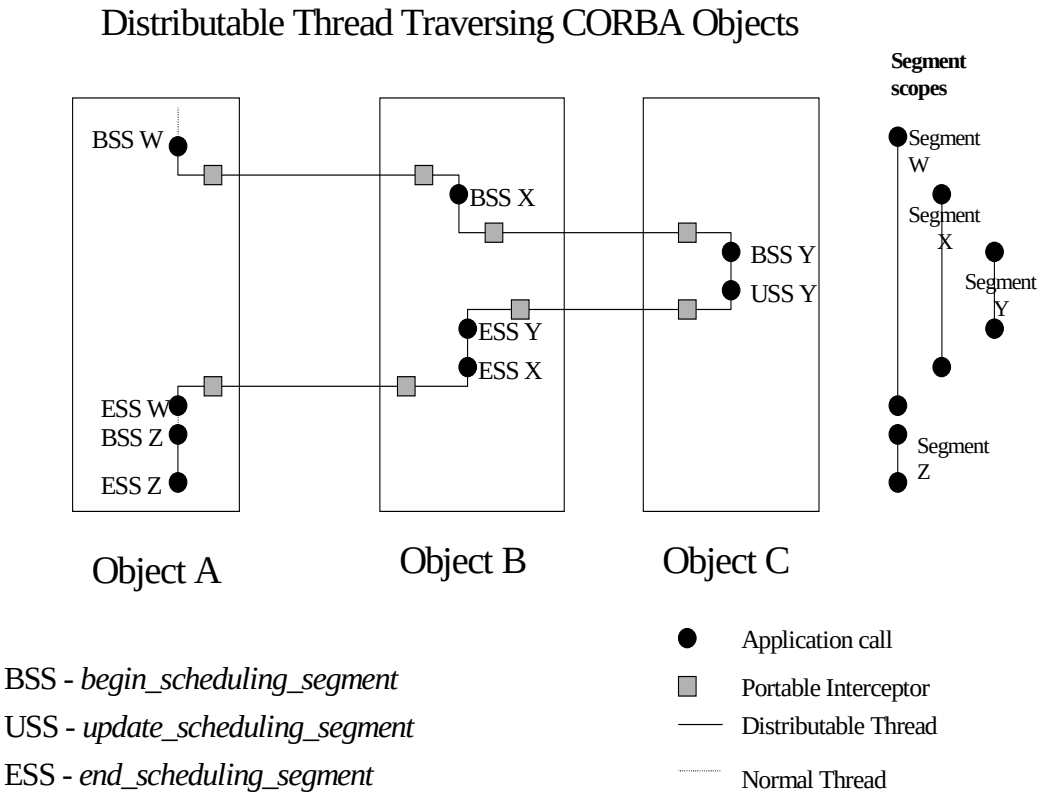

Figure 3-6 Distributable Thread with Nested Segments

Figure 3-6 illustrates the remaining features of scheduling segments, namely the use of multi-level nesting, updates, and the flexible placement of ends. Note that there are two levels of nesting within segment W. In this example, an **update_scheduling_segment** is called within segment Y. Any exceptions for the distributable thread could be delivered at this point, rather than waiting for the next portable interceptor call or the **end_scheduling_segment**. In addition, the application could provide new scheduling parameter elements on the update, without returning to the next upper scheduling scope. Note also, this in this example, segment Y is begun in object instance C, but ended in object instance B, which was the invoker of object instance C.

The application can obtain a list of the current scheduling segment names, innermost scope first, via the **current_scheduling_segment_names** operation.

3.10.7 Scheduling Points

There are a number of *scheduling points*, which are points in time and/or code at which the scheduler is run and may result in an alteration of the current schedule. These include all begins and ends, access to shared resources, and points at which control

transfers between processing nodes (i.e., CORBA requests). Because these scheduling points may result in schedule changes, they may also be a point at which dispatching occurs.

