
Real-Time CORBA Specification

August 2002
Version 1.1
formal/02-08-02



An Adopted Specification of the Object Management Group, Inc.

Copyright © 1998, 1999, Alcatel
Copyright © 1989- 2001, Hewlett-Packard Company
Copyright © 1998, 1999, Highlander Communications, L.C.
Copyright © 1998, 1999, Inprise Corporation
Copyright © 1995 - 2001, IONA Technologies, Ltd.
Copyright © 1998 - 2001, Lockheed Martin Federal Systems, Inc.
Copyright © 1998, 1999, 2001, Lucent Technologies, Inc.
Copyright © 1998, 1999, Nortel Networks
Copyright © 2002, Object Management Group, Inc.
Copyright © 1998, 1999, 2001, Objective Interface Systems, Inc.
Copyright © 1998, 1999, Object-Oriented Concepts, Inc.
Copyright © 1991 - 2001, Sun Microsystems, Inc.
Copyright © 1998, 1999, Tri-Pacific Software, Inc.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work

covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBA services™, CORBA facilities™, CORBAmed™, CORBA net™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are

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.

Contents

| | |
|---|-------------|
| Preface | v |
| 1. Real -Time CORBA Architecture | 1-1 |
| 1.1 Goals of the Specification | 1-1 |
| 1.2 Extending CORBA | 1-2 |
| 1.3 Approach to Real-Time CORBA | 1-3 |
| 1.3.1 The Nature of Real-Time | 1-3 |
| 1.3.2 Meeting Real-Time Requirements | 1-3 |
| 1.3.3 activities | 1-4 |
| 1.3.4 End-to-End Predictability | 1-4 |
| 1.3.5 Management of Resources | 1-5 |
| 1.4 Compatibility | 1-5 |
| 1.4.1 Interoperability | 1-5 |
| 1.4.2 Portability | 1-6 |
| 1.4.3 CORBA - Real-Time CORBA Interworking ... | 1-6 |
| 1.5 Real-Time CORBA Architectural Overview | 1-7 |
| 1.5.1 Real-Time CORBA Modules | 1-7 |
| 1.5.2 Real-Time ORB | 1-8 |
| 1.5.3 Thread Scheduling | 1-8 |
| 1.5.4 Real-Time CORBA Priority | 1-8 |
| 1.5.5 Native Priority and PriorityMappings | 1-8 |
| 1.5.6 Real-Time CORBA Current | 1-9 |
| 1.5.7 Priority Models | 1-9 |
| 1.5.8 Real-Time CORBA Mutexes and Priority Inheritance | 1-9 |
| 1.5.9 Threadpools | 1-9 |
| 1.5.10 Priority Banded Connections | 1-10 |
| 1.5.11 Non-Multiplexed Connections | 1-10 |
| 1.5.12 Invocation Timeouts | 1-10 |

| | | |
|-----------|--|------------|
| 1.5.13 | Client and Server Protocol Configuration | 1-10 |
| 1.5.14 | Real-Time CORBA Configuration | 1-10 |
| 1.5.15 | Scheduling Service | 1-11 |
| 2. | Real-Time CORBA Extensions | 2-1 |
| 2.1 | Real-Time ORB | 2-2 |
| 2.1.1 | Real-Time ORB Initialization | 2-2 |
| 2.1.2 | Real-Time CORBA System Exceptions | 2-3 |
| 2.2 | Real-Time POA | 2-3 |
| 2.3 | Native Thread Priorities | 2-4 |
| 2.4 | CORBA Priority | 2-5 |
| 2.5 | CORBA Priority Mappings | 2-6 |
| 2.5.1 | C Language binding for PriorityMapping | 2-6 |
| 2.5.2 | C++ Language binding for PriorityMapping | 2-7 |
| 2.5.3 | Ada Language binding for PriorityMapping | 2-7 |
| 2.5.4 | Java Language binding for PriorityMapping | 2-8 |
| 2.5.5 | Semantics | 2-8 |
| 2.6 | Real-Time Current | 2-9 |
| 2.7 | Real-Time CORBA Priority Models | 2-10 |
| 2.7.1 | PriorityModelPolicy | 2-10 |
| 2.7.2 | Scope of PriorityModelPolicy | 2-11 |
| 2.7.3 | Client Propagated Priority Model | 2-12 |
| 2.7.4 | Server Declared Priority Model | 2-13 |
| 2.7.5 | Setting Server Priority on a per-Object Reference Basis | 2-13 |
| 2.8 | Priority Transforms | 2-15 |
| 2.8.1 | C Language Binding for PriorityTransform | 2-16 |
| 2.8.2 | C++ Language Binding for PriorityTransform | 2-16 |
| 2.8.3 | Ada Language binding for PriorityTransform | 2-17 |
| 2.8.4 | Java Language binding for PriorityTransform | 2-17 |
| 2.8.5 | Semantics | 2-17 |
| 2.9 | Mutex Interface | 2-18 |
| 2.10 | Threadpools | 2-19 |
| 2.10.1 | Creation of Threadpool without Lanes | 2-21 |
| 2.10.2 | Creation of Threadpool with Lanes | 2-22 |
| 2.10.3 | Request Buffering | 2-22 |
| 2.10.4 | Scope of ThreadpoolPolicy | 2-23 |
| 2.11 | Implicit and Explicit Binding | 2-23 |
| 2.12 | Priority Banded Connections | 2-24 |
| 2.12.1 | Scope of PriorityBandedConnectionPolicy | 2-25 |
| 2.12.2 | Binding of Priority Banded Connection | 2-26 |
| 2.13 | PrivateConnectionPolicy | 2-27 |
| 2.14 | Invocation Timeout | 2-28 |

