OpenFusion
RTOrb Ada Edition

Version 1.0

User Guide

PRISMTECH

OpenFusion
RTOrb Ada Edition

User Guide

Augusta Ada King,
Countess of Lovelace

Copyright Notice

CONTENTS

Table of Contents

Real-time Programming

Preface

About this User Guide

This *User Guide* provides instructions and information needed to install, configure and use OpenFusion RTOrb Ada Edition.

Intended Audience

The *User Guide* is intended to be used by software developers who wish to use RTOrb to develop CORBA-based, real-time distributed applications in Ada. RTOrb can also be used as a conventional, non real-time, high performance enterprise Ada ORB for developers who do not need real-time capabilities.

Organisation

This *User Guide* is divided into three major sections: *Installation and Configuration* which provides information on installing and configuring RTOrb; *Real-time Programming* provides background information on CORBA, Ada and real-time programming; and *Programming with RTOrb* which describes how to create applications using RTOrb.

Conventions

The conventions listed below are intended to guide and assist the reader in understanding the User Guide.

Item of special significance or where caution needs to be taken.

i Item contains helpful hint or special information.

WIN Information applies to Windows (e.g. NT, 2000, XP) only.

UNIX Information applies to Unix based systems (e.g. Solaris) only.

On-Line (PDF) versions of this document: Items shown as cross references to other parts of the document, e.g. *Contacts* on page 2, behave as hypertext links: users can jump to that section of the document by clicking on the cross reference.

```
% Commands or input which the user enters on the command
line of their computer terminal
```

`Courier New`, **`Courier New Bold`** , or *`Courier New Italic`* fonts indicate programming code. The `Courier New` font also indicates file names.

Extended code fragments are shown as small `Courier New` font contained in shaded, full width boxes (to allow for standard 80 column wide text), as shown below:

```
NameComponent newName[] = new NameComponent[1];
// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, “”);

rootContext.bind (newName, demoObject);
```

`Courier Italics` and ***`Courier Italic Bold`*** indicate new terms or emphasise an item.

Arial Bold indicates user related actions, such as **File | Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be contacted at the following contact points. Users of the On-line version of this manual can *click* the e-mail addresses below to launch their e-mail client or Web browser to send e-mail direct to PrismTech.

Corporate Headquarters

PrismTech Corporation
6 Lincoln Knoll Lane
Suite 100
Burlington, MA
01803
USA

Tel: +1 781 270 1177
Fax: +1 781 238 1700

European Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead NE11 0NG
UK

Tel: +44 (0)191 497 9900
Fax: +44 (0)191 497 9901

Web: http://www.prismtech.com
General Enquiries: info@prismtech.com

INTRODUCTION

What is *Real-time*?

There are several definitions available which state what *real-time* means, such as:

"*Immediate, as an event is occurring.*"[1],

"*The actual time during which physical events take place.*"[2]

"*The processing and visibility of transactions and information as they occur, and not on a periodic or batch basis.*"[3]

"*...computer systems that update information at the same rate as they receive data...*"[4]

Current computer systems have physical restrictions which limit the ability to process information immediately, "*as an event is occurring*"- there are the inevitable processing speed and resource limits which affect how fast data can be processed. For the purpose of programming actual real-time applications, a more realistic definition of real-time has been adopted:

An application for which the requirements, design, or developers state that execution of application logic must or should occur within well-defined temporal conditions. [5]

---

1. http://www.hq.nasa.gov/office/pao/History/presrep95/r.htm
2. http://www.telemet.com/weather_gloss_q_r.htm
3. http://sun2.lenoir.cc.nc.us/~disted/distermc.htm
4. *The American Heritage® Dictionary of the English Language*, Fourth Edition, © 2000 Houghton Mifflin Company.

5. This definition is as given in Taking the Java™ Language into Uncharted Waters: Project Mackinac, Sun's RTSJ Implementation, Bollella et. al., Sun Microsystems, Inc.

Or in other words, the processing or completion of tasks is not instantaneous, but occurs within pre-defined time limits. This definition accepts the physical realities of our present computing machines and systems.

However, to complicate and possibly confuse matters, two different types of *real-time* have been identified, each relating to their ability to meet "well-defined temporal conditions". The types are:

- *hard real-time* where the execution of the application logic must *always* meet the temporal requirements,
- *soft real-time* where the execution of the application logic may *sometimes* meet the temporal requirements.

A system where there are no well-defined temporal conditions is referred to as a *non real-time system*.

These definitions are important (even if they appear to complicate matters) since they provide flexibility as to the temporal stringency and capability which a system will be designed to achieve. Some systems must perform strictly within the temporal limits, whereas others can be more flexible, appreciating that it is likely to be more difficult and costly to create the more stringent systems.

How RTOrb Provides for Real-time

The language and architectural components of RTOrb address the practical issues of developing real-time applications for the real world, whether they need to meet the more demanding hard real-time requirements or the less demanding soft ones. Real-time Ada and Real-time CORBA address the respective practical aspects of achieving hard or soft real-time requirements for distributed systems. Some aspects include:

- end-to-end predictable execution, thread scheduling and dispatching, along with the provision of distributable threads
- resource management, particularly memory management and allocation
- synchronization, resource sharing and avoidance of *priority inversion*[1]
- asynchronous event handling, transfer of control and thread termination
- interoperability and portability

1. These aspects are to ensure that things happen in the correct sequence in order to meet specified temporal requirements.

Features, Standards and Compliance

Amongst the key features of OpenFusion RTOrb Ada Edition are:

- CORBA 3.0 ORB
- RT CORBA v1.2 support
- CORBA Object Services – bundled Naming Service and Event Service
- Ada Language bindings based on the OMG's IDL to Ada Language Mapping specification
- Full POA implementation
- Multithreaded support
- Ultra-fast ANYs
- CORBA Messaging implementation (request Timeout, SyncScope and Rebind Policies, full AMI (Poll/Async))
- ORB interoperability including OpenFusion RTOrb Java Edition, TAO and JacORB.

The OpenFusion RTOrb Ada Edition product complies with the following standards and specifications, except as noted under *Limitations* below.

- OMG CORBA Specification, version 3
- GIOP Specification, version 1.3
- OMG Real-Time CORBA Specification, version 1.2

Limitations

- OpenFusion RTOrb Ada Edition does not support the "server per method" activation policy.
- It can only start local services (services which run on the same host as the orb), but it can start a script which can itself start a remote service.
- Persistence of objects must be managed by their implementation.

Support & Maintenance

PrismTech offers a wide range of support and maintenance packages for OpenFusion RTOrb Ada Edition that can be tailored to each customer's specific requirements.

PrismTech is renowned for the quality and responsiveness of its technical support services.

Scope of this Guide for RTOrb

The goal of this guide is to help developers use RTOrb as quickly and effectively as possible. Its scope includes essential background information, in addition to installation, configuration and usage information.

It is beyond the scope of this manual to provide full coverage of RTOrb's underlying technologies, such as explaining real-time programming techniques and theory, or covering the Real-time Ada or Real-time CORBA specifications. Information on these topics is available in the various documents listed in the bibliography.

This guide provides a technological overview which developers and architects can use as a starting point for understanding the intricacies of writing distributed, hard or soft real-time, CORBA-based, Ada programs.

A number of useful, if not essential, references are provided in the *Bibliography*: readers are encouraged to use these references to develop understanding of this powerful technology.

INSTALLATION AND CONFIGURATION

CHAPTER

1 Installation

This chapter describes how to install OpenFusion RTOrb Ada Edition (RTOrb). Please follow the procedures carefully.

1.0.1 Conventions

The following conventions are used in this chapter:

- Commonly used directories are shown as:

`<OFRT_DIR>` - where RTOrb is or will be installed

- The directory paths and environment variable separator shown here use the UNIX forward-slash (/) and colon (:) separator conventions; Windows™ users should substitute these separators with the standard DOS back-slash (\ ) and semi-colon (;) separators.
- Items which are unique to UNIX or Windows are shown using the *UNIX Only* or *Windows Only* icons, respectively. For example:

WIN
```
> SET CLASSPATH=.;%CLASSPATH%;
```

UNIX
```
% CLASSPATH=.:$CLASSPATH; export CLASSPATH
```

1.1 Prerequisites

RTOrb depends on underlying services and technologies. If these services and technologies are not properly installed and configured, then the OpenFusion RTOrb Ada Edition cannot perform as intended. Accordingly, please check that your system meets each of the prerequisites described below before installing OpenFusion RTOrb Ada Edition.

i The currently supported platforms are listed on the RTOrb *Supported Platforms* web page.
The *Supported Platforms* web page can be accessed from the *index.html* page located

in the root directory where RTOrb is installed (<OFRT_DIR>).
Please refer to *Supported Platforms* and other *Features* pages for the latest information about this distribution.

1.1.1 Supported Platforms

The OpenFusion RTOrb Ada Edition distribution is supported on a range of leading operating systems and has been built using a number of different Ada compiler variants, including:

- OS:

  Solaris, Linux, Windows, HP-UX, OpenVMS, MacOsX, Tru64
- RTOS:

  VxWorks, LynxOS
- Compilers:

  Rational Apex, Aonix ObjectAda, GnatPro, DDC-I Score, Green Hills Multi

Other platforms can be supported on request.

1.2 Installation Procedure

1.2.1 General

All installed RTOrb files are placed in the RTOrb installation directory specified during installation. No files are stored in any of the UNIX system directories.

1.2.2 Preparation

It is recommended that any existing RTOrb installation be removed before installing the current version (see *Uninstalling* on page 13). Please note the following warning.

Uninstalling OpenFusion RTOrb Ada Edition removes all RTOrb files, including the executables, license, configuration, and data files located in the RTOrb sub-directories. If these files are required, then they should be backed-up prior to uninstalling.

1.2.3 Installation

1.2.3.1 Installing the Development Environment

To install the RTOrb development environment,

UNIX Instructions

- First unzip the downloaded file and provide the unzip key (if the unzip key is not available , then contact)
- Open a shell and, cd to the directory where the `install_orbriver.bin` is located.
- At the prompt type: `sh ./install_orbriver.bin`
- Now open the RTOrb inline documentation and check the RTOrb configuration operations.

Notes

- A Java virtual machine is included with installers named VM_xx. It will be executed automatically when you run `install_orbriver.bin`.

WIN Instructions

- double-click on the downloaded file
- double-click on `install_orbriver.exe` and provide the unzip key (if the unzip key is not available , then contact) .

Notes

- A Java virtual machine is included with installers named VM_xx. It will be executed automatically when you run the installer.

MAC Instructions

- First unzip the downloaded file and provide the unzip key (if the unzip key is not available , then contact)
- After downloading, double-click `install_orbriver`
- Now open the RTOrb inline documentation and check the RTOrb configuration operations.

Notes

- Requires Mac OS X 10.0 or later
- The compressed installer should be recognized by Stuffit Expander and should automatically be expanded after downloading. If it is not expanded, you can

expand it manually using StuffIt Expander 7.0.1 or later.

- If you have any problems launching the installer once it has been expanded, make sure that the compressed installer was expanded using Stuffit Expander. If you continue to have problems, please contact technical support.

VMS Instructions

- Unzip the kit and then run `sys$update:vmsinstal` to install the unzipped kit.
- Proceed as usual with vmsinstal and answer the questions

Notes

- No Java VM included in this installer.