The following set of scheduling points is defined:

- Creation of a distributable thread (via **begin_scheduling_segment** or **spawn**).
- Termination or completion of a distributable thread.
- **begin_scheduling_segment**
- **update_scheduling_segment**
- **end_scheduling_segment**
- A CORBA operation invocation, specifically the request and reply interception points provided in the Portable Interceptor specification.
- Creation of a resource manager.
- Blocking on a request for a resource via a call to **RTScheduling::ResourceManager::lock or RTScheduling::ResourceManager::try_lock.**
- Unblocking as a result of the release of a resource via a call to **RTScheduling::ResourceManager::unlock.**

3.10.8 Schedule-Aware Resources

This specification permits the application to create a scheduler-aware resource locally via the **create_resource_manager** operation in a **ResourceManager**; these resources can have scheduling information associated with them via the **set_scheduling_parameter** operation. For example, a servant thread could have a priority ceiling if the application were using fixed priority scheduling. The scheduler will run when these resources are locked or released, so that the scheduling discipline is maintained.

Any scheduling information associated with these resources is scheduling discipline-specific.

3.10.9 Exceptions

This specification defines the following exceptions related to scheduling:

- CORBA::SCHEDULER_FAULT – this indicates that the scheduler itself has experienced an error.
- CORBA::SCHEDULE_FAILURE – this indicates that the distributable thread has violated the constraints of its scheduling parameter. For example, this exception could occur when a deadline has been missed or a segment has used more than its allowed CPU time.

- CORBA::THREAD_CANCELLED – indicates that the distributable thread receiving the exception has been cancelled. This may occur because a distributable thread cancels another distributable thread thereby causing the CORBA::THREAD_CANCELLED exception to get raised at the subsequent head of the cancelled distributable thread.
- RTScheduling:: UNSUPPORTED_SCHEDULING_DISCIPLINE – indicates that the scheduler was passed a scheduling parameter inappropriate for the scheduling discipline(s) supported by the current scheduler.

3.10.10 Summary

An application consists of one or more distributable threads (as well as possibly local processor threads that are not part of distributable threads). Each distributable thread will execute through one or a series of (distributed) scheduling segments, including some that may have nested segments. These segments represent regions of execution that have their own scheduling parameter elements. Within these scheduling segments, additional calls may be made to alter the scheduling parameter elements and/or to just allow the scheduler to run.

Distributable threads may evolve from application threads, due to a **begin_scheduling_segment** operation, a one-way operation, or be generated by **spawn** operations. Distributable threads may be cancelled by another distributable thread, and cancelled distributable threads will be notified of the cancellation via an exception.

These distributable threads may share local resources utilizing resource manager **lock**, **try_lock**, and **unlock** operations. These operations are schedule-respecting.

Section IV - Scheduler Interoperability and Portability

3.11 Scheduler Interoperability

A CORBA ORB supporting dynamic scheduling will interoperate with an ORB that does not support this capability. The scheduling parameter for a distributable thread is passed to the other ORB in the service context field and the other ORB can ignore them.

An ORB conformant with Real-time CORBA 1.0 will interoperate with an ORB compliant with this specification in the functional sense (i.e., without regard to timeliness). An ORB compliant with this specification that has no scheduler installed is fully interoperable in both terms of functionality and timeliness. If a scheduler is installed, then timeliness characteristics of the resulting system will depend on the installed scheduler and its backwards compatibility with the Real-time CORBA 1.0 fixed priority scheduling.

3.12 Scheduler Portability

This specification addresses the issue of portability between the ORB and scheduler, and between the application and the scheduler. This specification provides that capability in that it makes the ORB/scheduler interfaces available to applications.

3.13 Dynamic Scheduling Interoperation

This specification does not address interoperation between different dynamic scheduler implementations or between different scheduling disciplines.

Dynamic Scheduling is an extension of and modification to the RT CORBA specification. Application functions that are scheduled using the fixed priority methods will interoperate with dynamic scheduling tasks. This specification offers the application developer several options with regard to mixed mode operations. For example, a band of priorities can be reserved for dynamically scheduled activities. That band may be located at the high or low end of the priority range or it may be placed in the middle of the priority band. When activities have a priority higher than the dynamic scheduling band then dynamic scheduled activities will only run during what would otherwise be idle time. When dynamic scheduling is given top priority the scheduler resources might be dedicated to some activities while the remainder of the activities are dispatched during periods when the dynamically scheduled activities are not ready to execute.