| | | |
|---------------|--|-------------|
| 2.15 | Protocol Configuration | 2-28 |
| 2.15.1 | ServerProtocolPolicy | 2-29 |
| 2.15.2 | Scope of ServerProtocolPolicy | 2-31 |
| 2.15.3 | ClientProtocolPolicy | 2-31 |
| 2.15.4 | Scope of ClientProtocolPolicy | 2-32 |
| 2.15.5 | Protocol Configuration Semantics | 2-32 |
| 2.16 | Consolidated IDL | 2-33 |
| 3. | Real-Time CORBA Scheduling Service | 3-1 |
| 3.1 | Introduction | 3-1 |
| 3.2 | IDL | 3-3 |
| 3.3 | Semantics | 3-3 |
| 3.4 | Example | 3-4 |
| 3.4.1 | Server C++ Example Code | 3-4 |
| 3.4.2 | Client C++ Example Code | 3-5 |
| 3.4.3 | Explanation of Example | 3-6 |
| | Appendix A - Conformance and Compliance | A-1 |

Contents

Preface

About This Document

Under the terms of the collaboration between OMG and The Open Group, this document is a candidate for adoption by The Open Group, as an Open Group Technical Standard. The collaboration between OMG and The Open Group ensures joint review and cohesive support for emerging object-based specifications.

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/>.

The Open Group

The Open Group, a vendor and technology-neutral consortium, is committed to delivering greater business efficiency by bringing together buyers and suppliers of information technology to lower the time, cost, and risks associated with integrating new technology across the enterprise.

The mission of The Open Group is to drive the creation of boundaryless information flow achieved by:

- Working with customers to capture, understand and address current and emerging requirements, establish policies, and share best practices;
- Working with suppliers, consortia and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies;
- Offering a comprehensive set of services to enhance the operational efficiency of consortia; and
- Developing and operating the industry's premier certification service and encouraging procurement of certified products.

The Open Group has over 15 years experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification. The Open Group portfolio of test suites includes tests for CORBA, the Single UNIX Specification, CDE, Motif, Linux, LDAP, POSIX.1, POSIX.2, POSIX Realtime, Sockets, UNIX, XPG4, XNFS, XTI, and X11. The Open Group test tools are essential for proper development and maintenance of standards-based products, ensuring conformance of products to industry-standard APIs, applications portability, and interoperability. In-depth testing identifies defects at the earliest possible point in the development cycle, saving costs in development and quality assurance.

More information is available at <http://www.opengroup.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 or 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
pubs@omg.org
<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" on page 17-34.

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

Contents

This chapter contains the following topics.

| Topic | Page |
|--|-------------|
| “Goals of the Specification” | 1-1 |
| “Extending CORBA” | 1-2 |
| “Approach to Real-Time CORBA” | 1-3 |
| “Compatibility” | 1-5 |
| “Real-Time CORBA Architectural Overview” | 1-7 |

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 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 activities

This specification uses the word “activity” with a small “a.” It treats an “activity” as an analysis/design concept rather than as an implementation concept. Real-Time systems developers are interested in the particular relationship between the system under development and the system’s environment. This relationship describes: those external stimuli from the environment that impinge upon the system; the patterns with which these stimuli occur; and the extent of activity in the system resulting from each stimulus.