1.2.3.2 Installing for Production

When deploying RTOrb orbs, clients and services for production, the environment variable `<OFRT_DIR>` must to be set to a directory containing a copy of the `bin` (except idl2ada), `etc` and `doc` directories.

The `<OFRT_DIR>/etc/Orbs` file must be then setup to define the production orb topology (see Configuration).

You may need additional RTOrb license keys. Once they are installed (see 1.3.2 "Installing licence keys"), the production installation is completed. The different orbs can then be launched, beginning with the license server, and the production clients and services can then run.

1.2.4 Testing the ORB

The RTOrb is running properly by running an example such like Echo (see 6.2.2).

1.3 Licenses

1.3.1 Principles

Each license key is specific to the host where the license server runs and to the licensed tool, and may have an expiration date. Execution of the licensed tool is allowed on any machine capable to connect to the license server, providing that the number of simultaneous runs does not exceed the number of license tokens. The behaviour of the tool in case of license enfringements is tool dependent.

RTOrb and idl2ada execution is protected by license keys which are stored in the file

<OFRT_DIR>/etc/top_graph_x.lic. This file is read by the RTOrb daemon named License_Server which should be described in the <OFRT_DIR>/etc/Orbs file (see Configuration).

1.3.2 Installing license keys

The instructions to install the license keys will be provided by Top Graph'X with the license keys.

The tool <OFRT_DIR>/bin/add_license will be used for this purpose. When adding licenses, add_license first tries to update the running license server, then it updates the file <OFRT_DIR>/etc/top_graph_x.lic . If it cannot connect to the license server, a warning message is displayed (this generally means that the license server is not running). When installing RTOrb for the first time, you cannot run any daemon before installing a valid license key.

1.3.3 Requesting licenses

If you will use a single license server:

First execute <OFRT_DIR>/bin/machine_id on the computer which will run the license server. Email the result of this command and the host name to PrismTech support, you will receive the license keys in return.

If you have several licenses and want to run a license server on several hosts, do the following for each such host:

- write the result of <OFRT_DIR>/bin/machine_id on this host
- write the number of RTOrb licenses assigned to this host
- write the number of idl2ada licenses assigned to this host

Email the result to PrismTech support, you will receive the license keys in return.

If you need more licenses, ask PrismTech

1.4 Uninstalling

This section describes the procedure for uninstalling OpenFusion RTOrb Ada Edition.

Uninstalling RTOrb removes all RTOrb files, including the executables, license, configuration, and data files located in the RTOrb sub-directories. If these files are required, then they should be backed-up prior to uninstalling.

Step 1: Stop any running RTOrb services.

Step 2: Backup any data, license or other required files which are in the RTOrb directories.

Step 3: Run the *OrbAda_Uninstaller* utility (located in the *<OFRT_DIR>/UninstallData* directory):

```
% <OFRT_DIR>/UninstallData/OrbAda_Uninstaller
```

CHAPTER

2 Configuration

2.1 Configuration and Properties

2.1.1 Typing commands

The following tips and rules may help you for typing commands:

- Comment lines start with -- and blank lines are allowed.
- ORB names are not case sensitive.
- The format of the Orbs file is:
  <Orb_Name> ID=<Orb_Id> HOST=<Host_Name> TCP=<TCP port>
  where

  - <Orb_Name> indicates the name of the ORB (may be put in RTOrb variable)
  - <Orb_Id> indicates the identification number of the ORB
  - <Host_Name> indicates the host where the ORB runs
  - <TCP port> indicates the TCP port for IIOP connections to the ORB

- There should be no space on any side of the = sign. Example:

```
OrbAda_1 ID=1 HOST=localhost TCP=6060
License_Server ID=1 HOST=localhost TCP=6060
OrbAda_2 ID=2 HOST=omg.org TCP=6061
```

The command line environment variable `<OFRT_DIR>` must be set and contain the name of the directory where RTOrb is installed. The file `Orbs` in the `<OFRT_DIR>/etc` directory contains the list of the known RTOrb daemons.

There should be at most one daemon definition per line and an ORB definition should fit in one line. Several names may have the same definition. In this case, these different names are aliases of the same RTOrb daemon, which then can be accessed with any of the aliases.

There should exist a daemon named License_Server with "TCP=6060" which will be used as the license server for CORBA PrismTech tools (like `orbriver` and idl2ada).

RTOrb hangs if the daemon named "License_Server" is not running or if no license token is available.

2.1.2 POA Ports

POA names can be associated with TCP ports if desired. Add the property assignments as shown below in the `<OFRT_DIR>/etc/Poas` file to assign POA names with TCP ports.

To assign a POA name to a

- single port address use:

```
<POA name> TCP=<TCP port>
```

- range of port addresses use:

```
<POA name> TCP=<Min TCP port>-<Max TCP port>
```

where:

- there must be no space at the beginning of any line
- empty lines and lines beginning with hyphens (-) are ignored
- names which are in a POA hierarchy must be separated by forward slashes (/), for example:

```
<POA name>/<POA name> TCP=<TCP port>
<POA name> TCP=<Min TCP port>-<Max TCP port>
```

- the root POA name is called `RootPOA` by default. The name can be changed by running the required service with the `-OAName <Root POA name>` switch, noting that the name change is specific to that service.

Example of POA name TCP port assignments :

```
RootPOA TCP=10000-10999
RootPOA/Persist_Factories TCP=160001
```

2.1.3 Messaging Configuration

The `Messaging` module must be configured before messaging features are used by an application: a system exception is raised if the `Messaging` module is not configured. An application can configure messaging programmatically by using the following *`Messaging.Configure()`* method:

```
procedure Configure
   ( Sync_None_Tasks     : in Natural ;
     Sync_None_Priority  : in Priority ;
     Poll_Priority       : in Priority ;
     Async_Priority      : in Priority ;
```

```
Async_Min_Tasks    : in Natural := 0 ;
Async_Max_Tasks    : in Natural := 0 ;
Async_Peak_Tasks   : in Natural := 0 ;
Async_Stack_Size   : in Natural := 65_536);
```

The `Configure` procedure allows all of the relevant `Messaging` properties to be configured by the user.

2.1.3.1 Messaging Properties

The messaging properties which are set by the `Configure()` methods are as follows:

`Sync_None_Tasks` – the number of threads in the threadpool used for sending oneway sync_none requests.

`Sync_None_Priority` – the priority of the threads used for sending oneway sync_none requests.

`Poll_Priority` – the priority of the threads used for receiving asynchronous responses in poll mode.

`Async_Priority` – the priority of the threads used for receiving asynchronous responses in call_back mode.

`Async_Min_Tasks`, **`Async_Max_Tasks`**, and **`Async_Peak_Tasks`** – determines how the threadpool used for receiving asynchronous responses in `call_back` mode should be dimensioned. The values for these properties must follow the rule whereby `Async_Min_Tasks` ≤ `Async_Max_Tasks` ≤ `Async_Peak_Tasks`.

`Async_Min_Tasks` – the minimum number of threads in the threadpool used for receiving asynchronous responses in `call_back` mode.

`Async_Max_Tasks` – the maximum number of threads in the threadpool used for receiving asynchronous responses in `call_back` mode when all threads are not running (in other words, some threads are waiting).

`Async_Peak_Tasks` – the maximum number of threads in the threadpool used for receiving asynchronous responses in `call_back` mode when all threads are running.

`Async_Stack_Size` – `Async_Stack_Size` sets the size of the scoped memories used to run each thread of the threadpool which is used for receiving asynchronous responses in `call_back` mode.

2.1.3.1.1 Messaging Properties Example

Example using `Async_Min_Tasks`, `Async_Max_Tasks`, and `Async_Peak_Tasks`

A client application calls `Messaging.Configure()`, using the following parameters:

```
Async_Min_Tasks = 2
Async_Max_Tasks = 4
Async_Peak_Tasks = 6
```

The Async_Min_Tasks value is 2 in this case, therefore the Messaging module will create a threadpool containing two threads (the minimum).

The application is subsequently required to make three simultaneous, asynchronous responses during the course of its operation. The Messaging module's threadpool creates a third thread to handle the responses. The three threads will remain in the threadpool after the responses have been completed, even though they will be waiting, since the number of threads in the threadpool is less than the value of Async_Max_Tasks (4).

The application is then asked to handle five simultaneous, asynchronous responses: the required two new threads can be added to the threadpool since the total number of threads will be less than the value of Async_Peak_Tasks (6).

When one or more the threads are freed after handling their respective response, existing threads in the threadpool will be terminated until the total number of threads is equal to or less than the Async_Max_Tasks value (4).

The maximum number of threads that the application will be allowed to have at any one time is the value of Async_Peak_Tasks (6): any additional requests must wait until a thread is freed.

REAL - TIME PROGRAMMING

CHAPTER

3 Reviewing CORBA Concepts

CORBA stands for Common Object Request Broker Architecture. CORBA is the Object Management Group's (OMG):

"*open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.*"[1]

The Object Management Group is a non-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications.

3.1 Basic Concepts

3.1.1 The ORB

A core element of CORBA is the *Object Request Broker*, referred to as the *ORB*.

An ORB mediates between an object and one of its clients. A client is defined as any computing context that invokes operations on the object (that is, sends it a message, or invokes a method). ORBs can take many different forms. In common practice, ORBs are mechanisms that mediate between clients and objects on different computers, using some kind of network communication. ORBs are one of the principal enabling technologies in the field of distributed object computing.

3.1.1.1 Distributed Object Computing

Most popular object-oriented programming languages provide language constructs for encapsulation, inheritance, polymorphism, and other characteristic object-oriented concepts. These mechanisms have proven beneficial when building single-process applications. However, because they are implemented as programming language features, the benefits are not available when the application needs to interact with other processes or with remote machines. Programmers must generally resort to techniques such as sockets to build distributed applications.

1. The OMG's definition from its web site at http://www.omg.org

Distributed object technology extends the benefits of object-oriented technology across process and machine boundaries to encompass entire networks. In short, this technology makes remote objects appear to programmers as if they were local objects (that is, simple programming-language objects in the same process). This effect can be described as location transparency.

3.1.1.2 Transparencies

Transparencies occur when a software abstraction allows programmers to cross a computing boundary (such as a boundary between different languages, machines, network protocols, and so on) without having to be aware of the boundary at all, or without performing an explicit transformation to cross it.

In an object system, location transparency means that an object's client can invoke the object's methods in a natural manner, regardless of where the object actually resides. The target object may reside in the client program itself (as is inherently the case with most object-oriented programming languages), it may reside in another address space on the same machine as the client, or it may reside on a remote machine. The object's programming interface (from the client's perspective) is identical in all cases. See *Figure 1* for an illustration of this concept.