Schedulers may be constructed so that dynamic scheduling systems can provide services to non-dynamically scheduled CORBA client applications. Requests from such a client would be treated as any processing that occurs without a scheduling parameter set. When dynamically scheduled clients make requests to non-dynamically scheduled servants then the added information carried in the service contexts is ignored. The request is valid but is not dynamically scheduled.

Section V - Dynamic Scheduling Interfaces

3.14 ThreadAction Interface

3.14.1 do Operation

3.14.1.1 IDL

```
module RTScheduling
{

...
    local interface ThreadAction
    {
        void do(in CORBA::VoidData data);
```

```
    };
...
};
```

3.14.1.2 Semantics

The **ThreadAction** interface is used to provide an entry point for newly spawned distributable threads. The **ThreadAction** interface serves as a parent type for user implemented **ThreadAction** objects. The **ThreadAction::do** operation by default does nothing. User written overrides of the **do** operation are expected execute the application's thread-specific actions.

3.15 RTScheduling::Current Interface

The **RTScheduling::Current** interface is derived from **RTCORBA::Current**. An ORB that implements this specification returns a reference from a call to **CORBA::ORB::resolve_initial_references** with the **"RTCurrent"** value passed via the *identifier* parameter that can be narrowed to an **RTScheduling::Current** reference.

3.15.1 spawn Operation

3.15.1.1 IDL

```
module RTScheduling
{
    ...
    local interface Current
        : RTCORBA::Current
    {
        ...
        DistributableThread
            spawn
                (in ThreadAction      start,
                 in unsigned long     stack_size,
                     // zero means use the O/S default
                 in RTCORBA::Priority base_priority);
        ...
    };
    ...
};
```

3.15.1.2 Semantics

The **spawn** operation creates a new O/S thread and makes that thread a distributable thread with a stack size at least as large as the value passed in the **stack_size** parameter. The initial CORBA base priority is the value passed by the **base_priority** parameter. The new distributable thread calls the **do** operation on the **ThreadAction** object passed via the start parameter.

3.15.2 UNSUPPORTED_SCHEDULING_DISCIPLINE Exception

3.15.2.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        exception UNSUPPORTED_SCHEDULING_DISCIPLINE {};
        ...
    };
    ...
};
```

3.15.2.2 Semantics

The UNSUPPORTED_SCHEDULING_DISCIPLINE exception is raised when a scheduling parameter argument isn't appropriate for the installed scheduler instance.

3.15.3 begin_scheduling_segment Operation

3.15.3.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        void begin_scheduling_segment
            (in string      name,
             in CORBA::Policy sched_param,
             in CORBA::Policy implicit_sched_param)
```

```
                raises UNSUPPORTED_SCHEDULING_DISCIPLINE);
            ...
        };
        ...
};
```

3.15.3.2 Semantics

The **begin_scheduling_segment** operation raises the RTScheduling::UNSUPPORTED_SCHEDULING_DISCIPLINE exception when the **scheduling_parameter** argument didn't have an appropriate value for the active scheduling discipline.

The **begin_scheduling_segment** operation raises the CORBA::SCHEDULE_FAILURE exception when the scheduling segment failed its schedule.

The **begin_scheduling_segment** operation raises the CORBA::SCHEDULER_FAULT exception when the scheduler has had internal error.

The **begin_scheduling_segment** operation raises the CORBA::THREAD_CANCELLED exception when the distributable thread was cancelled.

The **begin_scheduling_segment** operation raises the CORBA::BAD_PARAM exception when the **scheduling_parameter**, any elements of the scheduling parameter, or the name parameter was invalid for the installed scheduler.

The **begin_scheduling_segment** operation begins a scheduling segment, and converts the currently executing thread into a distributable thread, if it is not already one. A scheduling segment is a window of execution where a distributable thread is executing a particular region of code. The scheduler conditions execution of a particular scheduling segment using the passed **scheduling_parameter** argument, until a **begin_scheduling_segment**. **update_scheduling_segment**, or **end_scheduling_segment** is encountered.