Most systems will not be purely CORBA systems. That is there may be I/O other than request and reply messages and there may be threads in addition to those handling the ORB and CORBA applications. Developers need to be able to treat such threads as part

of their activities. They also need to be able to treat non-CORBA Inputs as stimuli that trigger activities. It is a matter of application architecture whether or not a CORBA request message is treated as a stimulus that triggers an activity.

Real-Time CORBA does not define IDL for an activity. Instead of worrying about how to delimit an individual activity, it deals with invocations of IDL defined operations. These are well-formed concepts in the OMA. An operation invocation consists of a Request and a Reply. It is initiated by some client computational context; for example, a thread and passes through a client-role ORB, a transport protocol (TCP in the case of GIOP), a server-role ORB (possibly involving queuing) to a server application. Thereafter the operation passes through the same entities in reverse order, back to the client. An activity may encompass several, possibly nested, operation invocations.

This specification acknowledges that an abstract activity is represented by concrete entities: a message within a transport protocol, a request held in memory, and a thread scheduled to run on a processor. These three phases are termed “in-transit,” “static,” and “active” respectively. Real-Time CORBA provides the ability to effect these three phases of an activity. It leaves the developer to delimit their concept of an activity by the way they coordinate these concrete entities using the interfaces specified.

This specification provides a Real-Time CORBA Scheduling Service as an addition to the set of CORBA Core extensions. The Scheduling Service provides sufficient abstraction for the developer to work in terms of activities.

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 Real-Time CORBA Scheduling Service is specified in the Real-Time CORBA Scheduling Service 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.

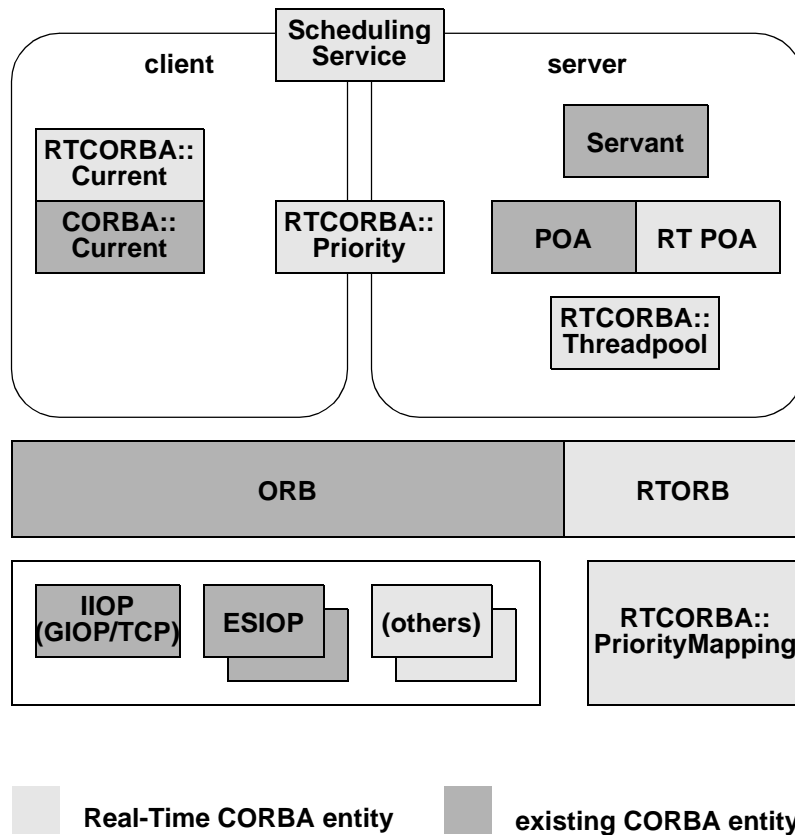


Figure 1-1 Real-Time CORBA Extensions

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.

1.5.15 Scheduling Service

The Scheduling Service provides an abstraction layer to hide the coordination of Real-Time CORBA scheduling parameters; for example, CORBA Priorities and Real-Time POA Policies. The Scheduling Service uses “names” for activities and for objects.