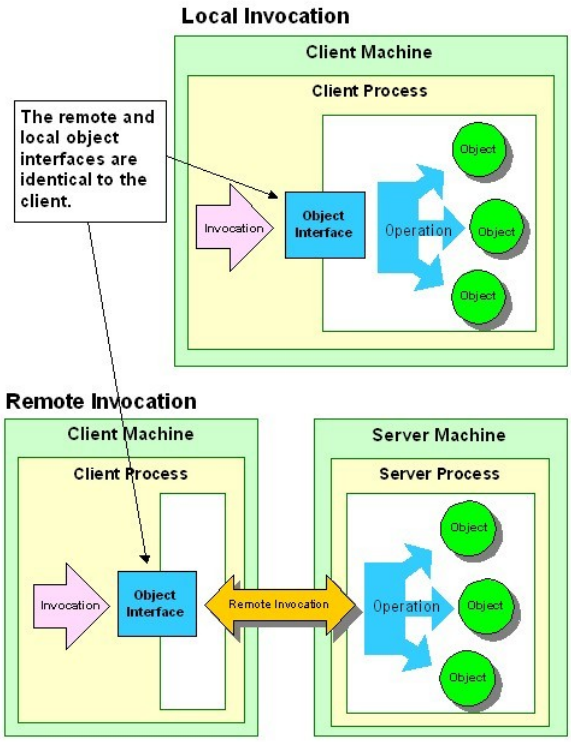


Figure 1 emote Invocations and Location Transparency

The ORB provides the location transparency in the CORBA model. ORBs also provide many other useful transparencies, including the following:

- **Programming language transparency**- The client and the object may be written in different programming languages and the ORB hides this fact; a Ada client is completely unaware that it is invoking an operation on a language-specific object, whether Java, C++, or Smalltalk, and vice versa.
- **Platform transparency**- The client and object implementation programs may be executing on different types of computing hardware, with different operating systems, in such a way that both programs are unaware of these differences.
- **Representation transparency**- Because of language, hardware, or compiler differences, processes communicating through an ORB may have different low-level data representations. The ORB automatically converts different byte orders, word sizes, floating point representations, and so on, so that application programmers can ignore the differences and avoid problems.

As lower- level distribution problems be come transparent, architects and programmers can focus their efforts on solving application problems, not plumbing problems. Expressed in other terms, distributed object technology raises the level of abstraction for distributed application design and development.

3.1.2 Distributed Object Computing (DOC) and CORBA

OMG specifications have emerged as the primary focus of industry standardization in distributed object computing, client/server computing, and large-scale object-oriented application development. The CORBA specifications provide the foundation for the most comprehensive platform for system interoperability and software portability that is foreseeable in today's computing market.

To this end, CORBA specifies:

- a concrete object model
- an abstract language for describing object interfaces
- abstract programming interfaces for implementing, using, and managing objects
- equivalent concrete programming interfaces in popular object-oriented programming languages (that is, language mappings)
- operational interfaces between ORBs to ensure interoperability between products from different vendors

Other OMG specifications include CORBAservices, which specifies standard interfaces for fundamental object services, such as naming and persistence, that are frequently required and generally useful for managing objects regardless of their

function or application domain.

3.1.2.1 Interfaces

In the CORBA object model, attention is primarily focused on the object's interface. An interface is the boundary layer that separates a consumer of an object's service (a client) from the supplier of the object's service (an object implementation). The interface defines what a client can know about an object and how a client may interact with it. As such, it hides the low-level details on one side of the boundary from the other side.

It may seem contradictory to describe interfaces as "hiding" things and providing "transparencies" at the same time, but it really isn't. The details that are hidden (such as network protocols, programming language idiosyncrasies, physical data organization, and so on) are like dirt on a window. They obscure what you really want to view the abstract behaviour of the object. By wiping these details out of the way (or hiding them) ORBs give an object's consumer clear, un-obscured access to the object's essential behaviour, expressed in terminology natural to the consumer.

An interface may also be viewed as a *contract* between an object's client and implementation. The implementation agrees to respond to a given request with certain results; both the client and the implementation agree on the information that will be exchanged in a given operation, and so on. If both sides abide by the contract and don't rely on any assumptions that aren't stated explicitly in the contract, then the interaction between client and object will behave properly.

A CORBA interface consists of a collection of operations, attributes, and definitions for data types that are used with the operations and attributes. CORBA interfaces may be composed from other interfaces through inheritance.

Almost every section of the CORBA specification deals with one aspect of interfaces or another, such as how interfaces are described, how the descriptions are stored and managed, how abstract de scriptions are mapped into concrete pro g ramming interfaces in various programming languages, how object implementations relate to and support an interface, and soon.

The CORBA specification defines a language for describing abstract object interfaces, called Interface Definition Language, or IDL.

3.1.2.2 Programming with CORBA Interfaces

IDL can be used to generate the *stubs* and *skeletons* that are actually used when programming. Since IDL is only an abstract interface description language, it must be transformed into equivalent constructs in a concrete programming language to be useful. The way in which these transformations are made for a particular language is called a *mapping* for that language.

Figure 2 illustrates the relationships between stubs, skeletons, clients, object implementations, and the ORB.

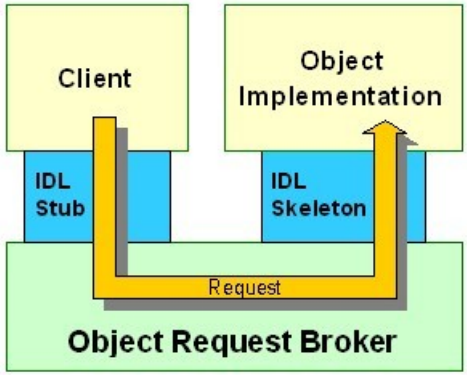


Figure 2 ORB Component Relationships

3.1.2.2.1 Stubs

Stubs are used by clients to invoke operations on target CORBA objects.

A stub is not the CORBA object itself. It *represents* a CORBA object and is, in part, responsible for propagating requests (invocations) made on itself to the real target object. In keeping with this role, stubs are sometimes called *proxies* or *surrogates*. When the target object resides in a remote process, the stub is responsible for packaging the request, with its parameters, into a message to send to the remote process across a network, then receiving the reply message from the object, unpacking the operation results from the message, and returning them to the calling program.

3.1.2.2.2 Skeletons and impls

Skeletons are used to call object *implementations*. An implementation of a CORBA interface is a package of code in a concrete programming language that provides the real be haviour of the object type . In some cases , the term implementation is used to indicate the body of code in an abstract sense, that is, the *type* (as opposed to an individual instance). In other cases, implementation can mean a specific instance of the implementation type. When there is a possibility of ambiguity, we will distinguish between the two as *implementation type* and *implementation instance*.

A skeleton takes the form of an base class declaration with functions that correspond to the operations in the IDL interface. Programmers construct an implementation by using the generated impl class, then providing method implementations for the operations.

The stub and impl have identical (or nearly identical) interfaces.

3.1.2.2.3 Clients and Servers

When a program includes the stub type and invokes operations on instances of the stub type, that program is acting in the role of a *client*, with respect to the target object represented by the particular stub instance. When a program includes an implementation type (built from the generated impl), creates instances of the implementation type, and makes them available for use by clients, the program is acting in the role of *server*, with respect to the implemented objects.

Note that the terms client and server merely describe *roles* that programs play with respect to a particular object or set of objects. In a distributed object context (or more specifically, a CORBA context), these terms do not indicate architectural roles played by the programs, as they do in the traditional sense of client/server computing. A client of one CORBA object may be the server for other clients. Programs sharing each others' objects in a variety of client/server roles may in fact be peers architecturally.

3.1.2.3 Delivering Requests Using an ORB

As described above, an ORB is anything that mediates between a client and its target object. By *mediate*, we mean to deliver the request from the client context to the server context, invoke the method on the target object, and deliver results, if any, back to the client. CORBA does not in any way prescribe or limit the mechanisms that an ORB may use to accomplish this task. The range of possible implementations is extremely large, and has interesting consequences, both practical and theoretical.

By leaving implementation decisions completely free, the CORBA specification allows highly specialized ORBs to be optimised for particular environments with unusual requirements, such as embedded real-time systems. For the purposes of this discussion, however, we will describe the OpenFusion RTOrb Ada Edition implementation.

3.1.2.3.1 Delivering Requests to Remote Objects

The ORB is a set of libraries that are linked into the client and server programs of the distributed CORBA-based application. When the client invokes an operation on the object, via the stub, the stub and the client-resident ORB library cooperate to assemble a message that describes the request. After assembling the message, the stub invokes the appropriate function in the client-resident class, transmitting the message to the server that contains the target object.

The message is received in the server by the server-resident ORB component. This component is responsible for decoding the message. The portable object adapter

(POA) locates the specific object targeted in the request and passes the message contents to the skeleton. The skeleton extracts the request parameters and invokes the requested operation on the object implementation instance. The process then reverses itself: the skeleton creates the reply message, sends it back to the client, where the stub decodes it and returns the results to the client that made the request.

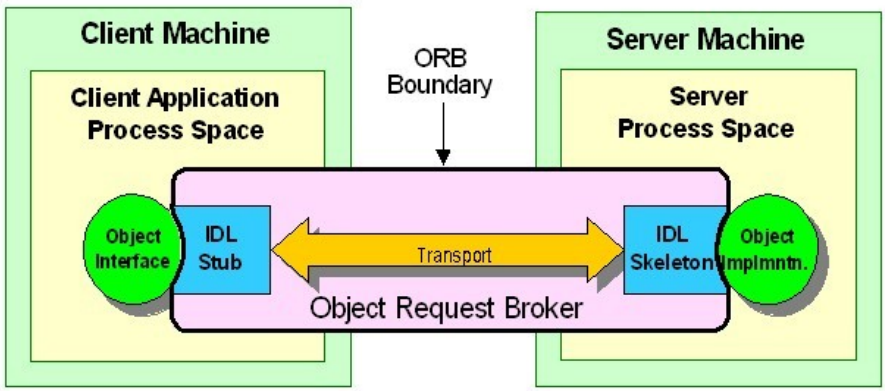
3.1.3 ORB Components

The ORB is composed of everything that intervenes between the client and the object to achieve location transparency. In a simple example, illustrated previously in *Figure 2*, the ORB encompasses the stub, the client-resident ORB classes, the server-resident ORB, and the skeleton. It can be argued that the network itself constitutes part of the ORB, because it mediates data transfer between processes - playing a major role in providing location transparency.

In an ORB's run-time environment, there may be a number of other processes (which are neither the client nor the server) that become involved in some aspect of the request delivery activity, to locate objects, start new server processes, monitor the status of requests in progress, and so on. It is usually not possible to point to a single process or software component and (accurately) call it *the ORB*.

Another way to determine what constitutes an ORB is to observe the two interface boundaries that the ORB mediates between. By boundary, we mean a specific API invocation (for example, function call, method invocation, and so on) through which non-ORB elements (clients and object implementations) interact with the ORB.

The client interacts with the ORB by invoking a member function on a stub. This boundary is labeled the client-ORB boundary in *Figure 3*. The object interacts with the ORB primarily by having one of its member functions invoked by the ORB. This boundary is labeled the ORB-object boundary in the figure. Anything between those boundaries may be considered as part of the ORB for conceptual purposes.



Note: The Client machine and Server machine can be the same physical machine.

Figure 3 The ORB as an Abstraction

3.1.3.1 Abstraction

Contrast the previous example with the following scenario. As mentioned above, stubs and skeletons are build from an IDL interface. When a programmer uses an ORB-based object, methods are invoked on the stub. Since both the stub class and the impl

class are buld from the IDL interface, client code that makes the invocation could be using either a stub that is bound to a remote object, or it could be invoking a method directly on an implementation instance that is in the same process. This use of polymorphism allows the client to use remote and local objects in exactly the same way, without ever having to (or in some cases, even being able to) distinguish between them.