The name parameter provides identification for the region of code that comprises the scheduling segment. Some schedulers may support nesting of scheduling segments. If a scheduler does not support nesting of scheduling segments this operation raises CORBA::SCHEDULE_FAILURE.

A **scheduling_parameter** contains elements that are a value or set of values appropriate for the active scheduling discipline. The **scheduling_parameter** used by the scheduler and set by the application.

The requirements for the "**scheduling_parameter**" and "name" parameters are dependant on both the scheduling discipline defined, and on the interaction style supported by the scheduler. It is expected that at least one these parameters ("**scheduling_parameter**" or "name") is a non-null argument.

In addition, the **begin_scheduling_segment** operation provides a scheduling point for the scheduler and gives the scheduler an opportunity to cancel a distributable thread by raising the CORBA::THREAD_CANCELLED exception while is executing in a scheduling segment.

3.15.4 update_scheduling_segment Operation

3.15.4.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
    void update_scheduling_segment
        (in string      name,
         in CORBA::Policy sched_param,
         in CORBA::Policy implicit_sched_param)
        raises (UNSUPPORTED_SCHEDULING_DISCIPLINE);
        ...
    };
    ...
};
```

3.15.4.2 Semantics

The **update_scheduling_segment** operation raises the RTScheduling::DistributableThread::UNSUPPORTED_SCHEDULING_DISCIPLINE exception when the **scheduling_parameter** argument didn't have an appropriate value for the active scheduling discipline.

The **update_scheduling_segment** operation raises the CORBA::SCHEDULE_FAILURE exception when the scheduling segment failed its schedule.

The **update_scheduling_segment** operation raises the CORBA::SCHEDULER_FAULT exception when the scheduler has had internal error.

The **update_scheduling_segment** operation raises the CORBA::THREAD_CANCELLED exception when the distributable thread was cancelled.

The **update_scheduling_segment** operation raises the CORBA::BAD_PARAM exception when the **scheduling_parameter** or any elements of the scheduling parameter are invalid for the installed scheduler.

The **update_scheduling_segment** operation provides the scheduler with a scheduling point and provides an opportunity for the scheduler to check for a scheduling failure. In addition, the **update_scheduling_segment** operation gives the scheduler an opportunity to raise the CORBA::THREAD_CANCELLED exception within a distributable thread while it is executing in a scheduling segment.

The **update_scheduling_segment** operation should only be called inside of a scheduling segment. A call to the **update_scheduling_segment** operation outside of a scheduling segment raises CORBA::SCHEDULE_FAILURE.

Any non-null value passed via the **scheduling_parameter** parameter allows an application to request that a scheduler update the scheduling parameter or the implicit scheduling parameter, or both, associated with enclosing scheduling segment. A null value indicates to the scheduler that there it should not update scheduling parameter associated with the enclosing scheduling segment.

3.15.5 end_scheduling_segment Operation

3.15.5.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        void end_scheduling_segment(in string name);
        ...
    };
    ...
};
```

3.15.5.2 Semantics

The **end_scheduling_segment** operation raises the CORBA::SCHEDULE_FAILURE exception when the scheduling segment failed its schedule.

The **end_scheduling_segment** operation raises the CORBA::SCHEDULER_FAULT exception when the scheduler has had internal error.

The **end_scheduling_segment** operation raises the CORBA::THREAD_CANCELLED exception when the distributable thread was cancelled.

The **end_scheduling_segment** operation raises the CORBA::BAD_PARAM exception when the name parameter was invalid for the installed scheduler.

The **end_scheduling_segment** operation ends a scheduling segment. Each call to a **end_scheduling_segment** operation should match a call to **begin_scheduling_segment** made in the same distributable thread. If **end_scheduling_segment** is called in a distributable thread that does not have a matching call to **begin_scheduling_segment** raises CORBA::SCHEDULE_FAILURE.

The **end_scheduling_segment** operation provides the scheduler with a scheduling point and provides an opportunity for the scheduler to check for a scheduling failure.

If a non-null string is passed via the name parameter, then the scheduler can verify the name with the name passed in the corresponding **begin_scheduling_segment** call. If a null string is passed, then no verification takes place.