The developer uses the run-time Scheduling Service by acting on these named activities and objects. The design-time part of the Scheduling Service determines how each of these named entities can be coordinated, using the interfaces defined for the Real-Time ORB, so that they meet their Real-Time requirements.

Real-Time CORBA Extensions

2

Note – Base text from CORBA 2.6.1. **Changes from Core 12/2000 RTF in Red.**

Contents

This chapter contains the following topics.

| Topic | Page |
|-----------------------------------|-------------|
| “Real-Time ORB” | 2-2 |
| “Real-Time POA” | 2-3 |
| “Native Thread Priorities” | 2-4 |
| “CORBA Priority” | 2-5 |
| “CORBA Priority Mappings” | 2-6 |
| “Real-Time Current” | 2-9 |
| “Real-Time CORBA Priority Models” | 2-10 |
| “Priority Transforms” | 2-15 |
| “Mutex Interface” | 2-18 |
| “Threadpools” | 2-19 |
| “Implicit and Explicit Binding” | 2-23 |
| “Priority Banded Connections” | 2-24 |
| “PrivateConnectionPolicy” | 2-27 |

| Topic | Page |
|--------------------------|------|
| “Invocation Timeout” | 2-28 |
| “Protocol Configuration” | 2-28 |
| “Consolidated IDL” | 2-33 |

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("RTCCurrent")** 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 ServicedId    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 );

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 **Objectld** of the target object is made available to the inbound transform while the **Objectld** 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 **Objectld** 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.

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.

Comment: [CORBA Core 12/2000 RTF Issue 4657](#)

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 IOP 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 marshal 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 IOP would be selected, rather than GIOP plus TCP being selected separately. IOP in particular is represented by the value **TAG_INTERNET_IOP** (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    rcv_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 rcv_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 **TCPPProtocolProperties** 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 Serviced RTCorbaPriority = 10;
    const Serviced RTCorbaPriorityRange = 11;
};
```

```
//File: RTCORBA.idl
#ifndef _RT_CORBA_IDL_
#define _RT_CORBA_IDL_
#include <orb.idl>
#include <iop.idl>
#include <TimeBase.idl>
#pragma prefix "omg.org"
// IDL
module RTCORBA {typedef short NativePriority;

    typedef short Priority;

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

```
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_
#include <orb.idl>
#include <PortableServer.idl>
#include <RTCORBA.idl>
#pragma prefix "omg.org"
// 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 );
    };
};

```

```
    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_
```


Real-Time CORBA Scheduling Service

3

Contents

This chapter contains the following topics.

| Topic | Page |
|----------------|-------------|
| “Introduction” | 3-1 |
| “IDL” | 3-3 |
| “Semantics” | 3-3 |
| “Example” | 3-4 |

3.1 Introduction

This chapter describes the Real-Time CORBA Scheduling Service. The Scheduling Service uses the primitives of the Real-Time ORB to facilitate enforcing various fixed-priority Real-Time scheduling policies across the Real-Time CORBA system in a way that abstracts away from the application some of the low-level Real-Time constructs. The Scheduling Service does not impose any new requirements on Real-Time or non-Real-Time ORBs beyond what appears in the RT CORBA specification or CORBA specification respectively.

The Scheduling Service makes use of the detailed information available at design-time regarding the associations among activities, objects, resources, and priorities. This information may be placed in the run-time Scheduling Service either by build tools or through proprietary, initialization-time interfaces.

The primitives added in Real-Time CORBA to create a Real-Time ORB are sufficient to achieve Real-Time scheduling, but effective Real-Time scheduling is complicated. For applications to ensure that their execution is scheduled according to a uniform

policy, such as global Rate Monotonic Scheduling, requires that the RT ORB primitives be used properly and that their parameters be set properly in all parts of the CORBA system.

Not only is determining the proper use and correct parameters difficult, but once it is done, the application code becomes substantially more complex - making analysis and modification very difficult. The Scheduling Service specified in this chapter addresses these problems because an instance of the Scheduling Service embodies a uniform scheduling policy, and because the simple Scheduling Service interface abstracts away much of the complexity from application code.

An application that uses an implementation of the Scheduling Service is assured of having a uniform Real-Time scheduling policy, such as global rate-monotonic scheduling with priority ceiling, enforced in the entire system. That is, a Scheduling Service implementation will choose CORBA priorities, POA policies, and priority mappings in such a way as to realize a uniform Real-Time scheduling policy. Different implementations of the Scheduling Service can provide different Real-Time scheduling policies.