When a client "sends" a request to a local implementation instance, what constitutes the ORB? You might be tempted to say that there is no ORB present but, in fact, there is. All of the necessary elements are present - the client, the target object, and something that delivers the request from the client to the object. The delivery mechanism (the ORB) in this case is the machine instruction that performs the function call on the target object's member function. The mediation between the client and the object takes place in a single stack frame in the local machine. Thinking of this as an ORB may seem too abstract, but from the programmer's point of view a local invocation is indistinguishable (if the ORB is properly implemented) from a remote invocation. If it communicates like an ORB, it's an ORB.

If you consider this scenario with respect to interface boundaries, the client-ORB and ORB-object boundaries from the previous example have coalesced into a single client-ORB-object boundary, creating for us the mental image that the ORB (in the case of local invocations) is a two-dimensional, infinitely thin surface between the client and the server.

3.1.4 Terminology Explained

Figure 4 is an adaptation from the CORBA 2.3 specification. The following subsections describe the elements shown in the figure and their roles in the overall activity of delivering requests. Some of the descriptions given here do not exactly match those in the CORBA specification. Where our descriptions vary, it is generally to achieve greater clarity and to provide a more consistent overall picture.

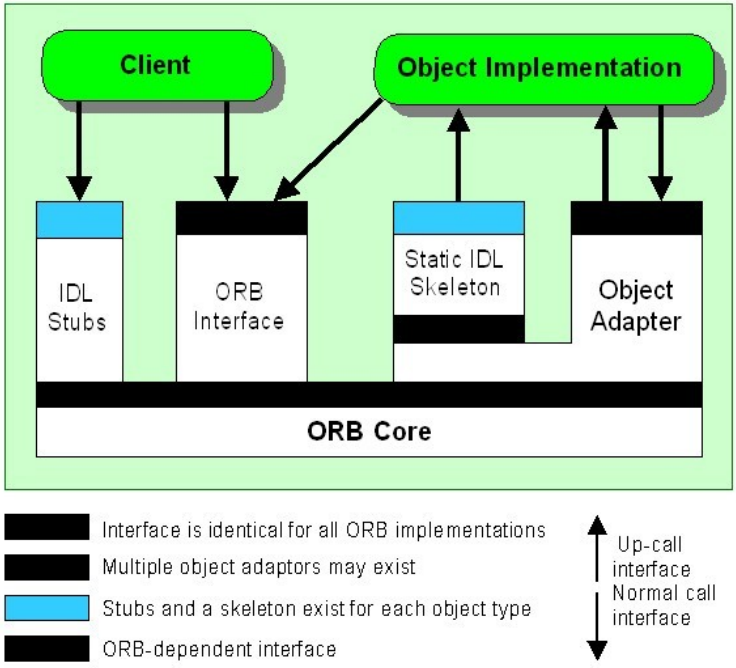


Figure 4 The Structure of Object Request Broker Interfaces

3.1.4.1 Clients and Servers

As mentioned above, the terms *client* and *server* in a distributed object context have a different meaning than the same terms used in the context of more traditional client-server computing. In CORBA, the terms refer primarily to roles played by different programs (or specific parts of programs) with respect to a particular object. The client of an object is the *processing context* from which a request is made on the object.

The term processing context is used advisedly, with some intentional ambiguity. Sometimes it may refer to the program (or process) that makes a request; it may also refer to a particular thread or a particular function from which an invocation is made. In some cases, it may refer to another object (an implementation instance) that contains a reference for the first object and makes requests on that object from within one of the containing object's methods. Though one object's methods may in fact constitute a client context for another object, there is formally no such thing as a *client object* in CORBA systems.

Likewise a *server* is the computing context in which an object is implemented. Sometimes the word server is used to indicate the object itself; other times it may denote the process in which an object resides. In general, its ambiguity is similar to that of the term *client*. Note again that the terms client and server apply to *roles* that components play, not the components themselves. Any given program may

simultaneously be a client of some objects and a server for other (or the same) objects.

3.1.4.2 Object References

The meaning of the term *object reference* is relative to the context in which it is used. When used in a programming context in the ORB, an object reference takes the form of an Ada interface. Programmatic object references may also be converted into character strings, which may be later converted back into object references. These strings capture the information model encapsulated in the programmatic reference. Even though the string is not usable as a reference in a program, it is thought of as an object reference because it potentially locates and identifies a particular implementation instance.

The term object reference may be used to denote the abstract concept of an object's identity and location. In the process of handling requests, the ORB maintains internal data structures that it uses to locate, identify, and connect to the target objects. Since these structures are opaque to ORB users, they may be discussed only as an abstraction. One might say, for instance, that an object reference is passed from a client to a server as a parameter in an invocation. The thing being passed inside the ORB is neither the stub nor the reference in string form. Though you may not know its concrete form, it is sometimes useful to refer to this abstraction in discussions as an object reference.

3.1.4.3 First Class Objects, Local Objects and Pseudo Objects

In CORBA terminology, a first class object is a fully functional CORBA object supporting all of the attributes ascribed to regular CORBA objects:

- It has a unique identity assigned and managed by the ORB
- The ORB can supply references to the object that can be used by remote clients to make invocations on the object through the ORB
- It supports at least one CORBA interface described in IDL
- Its references support all of the operations defined on *CORBA::Object*
- It behaves in a manner consistent with general descriptions of objects in the CORBA specification

A first class object may also be referred to as a *righteous* object.

For various reasons, the CORBA specification and some CORBAservices specifications define programming interfaces that, while object-oriented in style, cannot satisfy the requirements of a first-class object. In some cases the object is, of necessity, local to the process in which it is used, it is a local object. In other cases the interface cannot be properly expressed in IDL: it is a pseudo-interface. Local objects have a limitation: they cannot be remotely accessed. In general, pseudo interfaces are used to provide APIs for ORB components or utility objects specific to ORB or service functions, such as the ORB interface itself . Pseudo interfaces generally become programming objects in the language mappings (that is, a class in Ada), but

do not support required righteous object behaviours, such as:

- They cannot be remotely accessed
- They do not have real object references (although they do have programmatic references)
- They do not support *CORBA::Object* operations

Another characteristic of pseudo objects is that their interfaces are often described in pseudo-IDL, or PIDL. PIDL is not really a language at all; it is more of a dialect of IDL that is used to describe interfaces for pseudo objects in a convenient, familiar manner, while recognizing that the PIDL need never actually be compiled into stubs and skeletons. Because this is the case, some pseudo interfaces described in PIDL contain syntax or data types that are not legal IDL but are intended to describe interface elements that are not allowed for righteous objects (hence, the need for pseudo objects). The following subsections describe some of the more important pseudo and local-objects.

3.1.4.3.1 The ORB Pseudo Object

The definition of ORB - given above - described the ORB as an abstract functional entity that mediates requests. The CORBA specification also desc ribes a programming interface called the *ORB pseudo object*. This interface supports operations that interact with the computing environment provided by the CORBA implementation (the ORB in the abstract sense) such as initialization, and operations that perform utility functions, such as converting object references to and from strings. Although this pseudo object interface is called the ORB and it is a component of the abstract ORB entity, do not confuse the ORB pseudo object with the actual ORB, or infer from the way the interface is described that the ORB is a physical, identifiable object.

3.1.4.3.2 Object Adapters

The CORBA specification describes local objects called *object adapters* that provide part of the interface between the ORB and object implementations. In particular, CORBA specifies an interface for the POA. The POA interface supports the following capabilities:

- It allows implementations to associate ORB-managed object identities with instances of user-supplied implementation classes
- It allows an implementation to inform the ORB that it (or one of its instances) has undergone a state change that affects its relationship with the ORB, such as activation (that is, the implementation or object is prepared to receive requests) or deactivation (the object is not available to receive requests)

3.2 Portable Object Adapter

The Portable Object Adapter is the link between the ORB and individual servants created in various programming languages. It is responsible for creating object references and for routing requests from the ORB to the appropriate servant.

The CORBA specification defines the Portable Object Adapter (POA) with the following features:

- source-level portability between ORB products
- allows multiple and distinct instances of the POA to exist in a server
- allows individual servants to support multiple object identities simultaneously
- provides a mechanism by which policy information can be associated with individual POA instances
- supports both persistent and transient objects
- supports object implementations that inherit from static skeleton classes, as well as Dynamic Skeleton Interface (DSI) implementations (DSI is not supported by the OpenFusion RTOrb Ada Edition)

All references to the POA in this section regard POA characteristics as defined in the CORBA specification.

3.2.1 How the POA Works

In simplistic terms, after the client obtains an object reference it invokes a request on that object. That request is transmitted via the ORB to the server application. Refer to Figure 5, *Request Dispatching*. The POA is responsible for routing the request to the appropriate servant, which incarnates the target object responsible for processing the request.

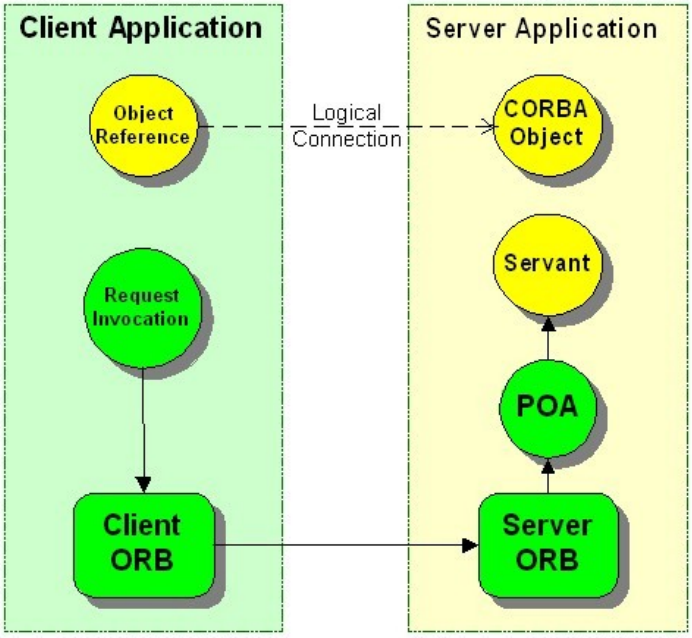


Figure 5 Request Dispatching

The POA maintains an association between the `ObjectId` (embedded in the object reference) and the servant (a programming language implementation of a CORBA object). This association is maintained in a table called the Active Object Map. When a request is received, the object adapter looks at the `ObjectId` that came with the request and finds the servant associated with that `ObjectId` from its Active Object Map. Then it dispatches the request on that servant. A CORBA server process can contain a number of different POAs, each having their own Active Object Map. POAs are created in a hierarchical fashion, with the special `RootPOA` serving as a common ancestor to all other POAs.

The ability to create multiple POAs and to set characteristics on the POA using policies allows you to control POA behaviour and, consequently, the scalability and performance of your application.

3.2.2 POA Policies

Key to the POA definition is the ability to create multiple POAs and to customize each instance by setting policies. In general, you will define a list of policies, then assign them to a POA when it is created. Once a POA is created with an assigned set of policies, those policies cannot be changed for the life of the POA. A new POA does not inherit policies from its parent POA.

Interfaces that define policies to be assigned to a POA must be derived from

`CORBA::Policy`.

3.2.2.1 Standard POA Policies

The following standard POA policies are defined by CORBA.

