Spectra ORB
C Edition
User Guide
Version 2.0

Guide edition: 05 (01/04/14)

Table of Contents

1 Preface

About the User Guide

The *Spectra ORB (formally know as e*ORB) C Edition User Guide* provides instructions and information needed to program using the product.

The *User Guide* should be read in conjunction with the *Product and Installation*, *Reference* and *IDL Guides*, as well as the other documents included with the product; please refer to the *Product and Installation Guide* for a complete list of documents.

Intended Audience

The *User Guide* is intended to be used by developers who use Spectra ORB to develop CORBA-based distributed applications.

Organisation

The User Guide is logically organized into two major parts. The first part of the guide describes the CORBA architecture including the Portable Object Adaptor (POA). In the second part of the guide the specifics of the Spectra ORB implementation are covered.

The first part of the guide is preceded by an *Introduction*, providing a high level description of the ORB's features. At the end of the guide Appendices are included providing details of each of the CORBA profiles supported by the product, there is also a *Bibliography*, listing a recommended set of useful resources.

The *CORBA* section of the guide is sub-divided into the following topics:

- Chapter *3, CORBA Basics*, provides a basic introduction to CORBA
- Chapter *4, Portable Object Adapter*, describes how to use the ORB's Portable Object Adapter

The section detailing product specifics is organised into these topics:

- Chapter *5, CORBA Extensions*, describes the additional CORBA features support by Spectra ORB which are not specified in the supported Profiles
- Chapter *6, Additional Features*, describes additional Spectra ORB proprietary features
- Chapter *7, Extensible Transport Framework*, covers Spectra ORB's Extensible Transport Framework (ETF)
- Chapter *8, Using the ORB*, provides information necessary to create applications which use Spectra ORB

Conventions

The conventions listed below are used to guide and assist the reader in understanding the 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.* XP, Vista, Windows 7) only

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

C C language specific

C++ C++ language specific

Java Java language specific

Hypertext links are shown as *blue italic underlined*.

On-Line (PDF) versions of this document: Items shown as cross-references to other parts of the document, *e.g.* *Contacts* on page 8, behave as hypertext links: 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`, **`Courier Bold`**, or *`Courier Italic`* fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

```
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);
```

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

Sans-serif Bold indicates user-related actions, *e.g.* **File > Save** (a sequence of selections from menus, or buttons or check-boxes).

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

Contacts

PrismTech can be contacted at the following contact points.

Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

European Head 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*

Technical questions: technical-support*@prismtech.com*

Sales enquiries: *sales@prismtech.com*

2 Introduction

About Spectra ORB C Edition

Spectra ORB C Edition is a lightweight, high performance Object Request Broker (ORB) designed specifically for Distributed Real-time Embedded (DRE) systems and that can be configured to support different CORBA profiles based on a user's needs. It is the first interoperable ORB solution that can be used to support CORBA applications written in C on both GPPs and DSPs.

Spectra ORB C++ Edition v2 provides a pluggable set of libraries that can be used to configure the product to support the following CORBA profiles:

- Minimum CORBA Profile
- CORBA/e Compact and Micro Profiles
- Software Communication Architecture v4.0 Full and Lightweight Profiles

Specifically Spectra ORB supports provides support for CORBA APIs and features defined in the following standards:

- Minimum CORBA Specification version 1.0: OMG Document formal/02-08-01, August 2002
- Common Object Request Broker Architecture for Embedded (CORBA/e) Specification Version 1.0: OMG Document Number: formal/2008-11-06
- Software Communications Architecture Specification – Joint Program Executive Office (JPEO) Joint Tactical Radio System (JTRS)– SCA v2.2.2 – FINAL / 15 May 2006
- Software Communications Architecture Specification V4.0, 28th February 2012, - Appendix E: Platform Specific Model (PSM) – Common Object Request Broker Architecture (CORBA)
- Interoperable Naming Service Specification: OMG Document formal/00-11-01
- Lightweight Log Service Specification: OMG Document formal/05-02-02: v1.1
- Event Service Specification: OMG Document formal/05-02-02: v1.1
- Real-time CORBA Specification, January 2005 Version 1.2 formal/05-01-04
- C++ Language Mapping Specification, July 2012, Version 1.3, formal/2012-07-02

2.1 Key Features

- *Micro ORB kernel*
- *IDL to C compiler*
- *GIOP 1.2 support (with the exception of bi-directional support)*
- *Pluggable Portable Object Adaptor (POA) architecture* containing an extensible and highly scalable plug-in POA architecture (including child POAs)

- *Extensible Transport Framework* – providing multi-transport plug-in support, TCP transport by default; provides users with the ability to develop additional custom transports as required (e.g. UDP, RapidIO, Compact PCI, other..)
- *Multi-thread safe*
- *User configurable server-side threading* – Thread-Per-Request and Thread-Per-Connection models supported
- *Pluggable Any data type support*
- *Request time-outs*
- *Pluggable Real-time CORBA support*
- *Suite of Lightweight Common Object Services* – Naming Service (both Full and Lightweight), Event Service, Log Service

2.2 CORBA Profiles

Spectra ORB was specifically designed for use in resource constrained embedded environments. In order to optimize memory footprint and performance it implements a number of specific subsets of functionality from the OMG's *CORBA Specification* required by DRE systems. These subsets of functionality are referred to as Profiles.

The OMG's *Minimum CORBA Specification* was the first CORBA Profile and was created in order to provide support for restricted environments where the full CORBA version was too large to meet the size and performance requirements. Domain specific standards such a the Software Communications Architecture (v2.2.2) mandate the use of ORB implementations that can support Minimum CORBA.

Minimum CORBA was eventually superseded by the OMG's *CORBA for Embedded Specification (CORBA/e)* which introduced two new Profiles; the CORBA/e Compact and Micro Profiles. The CORBA/*e* Compact Profile supports sophisticated applications such as real-time image and signal processing on board based systems running a standard Real-Time Operating System (RTOS). The CORBA/*e* Micro Profile supports basic functionality on the smallest networked systems, including Digital Signal Processors (DSPs) and the low-powered microprocessors found on typical hand-held devices.

The most recent version of *Software Communications Architecture Specification*, V4.0, defines a Platform Specific Model (PSM) specifying three new CORBA Profiles derived from CORBA/*e* with additional features from RT CORBA and a feature set tuned specifically for Software Defined Radio (SDR) applications.

The SCA CORBA profiles are characterized as follows:

1. SCA Full CORBA (Full) Profile – is the Full CORBA profile and is intended for SDR applications that will be hosted on most General Purpose Processor (GPP) platforms
2. SCA Lightweight CORBA (LW) Profile – is more constrained than the SCA Full CORBA Profile and is targeted towards environments with limited computing support such as a DSP.
3. SCA Ultra-Lightweight CORBA (ULW) Profile – is more constrained than the SCA Lightweight CORBA Profile and is specifically intended for processing elements with even more limited computing support such as those hosted on FPGAs.

2.2.1 Minimum CORBA Profile

The Minimum CORBA Specification, which refers to itself as the *minimumCORBA* specification, excludes features which were not felt to be essential in restrictive, high performance environments:

- Compiles all OMG IDL (although dynamic aspects of CORBA – IFR, DII, DSI, recursive Valuetypes, dynamic Any – do not execute).

- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile,CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any, recursive Valuetypes.
- Server-side: POA Supporting Transient or Persistent objects; Retained servants (disallows Implicit Activation).
- DCE ESIOP is omitted.
- Interworking between COM and CORBA is omitted.
- Interceptors are omitted.

All other features from the mandatory parts of OMG's CORBA Specification are supported in Minimum CORBA.

2.2.2 CORBA/e Compact Profile

Compact yet powerful: Fits resource-constrained systems (32-bit processor running a RTOS), but supports sophisticated applications such as signal or image processing in Real-time:

- Compiles all OMG IDL (although dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any – do not execute) with exception of Abstract Interface, Context clauses and Import.
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Supports Real-time CORBA with Static Scheduling; Propagates Real-time CORBA priorities over the wire.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any.
- Restricted Valuetypes – no Value Boxes or Custom Valuetypes.
- Messaging QoS - Rebind Support, Synchornisation Scope, Request and Reply Timeouts.
- Server-side: POA Supporting Transient or Persistent objects; Retained servants (disallows Implicit Activation); Prioritized multi-threading under ORB control.
- Includes Naming, Events, and Lightweight Logging Services.

2.2.3 CORBA/e Micro Profile

Truly Micro: Fits on a mobile or similar device with a lowpower microprocessor, or high-end DSP.

- Compiles all OMG IDL (Dynamic aspects of CORBA – IFR, DII, DSI, transient Servants – do not execute) with exception of Abstract Interface, Context clauses and Import.
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any.
- Messaging QoS - Rebind Policy, Synchronization Scope Policy, Request and Reply Timeout Policies.
- Supports only statically defined Interfaces, Interactions, and Scheduling.
- Supports Real-time CORBA MUTEX interfaces.

- Server-side: For compactness and deterministic behavior supports exactly one POA; allows only transient, retained servants with unique, system-assigned IDs; and multithreading under ORB control.

2.2.4 SCA 4.0 Full Profile

Targets high performance SDR applications based on the SCA and typically hosted on a GPP:

- Compiles all OMG IDL (although dynamic aspects of CORBA – IFR, DII, DSI, Valuetypes, dynamic Any – do not execute) with exception of WChar data type, Abstract Interface, Context clauses and Import.
- Restricted Any data type that can only contain:
  - Basic CORBA Types
  - Sequences of basic types (such as String)
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.
- Supports Real-time CORBA with Static Scheduling; Propagates Real-time CORBA priorities over the wire.
- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any.
- Messaging QoS - Synchronization Scope Policy.
- Server-side: POA Supporting Transient or Persistent objects; Retained servants (disallows Implicit Activation); Prioritized multi-threading under ORB control.
- The Full Profile supports the additional standardized parameters identified in Table 1 to the `ORB_init` call to allow the root POA to be created with non-default policies.