After an **end_scheduling_segment** operation, the distributable thread is either operating with the scheduling parameter of the next outermost scheduling segment scope. If this operation is performed at the outermost scope, the result is that the processing for that thread reverts back to the fixed priority scheduling where the active thread priority is the sole determinant of the threads eligibility for execution.

3.15.6 Id Related Operations

3.15.6.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...

        IdType get_current_id();
        // returns id of thread that is running

        DistributableThread lookup(in IdType id);
            // returns a null reference if
            // the distributable thread is
            // not known to the local scheduler

        typedef sequence<octet> IdType;

        readonly attribute IdType id;
            // a globally unique id
        ...
    };
    ...
};
```

3.15.6.2 Semantics

Each distributable thread has a globally unique id within the system, which can be accessed via the **get_current_id** operation. The distributable thread id can be used to obtain a reference to a distributable thread, via the **lookup** operation. This reference can then be used to cancel that distributable thread, via the **RTScheduling::DistributableThread::cancel** operation. This **cancel** operation results in a CORBA::THREAD_CANCEL system exception being raised at the head of the cancelled distributable thread.

3.15.7 scheduling_parameter and implicit_scheduling_parameter Attributes

3.15.7.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        readonly attribute CORBA::Policy
                        scheduling_parameter;
        readonly attribute CORBA::Policy
                        implicit_scheduling_parameter;
        ...
    };
    ...
};
```

3.15.7.2 Semantics

Each distributable thread has a current scheduling policy if it is operating in a scheduling segment (and a null scheduling policy otherwise). The **scheduling_parameter** attribute returns the scheduling parameter for the innermost segment name.

The **implicit_scheduling_parameter** attribute returns the implicit scheduling parameter as last set by a **begin_scheduling_segment** or **update_scheduling_segment** call for the current distributable thread.

If the distributable thread is executing outside the context of the scheduling segment, then a null reference is returned from either of these attributes.

3.15.8 current_scheduling_segment_names Attribute

3.15.8.1 IDL

```
module RTScheduling
{
    ...
    local interface Current : RTCORBA::Current
    {
        ...
        typedef sequence<string> NameList;

        readonly attribute NameList
                        current_scheduling_segment_names;
            // Ordered from innermost segment name
            // to outmost segment name
        ...
    };
    ...
};
```

The **current_scheduling_segment_names** attribute returns a list of the current scheduling segment names, innermost scope first.

3.16 RTScheduling::ResourceManager Interface

3.16.1 IDL

```
module RTScheduling
{
    ...
    local interface ResourceManager : RTCORBA::Mutex
    {
    };
    ...
};
```

3.17 RTScheduling::DistributableThread Interface

3.17.1 IDL

```
module RTScheduling
{
    ...
    local interface DistributableThread
    {
```

```
        void cancel();
            // raises CORBA::OBJECT_NOT_FOUND if
            // the distributable thread is
            // not known to the scheduler
    };
    ...
};
```

3.17.2 cancel Operation

The **cancel** operation causes the CORBA::THREAD_CANCELLED exception to be raised at the head of the distributable thread. Note that while the **DistributableThread** is a local interface the head of the distributable thread may not be executing within the same address space as thread calling **cancel**.

3.18 RTScheduling::Scheduler Interface

The scheduler interface is a local interface with the semantics of an abstract interface. Its purpose is to delineate the core interface of a scheduler such that the **Scheduler** interface is used as a parent interface of a scheduler plug-in.

An object reference to the currently installed scheduler is obtained by calling **CORBA::ORB::resolve_initial_references** with the *identifier* parameter set to the value "**RTScheduler**." If no scheduler is installed, a null object reference is returned.