3.2.2.1.1 Lifespan Policy

`POA::create_lifespan_policy` allows you to specify the lifespan of objects.

- `TRANSIENT` objects cannot outlive the processes in which they are first created.
- `PERSISTENT` objects can outlive the process in which they are created. The default value for this policy is `TRANSIENT`.

Setting the `TRANSIENT` policy does not prevent explicit reactivation of a servant with the same object key. Change the object keys to enforce transient behaviour. The easiest way to do this is to create new POAs for servant reactivation.

3.2.2.1.2 Object Id Uniqueness Policy

`POA::create_id_uniqueness_policy` specifies whether servants activated by the POA must have unique `ObjectIds`.

- `UNIQUE_ID` specifies that each servant activated by that POA can support only one `ObjectId`.
- `MULTIPLE_ID` specifies that servants activated by that POA can support more than one `ObjectId`.

The default value for this policy is `UNIQUE_ID`.

3.2.2.1.3 Id Assignment Policy

`POA::create_id_assignment_policy` specifies whether ID assignment is performed by the POA or by the application.

- `SYSTEM_ID` specifies that the POA generates and assigns `Object Ids`.
- `USER_ID` specifies that `ObjectIds` are assigned by the application.

The default value for this policy is `SYSTEM_ID`.

3.2.2.2 POA Policy Extensions

OpenFusion RTOrb Ada Edition provides the following extensions to the

CORBA-standard POA policies for use with the POA in "Enterprise" mode (they are useless in RT mode):

3.2.2.2.1 ThreadCount Policy

`POA.create_thread_count_policy` specifies the number of extra threads that may be needed to fluently handle the requests for this POA.

The default value for this policy is `5` for the root POA and 0 for its descendants.

3.2.2.2.2 ProtocolPolicy

OpenFusion RTOrb Ada Edition provides a pluggable transport mechanism. The `POA.create_Protocol_Policy` is used to select and configure one or more communication protocols used by a POA.

3.2.2.3 POA Policy Summary

All POA policy objects are locality constrained (they are local objects); that is, you cannot pass their references as arguments to normal CORBA operations or convert them to strings using `ORB::object_to_string`. They can be accessed only within the context of the ORB in which they were created.

Once you define the policies to be assigned to a POA, you can create the POA by calling `create_POA` on an existing POA. The new POA becomes the child of the POA on which the call was made. `create_POA` takes three arguments: the name for the new POA, a reference to the POAManager for that POA, and a list of policies to be applied to the new POA. If no POAManager is specified, a new POAManager is created.

3.2.3 POA Manager

The POAManager controls the flow of requests to one or more POA objects. The POAManager interface supports operations to change the state of a POA to one of the following:

POA State	Meaning
ACTIVE	Calling activate on the POAManager allows requests to flow to the POAs that it controls.
HOLDING	Calling hold_requests on the POAManager allows requests to be blocked by the POAs that it controls.
DISCARDING	Calling discard_requests on the POAManager allows requests to fail with a TRANSIENT system exception with standard minor code 1returned to the client by the POAs that it controls.
INACTIVE	Calling deactivate on the POAManager allows requests to fail with an ADAPTER_INACTIVE system exception returned to the client by the POAs that it controls.

3.2.4 Object References, Keys, and IDs

The POA is responsible for creating an object reference, which the client can use to contact the target object. The object key is embedded within the object reference and the object identifier is embedded within the object key. The policies you set on the POA determine whether or not your application controls the content of the `ObjectId` and whether servants can support multiple IDs. `ObjectId`s must be unique within each individual POA; however different POAs can assign the same `ObjectId`.

3.2.5 Servants

The IDL compiler generates server-side skeleton and servant classes. These skeletons are internal classes needed by the POA to call your servant classes. Servant classes are obliged to implement all of the functions declared in their generated specification. Servants are responsible for incarnating CORBA objects. A servant is an Ada instance used to service a request.

3.2.6 Object Creation and Activation

A CORBA object must be created and activated before the client can invoke operations on it. The POA remembers the relationship between the object and the servant which created it.

Depending on the policies set on the POA, you will either:

- use `POA::activate_object or POA::activate_object_with_id` to activate the object. Once the object is activated, the POA can dispatch requests arriving for that object. After activation, you may use the `POA::servant_to_reference()` operation to obtain an object reference from the servant.

or

- use `POA::create_reference_with_id` to create an object reference without activating it

Use deactivate object to remove the association of the object with its servant.

3.2.7 Request Processing

When the ORB receives a request, it attempts to locate the appropriate POA and deliver the request. It uses the received object reference, which contains the `ObjectId` and POA identification, to locate the appropriate server and POA within that server. The request is then handed off to the POA.

The POA now takes over and tries to locate the target object. The POA searches for the servant associated with the `ObjectId` in its Active Object Map or through its servant adapters. Once a reference to the servant is obtained, the appropriate method is invoked. Otherwise, an exception is thrown.

3.2.8 Designing an Application

OpenFusion RTOrb Ada Edition fully support the dynamic features supported by the full-version of the CORBA specification. As a result, it support certain features, such as *activation on demand*, which must be taken account of when designing servers. For example, POAs can be activated on demand because OpenFusion RTOrb Ada Edition fully supports adapter activators . The server can also be designed to dynamically activate objects when needed using servant adapters.

CHAPTER

4 Introduction to Real-time CORBA

This chapter introduces the essential aspects of the Real-time CORBA ORB.

Please note that real-time CORBA examples are provided in the OpenFusion RTOrb Ada Edition distribution's html pages.

4.1 Real-time Specification

The *Real-time CORBA Specification* defines a set of real-time extensions to standard CORBA specification.

Figure 6 shows the key Real-time CORBA entities. The features that these relate to are described below.

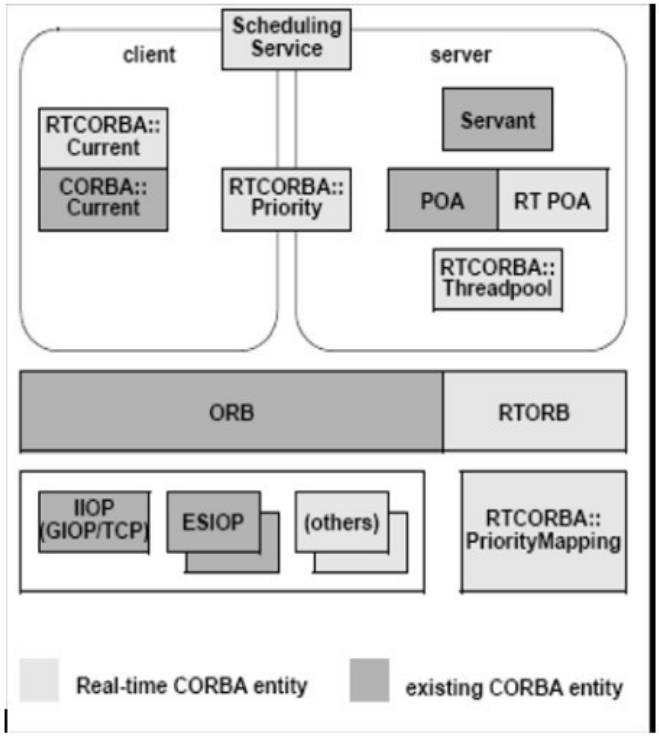


Figure 6 Real-time CORBA Extensions

4.1.1 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.)

4.1.2 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.

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

4.1.4 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 prioritised 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.

4.1.5 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.

Real-time CORBA defines a PriorityMapping interface in order to allow the Real-

time ORB and applications to do both of these things.

4.1.6 Real-time CORBA Current

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

4.1.7 Priority Models

One goal of Real-time CORBA is to bound and to minimize priority inversion in CORBA in vocations. One mech an ism that is employed to achieve this is prop agatio n o f the activ ity p riority fro m 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 honours 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.

4.1.8 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 Ada implementation need not provide an implementation of `Mutex` as Ada already provides a far better solution through the use of protected objects. This allows a consistent priority inheritance scheme to be delivered across the whole system.

4.1.9 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.

4.1.10 Priority Banded Connections

In order 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.

4.1.11 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.

4.1.12 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.

4.1.13 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.

4.1.14 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 CORBA policy application points (ORB, POA, Current) that 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.

4.2 Real-time Portable Object Adapters

Real-time Portable Object Adapters (RTPOA) configuration is one of the most important features in real-time CORBA. Application developers can configure and control hardware resources using real-time policies associated with real-time POAs. This section describes priority models, the pluggable RTPOA, threads and threadpools, and priority banded connections.

4.2.1 Priority Model

OpenFusion RT Orb Ada Edition only supports both the `RTCORBA::SERVER_DECLARED` and `RTCORBA::CLIENT_PROPAGATED` priority models. Refer to the CORBA Priority Model example included in the RTOrb examples to see how to set the `RTCORBA::SERVER_DECLARED` priority model policy for an `RTPOA`.

4.2.2 RTPOA

The RTPOA module which extends the standard POA interface with respect to priority and resource configuration.

4.2.2.1 POA Activation Methods with Priority

```
function create_reference_with_priority
   ( Self : in Ref;
     intf : in CORBA.RepositoryId;
     priority : in RTCORBA.Priority)
   return Corba.Object.Ref ;

function create_reference_with_id_and_priority
   ( Self : in Ref;
     oid : in PortableServer.ObjectId;
     intf : in CORBA.RepositoryId;
     priority : in RTCORBA.Priority)
   return Corba.Object.Ref ;

function activate_object_with_priority
   ( Self : in Ref;
     p_servant : in PortableServer.Servant;
     priority : in RTCORBA.Priority)
   return PortableServer.ObjectId ;

procedure activate_object_with_id_and_priority
   ( Self : in Ref;
     oid : in PortableServer.ObjectId;
     p_servant : in PortableServer.Servant;
     priority : in RTCORBA.Priority) ;
```

4.2.3 Threads and Threadpools

There are two basic ways of manipulating threads in RT CORBA, `RTCORBA::Current` and Threadpools (via policies at POA creation time).

4.2.3.1 Current

RT CORBA defines a RTCORBA::Current interface to provide access to the CORBA priority of a thread. Please refer to the CORBA Priority example included with this product on how to access the priority of a thread.

4.2.3.2 Threadpools

Thread pools are one of the most important features in Real-time CORBA. Threads in pools can be pre-allocated and partitioned amongst active Real-time POA's. Application developers and end-users configure and control processor resources using thread pools. The possibility of experiencing priority inversion can be bounded and

reduced by configuring real-time POA's with threadpools where each POA associates with one thread pool (see *Figure 7*). Note that threadpools are independent of the POA lifecycle.

4.2.3.3 Thread Pool Operation Basic Mode

Application developers and end-users configure and control processor resources using thread pools (see *Figure 7*). Threads in the threadpool execute requests at the object priority for which each request is targeted. Each POA associates with one thread pool. However, you are reminded that thread pools are independent of the POA lifecycle.

To dispatch requests to the correct queue and to the right servant on the server side, each request needs to be handled by the right priority thread. To achieve this, requests are pushed onto the queue of appropriate priority and are processed synchronously by the waiting threads within a lane. There is a queue assigned to each thread pool.