Policy	Default Value	Alternate Value	Optional Parameter to Override	Full Profile	LW Profile
Lifespan Policy	TRANSIENT	PERSISTENT	-ORBPOAPersistent	✓	✓
ID Uniqueness Policy	UNIQUE_ID	MULTIPLE_ID	-ORBPOAMultipleId	✓	✓
ID Assignment Policy	SYSTEM_ID	USER_ID	-ORBPOAUserId	✓	✓

Table 1: ORB_init() Parameters

- Supports Real-time CORBA with Static Scheduling; Client Propagated and Server Declared Priority Models, RT Thread Pools

2.2.5 SCA 4.0 Ligthtweight Profile

Targets high performance embedded SDR applications based on the SCA and hosted on an extremely resource limited processor such as a DSP:

- Compiles all OMG IDL (although dynamic aspects of CORBA – IFR, DII, DSI, Valuetypes, dynamic Any – do not execute) with exception of Any and WChar data type, Abstract Interface, Context clauses and Import.
- Integrates with applications running full CORBA, Minimum CORBA, CORBA/*e* Compact Profile, CORBA/*e* Micro Profile, SCA 4.0 Full, Lightweight and Ultra-Lightweight Profiles.
- Supports GIOP and native IIOP.

- Disallows dynamic aspects of CORBA – IFR, DII, DSI, dynamic Any.
- Server-side: For compactness and deterministic behavior supports exactly one POA; allows only transient, retained servants with unique, system-assigned IDs; and multithreading under ORB control.
- The Lightweight Profile supports the additional standardized parameters identified in Table 1 to the ORB_init call to allow the root POA to be created with non-default policies.

For a detailed mapping between functionality defined in each CORBA profile and the functionality provided by Spectra ORB C++ Edition please refer to Table 6.

3 CORBA Basics

3.1 Introduction to CORBA

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.*"

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

3.2 The ORB

A core element of CORBA is the *Object Request Broker*, commonly 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. In this setting, ORBs are one of the principal enabling technologies in the field of distributed object computing.

3.2.1 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.

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.

3.2.2 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.

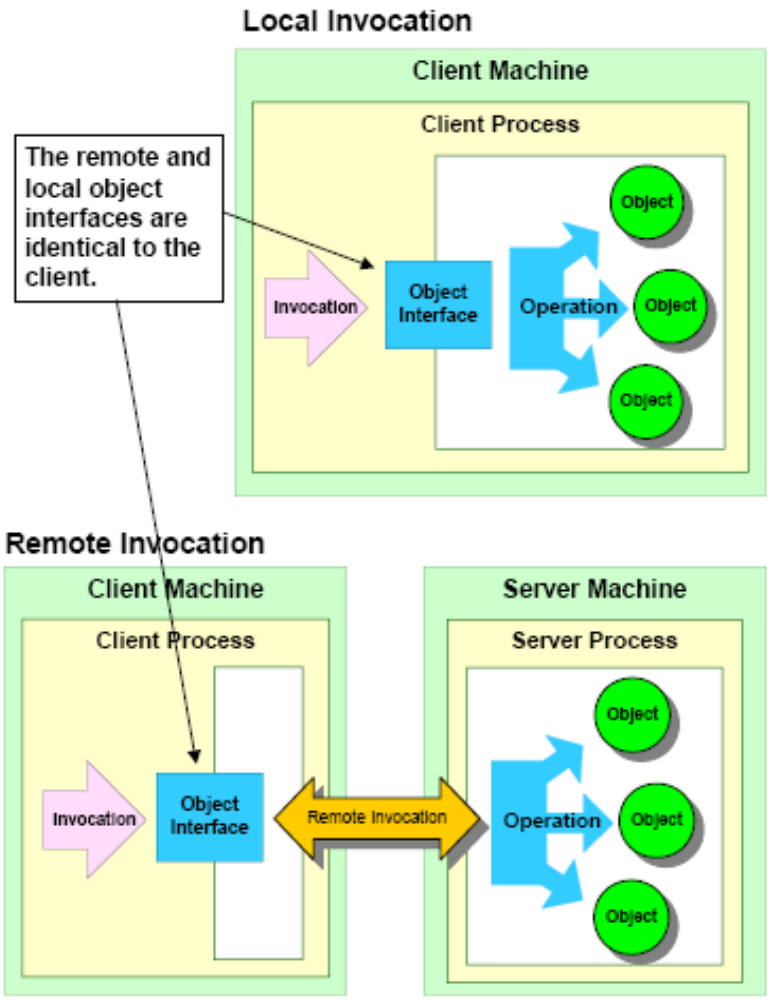

Figure 1 Remote Invocations and Location Transparency

In the CORBA model, the ORB provides the location transparency. 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 Java client is completely unaware that it is invoking an operation on a language-specific object, whether Java, C++, or C, 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 become 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.

3.3 Distributed Object Computing 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 CORBA services, 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.

3.3.1 Interfaces

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 descriptions are mapped into concrete programming interfaces in various programming languages, how object implementations relate to and support an interface, and so on.

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

3.3.2 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.

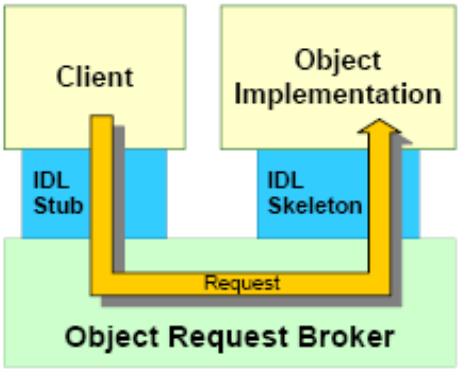

Figure 2 ORB Component Relationships

3.3.2.1 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.

3.3.2.2 Skeletons

Skeletons are used to build object *implementations*. An implementation of a CORBA interface is a package of code in a concrete programming language that provides the real behaviour 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 abstract base class declaration with abstract functions that correspond to the operations in the IDL interface. Programmers construct an implementation by deriving a new type from the skeleton class, then providing method implementations for the operations inherited from the skeleton class.

The stub and skeleton have identical (or nearly identical) interfaces. They are type compatible (i.e., can be substituted for one another) at the level of the common base interface.

3.3.2.3 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 (derived from the skeleton), 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.

3.3.3 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 Spectra ORB implementation.

3.3.3.1 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.

3.4 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 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 labelled 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 labelled the ORB-object boundary in the figure. Anything between those boundaries may be considered as part of the ORB for conceptual purposes.

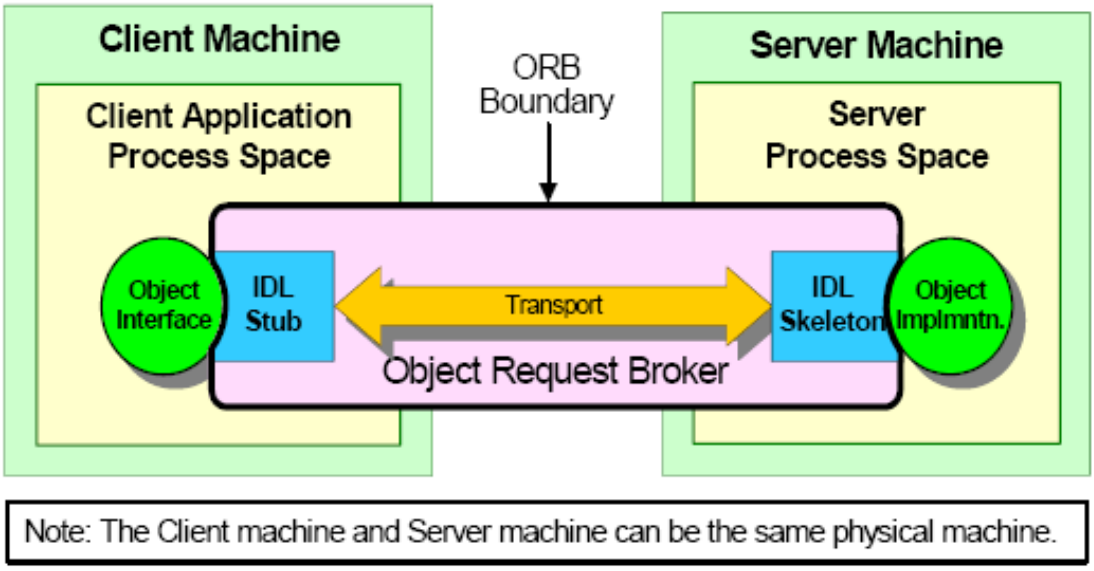

Figure 3 The ORB as an Abstraction

3.4.1 Abstraction

Contrast the previous example with the following scenario. As mentioned above, stubs and skeletons are derived from an interface. When a programmer uses an ORB-based object, methods are invoked on the common interface, not the derived stub or skeleton. Since both the stub class and the skeleton class (and, thus, the implementation class) are derived from the interface base class, 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 C++ 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.

Spectra ORB always routes operation invocations via stubs. Local calls are not made directly on the implementation instance.

3.5 Terminology Explained

Figure 4 is an adaptation from the CORBA specification. The official OMG illustration of ORB architecture is modified to show only Minimum CORBA. 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.

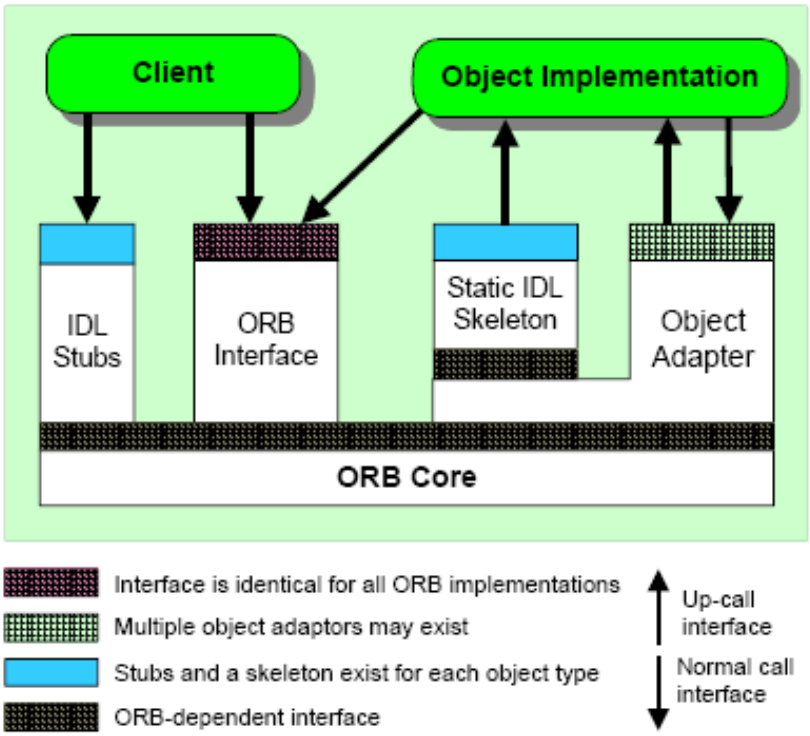

Figure 4 The Structure of Object Request Broker Interfaces

3.5.1 Clients and Servers

As mentioned above, the terms *client* and *server* in a distributed object context have a different meaning from 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.