The Scheduling Service abstraction of scheduling parameters (such as CORBA Priorities) is through the use of “names.” The application code uses names (strings) to specify CORBA Activities and CORBA objects. The Scheduling Service internally associates those names with scheduling parameters and policies for the named Activity or the named CORBA object. This abstraction improves portability with regard to Real-Time features, eases uses of the Real-Time features, and reduces the chance for errors.

Each name used by the Scheduling Service method invocations must be unique. The Scheduling Service is designed to work in a “closed” CORBA system where fixed priorities are needed for a static set of clients and servers. Therefore, it is assumed that the system designer has identified a static set of CORBA Activities, the CORBA objects that the Activities use, and has determined scheduling parameters, such as CORBA priorities, for those Activities and objects. In that process, names are uniquely assigned to those Activities and Objects and the names are associated to scheduling parameters. This association of names to scheduling parameters is then used to configure the Scheduling Service.

The capabilities provided by the Scheduling Service are not orthogonal to the primitives provided by the Real-Time ORB. In fact, most of the capabilities provided by the Scheduling Service are expected to be implemented by the Scheduling Service invoking the Real-Time CORBA primitives in a way that ensures a uniform Real-Time scheduling policy is enforced.

3.2 IDL

```

//File: RTCosScheduling.idl
#ifndef _RT_COS_SCHEDULING_IDL_
#define _RT_COS_SCHEDULING_IDL_
#include <orb.idl>
#include <PortableServer.idl>
#pragma prefix "omg.org"
// IDL
module RTCosScheduling {
    exception UnknownName {};

    // locality constrained interface
    interface ClientScheduler {

        void schedule_activity(in string name)
            raises(UnknownName);
    };

    // locality constrained interface
    interface ServerScheduler {

        PortableServer::POA create_POA (
            in PortableServer::POA parent,
            in string adapter_name,
            in PortableServer::POAManager a_POAManager,
            in CORBA::PolicyList policies)
            raises ( PortableServer::POA::AdapterAlreadyExists,
                    PortableServer::POA::InvalidPolicy );

        void schedule_object(in Object obj, in string name)
            raises(UnknownName);
    };
};
#endif // _RT_COS_SCHEDULING_IDL_

```

3.3 Semantics

A CORBA client obtains a local reference to a **ClientScheduler** object. Whenever the client begins a region of code with a new deadline or priority (indicating a new CORBA Activity), it invokes “**schedule_activity**” with the name of the new activity. The Scheduling Service associates a CORBA priority with this name (assuming the name is valid; otherwise, an exception is thrown), and it invokes appropriate RT ORB and RTOS primitives to schedule this activity.

The “**create_POA**” method accepts parameters allowing it to create a POA. This POA will enforce all of the non-Real-Time policies in the Policy List input parameter. All Real-Time policies for the returned POA will be set internally by this scheduling service method. This ensures a selection of Real-Time policies that is consistent with

the scheduling policy being enforced by the Scheduling Service implementation. The Scheduling Service implementation should clearly document what POA RT policies it will use under various conditions.

“**Schedule_object**” is provided to allow the Scheduling Service to achieve object-level control over scheduling of the object. RT POA policies in the RT ORB allow some control over the scheduling of object invocations, but must do so for all objects managed by each POA. Some Real-Time scheduling, such as priority ceiling concurrency control, requires object-level scheduling. The “**schedule_object**” call will install object-level scheduling with scheduling parameters, for example, the priority ceiling for the object. These scheduling parameters are derived internally by the Scheduling Service using the name passed into the call.

3.4 Example

The following example use of the Scheduling Service, in C++, uses two CORBA objects each supporting two operations: “method1” and “method2.” A client wishes to call method1 on both objects under one deadline and subsequently call method2 on both objects under a different deadline.

For both client and server it is assumed that the relevant Scheduling Service is started, that Scheduling Service instance is available, and that an appropriate **PriorityMapping** has overridden the ORB vendor’s default.

The use of names instead of actual CORBA priorities in application code has two major advantages.