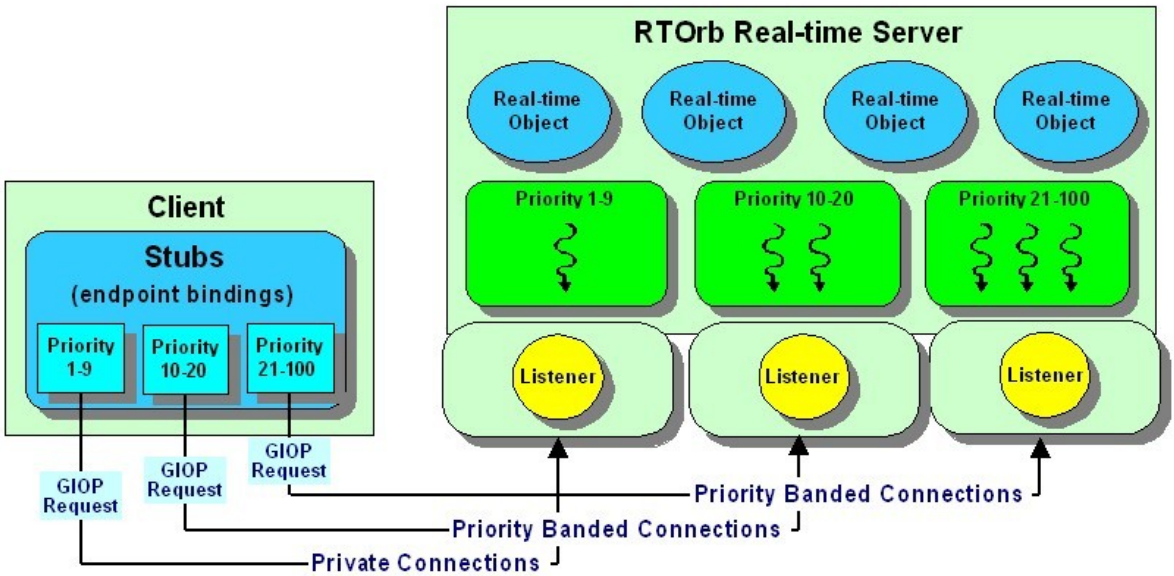


Figure 7 Controlling Network Resources

The client side may hold multiple connections open through the use of individual object references to end points in the server, based on priority band.

Threadpools can be configured for use with RTPOA's in one of two forms:

- Non-laned Threadpool
- Laned Threadpool

4.2.3.4 Laned Threadpool

A threadpool can be created that has *n* partitions (lanes) each created to serve requests at a specific priority. Each lane has m static threads running at the priority defined for the lane. Whenever a request arrives, a lane is chosen based on the priority associated with the activated object. Please refer to the Threadpools example included with this product on how to create a Threadpool with lanes.

As seen above, the half-sync layer consists of a thread pool associated with a POA. A thread pool can be shared among POAs.

4.2.3.5 Priority Banded Connections

RT CORBA introduces the concept of priority banded connections. A real time POA (RTPOA) supporting priority banded connections is capable of accepting requests across transport with some concept or awareness of the requestors priority at which the server should execute. Each client can open a number of connections with a server, each connection handling a range of priorities defined in the priority banded connection policy.

Priority banded connections are useful when used in conjunction with a transport protocol that does not respect priorities. Transports like TCP that are not easily pre-emptable and do not respect priorities can incur head of line blocking where requests of higher priority are blocked and unable to pre-empt requests at lower priority. This leads to unbounded delays and the potential of priority inversion. Priority bands allow multiple connections to be utilized to minimize the head of line blocking that can occur where one connection is used for multiple priority requests. An RTPOA th at is configured with laned th readpools and prio rity banded connections can provide more predictability. Please refer to the Connections example included with this product on how to create priority banded connections.

4.2.4 RTPOA Current

This interface is available to perform operations to access the identity of the object on which a call was invoked. This is supplied for supporting servants that may implement multiple objects.

4.2.5 Associations Between Pools and RTPOA

Each POA must have one thread pool attached to it. This is done by passing a thread pool policy to the POA. In the case where no policy is specified or an invalid threadpool identifier is used, the ORB will use the default thread pool. One thread pool can be shared among multiple POAs. The default pool must be created and set before the first inquiry for the RTRootPOA.

4.3 Priority Machinery

Priority is the medium used to achieve QoS in real-time CORBA, hence the focus of RTOrb application design. With the RTOrb priority scheduling is achieved via the RTOS scheduler. Tasks or threads that comprise the application execute in a stable, predictable manner as a result of this priority scheduling. In addition if using only the RTOS for scheduling purposes, it must provide proper mutexes and semaphores to resolve resource contention, such as priority-aware application objects and/or code segments.

The central theme in real-time CORBA programming is the notion of prioritised scheduling of activities, tasks, or threads.

This section provides:

- background information on the phenomenon of priority inversion
- discussion of protocols used to overcome priority inversion
- discussion of priority mapping and CORBA priority scheme

4.3.1 Priority Phenomena and Protocols

Priority inversion is a commonly known phenomenon in real-time systems. It usually manifests in the form of unbounded delays of high priority tasks. Normally, when priority inversion occurs, high priority tasks are forced to wait on low priority tasks. This occurs when the high priority tasks are sharing common resources with low priority tasks. If a low priority task locks the resource for its own use but is pre-empted by a higher priority task, which also needs access to the common resource, the high priority task will have to wait on the lower priority task.

To illustrate the concept of priority inversion more clearly, consider *Figure 8*. Here, 3 tasks or threads are executing, T1, T2, T3. The tasks are illustrated in order of decreasing priority such that the priority of T1 is the greatest and that of T3 the least of the 3. In addition, we assume that T1 and T3 share a common resource, such as a critical section, to which only one can have access at any point in time. The following is a typical scenario illustrating priority inversion.

At time t0 task T3 starts to run. At time t1 task T3 locks and enters a critical section, continuing to execute until time t2. The portion of time for which task T3 is in a critical section is shown as shaded. At time t2, task T1 pre-empts task T3 because T1 has a higher priority. Task t1 now executes from time t2 until time t3, at which point it attempts to gain access to the critical section, which has previously been locked by lower priority task T3. Task T1 is therefore forced to wait or block until such time as T3 releases the lock on the critical section shared between T1 and T3. Task T3 is allowed to run next. So at time t3, task T3 resumes execution and continues to work its way through the critical section.

Now at time t4, task T2 pre-empts task T3 and starts to run because task T2 has a higher priority than that of task T3.

Task T1 is now blocked by task T3 because of the shared resource, and task T3 is blocked by task T2. Therefore T2 is now indirectly blocking task T1 as well. Task T2 blocks task T3 until T2 completes at time t5. As a consequence T3 is forced to block for a significant amount of time (the length of the shared critical section plus the execution time of task T2).

For an actual system, when several medium priority tasks exist with priorities greater than that of task T3 but less than that of task T1, it can lead to unbounded delay or blocking.

This effect is known as priority inversion and occurs in the time interval t3 to t6.

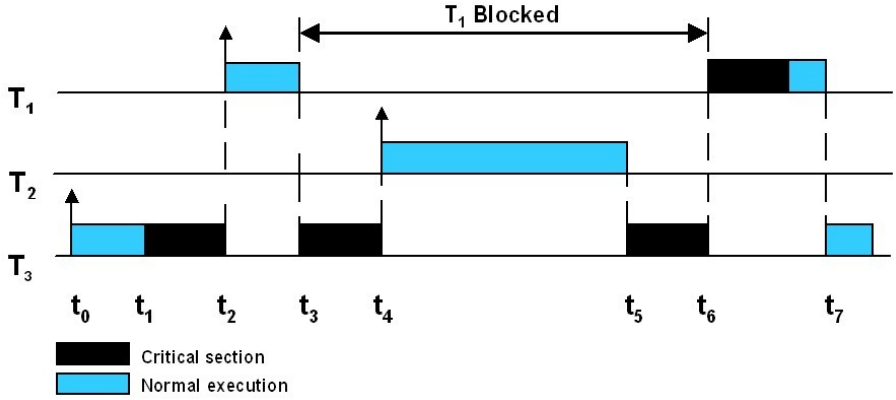


Figure 8 Priority Inversion

The priority inversion phenomenon in real-time systems is one that can manifest any time several tasks want to execute in the presence of services that are shared among them.

Several approaches have been proposed to alleviate the priority inversion phenomenon in real-time systems and much literature is available. A complete description and analysis is beyond the scope of this document. The reader is directed to further reading under *Bibliography* on page 93, particularly Buttazo.

The Real-time Extension aids the RT CORBA developer by providing priority inheritance protocols in the ORB. Specifically, RTOrb's RT CORBA mutex supplies a default implementation that uses the simple priority inheritance protocol as an example. Other protocols are also possible, but this is used to illustrate the concept and its applicability.

The priority inheritance protocol bounds any priority inversion that could possibly

occur. Although the ORB's initial design is such that it tries to eliminate the possibility, it can still occur as a result of unusual transports, or hardware specifics that are used in a particular setup.

Figure 9 and the following text explain how priority inheritance protocols bound any possible priority inversion. The same three tasks are illustrated as in *Figure 8*. Additionally, the relative priorities of the three tasks are depicted at the bottom of the figure as P_1, P_2, and P_3.

Up to time t_3, the behaviour of tasks T_1 and T_3 are the same as in *Figure 8*. At time t_3, T_1 is forced to block on T_3 due to T_3 holding a lock on a critical section to which T_1 needs access. At this point the mechanism of priority inheritance is employed. This mechanism causes T_3 to inherit the priority P_1 of task T_1, which forces task T_3 to execute immediately and run through the remaining part of its critical section. This forces T_3 to execute from t_3 to t_5 at the T_1 priority P_1, which is the highest priority in this illustration. Note that task T_2 cannot pre-empt task T_3 as task T_2 has lower priority than the temporarily assigned priority (P_1 of task T_3, through priority inheritance).

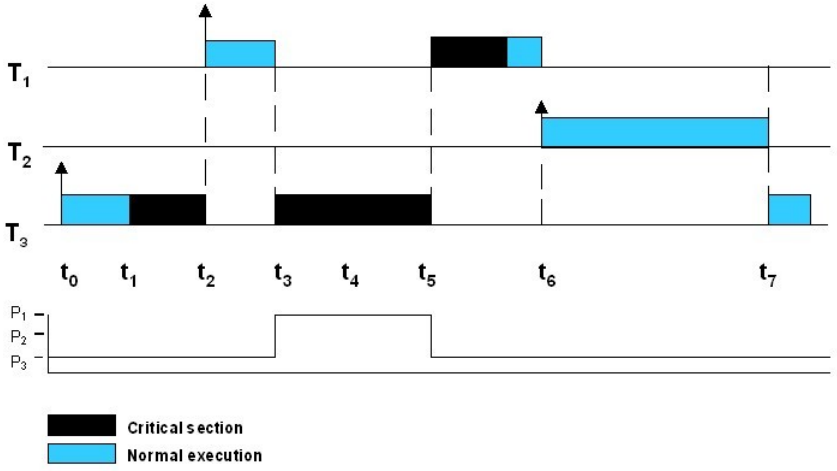


Figure 9 Priority Inheritance Protocol to Bound Priority Inversion

As task T_3 exits the critical section, its priority is returned to its original value P_3 as shown in *Figure 9*. At time t_5, task T_1 can run because priority P_1 is greater than priority P_2 of task T_2. Thus it no longer needs to block on task T_3, which was holding a lock on the critical section. Task T_1 now runs through the critical section and to completion at t_6. At time t_6, task T_2 has the highest priority and executes as shown in *Figure 9*.