3.5.2 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 a C++ 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.

3.5.3 First Class 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

For various reasons, the CORBA specification and some CORBA services 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; in other cases the interface cannot be properly expressed in IDL. 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 or the interface for the POA. Pseudo interfaces generally become programming objects in the language mappings (that is, a class in C++), 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-objects.

3.5.3.1 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 describes 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.

3.5.3.2 Object Adapters

The CORBA specification describes pseudo 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)

4 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

All references to the POA in this section describe the POA characteristics supported by Spectra ORB with the exception of the object adapter's dynamic behaviour which is not supported.

4.1 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.

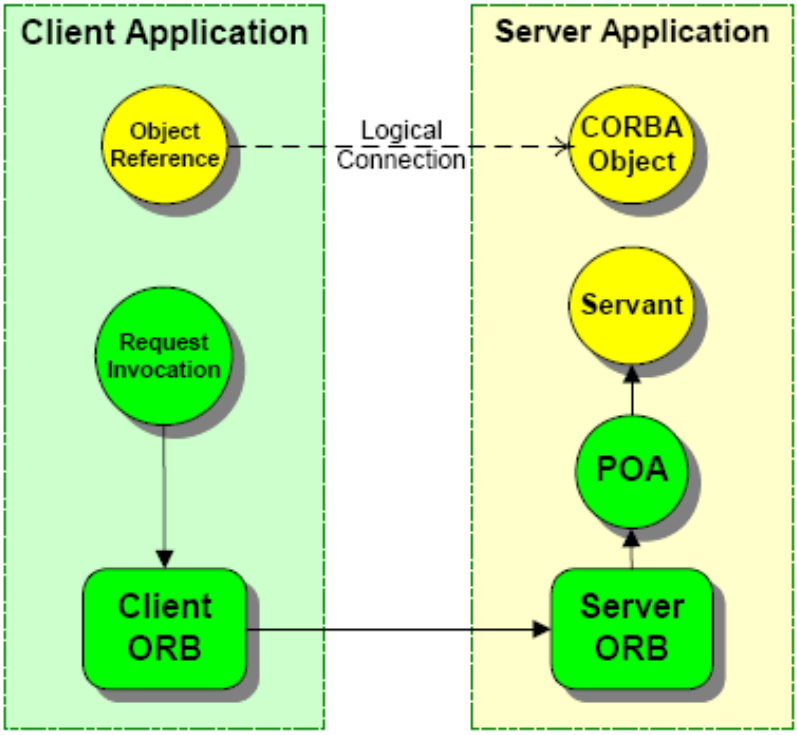

Figure 5 Request Dispatching

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.

4.2 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.

4.2.1 Standard POA Policies

The following standard POA policies are supported by Spectra ORB.

4.2.1.1 Lifespan Policy

The POA `create_lifespan_policy` operation allows you to specify the lifespan of objects with the following values:

- `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 POA lifespan policy as `TRANSIENT` does not prevent explicit reactivation of a servant with the same object key.

4.2.1.2 Object Id Uniqueness Policy

The POA `create_id_uniqueness_policy` operation 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`.

4.2.1.3 Id Assignment Policy

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

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

The default value for this policy is `SYSTEM_ID`.

4.2.2 POA Policy Summary

All POA policy objects are locality constrained; 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.

Spectra ORB supports only very basic POA manager functionality. An activate call is supported which activates request handling on all related POAs. In Spectra ORB there is a single `POAManager` which controls all POAs. No request handling is done until the POAs are activated. The ORB run call will implicitly activate the POA manager if not already active.

4.3 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. ObjectIds must be unique within each individual POA; however different POAs can assign the same ObjectId.

4.4 Servants

The IDL compiler, `idlc`, generates server-side skeleton code in C. These skeletons are abstract base classes (in the form of C structs) from which servant structs are derived. The user is obliged to implement all of the functions declared in that interface. Servants are responsible for incarnating CORBA objects. A servant is a pointer to the target implementation function.

4.5 Servant Mapping

A *servant* is a language-specific entity that can incarnate a CORBA object. In C, a servant is composed of a data structure that holds the state of the object along with a collection of *method functions* that manipulate that state to *implement* the CORBA object.

For example, the `PortableServer::Servant` IDL type maps to a C type as:

```
typedef void * PortableServer_Servant;
```

In C, a *servant* is mapped to a *void** rather than a pointer to *ServantBase*, allowing all servant types of derived interfaces to be passed to all the operations that take a Servant parameter without requiring casting. However, it is expected that an instance of PortableServer_Servant points to an instance of a PortableServer_Servant or its equivalent for derived interfaces.

Due to a lack of support for object-oriented features in C, the inheritance and polymorphism of the C server-side mapping is effected using embedded pointers in structs of various types. A servant is associated with a table of pointers to method functions. This table is called an *entry point vecto*r, or EPV. The EPV has the same name as the servant type with *__epv* appended (note the double underscore). The EPV for PortableServer_Servant is defined as follows:

```
typedef struct PortableServer_Servant__epv
{
   void* _private;
   void (*finalize)(PortableServer_Servant, CORBA_Environment*);
   PortableServer_POA (*default_POA)
   (
      PortableServer_Servant,
```

```
      CORBA_Environment*
   );
} PortableServer_Servant__epv;

extern PortableServer_POA PortableServer_Servant__default_POA
{
  PortableServer_Servant,
  CORBA_Environment*
);
```

The PortableServer_Servant__epv *_private* member is opaque to application writers. EPVs are shared among multiple servants. The second member is a pointer to the finalization function for the servant, which is invoked when the servant is etherealized. The other function pointers correspond to the usual Servant operations. The finalization function may perform or call operations that perform C++ destructor-like deallocations and clean-ups that may be required.

The `PortableServer_Servant` structure combines an EPV with per-servant data, as shown below:

```
typedef PortableServer_Servant__epv * PortableServer_Servant__vepv;
typedef struct PortableServer_Servant
{
   void* _private;
   PortableServer_Servant__vepv * vepv;
} PortableServer_Servant;
```

The first member is a void* that points to application-opaque, ORB implementation-specific data. The second member is a pointer to a pointer to a PortableServer_Servant__epv. Double indirection is used so that servants for derived classes contain multiple EPV pointers, one for each base interface as well as one for the interface itself. The name of the second member, *vepv*, is standardized to allow portable access through it.

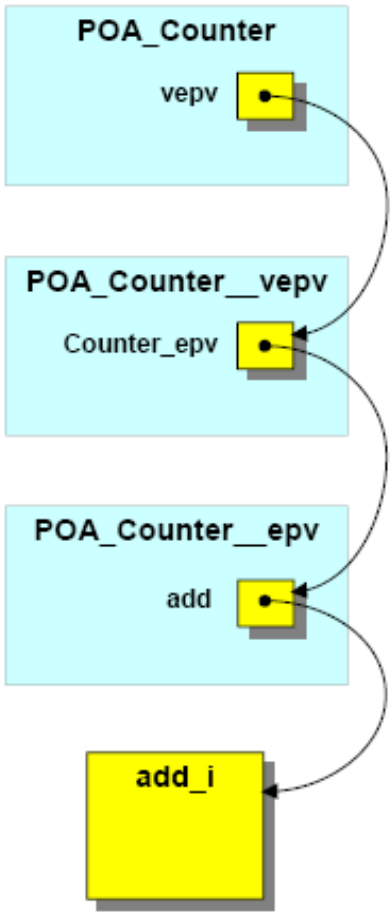

Figure 6 Object Implementation Using Static Invocation Interface

The pointers to EPVs in the VEPV structure are in the order that the IDL interfaces appear in a top-to-bottom left-to-right traversal of the inheritance hierarchy of the most-derived interface.

4.6 Servant Structure Initialization

The CORBA specification mandates that each servant must have initialization and etherealization, or finalization functions. The ORB implementation provides the following functions:

```
void PortableServer_Servant__init
(
   PortableServer_Servant,
   CORBA_Environment*
);

void PortableServer_Servant__fini
(
   PortableServer_Servant,
   CORBA_Environment*
);
```

The first argument to the `init` function is a valid `PortableServer_Servant` whose *`vepv`* member has already been initialized to point to a `VEPV` structure. The `init` function performs ORB-specific initialization of the `PortableServer_Servant`, and initializes the *`finalize`* `struct` member of the pointed-to `PortableServer_Servant__epv` to point to the `PortableServer_Servant_fini` function if the *`finalize`* member is `NULL`.

If the `finalize` member is not `NULL`, it is presumed that it has already been correctly initialized by the application, and is thus not modified. Similarly, if the `default_POA` member of the `PortableServer_Servant__epv` structure is `NULL` when the `init` function is called, its value is set to point to the `PortableServer_Servant__default_POA` function, which returns an object reference to the root POA.

If a servant pointed to by the `PortableServer_Servant` passed to an `init` function has a `NULL vepv` member, or if the `PortableServer_Servant` argument itself is `NULL`, no initialization of the servant is performed, and the `CORBA_BAD_PARAM` standard exception is raised via the `CORBA_Environment` parameter. This also applies to interface-specific `init` functions.

The `fini` function cleans up ORB-specific private data. It is the default finalization function for servants. It does not make any assumptions about where the servant is allocated. Applications may “override” the `fini` function for a given servant by initializing the `PortableServer_Servant__epv finalize` pointer with a pointer to a finalization function made specifically for that servant. Any such overriding function must always ensure that the `PortableServer_Servant_fini` function is invoked for that servant as part of its implementation.

If a servant passed to a `fini` function has a `NULL epv` member, or if the `PortableServer_Servant` argument itself is `NULL`, no finalization of the servant is performed, and the `CORBA_BAD_PARAM` standard exception is raised via the `CORBA_Environment` parameter. This also applies to interface-specific `fini` functions.

Normally, the `PortableServer_Servant__init` and `PortableServer_Servant__fini` functions are not invoked directly by applications, but rather by interface-specific initialization and finalization functions generated by an IDL compiler. For example:

```
void POA_Counter__init (POA_Counter * servant, CORBA_Environment * ev)
void POA_Counter__fini (POA_Counter * servant, CORBA_Environment * ev)
```

The address of a servant should be passed to the `init` function before the servant is allowed to be activated

or registered with the POA in any way. The results of failing to properly initialize a servant via the appropriate `init` function before registering it or allowing it to be activated are implementation-specific.