1. The use of names instead of priority numbers allows changing of scheduling policy; for example, from Deadline Monotonic to Rate Monotonic without changing or re-compiling application code. If the chosen Scheduling Service was enforcing Deadline Monotonic Scheduling it might, for instance, internally use CORBA priority 10 for “activity1” and CORBA priority 12 for “activity2.” If a different implementation of the Scheduling Service were being used, it might internally use completely different CORBA priorities for these two CORBA activities to realize a different scheduling policy; for example, Rate Monotonic Scheduling.
2. The use of names instead of priority numbers allows changing *any* CORBA priority without having to find and possibly re-order CORBA priority numbers in application code. The Scheduling Service is the central place to change CORBA priorities. Again, changes in priority can be made without re-compiling application code.

3.4.1 Server C++ Example Code

```
// SERVER C++
// Initialize ORB

CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

// Get Root POA
```

```

CORBA::Object_var rpoa = orb ->
    resolve_initial_references("RootPOA");

PortableServer::POA_var rootPOA =
    PortableServer::POA::_narrow(rpoa);

// create some policies

CORBA::PolicyList policies(2);
policies[0] = rootPOA -> create_thread_policy(
    PortableServer::ThreadPolicy::ORB_CTRL_MODEL);
policies[1] = rootPOA -> create_lifespan_policy(
    PortableServer::LifespanPolicy::TRANSIENT);

// create my RT scheduling POA.

RTCosScheduling::ServerScheduler_var server_sched ;

PortableServer::POA_var RTPOA =
    server_sched -> create_POA(
        rootPOA,
        "my_RT_POA",
        PortableServer::POAManager::_nil(),
        policies ) ;

// create object references and then schedule the objects

CORBA::Object_var obj1 = RTPOA -> create_reference (
    "IDL:Object1:1.0" ) ;
CORBA::Object_var obj2 = RTPOA -> create_reference (
    "IDL:Object2:1.0" ) ;

...

server_sched -> schedule_object ( obj1, "Object1" ) ;
server_sched -> schedule_object ( obj2, "Object2" ) ;

...

```

3.4.2 Client C++ Example Code

```

// CLIENT C++
// Initialize ORB

CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);

// create the instance of the client scheduler.

RTCosScheduler::ClientScheduler_var client_sched ;

```

```
// get and bind Objects

object1_var obj1 = /* something */
object1_var obj1 = /* something */

// invoke methods

client_sched -> schedule_activity ("activity1") ;

obj1 -> method1 () ;

obj2 -> method1 () ;

...

client_sched -> schedule_activity ("activity2") ;

obj1 -> method2 () ;

obj2 -> method2 () ;

...
```

3.4.3 Explanation of Example

The **PriorityMapping** is consistent with the policy being enforced by the implementation of the Scheduling Service. For instance, a priority mapping for an analyzable Deadline Monotonic policy might be different than the priority mapping for an analyzable Rate Monotonic policy. Thus the Scheduling Service will have determined the appropriate **PriorityMapping** prior to run-time.

Note that there are no calls to the Real-Time CORBA APIs (**RTORB**, **RTCORBA::Current**, **RTPortableServer::POA**, etc.) in the example. The Scheduling Service shall be capable of making all the necessary calls with the implementation of its own operations.

Note that there are no CORBA priorities specified - only names for the two CORBA Activities in the client. This facilitates plugging in different fixed priority scheduling policies by choosing an implementation of the Scheduling Service. Recall that the server in the example has two Scheduling Service calls. The first call accepts the normal parameters to create a POA. The Scheduling Service is capable of creating all the necessary Real-Time policies; therefore, only non-Real-Time policies need to be provided by the developer. The Scheduling Service creates the POA itself within the provided wrapper. It coordinated the POA with other aspects of the system. For example, it can select Real-Time policies (thread pools, protocols, concurrency, server priority, etc.) that make sense under the uniform scheduling policy being enforced. It also relieves the application programmer from having to determine all of those (relatively complicated) policies themselves.

The Scheduling Service calls to “**schedule_object**” allows the Scheduling Service to associate a name with the object. Any Real-Time scheduling parameters for this object, such as the priority ceiling for the object, are assumed to be internally associated with the object’s name by the Scheduling Service implementation. Thus, the call associates the scheduling parameters (for example, priority ceiling) with the object reference, perhaps to enforce priority ceiling concurrency control on that object.