4.3.1.1 CORBA Priority

CORBA uses a standard (canonical) form of priority that can be mapped to any RTOS priority scheme. In effect, CORBA su bsumes th e heterogen e ity in RTOS-specific priority schemes and thus achieves uniformity. This allows CORBA invocations to be made across multiple, different RTOS platforms - which may have different native priority schemes - in a consistent manner. Therefore, CORBA priority is a wrapper for native priority schemes.

4.3.1.1.1 CORBA Priority Mapping

Priorities may be mapped from the CORBA priority scheme to the RTOS native priority scheme. This is accomplished with an interface defined in IDL, and allows you to forward and reverse map CORBA and native priorities as shown in *Figure 10*.

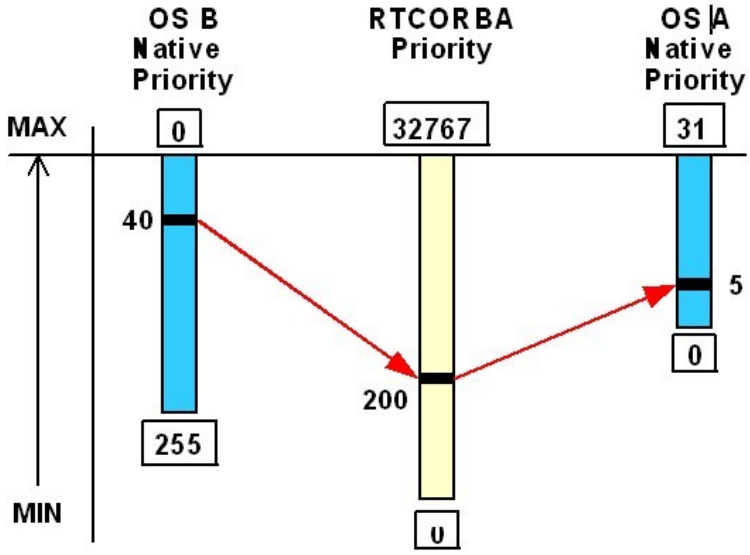


Figure 10 Priority Mapping

An `RTCORBA` priority type id, defined in IDL to be of type CORBA short, is as follows:

```
module RTCORBA {
typedef short Priority;
const Priority minPriority = 0;
const Priority maxPriority = 32767;
};
```

It spans the interval 0 to 32767. Higher values of `RTCORBA` priorities map to higher native RTOS priorities.

4.3.1.1.2 RTCORBA Current Interface

The `Current` interface in RTCORBA allows a developer access to the priority data of the current locus of execution or thread. The interface allows for setting and getting a thread's CORBA priority.

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

A thread has native base and elevated priorities, which may be different than the observed CORBA mapped value.

This is a local interface, which also stores information about its current CORBA and native priorities in a thread-local storage structure. It is a singleton within the context of its present locus of execution. A typical application's use of the RTCORBA current interface is illustrated below: Please refer to the CORBA Priority example included with this product on how to use the RTCurrent get and set methods, and use of the default priority mapping.

CHAPTER

5 Introduction to Real-time Systems

This chapter expands on the short introduction given earlier and introduces some of the essential aspects of real-time systems programming.

5.1 Real-time Systems

The term *real-time* is used to define systems where the time taken for the execution of a task is temporally deterministic (predictable). This yields, at the task level, the notion of *hard* deadlines: a task must complete within the specified time. Thus a real-time system executes tasks in a predictable manner with respect to time.

The degree of predictability is the basis for the terminology used to describe real-time systems. Widely used categories are *hard* real time and *soft* real time. This degree-of-predictability classification conveys relative descriptive utility, but more precise definitions are implied for a given application.

In hard real-time systems, task execution that completes at an incorrect time means system failure. A missed deadline is the same as a wrong answer.

In soft real-time systems, task execution that completes at an incorrect time means reduced system performance. A missed deadline is not catastrophic, but rather degrades system performance.

Examples of hard real-time activities are:

- flight control (inertial guidance and navigation)
- nuclear power plant control
- pacemakers (human heart)
- vehicle anti-lock braking
- air-bag deployment systems

Examples of soft real-time activities are:

- command interpretation of inputs from a user interface
- saving or displaying management data
- ship navigation
- certain types of telecommunications traffic shaping functions

In general, real-time applications consist of soft and hard deadlines. Operating systems try to guarantee the individual timing constraints of the hard deadline tasks while attempting to minimize the average response times of the soft ones. Real-time operating system (RTOS) kernels achieve this through the use of appropriate features:

- near constant time system calls
- the ability to associate priority not only with the threads (or tasks) executing, but also the synchronization constructs such as mutates
- pre-emption to achieve greater determinism
- appropriate scheduling strategies

5.1.1 Time- and Event-Triggered Systems

Another way to classify real-time systems is based on whether they are time-triggered or event-triggered. A trigger is an event that causes the start of some action, for example, the execution of a task or the transmission of a message.

There are two distinctly different approaches to the design of real-time computer applications: the event-triggered (ET) approach, and the time-triggered (TT) approach. A triggering mechanism is used to start communication and processing activities in each node of a computer system (network).

In the ET approach, all communication and processing activities are initiated upon occurrence of a significant change of state. The regular event of a clock tick is not such an event. In the TT approach, all communication and processing activities are initiated at predetermined times. While ET systems are flexible, TT systems are temporally predictable. In this guide, the systems discussed are event-triggered.

5.1.2 Developing Real-time Systems with RTOS

Real-time Operating System (RTOS) kernels are built to support real-time tasking through a number of important features that real-time systems use:

- priority based scheduling to perform real time inter-kernel process management
- priority aware synchronization constructs (semaphores for instance)
- concurrency constructs such as multi-tasking or multi-threading
- real-time clock for a time reference for internal kernel task management and housekeeping tasks
- mechanisms for inter-process and intra-process communication with associated synchronization primitives
- bounded, constant-time fast context switch, and often an associated minimal base kernel size (typically 16-32kb)
- internal kernel architecture geared to respond to external interrupts in a fast manner, and so separate their execution from intra-kernel tasks

Pre-emption and priority-based scheduling are the most important characteristics of

real-time kernels. Together they give rise to the notion of priority, the central mechanism used to achieve predictable, deterministic behaviour. These characteristics are sufficient for soft real-time systems. Behavioural characteristics include quick response and small execution times for higher priority tasks - while yielding small average response times for other tasks. For hard real-time applications however, a centrally important theme is missing in such kernels. It is the notion of some form of guarantee, which is necessary for time-critical, hard real-time behaviour.

To achieve hard real-time, distributed applications, the most important properties of a distributed, mission-critical system RTOS and ORB tuple are:

- *predictability* - The RTOS must be able to predict in an *a priori* fashion the consequences of scheduling any and all tasks under its control. If it is not possible to guarantee an upper bound for the execution time of any task, the RTOS must be able to take an alternative course of action to cope with such events. Predictability is by far the single most important requirement on an RTOS, especially for hard real-time application hosting.
- *timeliness* - The RTOS must comprise internal clocks for effective handling of tasks with differing time constraints, and degree of importance or criticality.
- *fault-tolerance* - The RTOS should be immune (to some degree) to certain classes of hardware and software failures. Mission critical components in such high availability RTOS models should have fault-tolerance features inherent in their design.
- *design for peak load* - The RTOS should provide some continued minimal level of performance when subjected to unusually high peak loads. RTOS failure and crash under such circumstances is an unacceptable scenario for hard real-time applications. Therefore, they must be designed to cope with anticipated scenarios of high sporadic load.
- *maintainability* - The RTOS kernel and ORB need to be designed in a modular, pluggable fashion to ensure a minimal, optimised use of RTOS resources under any load. In addition, the ability to make modification/customisations to the kernel - as the ORB based application might require - should be minimally cross-coupled so as to be able to make the changes easily.

5.1.3 Predictability in Distributed Applications

Predictability of a complex, distributed, real-time application is achieved through the careful combination of RTOS features, networking transport, IPC mechanism implementations, and constant-time ORB internals design. A sum of these, yields a degree of predictability that enables some level of Quality of Service (QoS) to be furnished to the application built on the RTOS-ORB combination.

As far as the RTOS is concerned, it should be able to plot the evolution of tasks and events ahead of time in a given situation such that it can guarantee in advance that all

critical timing constraints are met by suitable scheduling of its internals. Components that contribute to the possibility of predictably scheduling deadline-restricted tasks are:

- the features and numbers of CPUs and the scheduling policies they support
- internal CPU features such as pre-fetch, pipe lining, cache memory, and direct memory access, which can contribute to non-determinism
- types of scheduling algorithms employed in the kernel
- synchronization mechanisms
- types of priority-aware semaphore
- memory management policies, especially heap management
- communication mechanism, e.g., whether the kernel is based on messages or signals
- interrupt handling mechanisms

5.1.4 Features and Non-Determinism

It is important for the distributed real-time application designer to understand the features that will most contribute to non-determinism. These are discussed briefly in the context of an RTOS and ORB.

Probably the single greatest contributor, at the ORB level, of non-determinism is a transport that is not QoS aware or priority respecting. In essence, the management of ORB, application, and RTOS internal tasks needs to be efficiently managed by the RTOS.

Perhaps the single greatest enemy of an effective hard real-time system design is the phenomenon referred to as priority inversion.

Priority inversion occurs when a high priority task (that is, of possibly greater importance and criticality) is blocked by a less critical, lower priority thread for an unbounded period of time. This type of situation is often seen when the high priority thread is trying to get access to a shared (with the low priority task) resource, which the low priority task has locked for its own use. There is much detailed real-time literature on this subject, and designs for its avoidance. For further reading, see *Bibliography* on page 93, particularly Rajkumar and Buttazo.

The integration of ORB and application tasks is under the control of the application designer, but the tasking and priority level control of the transport threads is not, and can give rise to priority inversions.

Other major contributors to non-determinism include:

- *DMA* - Certain methods of direct memory - such as cycle-stealing access, used to transfer data between devices and main memory - give rise to unbounded

delays. However, this can be overcome by using other techniques, such as time-slice methods.

- *cache* - This procedure buffers CPU-RAM exchanges in an attempt to reduce task execution times. Under certain circumstances, this can contribute to non-determinism.
- *interrupts* - These events can be sporadically triggered due to I/O devices and can impair predictability of a real time system due to the fact that they introduce unbounded delays into the execution times of other processes.
- *system calls* - The calls for hard real-time kernel primitives need to be pre-emptible and implemented to have bounded execution times. These are then used by the scheduling subsystem of the kernel to pr oduce the necessary guaranteed, temporally-correct behaviours internally in the kernel.
- *semaphores* - These should be modified to be priority aware and thus avoid the priority inversion phenomenon. RTOS' normally furnish priority protocols when implementing this modification. Examples include basic priority inheritance, priority ceiling, and stack resource policy. These protocols temporarily modify task priorities to avoid deadlock and anomalous priority assignments, which cause non-determinism.
- *memory management* - This must not produce unbounded delays in the course of execution of real-time tasks. A common practice is to use fixed, constant time type schemes to allocate, and address memory partitions to achieve predictable memory access. It is usual to see a greater degree of static allocation, which reduces flexibility for dynamic environments. The designer of real-time systems must make trade off decisions when implementing on an RTOS using languages that permit dynamic heap memory allocations, such as C++.