4.6.1 Auto-Initialization of Servants

The Spectra ORB IDL compiler has several switches which help with the generation of EPV structures and initializing them to refer to implementation functions.

The *`-gen_vepv`* flag causes the generation of EPV structures for operations. The *`-gen_impl`* flag causes the generation of skeleton implementation functions for operations.

For example:

```
% idlc -both -gen_vepv hello.idl
```

4.7 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.

or

- use `POA_create_reference` or `POA_create_reference_with_id` to create an object reference without activating it

However, because Spectra ORB does not support implicit activation of objects, you must explicitly activate the object to process requests to it. Use *`deactivate_object`* to remove the association of the object with its servant.

4.8 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 object id 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 object id in its Active Object Map. Once a reference to the servant is obtained, the appropriate method is invoked. Otherwise, an exception is thrown.

4.9 Designing an Application

Spectra ORB does not support thedynamic features of a full POA - such as activation on demand – as such you must take this into consideration before designing the server. As adapter activators are not supported in Spectra ORB , non-existent POAs are not activated on demand.

The ORB POAs must exist when the server comes up. The server must also be designed to activate all objects on which clients can invoke requests. The process of activation registers a servant associated with an object id in the Active Object Map. If a request arrives for a servant that is not active, the POA will throw an exception.

4.10 Application Servants

Applications may create their own servant structures so that they can add their own servant-specific data members to store object state. For the `Counter` example shown above, an application servant would probably have a data member used to store the counter value:

```
typedef struct AppServant
```

```
{
   POA_Counter base;
   CORBA_long value;
} AppServant;
```

The application might contain the following implementation of the `Counter::add` operation:

```
CORBA_Long app_servant_add
(
   PortableServer_Servant _servant,
   CORBA_long val,
   CORBA_Environment* _env
)
{
   AppServant * self = (AppServant*) _servant;
   self->value += val;
   return self->value;
}
```

The user may initialize the servant using a VEPV table by adding following code to the server application:

```
PortableServer_Servant__epv base_epv =
{
   NULL, /* ignore ORB private data */
   NULL, /* no servant-specific finalize function needed */
   NULL, /* use base default_POA function */
};

POA_Counter__epv counter_epv =
{
   NULL, /* ignore ORB private data */
   app_servant_add /* point to our add function */
};

/* Vector of EPVs */

POA_Counter__vepv counter_vepv = {&base_epv, &counter_epv};

AppServant my_servant =
{
   /* initialize POA_Counter */
   {
       NULL, /* ignore ORB private data */
       &counter_vepv /* Counter vector of EPVs */
   },
   0 /* initialize counter value */
};
```

Before registering or activating this servant, the application must call:

```
CORBA_Environment ev;
POA_Counter__init (&my_servant, &ev);
```

If the application requires a special destruction function for `my_servant`, it must set the value of the `PortableServer_Servant__epv` "finalize" member either before or after calling `POA_Counter__init`:

```
my_servant.epv._base_epv.finalize = my_finalizer_func;
```

Note that if the application statically initialized the “finalize” member before calling the servant initialization function, explicit assignment to the “finalize” member as shown here is not necessary, since the `PortableServer_Servant__init` function will not modify it if it is non-NULL.

The example shown above illustrates static initialization of the EPV and VEPV structures. While portable, this method of initialization depends on the ordering of the VEPV `struct` members for base interfaces—if the top-to-bottom left-to-right ordering of the interface inheritance hierarchy is changed, the order of these fields is also changed.

A more robust method of initializing these fields is to perform the initialization at runtime, relying on assignment to the named `struct` fields. Since the names of the fields are used in this approach, it does not break if the order of base interfaces changes. Performing field initialization within a servant initialization function also provides a convenient place to invoke the servant initialization functions. In any case, both approaches are portable, and it is ultimately up to the developer to choose the one that is best for each application. If the application developer wishes to use these models, the IDL compiler auto generates initialization `structs` by use of the `-gen_vepv` switch.

4.11 Method Signatures

Implementation methods have signatures that are identical to the stubs except for the first argument. If the following interface is defined in OMG IDL:

```
interface example
{
   long op (in long arg);
};
```

A method function for the `op` operation must have the following function signature:

```
CORBA_long example_op
(
   PortableServer_Servant _servant,
   CORBA_long arg,
   CORBA_Environment * _env
);
```

The `_servant` parameter is the pointer to the servant incarnating the CORBA object on which the request was invoked. The method can obtain the object reference for the target CORBA object by using the `POA_Current` object. The `_env` parameter is used for raising exceptions. Note that the names of the `_servant` and `_env` parameters are standardized to allow the bodies of method functions to refer to them portability.

The method terminates successfully by executing a return statement returning the declared operation value. Prior to returning the result of a successful invocation, the method code must assign legal values to all out and inout parameters.

The method terminates with an error by executing the `CORBA_exception_set` operation prior to executing a return statement. When raising an exception, the method code is not required to assign legal values to any out or inout parameters. Due to restrictions in C, however, it must return a legal function value.

Note that when a method raises an exception it must not also return allocated out arguments as this will cause a memory leak. A conformant method implementation must either complete successfully and allocate all out arguments that require allocation or raise an exception and not allocate any out or return arguments.

4.12 Child POA Creation and Destruction Mechanism

The ORB supports the creation of child POA constructs. These are created using the `PortableServer_POA_create_POA` API with a given POA policy list. The ORB then spawns a POA listener. This listener/reactor thread is dedicated to accepting new connections on the pre-specified port. When a new connect request is received, this thread spawns a new thread to handle the GIOP request. The pointer returned by `PortableServer_POA_create_POA` points to the associated POA.

The following creation and destruction functions are supported:

```
PortableServer_POA PortableServer_POA_create_POA
(
   PortableServer_POA poa,
   const char * name,
   PortableServer_POAManager poaManager,
   CORBA_PolicyList * policies,
   CORBA_Environment* ev
);
void PortableServer_POA_destroy
(
   PortableServer_POA self,
   CORBA_Environment * ev
);
```

4.13 Object Initialization Mechanism

A `PortableServer_Servant` construct is defined as a `void*` so all servant types for derived interfaces can be passed to all of the operations that take a `PortableServer_Servant` parameter without requiring casting. A servant is associated with a table of pointers to method functions.

However, Spectra ORB C Edition provides modifications to the IDL-to-C compiler which generates the correct code for supporting the EPV `structs`. In addition, if the IDL compiler is used with `-disptab` flag, then the compiler automatically generates a sorted servant virtual function table in the skeleton C file.

The appropriate servant initialization functions are generated in the skeleton C file. In accordance with the C mapping specification, the server application calls:

```
Myservant = malloc (sizeof (POA_hello));
POA_Myservant__init (Myservant, &ev);
```

4.14 Object Activation and Deactivation Mechanism

Object activation is supported with both `USER_ID` and `SYSTEM_ID`:

```
PortableServer_ObjectId * PortableServer_POA_activate_object
(
   PortableServer_POA self,
   PortableServer_Servant servant,
   CORBA_Environment * ev
);

void PortableServer_POA_activate_object_with_id
(
   PortableServer_POA self,
   PortableServer_Servant servant,
   const PortableServer_ObectId * oid,
   CORBA_Environment * ev
);
```

4.15 Reference, Servant and Object Identity Conversion

The ORB supports the following POA mapping functions:

```
PortableServer_ObjectId * PortableServer_POA_servant_to_id
(
   PortableServer_POA self,
   PortableServer_Servant servant,
   CORBA_Environment * ev
);

CORBA_Object PortableServer_POA_servant_to_reference
(
   PortableServer_POA self,
   PortableServer_Servant servant,
   CORBA_Environment * ev
);

PortableServer_Servant PortableServer_POA_reference_to_servant
(
   PortableServer_POA self,
   CORBA_Object ref,
   CORBA_Environment * ev
);

PortableServer_ObjectId * PortableServer_POA_reference_to_id
(
   PortableServer_POA self,
   CORBA_Object ref,
   CORBA_Environment * ev
);

PortableServer_Servant PortableServer_POA_id_to_servant
(
   PortableServer_POA self,
   const PortableServer_ObjectId * oid,
   CORBA_Environment * ev
);

CORBA_Object PortableServer_POA_id_to_reference
(
   PortableServer_POA self,
   const PortableServer_ObjectId * oid,
   CORBA_Environment * ev
);
```

5 CORBA Extensions

This section describes features which are part of the full CORBA specification and are provided by Spectra ORB, but which are not specified in the supported Profiles.

5.1 URL Object References

There are several standard portable ways for applications to obtain object references.

- A small set of initial references to essential objects, *RootPOA* and *POACurrent*, are available to applications using the ORB reference resolution operation, `resolve_initial_references()`.
- If a Naming Service is available, it can be resolved as an initial reference then additional object references can be obtained from it.
- Alternatively a server may use the ORB `object_to_string()` method to convert an object reference to a string and then make the string available to clients by displaying it or writing it to a file. Clients can convert the string to an object reference by calling the ORB `string_to_object()` method.
- The URL object reference provides an additional way for clients to obtain object references.

A Uniform Resource Locator (URL) is a networked extension of the standard directory/filename concept. Object references can be constructed from a `corbaloc` type URL:

```
corbaloc:[<protocol>]:<endpoint>/<key>
```

where:

`<protocol>` is the protocol used, for example, *iiop*. Note that *iiop* is a special case in that it is the default protocol so need not be set.

`<endpoint>` is the endpoint for the specified protocol.

<key> is the object key (or short key). A short key is the name by which the object has been registered with the ORB with the `register_initial_reference` operation.

For the iiop protocol the endpoint has the format:

```
<host>:[<port>]
```

`<host>` is the internet host IP address or DNS-style name of the target server.

`<port>` is the port number where the object receives connections. The default value, `2908`, is used when `<port>` is omitted.