3.18.1 Scheduler:: INCOMPATIBLE_SCHEDULING_DISCIPLINES Exception

3.18.1.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        exception INCOMPATIBLE_SCHEDULING_DISCIPLINES {};
        ...
    };
    ...
};
```

3.18.2 Scheduler::scheduling_policies Attribute

3.18.2.1 IDL

```
module RTScheduling
{
    …
    local interface Scheduler
    {
        …
        attribute CORBA::PolicyList
            scheduling_policies;
        …
    };
    …
};
```

3.18.3 Scheduler::poa_polices Attribute

3.18.3.1 IDL

```
module RTScheduling
{
    …
    local interface Scheduler
    {
        …
        readonly attribute CORBA::PolicyList poa_policies;
        …
    };
    …
};
```

3.18.3.2 Semantics

The **scheduling_policies** attribute allows the ORB to request the list of POA policies that the scheduler requires to be applied to all POA's associated with this ORB. A null list is an acceptable result value.

3.18.4 Scheduler::scheduling_discipline_name Attribute

3.18.4.1 IDL

```
module RTScheduling
{
    …
```

```
    local interface Scheduler
    {
        ...
        readonly attribute string
            scheduling_discipline_name;
        ...
    };
    ...
};
```

3.18.4.2 Semantics

A simple string containing the textual name of the scheduling discipline for use by both the ORB and application.

3.18.5 Scheduler::create_resource_manager Operation

3.18.5.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        ResourceManager
            create
                (in string       name,
                 in CORBA::Policy scheduling_parameter);
        ...
    };
    ...
};
            // raises (CORBA::BAD_OPERATION, CORBA::BAD_PARAM,
CORBA::NO_RESOURCES);
```

3.18.5.2 Semantics

Used by application developers to create a scheduler aware resource protection primitive, and associated a name with the resource.

3.18.6 Scheduler::set_scheduling_parameter Operation

3.18.6.1 IDL

```
module RTScheduling
{
    ...
    local interface Scheduler
    {
        ...
        void set_scheduling_parameter
            (inout PortableServer::Servant resource,
             in    string       name,
             in    CORBA::Policy  scheduling_parameter);
        ...
    };
    ...
};
```

3.18.6.2 Semantics

The **set_scheduling_parameter** operation associates the supplied scheduling parameter and name parameter with the supplied servant resource.

The "**resource**" parameter is a required parameter.

The requirements for the "**scheduling_parameter**" and "**name**" parameters are scheduling discipline defined. It is expected that at least one these parameters ("**scheduling_parameter**" or "**name**") is a non-null argument.

This is useful for schedulers that associate some scheduling information with a shared resource. An example of this type of scheduler would be a fixed priority scheduler that uses some form of priority ceiling protocol.

Compliance *A*

This appendix specifies the points that must be met for a compliant implementation of Real-time CORBA Base Architecture, Dynamic Scheduling, and any one of the schedulers listed in Chapter 3, “Dynamic Scheduling".

The Real-time CORBA Base Architecture is an extension of CORBA Core. Compliance can only be claimed in conjunction with compliance to CORBA Core. Note that compliance with the Real-time CORBA Base Architecture is not necessary for compliace to CORBA Core.

An ORB implementation compliant with the Real-time CORBA Base Architecture must implement all of Real-time CORBA, as defined in the Chapter 1, “Real-time CORBA Base Architecture" and Chapter 2, “Real-time CORBA Extensions". Hence there is a single mandatory compliance point.

Dynamic Scheduling, as defined in the Chapter 3, “Dynamic Scheduling", is a separate and optional compliance point. An ORB implementation compliant with the Real-time CORBA Base Architecture may or may not choose to offer an implementation of the Dynamic Scheduling.

While, the implementation of Dynamic Scheduling is optional for implementations of the Real-time CORBA Base Architecture the opposite is not true. For an ORB to comply with Dynamic Scheduling the ORB must conform to the Real-time CORBA Base Architecture specification. In particular, an ORB that conforms to Dynamic Scheduling must implement the fixed priority scheduling of the Real-time CORBA Base Architecture when no scheduler is installed.

The implementation of the basic Dynamic Scheduling infrastructure (i.e., the implementation of all interfaces and associated semantics not associated with a particular scheduler) is the most basic form of compliance with Dynamic Scheduling.

Implementing a scheduler that conforms to one of the scheduling disciplines in this specification (i.e., the scheduler implements of all interfaces and associated semantics for that scheduling discipline) is an optional and separate compliance point for a

conforming implementation of the basic Dynamic Scheduling infrastructure. Nesting of scheduling segments is not a required feature for the compliance of a scheduler that implements any one of the specified scheduling disciplines in this specification.

Index

Index