PROGRAMMING WITH RTORB

CHAPTER

6 Creating Applications

6.1 General

This section describes how to write the applications themselves and covers:

- How to write a simple *non real-time* application, called *Echo*. This application contains the minimal, essential elements needed to create a distributed client-server application
- How to write a simple *real-time* version of the *Echo* application. This application demonstrates basic real-time programming using RTOrb

i It is assumed that readers understand basic CORBA programming with Ada concepts and practice. The descriptions given here concentrate on those aspects which may be of most help, with basic operations (which readers should be familiar with) being only lightly covered.

6.2 A Simple Application

This example, the Echo application, is very simple: it contains the minimum elements needed to create a working client-server application using RTOrb.

- has an IDL specification in `echo.idl` which
  - declares the `Echo` interface and `EchoString` function (see 7.2.1)

- has a server which
  - performs the basic initialisation tasks required by all servers
  - creates an `Echo` servant object; the servant's single method prints a greeting for the client which called the server.
  - makes the Echo servant accessible to clients by saving the servant's stringified IOR to a file
  - listens for requests from clients

- has a client which

- performs the basic initialisation tasks required by all clients
- obtains references to the Echo servant object by reading its stringified IOR from a file
- all the *EchoString* method on the Echo object which displays a greeting with the client's name

This example uses files for object resolution. Other methods of object resolution must be used on platforms which do not have a file system.

The complete source code for the Echo application is in the *<OFRT_DIR>/examples/ada/echo* directory of the RTOrb distribution.

6.2.1 IDL Specification

The IDL specification for Echo (*echo.idl*) is very simple: it declares a single interface, *Echo*, with a single method, *EchoString*. The EchoString takes a string (the name of the client calling the method) and returns a string (a greeting with the client's name).

```
interface Echo {
  string echoString(in string mesg);
};
```

6.2.2 Running the example Echo

It is assumed you have installed and configured a compiler and a library beforehand.

WIN The Echo example is located in *<OFRT_DIR>\examples\ada\echo*. If it does not already exist, a *tmp* directory must be created in the system root (i.e. if RTOrb was installed on *C:*, then you must create *c:\tmp*)

Step 1 Compile the needed files to run the application. Run *compile.bat*, located in the Echo folder:

```
<OFRT_DIR>\examples\ada\echo\compile.bat
```

The executables files *echo_server.exe* and *echo_client.exe* are generated.

Step 2 Start the server by executing *echo_server.exe*

```
<OFRT_DIR>\examples\ada\echo\echo_server.exe
```

The console indicates the IOR \tmp\ior.dat has been created. The server is running, as long as the console it was launched in remains open.

Step 3 In another console, or through Windows Explorer, launch `echo_client.exe`

The console where the server is running indicates he has received the greeting from the client. The client and the server you have created communicate well.

6.2.3 Client-side

First, the client side packages of the interfaces of the objects this client needs to access must have been generated using idl2ada. The generated code must not be edited in any case.

6.2.3.1 Initialization

Before doing any method invocation, the client may need to start the OpenFusion orb. This is done by calling Corba.Orb.Orb_Init.

This procedure requires two parameters:

- Argv : an argument list to supersede command line parameters
- Orb_Identifier : the name of the RTOrb daemon to connect to if this name is the null string, then the command line options () are used to determine the orb (see Client and Service switches), else the RTOrb environment variable is tried. If the name is still a null string, the client uses its own ORB library only, else if the named RTOrb daemon is not found, a message is displayed and the exception Ada.Io_Exceptions.Name_Error is raised.

Example:

```
with Corba.Orb ;
procedure Client is
        Args : Corba.Orb.Arg_List ;
        Orb_Name : Corba.Orb.Orbid ;
begin
        -- Initialization of the ORB connection
        -- Use null argument list and name to allow command line options
        -- and RTOrb environment variable
        Corba.Orb.Orb_Init (Args, Orb_Name) ;
        -- Put client code here
        -- Corba processing termination
        Corba.Orb.Shutdown (Wait_For_Completion => True) ;
end Client ;
```

If needed, the client can establish a connection with RTOrb daemons (for example, an administrator who wants to interconnect all the Naming Services will read the etc/Orbs file, connect to all the RTOrb daemons, get their Naming Service root reference, and bind it in all the others).

Corba.Orb.Orb_Init can be called several times for the same daemon (in independent parts of the client). In this case, as required by the OMG CORBA specification, the daemon reference count is incremented. There must be the same number of calls to Corba.Orb.Stop to effectively terminate the connection with this daemon. Corba.Orb.Shutdown also closes all the connections wherever it is called.

6.2.3.2 Getting object references

In order to invoke methods on an object, the client needs to get a reference to this object. This reference can be obtained by different means:

- by a stringified object reference (IOR) which is transformed into a reference by calling Corba.Orb.String_To_Object
- by a corbaloc, corbaname or file URL object reference which is transformed into a reference by calling Corba.Orb.String_To_Object
- by searching the root object of the service in the initial references of the orb by calling Corba.Orb.Resolve_Initial_References
- by searching in the NamingService under an already defined name
- by calling Corba.Implementation_Repository.Get_Implementation and Corba.ImplementationDef.Get_Root_Object to start a new unshared service

- as the result of an operation on another object

If the object corresponding to the reference exists but is not active, the target orb does the job to make it active at the first method invocation.

To allow its resolution by a corbaloc URL, a reference must be added to the initial references using Corba.Orb.Register_Initial_Reference. Any server created with RTOrb can resolve the references it registers by this mean.

6.2.3.3 Tasking

RTOrb client and service library operations are tasking-safe. They are potentially blocking at the Ada task level.

Asynchronous Transfer of Control (ATC) is not supported during method invocation. This would introduce a big performance penalty due to the mandatory extra actions which would be needed, and in most of the cases useless (when ATC is not used or pending).

If ATC has to be used, the client implementor should use an extra task to perform the method invocations while ATC is pending.

6.2.3.4 CORBA exception handling

CORBA exceptions carry information when they are raised. To get these data, an exception handler must have a choice parameter specification (see Ada RM95 11.2) and must be specific to the raised exception. The 'Get_Members' method associated with the exception can then retrieve the data. Several exception handlers can retrieve the data if the exception is re-raised with the simple instruction 'raise ;' (e.g. if another exception is re-raised, the access to the data is lost).

The CORBA exception Impl_Limit is raised during method invocation if a non CORBA exception is raised during the execution of the method in the server side.

Example (taken from a Naming Service context operation) :

```
exception
when NotFound_Exception : CosNaming.NamingContext.NotFound =>
  Ada.Text_Io.Put_Line ("Exception : Not Found");
  declare
    Members : CosNaming.NamingContext.NotFound_Members :=
      CosNaming.NamingContext.Get_Members (NotFound_Exception);
  begin
    Ada.Text_Io.Put_Line ("Reason : " &
      CosNaming.NamingContext.NotFoundReason'Image (Members.Why));
end;
```

6.2.3.5 Termination

Corba.orb.Stop must be called to close the CORBA session. If Corba.orb.Orb_Init was called several times, the session will be effectively closed after Corba.orb.Stop is called the same number of times, in order to protect the CORBA work from the other tasks of the process. The standard procedure Corba.orb.Shutdown forces the termination of the session.

If Corba.orb.Stop or Corba.orb.Shutdown are not called, the RTOrb daemon itself (if used) will do the cleaning work when the connection is closed (for example at least when the client is killed).

6.2.4 Server-side

6.2.4.1 Initialization

```
Corba.Orb.Orb_Init (Args, Orb_Name) ;
```

6.2.4.2 Getting the Root POA and activating it

```
Corba.Orb.Resolve_Initial_References
      (Corba.To_Unbounded_String ("RootPOA"), Root_Poa) ;
Manager := Corba.PortableServer.Poa.Get_The_POAManager (Root_Poa) ;
Corba.PortableServer.PoaManager.Activate (Manager) ;
```

6.2.4.3 Creating and associating the object

```
Root_Ptr := new Echo.Impl.Object ;
Corba.PortableServer.Poa.Servant_To_Reference
      ( Self      => Root_Poa,
        P_Servant => Corba.PortableServer.Servant (Root_Ptr),
        Result    => Root_Ref);
```

6.2.4.4 Making the object available

You can make the object available either as an initial reference or having its reference stored in a file.

As an initial reference:

```
Corba.Orb.Register_Initial_Reference
   ( Id     => Echo.Tgx_Service_Name,
     Object => Root_Ref) ;
```

Store its reference in a file:

```
Ada.Text_Io.Create ( File => File,
                     Mode => Ada.Text_Io.Out_File,
                     Name => "/tmp/ior.dat") ;
Ada.Text_Io.Put (File, Corba.To_String
                    (Corba.Orb.Object_To_String (Root_Ref))) ;
Ada.Text_Io.Put (File, Character'val (0) ) ;
Ada.Text_Io.Close (File) ;
Ada.Text_Io.Put_Line
   ("The reference of the object is written in /tmp/ior.dat") ;
```

6.2.4.5 Starting method processing

The current thread is blocked with this call. It will only exit when Orb.shutdown is called from another task.

```
Corba.Orb.Run ;
```

Invocation processing is now started.

BIBLIOGRAPHY

Bibliography

The documents and articles listed below are referred to in the text or are recommended reading.

[1] *A Comprehensive Source of Information on Real-time Systems and Desig*n, Jensen D.,*http://www.real-time.or*g.

[2] *Concurrent and Real-Time Programming in Jav*a, Andy Wellings, John Wiley & Sons Ltd., 2004.

[3] *Patterns for Concurrent and Networked Object*s, Pattern Oriented Software Architecture -Volume 2, Schmidt D., et. al.,. J Wiley, 2000.

[4] *Predictable Scheduling Algorithms and Application*s, Hard Real-time Computing Systems, Buttazo G., Kluwer Academic Press, 1997.

[5] *Programming for the Real Worl*d, Posix.4, Gallmeister B.O., O'Reilly and associates, 1995.

[6] *Real-Time Java Programmin*g, Peter C. Dibble, Sun Microsystems Press Java Series, 2002.

[7] *Real-Time Specification for Java (RTSJ) v1.0.*1, Rudy Belliardi, et. al., http://www.rtsj.org

[8] *Real-Time Specification for Jav*a, Bollella G., et. al., Addison Wesley, 2000.

[9] *Real-Time Systems and Programming Language*s, Alan Burns and Andy Wellings, Addison Wesley, Third Edition, 2001.

[10]*Real-Time Systems: Design Principles for Distributed Embedded Application*s, Kopetz, H., Kluwer Academic Press, Fourth Edition, 1997.

[11]*Sun Java Real-Time Syste*m, *http://java.sun.com/j2se/realtim*e, Sun Microsystems.

[12]*Synchronization in Real-time Systems: A Priority Inheritance Approac*h, Rajkumar R., Kluwer Academic Press, 1991.

[13]*What is Predictability for Real-time System*s, Stankovic J.A. and Ramamritham K., Journal of Real-time Systems, Issue 2, 1990.

INDEX

Index

A

B

C

D

E

F

I

L

M

N

O

P

Q

R

S

T

U

V