For example: `corbaloc:iiop:10.1.0.28:1234/Myserver`

5.1.1 Client Side Implementation

To use a URL to locate an object, a client must do the following:

Step 1: Start the application with a command line that includes the `-ORBInitRef` option in the form:

```
-ORBInitRef <name>=<url>
```

Step 2: Separate sequences of `-ORBInitRef <name>=<url>` are required for each object that the client will locate by URL. For example:

```
% Client -ORBInitRef payroll_server=corbaloc:iiop:money:8872/Payroll \
         -ORBInitRef document_server=corbaloc:iiop:docs:8213/DocSvr
```

Step 3: Call `CORBA_ORB_init (&argc, argv, "eorb-ce", &env)`. The ORB converts the URLs to object references and adds them to the list of initial references.

Step 4: Call `CORBA_ORB_resolve_initial_references(<name>)` to obtain a reference to the named object.

5.1.2 Server Side Implementation

Objects created by a server are not automatically accessible by URL. In order to make an object accessible via URL, a server must:

Step 1: Create and activate a servant in a POA.

Step 2: Get a reference to the servant with the POA `servant_to_reference` operation.

Step 3: A Register the reference with the ORB `register_initial_reference` operation. The name by which the reference is registered can then be used as the key to the corbaloc URL by the client.

6 Additional Features

This section describes the additional, proprietary features provided by Spectra ORB.

Features and topics covered include:

- the pluggable transport mechanism
- the pluggable module API
- the ORB libraries
- policy definitions
- exceptions
- logging
- GIOP 1.2

6.1 Extensible Transport Framework

The ORB supports a pluggable transport layer that provides users with the ability to develop additional, custom transports as required. Please note that a default TCP transport is included with the ORB. (Please also refer Chapter 7 of this guide for additional information.

6.2 Pluggable API

The ORB micro kernel can be extended simply by adding PrismTech-supplied, or user-developed modules. This allows you to conserve footprint if the feature is not needed. It also allows the user to select different implementations of the same feature. For instance in, the transport library allows proprietary transports to be plugged in to the ORB. So, the user can plug in any one of the transports to the ORB core. The ORB provides an API to plug in the modules that the application needs to use. The plug-in modules are any, POA, and the choice of transport.

Plugin Function	Library	Description
`EORB_POA_plugin ()`	`poa`	POA
`EORB_Any_plugin ()`	`any`	Any type support
`EORB_Fixed_plugin ()`	`fixed`	Fixed type support
`EORB_Codec_plugin ()`	`codec`	IOP::Codec module support
`EORB_IIOP_plugin ()`	`iiop`	IIOP (tcp) transport profile
`EORB_UIOP_plugin ()`	`uiop`	UIOP (domain socket) transport profile
`EORB_DIOP_plugin ()`	`diop`	DIOP (udp) transport profile
`EORB_MIOP_plugin ()`	`miop`	MIOP (unreliable multicast) transport profile
`EORB_MQIOP_plugin ()`	`mqiop`	MQIOP (POSIX message queue) transport profile
`EORB_MSGQIOP_plugin ()`	`msgqiop`	MSGQIOP (TI DSP message queue) transport profile
`EORB_INTIOP_plugin ()`	`intiop`	INTIOP (Integrity connect) transport profile
`EORB_QCIOP_plugin ()`	`qciop`	QCIOP (QuicComm) transport profile
`EORB_RTORB_plugin ()`	`rt`	Realtime ORB
`EORB_RTPOA_plugin ()`	`rtpoa`	Realtime POA

Table 2: Standard Plug-ins

6.3 Using Plugin Modules

To use a plug-in module, call the plugin function in main before initializing the ORB. All plugin functions have the following form:

```
void EORB_<module>_plugin ();
```

These steps describe how to plug in and link the `any` module:

Step 1: Include `eOrbC/CORBA/any.h`.

Step 2: Call `EORB_Any_plugin ()` before `CORBA_ORB_init ()`.

Step 3: Link executables with the `ec_any` library.

6.3.1 Libraries

The ORB runtime is distributed as a set of libraries to be linked with the ORB application code. This section lists each library, explaining its purpose and the circumstances in which it should be used.

Note that this section refers to the ORB libraries by their base name. The actual name of the libraries will be platform specific, according to the naming conventions of each platform. For example, the *`ec_orb`* library may be called *`ec_orb.lib`*, *`libec_orb.so`*, *`libec_orb.a`*, *`libec_orb.lib`*, etc. depending on the platform it is intended for.

Library	Description
ec_orb	The ORB implementation
ec_lworb	The lightweight ORB implementation (SCA LightWeight profile)
ec_core	Shared part of core ORB (with C++ ORB)
ec_iiop	IIOP transport profile
ec_diop	DIOP transport profile
ec_uiop	UIOP transport profile
ec_mqiop	MQIOP transport profile
ec_miop	MIOP transport profile
ec_tcp	TCP transport (for IIOP)
ec_udp	UDP transport (for DIOP)
ec_un	Domain socket transport (for UIOP)
ec_mq	POSIX message queue transport (for MQIOP)
ec_uipmc	UIPMC transport (for MIOP)
ec_codec	IOP::Codec support
ec_fixed	Fixed type support
ec_any	Any type support
ec_poa	The POA implementation.
ec_lwpoa	The lightweight POA implementation (SCA LightWeight profile)
ec_goa	Group Object Adapter (GOA) (for MIOP)
ec_os	Operating system portability layer
ec_rtorb	Real-time ORB
ec_rt	Shared part of Real-time ORB (with C++ ORB)
ec_rtpoa	Real-time ORB

Table 3: Standard Libraries

Any ORB application, however simple, must link with at least these libraries:

- The ORB (ec_orb or ec_lworb)
- The shared ORB core (ec_core)
- A protocol profile (e.g. ec_iiop)

- A transport plug-in (e.g. `ec_tcp`)
- The platform abstraction library (`ec_os`)
- Optionally for servers the POA (`ec_poa` or `ec_lwpoa`)

Server applications must also be linked to a POA library, `ec_poa` or `ec_lwpoa`. Server application that use multicast transport must also be used with the GOA library, `ec_goa`. The ORB may also provide additional platform specific libraries, for example to support specific transport implementations on embedded systems. In addition to these mandatory libraries, link to other libraries as required to provide the functionality your application uses. There will also be additional, platform-specific system libraries which the ORB requires.

6.4 Object Reference Resolution

Typically servers need to be able to publish object references for their servants and clients need to resolve these references to call onto the server. The standard way to do this in the ORB without using an explicit service such as the Naming service is to use the standard ORB resolve_initial_references and register_initial_reference operations.

The ORB resolve_initial_reference function resolves a named object reference. This can be configured via the `-ORBInitRef` ORB initialization argument. Various URL formats are supported, for example to resolve a named reference from an IOR written to file. For example:

```
-ORBInitRef MyServer=file:/home/user/server.ior
```

To support the registration of object references the register_initial_reference operation can be configured to support the writing of object references either to standard output or to file. This is supported via two custom plug-ins:

```
EORB_Stdio_plugin ();
EORB_File_plugin ();
```

These plug-ins install support for either writing registered references to standard output or file, respectively. By default if both are installed the standard output plug-in is used by default. The plug-ins can be controlled via the ORB initialization argument `-ORBRegistry`, which takes a comma separated list of plug-in initialization arguments. For example:

```
-ORBRegistry stdio:
-ORBRegistry file:
-ORBRegistry stdio:,file:
-ORBRegistry file:/tmp
```

The first case enables just standard output, the second just file output and the third both types of registration. By default the file registry writes the named object reference to a file called `<name>.ior` in the current directory unless the last form of the file registry initializer is used, which specifies a directory in which files are to be written.

6.5 Policies

Policies are settings which affect the operation of the ORB. Policies are accessed in a structured manner using interfaces derived from the *`Policy`* interface defined in the *`CORBA`* module. The ORB's policies, and the functions used to access them, are described in the *Reference Guide*.

Policies can be set with ORB, POA, Thread, or Object scope. The effective policy for an Object is the one with the narrowest scope; for example, a policy set at the Object level will override a policy set at Thread level. If a policy is not set explicitly at any level, the default policy value is used.

The operations used to *get* and *set* Policy values are defined in the file *`policy.h`* and are implemented in the *`ec_orb`* library. These operations are described in the *Reference Guide*.

6.6 Exceptions

Since the C language does not provide native exception handling support, applications pass and receive exceptions via the special `CORBA_Environment` parameter passed to each IDL operation. The `CORBA_Environment` type is partially opaque; the C declaration contains at least the following:

```
typedef struct CORBA_Environment
{
   CORBA_exception_type _major;
   ...
} CORBA_Environment;
```

Upon return from an invocation, the `_major` member indicates whether the invocation terminated successfully; `_major` can have one of the values `CORBA_NO_EXCEPTION`, `CORBA_USER_EXCEPTION`, or `CORBA_SYSTEM_EXCEPTION`; if the value is one of the latter two, then any exception parameters signalled by the object can be accessed.

Four functions are defined on a `CORBA_Environment` structure for accessing exception information. Their signatures are:

```
extern void CORBA_exception_set
(
   CORBA_Environment * ev,
   CORBA_exception_type major,
   const CORBA_char * except_repos_id,
   void * param
);
extern CORBA_char * CORBA_exception_id (CORBA_Environment * ev);
extern void * CORBA_exception_value (CORBA_Environment * ev);
extern void CORBA_exception_free (CORBA_Environment * ev);
```

`CORBA_exception_set` operation allows a method implementation to raise an exception.

The `ev` parameter is the environment parameter passed into the method. The caller must supply a value for the `major` parameter. The value of the `major` parameter constrains the other parameters in the call as follows:

- If the major parameter has the value `CORBA_NO_EXCEPTION`, this is a normal outcome to the operation. In this case, both `except_repos_id` and `param` must be `NULL`. Note that it is *not* necessary to invoke `CORBA_exception_set` to indicate a normal outcome; it is the default behaviour if the method simply returns.
- For any other value of major, it specifies either a user-defined or system exception. The `except_repos_id` parameter is the repository ID representing the exception type. If the exception is declared to have members, the `param` parameter must be the address of an instance of the exception `struct` containing the parameters according to the C language mapping, coerced to a `void*`. In this case, the exception `struct` must be allocated using the appropriate `T__alloc` function, and the `CORBA_exception_set` function adopts the allocated memory and frees it when it no longer needs it. Once the allocated exception `struct` is passed to `CORBA_exception_set`, the application is not allowed to access it because it no longer owns it. If the exception takes no parameters, `param` must be `NULL`.

If the `CORBA_Environment` argument to `CORBA_exception_set` already has an exception set in it, that exception is properly freed before the new exception information is set.