Scheduling Service implementation associates the names “activity1” and “activity2” in the **schedule_activity** calls in the client with CORBA priorities. This association was made prior to run-time. The **sched_activity** calls allow the users code to be configured correctly for performing activity1 (or activity2). When the client invokes the server, either the client priority is propagated (implicitly) or there is declared priority at the server for the target object. The server-side ORB will always make a call to the inbound **PriorityTransform** and with the **ObjectId** the available the transform is capable of retrieving the name “object1” and, primed by the **SchedulingService**, returning a priority for the upcall appropriate to the scheduling policy being enforced.

A.1 Conformance

This section specifies the points that must be met for a compliant implementation of Real-Time CORBA. Real-Time CORBA is an extension of CORBA. Conformance can only be claimed in conjunction with conformance to CORBA. Note that, Real-Time CORBA Extension is not necessary for conformance to CORBA.

A.2 Compliance

An ORB implementation compliant with Real-Time CORBA must implement all of Real-Time CORBA, as defined in the “Real-Time CORBA Extensions” chapter. Hence there is a single mandatory compliance point.

The Real-Time CORBA Scheduling Service, as defined in the “Real-Time CORBA Scheduling Service” chapter, is a separate and optional compliance point. An ORB implementation compliant with Real-Time CORBA may or may not choose to offer an implementation of the Real-Time CORBA Scheduling Service.

A

active phase 1-4
activities 1-4
Ada Language binding for PriorityMapping 2-7
Ada Language binding for PriorityTransform 2-17
Algorithms 1-2
Architectural overview 1-7

B

Binding 2-23
Bounding of thread usage 1-10
Buffering of additional requests 1-10

C

C Language binding for PriorityMapping 2-6
C Language Binding for PriorityTransform 2-16
C++ Language binding for PriorityMapping 2-7
C++ Language Binding for PriorityTransform 2-16
Client and server protocol configuration 1-10
Client C++ Example Code 3-5
Client Propagated Priority Model 2-12
Client propagated priority model 1-9
ClientProtocolPolicy 2-31
Compatibility 1-5
CORBA - Real-Time CORBA Interworking 1-6
CORBA priority 2-5
CORBA priority mappings 2-6
Current 1-9

E

End-to-end predictability 1-4
Exceptions 2-3
Extending CORBA 1-2

H

hard real-time 1-2

I

IDL 2-33, 3-3
Interoperability 1-5
in-transit phase 1-4
Invocation Timeout 2-28
Invocation timeouts 1-10

J

Java Language binding for PriorityMapping 2-8
Java Language binding for PriorityTransform 2-17

M

Messaging
 module A-1
Models 2-10
Modules 1-7
Mutex interface 1-9, 2-18

N

Native Priority 1-8
Native thread priorities 2-4
nature 1-3
Non-multiplexed connections 1-10

O

ORB Initialization 2-2

P

Partitioning of threads 1-10
POA 2-3
Portability 1-6
POSIX Real-time extensions 1-2
Preallocation of threads 1-10
Priority 1-8, 2-5
Priority banded connections 1-10, 2-24
Priority inheritance 1-9
Priority mappings 2-6
Priority models 1-9
Priority Transforms 2-15
PriorityMappings 1-8
PriorityModelPolicy 2-10
PrivateConnectionPolicy 2-27
Protocol Configuration 2-28
Protocol Configuration Semantics 2-32
Protocols 1-2

R

Real-Time 1-3
Real-Time CORBA configuration 1-10
Real-Time CORBA Current 1-9
Real-Time CORBA Priority Models 2-10
Real-Time CORBA system exceptions 2-3
Real-Time Current 2-9
Real-Time ORB 1-8, 2-2
Real-Time ORB initialization 2-2
Real-Time POA 2-3
Real-Time requirements 1-3
Reply 1-4
Request 1-4
Request Buffering 2-22
Resources, management 1-5
rotocol 2-28

S

Scheduling Service 1-11
Semantics 2-8, 2-17, 3-3
Server C++ Example Code 3-4
Server Declared Priority Model 2-13
Server declared priority model 1-9
Server priority 2-13
ServerProtocolPolicy 2-29
soft real-time 1-2
static phase 1-4
System exceptions 2-3

T

Thread scheduling 1-8
Threadpool 1-9
Threadpool with Lanes 2-22
Threadpool without Lanes 2-21
ThreadpoolPolicy 2-23
Threadpools 2-19
Transforms 2-15
Transport protocol 1-4

Index