`CORBA_exception_id` returns a pointer to the character string identifying the exception. The character string contains the repository ID for the exception. If invoked on a `CORBA_Environment` that identifies a non-exception (`_major==CORBA_NO_EXCEPTION`), a null pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until `CORBA_exception_free` is called.

`CORBA_exception_value` returns a pointer to the structure corresponding to this exception. If invoked on a `CORBA_Environment` that identifies a non-exception or an exception for which there is no associated information, a null pointer is returned. Note that ownership of the returned pointer does not transfer to the caller; instead, the pointer remains valid until `CORBA_exception_free` is called.

`CORBA_exception_free` frees any storage allocated in the environment set by `CORBA_exception_set`, and sets the `_major` field to `CORBA_NO_EXCEPTION`. It is permissible to invoke `CORBA_exception_free` regardless of the value of the `_major` field.

Consider the following example IDL:

```
interface exampleX
{
   exception BadCall
   {
      string<80> reason;
   };
   void op () raises (BadCall);
};
```

This utility function is used for checking exceptions. It checks to see whether any ORB function has thrown an exception.

```
void checkException (char * str, CORBA_Environment * ev)
{
   CORBA_System_Exception * sysEx;
   printf ("Inside %s\n", str);

   switch (ev->_major)
   {
      case CORBA_SYSTEM_EXCEPTION:
      {
         sysEx = (CORBA_System_Exception*) CORBA_exception_value (ev);
         printf ("Got system Exception %s: ", CORBA_exception_id (ev));
         printf ("With Minor Code: %d ", sysEx->minor);
         printf ("\tCompletion Status:%d\n\n ", sysEx->completed);
         CORBA_exception_free (ev);
         exit (1);
      }
      case CORBA_NO_EXCEPTION:
      break;
      default:
      {
         printf ("Unknown exception thrown\n\n");
         exit (1);
      }
   }
}
```

For descriptions of specific exceptions, refer to the *Reference Guide*.

6.7 Logging

The ORB supports logging that can be enabled to help debugging and monitor any errors or warnings generated by the ORB. There are four log levels supported, identified via the *`EORB_Log_Level`* enum:

```
typedef enum
{
   EORB_LOG_DEBUG,
   EORB_LOG_WARN,
   EORB_LOG_ERROR,
   EORB_LOG_FATAL
```

```
} EORB_Log_Level;
```

These log levels are defined in the `eOrbC/EORB/Log.h` header file and are ordered in increasing level of severity.

Logging is only supported in the debug version of the ORB libraries (debug libraries are identified by the *_g* suffix). A default logger implementation is supported that simply logs to standard output via *printf*. The default log level is `WARN`. The log level can be set via the ORB initialisation argument *-ORBLogLevel* passed to the *CORBA_ORB_init* call. This should be followed by the log level required, namely `DEBUG`, `WARN`, `ERROR` or `FATAL`.

The ORB also supports programmatic configuration of the logger at runtime when the *EORB_USE_LOGGING* compiler define is defined as true (*1*). By default *EORB_USE_LOGGING* is defined to be true when debugging is enabled (*E_DEBUG* is defined).

To modify the active log level use the *EORB_Log_setLevel* function defined as:

```
void EORB_Log_setLevel (EORB_Log_Level level);
```

It is also possible to redirect log messages to a plug-in logger of type *EORB_Logger* defined as:

```
typedef void (*EORB_Logger)
(
   const char * facility,
   EORB_Log_Level level,
   const char * message
);
```

A custom logger can be installed via the `EORB_Log_setLogger` function defined as:

```
void EORB_Log_setLogger (EORB_Log_Logger logger);
```

6.8 Logger Macros

Logger macros are provided to generate messages passed to the logger when *EORB_USE_LOGGING* is defined as true (*1*). If `EORB_USE_LOGGING` is defined as false (*0*) then no code is actually generated. By default `EORB_USE_LOGGING` is defined to be true when debugging is enabled (*E_DEBUG* is defined).

The following macros are supported: each takes a string identifying the log facility as its first argument, the log level as its second argument and the log message as its third argument. Up to five additional arguments are supported: these are inserted into the message string using *sprintf* formatting. The log macros differ only in the number of optional arguments they support and are called:

- *EORB_LOG*
- *EORB_LOG1*
- *EORB_LOG2*
- *EORB_LOG3*
- *EORB_LOG4*
- *EORB_LOG5*

Their usage can best be seen through an example:

```
char * str;
EORB_LOG1 ("Test", EORB_LOG_DEBUG, "setName called with %s", str);
```

Log message strings should not contain new line characters (*\n*) as the default logger prints each log on a new line.

6.9 Log Facilities

The following logging facilities are currently defined within the ORB:

- *ORB* - the ORB core
- *Util* - internal utility code
- *TypeCode* - the typecode support subsystem
- *Memory* - the memory management subsystem
- *Transport* - generic transport management subsystem
- *TCP* - the TCP-based IIOP transport
- *UDP* - the UDP-based DIOP transport
- *POA* - the POA

6.10 GIOP 1.2

There are eight GIOP messages defined in GIOP 1.2:

- `Request`
- `Reply`
- `CancelRequest`
- `LocateRequest`
- `LocateReply`
- `CloseConnection`
- `MessageError`
- `Fragment`

`Request` is issued by a CORBA client and handled by the server.

`Reply` is issued by the server in response to a `Request` and is handled by the client. A reply can have a reply status of `LOCATION_FORWARD`, and contain an IOR to a different CORBA server. The ORB will handle a `LOCATION_FORWARD` reply status and connect to the new server location.

`CancelRequest` is issued by the client, indicating that the client is no longer interested in the Reply to that Request. Note that Spectra ORB clients do not issue `CancelRequest` messages. Spectra ORB servers can handle a `CancelRequest` message, but they do not interrupt the server-side execution of the request. This behaviour is compliant with the specification.

`LocateRequest` is an optional message issued by the client. Note that Spectra ORB clients do not issue `LocateRequest` messages by default (can be enabled as an ORB QoS). Spectra ORB servers can handle `LocateRequest` messages and respond with the correct reply message. A `LocateReply` is issued by the server in response to a `LocateRequest` message. Spectra ORB servers will handle a `LocateRequest` message by responding with a `LocateReply` message with a status of `UNKNOWN_OBJECT` or `OBJECT_HERE`. The ORB will not reply with an `OBJECT_FORWARD` status, as the ORB has no location forwarding capabilities.

`CloseConnection` is issued by a server to inform the client that the server is about to close the connection. Spectra ORB servers do not issue `CloseConnection` messages by default (can be enabled as an ORB QoS). Spectra ORB clients handle `CloseConnection` messages by closing the connection, reclaiming resources associated with the connection, and attempting to re-establish the connection.

`MessageError` messages can be sent by both client and server. This message is sent if there is a malformed GIOP header. Clients and servers handle `MessageError` messages by closing the connection, reclaiming resources associated with the connection, and throwing a `COMM_FAILURE` exception to the application layer.

`Fragment` messages can be sent by both client and server. This message is sent when a request or reply message has been split into multiple fragments. Spectra ORB can receive fragmented messages but will not send them unless configured as an ORB QoS.

7 Extensible Transport Framework

This section describes Spectra ORB's Extensible Transport Framework (ETF) and pluggable profile support.

7.1 Implementing a Transport

The Extensible Transport Framework is based on an adopted Object Management Group (OMG) specification for a pluggable ORB transport architecture. The purpose of the ETF specification is to provide a high level of abstraction between the transport and ORB messaging layer allowing development of interoperable transport plug-ins. IIOP (GIOP over TCP/IP) imposes limitations in the form of unpredictable latencies and communication overheads unsuitable for many embedded and real-time systems. ETF aims to separate the concerns of the messaging layer (GIOP) with that of the transport layer thus facilitating independent development of alternative transports (for example, TCP) without the need for a custom messaging layer. Furthermore the ETF specification defines a standard object oriented approach to transport functionality and interaction with the ORB greatly simplifying the process of third party transport development.

An ETF transport can be viewed at the highest level as a collection of objects each of which is responsible for representing a particular subset of transport functionality. The following objects are described by the ETF specification:

- `Factories`
- `ProtocolProperties`
- `Profile`
- `Connection`
- `Listener`

For both Spectra ORB C and C++ Editions a common C transport implementation of these interfaces is used.

7.2 Factories

The `Factories` object is the entry point through which the ORB can gain access to the underlying transport. The primary purpose of the `Factories` object is to create other ETF objects on demand such as `Connection` and `Listener` objects in addition to providing information about the transport such as an identifier and version number.

7.2.1 Protocol Properties

The purpose of the `ProtocolProperties` object is to represent attributes and customisable properties for a transport plug-in. The transport attributes held by a `ProtocolProperties` object usually concern the performance and/or behaviour of the transport. As an example, packet size or send and receive window size are typical attributes that might be used for a TCP transport plug-in.

7.2.2 Profile

The `Profile` object holds information about the server endpoint. This information is required by a client to be able to connect to the server. In a typical example of a TCP/IP transport, this information would usually be a host name and port number.

7.2.3 Connection

The `Connection` object holds state information regarding a client's connection to a server and vice versa. It provides operations to establish, interact with and shut down a connection. A `ConnectionZeroCopy` object inherits from the basic connection, implementing additional support for zero copy read and write operations. Currently Spectra ORB supports the zero copy write operation.

7.2.4 Listener

A `Listener` object is associated with a server endpoint and is responsible for accepting incoming connection requests and establishing and managing server-side connections.

7.3 Pluggable Profiles

The ETF specification relates to the functionality required to implement a transport used to transmit data between client and server and the management of these logical connections. However to integrate a transport with an ORB additional functionality is required essentially to manage transport endpoints (Profiles) within the ORB. This is implemented in Spectra ORB as a pluggable profile. For each pluggable profile there is a corresponding pluggable transport, each implemented in its own library. The following table shows the currently supported pluggable transports and corresponding profiles.

Profile	Transport	Implementation	Type	Specifier
`iiop`	`tcp`	TCP sockets (standard default)	Two-way reliable	OMG
`miop`	`uipmc`	Multicast datagram sockets	One-way unreliable multicast	OMG
`uiop`	`un`	Stream Unix domain sockets	Two-way reliable	TAO
`diop`	`udp`	Datagram sockets	One-way unreliable	TAO
`mqiop`	`mq`	POSIX message queues	Two-way reliable	PrismTech
`intiop`	`int`	Integrity connections	Two-way reliable	PrismTech
`pgmiop`	`pgm`	Multicast datagram sockets	One-way reliable multicast	PrismTech
`sliop`	`sl`	Unix devices (example)	Two-way reliable	PrismTech
`msgqiop`	`msgq`	TI BIOS Message Queue	Two-way reliable.	PrismTech
`fliop`	`fl`	File system (example)	Two-way reliable.	PrismTech
`qciop`	`qc`	QuicComm	Two-way reliable.	PrismTech
`vdkiop`	`vdk`	VDK message queues	Two-way reliable	PrismTech

Table 4: Spectra ORB Pluggable Profiles

Apart from the OMG defined transports (IIOP and MIOP), most of these transports are specific to Spectra ORB, however some of these transports are specified by the open source TAO ORB (UIOP and DIOP).

A pluggable profile in Spectra ORB supports three pieces of key functionality:

- Reading and writing a transport profile from/to a data stream. This supports the encoding/decoding of a transport profile within an object reference (IOR).
- Reading and writing a transport profile from a string. This supports the parsing of listen endpoints from an ORB initialization argument and the generation of `corbaloc` format IORs.
- The registration of the profile and corresponding transport with the ORB runtime.

7.4 Implementing a Transport Plugin

The following section describes how to implement a custom transport. The descriptions will mainly refer to the example `sl` transport and corresponding `sliop` profile provided in the `examples/c/transport` directory.

Headers corresponding to the main transport components (`Factories`, `Listener`, `Connection` and `Profile`) can be found in the `eOrbC/ETF` include directory and all can be included from `eOrbC/ETF.h`. The implementation of each of these components should include the relevant header. If any transport protocol properties are required then `eOrbC/RTCORBA/ProtocolProperties.h` should be included.

The ETF include files themselves include the corresponding implementation base functionality and data types defined in `eOrbC/EORB/ETF`. Most of the standard ETF include files simply define macros that are implemented in the corresponding Spectra ORB implementation support classes. In general the support classes provide a base data structure to support a specific component, this contains function pointers that a specific implementation must set. The core ETF component types are simply pointers to the implementation structures:

```
typedef struct EORB_ETF_ConnectionBase * ETF_Connection;
typedef struct EORB_ETF_ConnectionBase * ETF_ConnectionZeroCopy;
typedef struct EORB_ETF_FactoriesBase * ETF_Factories;
typedef struct EORB_ETF_ListenerBase * ETF_Listener;
typedef struct EORB_ETF_ProfileBase * ETF_Profile;
```

For example in `Factories.h` the logical function `ETF_Factories_create_listener` is implemented as a macro that calls onto a `create_connection` function pointer defined in the `EORB_ETF_FactoriesBase` base implementation structure. The `sl` transport implementation of `Factories` creates an instance of the base support class and sets the required function pointers.

```
static EORB_ETF_FactoriesBase factories =
{
   EORB_SL_Connection_create,
   EORB_SL_Listener_create,
   EORB_TAG_SLIOP
};
```

The following table shows what base implementation structures are used to support the core logical components:

Component	Implementation Structure	Function Pointer Structure
`Factories`	`EORB_ETF_FactoriesBase`	`EORB_ETF_FactoriesBase`
`Listener`	`EORB_ETF_ListenerBase`	`EORB_ETF_ListenerBase__vtable`
`Profile`	`EORB_ETF_ProfileBase`	`EORB_ETF_ProfileBase__vtable`
`Connection`	`EORB_ETF_ConnectionBase`	`EORB_ETF_ConnectionBase__vtable`
`ConnectionZeroCopy`	`EORB_ETF_ConnectionBase`	`EORB_ETF_ConnectionBase__vtable`

Table 5: ETF Base Implementation Structures

The `Factories` component is a special case in that there is only ever a single instance of a `Factories` object for any transport so the required function pointers are simply set on this structure. For the other components, multiple instances may be created so there is a split between the per instance component data and the static component data held in a separate `__vtable` structure.

All the component implementation structures (apart from `EORB_ETF_FactoriesBase`) have as their first member a pointer to the corresponding `_vtable`. This means that an instance of a component implementation such as `Connection`, can be used to access the `__vtable` structure from which the function pointers acting on the component can be called. Convenience macros are provided for this purpose in the standard ETF include files where the functions are not directly implemented. For example in `Listener.h`:

```
#define ETF_Listener_listen(l,e) \
   ((l)->m_vtable)->m_listen (l, e)
```

Each component implementation structure may also contain data fields used by instances of the component. These fields are described in the following component specific sections.

7.5 Factories Implementation

The `Factories` component is the logical entry point into a transport implementation and supports functions to create new instances of `Listener` and `Connection` components, the last field of the `Factories` implementation structure is the tag identifier value for the profile, these are assigned and managed by the OMG. A `Factory` implementation simply needs to create a static instance of the implementation structure and provide an accessor function. For example for the `sl` transport:

```
#include "eOrbC/EORB/SL/Factories.h"
#include "eOrbC/EORB/SL/Listener.h"
#include "eOrbC/EORB/SL/Connection.h"

static EORB_ETF_FactoriesBase factories =
{
   EORB_SL_Connection_create,
   EORB_SL_Listener_create,
   EORB_TAG_SLIOP
};

ETF_Factories EORB_SL_Factories_create (void)
{
   return (ETF_Factories) &factories;
}
```

The `ETF_Factories` component returned by the accessor function is associated with the related pluggable profile registration entry. As the accessor function is called only once it is the logical place to put any additional, one off, transport initialization code.

7.6 Profile Implementation

The `Profile` base implementation structure (`EORB_ETF_ProfileBase`) is defined as:

```
typedef struct EORB_ETF_ProfileBase
{
   struct EORB_ETF_ProfileBase__vtable * m_vtable;
   IOP_ProfileId m_id;
}
EORB_ETF_ProfileBase;
```

The `m_id` data field is the identifier tag for the `Profile` defined by the OMG. The associated function pointer structure (`EORB_ETF_ProfileBase__vtable`) has a single function pointer for the implementation of the `Profile::is_match` operation. The transport specific `Profile` implementation structure should derive from the base implementation structure and add additional elements corresponding to the types that make up an endpoint for the transport. For example the `sl` transport profile has an endpoint that consists of server and client, names and devices and is defined as:

```
typedef struct EORB_SL_Profile
{
   EORB_ETF_ProfileBase m_base;
   char * m_server_host;
   char * m_server_device;
   char * m_client_host;
   char * m_client_device;
}
EORB_SL_Profile;
```

The fields in this structure are those that will be encoded into an IOR to represent a transport listen endpoint. The profile implementation should also create a static instance of the function pointer structure (`EORB_ETF_ProfileBase__vtable`) and set the appropriate function pointers:

```
static EORB_ETF_ProfileBase__vtable EORB_SL_Profile_vtab =
{
   (EORB_ETF_Profile_MatchFN) EORB_SL_Profile_is_match
};
```

An allocator function should be provided to create a new instance of the `Profile` and initialize it with the static function pointer structure and set the id:

```
ETF_Profile EORB_SL_Profile__alloc (void)
{
   EORB_SL_Profile * profile = EORB_allocVar
   (
      (EORB_DtorFN) EORB_SL_Profile_finalize,
      sizeof (EORB_SL_Profile)
   );
   profile->m_base.m_id = EORB_TAG_SLIOP;
   profile->m_base.m_vtable = &EORB_SL_Profile_vtab;
   EORB_ETF_ProfileBase__init ((ETF_Profile) profile);

   return (ETF_Profile) profile;
}
```

The `Profile` function pointer structure and id must be set before the base initializer is called (`EORB_ETF_ProfileBase__init`) as this checks whether both of these fields are non zero. The function returns the base `ETF_Profile` type as the function is installed in the associated pluggable profile, and requires a generic function signature. A clean up function should be associated with the allocated instance to free any storage associated with the structure and call the base finalizer function:

```
static void EORB_SL_Profile_finalize (EORB_SL_Profile * prof)
{
   CORBA_string_free (prof->m_server_host);
   CORBA_string_free (prof->m_server_device);
   CORBA_string_free (prof->m_client_host);
   CORBA_string_free (prof->m_client_device);
   EORB_ETF_ProfileBase__fini ((ETF_Profile) prof);
}
```

7.6.1 Implementing the Profile::is_match Function

This function determines whether two profiles are equivalent and represent the same transport endpoint. For the example SL profile, the function compares the client host and device strings:

```
static CORBA_boolean EORB_SL_Profile_is_match
(
   EORB_SL_Profile * p1,
   EORB_SL_Profile * p2
)
{
   return
   (
      (CORBA_string_cmp (p1->m_client_device, p2->m_client_device) == 0)
      &&
      (CORBA_string_cmp (p1->m_client_host, p2->m_client_host) == 0)
   );
}
```

Note that the ORB will check for pointer equivalence before calling the profile match function so there is no need to check for address equality (`p1 == p2`) in the function implementation.

7.7 Protocol Properties Implementation

`ProtocolProperties` are supported as part of the core ORB implementation and so do not have any ETF related support. They are the means by which any transport related quality of service or endpoint information is passed to the transport. An instance of `RTCORBA_ProtocolProperties` is passed to the `Connection` and `Listener` creation functions provided by the `Factories` component. Client-side `ProtocolProperties` can be set as a standard client-side policy, server-side properties can be passed explicitly to the `POA` creation operation.

A `ProtocolProperties` component needs to derive from `RTCORBA_ProtocolProperties` and has fields corresponding to any qualities of service or endpoint information that may be required for `Listener` or `Connection` creation. Standard accessor (read and write) functions are typically implemented for these fields. For example:

```
typedef struct
{
   RTCORBA_ProtocolProperties _base;

   char * m_server_host;
   char * m_server_device;
   char * m_client_host;
   char * m_client_device;
}
EORB_SL_ProtocolProperties;

EXPORT char * EORB_SL_ProtocolProperties__get_client_host
(
   EORB_SL_ProtocolProperties * props,
   CORBA_Environment * ev
);

EXPORT void EORB_SL_ProtocolProperties__set_client_host
(
   EORB_SL_ProtocolProperties * props,
   char * host,
   CORBA_Environment * ev
);
```

7.8 Listener Implementation

The base `Listener` implementation structure has a number of generic members:

```
typedef struct EORB_ETF_ListenerBase
{
   struct EORB_ETF_ListenerBase__vtable * m_vtable;
   struct EORB_ETF_ConnectionBase * m_connections;
   CORBA_boolean m_destroyed;
   ETF_Profile m_endpoint;
}
EORB_ETF_ListenerBase;
```

- `m_connections`: a list of connections created by the listener. This is managed internally by the ORB and should not be used directly by the transport implementation.
- `m_destroyed`: a flag that is set when the listener is destroyed. The `Listener::accept` operation should return and raise an `ETF::Shutdown` exception when the listener has been destroyed. This allows the ORB listening thread to exit when the ORB is being shutdown or the POA associated with a specific listener has been destroyed.
- `m_endpoint`: the profile holding the endpoint on which the listener is listening. This should be created in the listener creation function and the fields of the endpoint set in the listen function. This profile is returned by the `ETF_Listener__get_endpoint` function.

The `Listener` implementation should also create a static instance of the function pointer structure (`EORB_ETF_ListenerBase__vtable`) and set the appropriate function pointers:

```
static EORB_ETF_ListenerBase__vtable EORB_FL_Listener_vt =
{
   EORB_FL_Listener_accept,
   EORB_FL_Listener_listen,
   EORB_FL_Listener_destroy
};
```

7.8.1 Implementing the Listener Creation Function

This function should create a new instance of a listener implementation structure and an associated profile endpoint (`m_endpoint`). The base implementation function pointer table (`m_vtable`) should be set and then the base initializer function called:

```
   EORB_FL_Listener * listener = NULL;
   EORB_FL_ProtocolProperties * pp = (EORB_FL_ProtocolProperties*) props;
   EORB_FL_Profile * profile = (EORB_FL_Profile*) EORB_FL_Profile__alloc ();

   /* Create and initialize new Listener */

   listener = EORB_allocVar
   (
      (EORB_DtorFN) EORB_FL_Listener_finalizer,
      sizeof (EORB_FL_Listener)
   );
   listener->m_base.m_vtable = &EORB_FL_Listener_vt;
   EORB_ETF_ListenerBase__init ((EORB_ETF_ListenerBase*) listener);
   listener->m_base.m_endpoint = (ETF_Profile) profile;
```

The instance should be created with the `EORB_allocVar` allocation function that allows a destructor function to be registered. The destructor function should free any connection related fields of the implementation structure and call the base finalizer:

```
static void EORB_SL_Listener_finalizer (EORB_SL_Listener * listener)
{
```

```
   EORB_ETF_ListenerBase__fini ((ETF_Listener) listener);
}
```

A `ProtocolProperties` object is passed as an argument to the creation function. This can be used to set the endpoint for the listener or other qualities of service (buffer size etc.). The `ProtocolProperties` object will have either been explicitly created in application code and passed as a policy at `POA` creation time to associate a specific listener with a `POA`, or created implicitly by the `ORB` as a way of passing command line endpoint information (`-ORBListenEndpoints`). The functionality to convert command line endpoint information into a `ProtocolProperties` object is implemented in the related pluggable profile. In the example SL transport the host and device members of the `ProtocolProperties` object are used to establish the endpoint profile for the listener:

```
EORB_SL_ProtocolProperties * pp = (EORB_SL_ProtocolProperties*) props;
EORB_SL_Profile * profile;

profile = (EORB_SL_Profile*) EORB_SL_Profile__alloc ();
profile->m_server_host = CORBA_string_dup (pp->m_server_host);
profile->m_server_device = CORBA_string_dup (pp->m_server_device);
profile->m_client_host = CORBA_string_dup (pp->m_client_host);
profile->m_client_device = CORBA_string_dup (pp->m_client_device);
listener->m_base.m_endpoint = (ETF_Profile) profile;
```

If a `ProtocolProperties` object is not provided or it does not contain endpoint information, then an endpoint profile should be created dynamically. Whether this can be done depends on the underlying transport implementation. If this cannot be done a `CORBA::COMM_FAILURE` system exception should be thrown.

7.8.2 Implementing the Listener::listen Function

The `listen` function is called when the `ORB` is initialized or a `POA` with a specific transport is created. This function should simply try to establish (open) the listen endpoint (`m_endpoint`). If this cannot be done then a `CORBA::COMM_FAILURE` system exception should be thrown:

```
   EORB_SL_Listener * listener = (EORB_SL_Listener*) obj;

   listener->m_channel = open
   (
      ((EORB_SL_Profile*) obj->m_endpoint)->m_server_device,
      O_RDWR | O_NONBLOCK,
      0
   );

   if (listener->m_channel == -1)
   {
      CORBA_exception_set
      (
         ev,
         CORBA_SYSTEM_EXCEPTION,
         ex_CORBA_COMM_FAILURE,
         CORBA_COMM_FAILURE__alloc ()
      );
      ev->system.minor = EORB_COMM_FAILURE_M11;
      ev->system.completed = CORBA_COMPLETED_NO;
   }
```

7.8.3 Implementing the Listener::accept Function

The `accept` function has a dedicated `ORB` thread. This function should block on the listener endpoint waiting for a new client connection (via `Connection::connect`). For each new connection it should create a new server-side `Connection` object which is returned from the function.

If an error occurs then `NULL` should always be returned from this function, optionally a `CORBA::COMM_FAILURE` system exception may also be raised. If a system exception has been raised then the calling thread will exit so no more connections can be created on the endpoint. If a system exception has not been raised then the calling thread will repeatedly call the function, having cleared the system exception. It is up to an implementation to determine what behaviour is most appropriate. This will depend on whether it is possible to recover from a failed connection attempt to an endpoint, if this is the case then a system exception should generally not be raised.

The `Listener accept` function should exit when the listener is destroyed. This will be flagged in the `m_destroyed` field in the base implementation structure.

7.8.4 Implementing the Listener::destroy Function

The `destroy` function is called when the `ORB` is destroyed or a `POA` with a dedicated listener is destroyed. This function should simply close the endpoint on which the listener is accepting. The `m_destroyed` field in the base implementation structure will have been set by the `ORB` core before this function is called. If the `Listener accept` function is blocked this function should try and wake it so it can detect that the listener has been destroyed and exit to terminate the accepting thread. This functionality is implementation specific.

7.9 Connection Implementation

The base `Connection` implementation structure has a number of generic members:

```
typedef struct EORB_ETF_ConnectionBase
{
   struct EORB_ETF_ConnectionBase__vtable * m_vtable;
   struct EORB_ETF_ConnectionBase * m_next;
   struct EORB_ETF_ListenerBase * m_listener;
   ETF_Profile m_server_profile;
   CORBA_boolean m_close;
   CORBA_boolean m_connected;
   CORBA_boolean m_server;
}
EORB_ETF_ConnectionBase;
```

- `m_vtable:` pointer to function pointer structure.
- `m_next`: a list of connections used to associate connections with a listener. This is managed internally by the `ORB` and should not be used directly by the transport implementation.
- `m_listener`: for server-side connections, this refers to the listener that created the connection. This is managed internally by the `ORB` and should not be used directly by the transport implementation.
- `m_server_profile`: for client-side connections this is the server profile (endpoint) to which the connection is connected. This is set automatically be the `ETF_Connection_connect` function and should not be used directly by the transport implementation. The `ETF_Connection_get_server_profile` function returns this member.
- `m_close`: this flags whether the `GIOP::CloseConnection` message should be sent by the ORB when the connection is closed. This may be required if at the transport level the server-side connection cannot detect when the client-side connection is closed or vice versa. It can be set in the `Connection::connect` function for client-side connections or in the `Listener::accept` function for server-side connections.

- `m_connected`: this flags whether a connection has been established. It is used internally by the ORB to ensure that a connection can only be closed if it has been connected and also to make sure that a connection can only be closed once.
- `m_server`: this flags whether a connection is server-side or client-side. Server-side connections are created via the `Listener::accept` function and client-side connections via the `Factories::create_connection` function.

The C`onnection` implementation should also create a static instance of the function pointer structure (`EORB_ETF_ConnectionBase__vtable`) and set the appropriate function pointers:

```
static EORB_ETF_ConnectionBase__vtable EORB_SL_Conn_vt =
{
   NULL,
   EORB_SL_Connection_write,
   EORB_SL_Connection_read,
   EORB_SL_Connection_connect,
   EORB_SL_Connection_close
};
```

Note that the first field in the `vtable` is `NULL`. This corresponds to the optional zero copy write function that can be supported on a transport. A connection implementation must support at least one of the two write functions (both may be supported).

7.9.1 Implementing the Connection Creation Function

This function should create a new instance of a `Connection` implementation structure. The implementation function pointer table pointer should be set and then the base initializer function called:

```
   EORB_SL_Connection * conn = EORB_allocVar
   (
      (EORB_DtorFN) EORB_SL_Connection_finalize,
      sizeof (EORB_SL_Connection)
   );
   conn->m_channel = -1;
   conn->m_base.m_vtable = &EORB_SL_Conn_vt;
   EORB_ETF_ConnectionBase__init ((ETF_Connection) conn);
   return (ETF_Connection) conn;
```

The instance should be created with the `EORB_allocVar` allocation function that allows a destructor function to be registered. The destructor function should free any connection related fields of the implementation structure and call the base finalizer:

```
static void EORB_SL_Connection_finalize (ETF_Connection obj)
{
   EORB_ETF_ConnectionBase__fini (obj);
}
```

7.9.2 Implementing the Connection::connect Function

The `connect` function is called by a client when a call is first made on an object reference. The transport implementation should create a new connection with the listener, with the underlying implementation causing the listener accept function to return a new server-side connection, connected to the client-side connection. The pair of connections, one on the server and one on the client, make up a logical connection between client and server, supporting reading and writing of data across the connection.

When a client connection is closed, normally this can be detected by the server-side connection (typically a thread blocked in the read function), where it should throw a `CORBA::COMM_FAILURE` exception so that the server-side connection and thread can be removed. If this cannot be done at the transport level, then the `m_close` flag should be set in the connected connection so that the ORB will send the `GIOP::CloseConnection` message when the client-side connection is closed. In effect closing the server-side connection is then handled at the protocol rather than the transport level.

7.9.3 Implementing the Connection::write Function

The `write` function is called on connected connections to write data from one to another. The data written on one connection is read from the other. Usually, unless GIOP fragmentation is enabled, a single GIOP message will be sent by the ORB as a single write operation on the transport.

7.9.4 Implementing the ConnectionZeroCopy::write_zc Function

The `write_zc` function has exactly the same semantics as the basic `Connection::write` function. The only difference is that it takes a vector of output buffers as opposed to a single output buffer. This supports the optimal direct copy of some user data types to the transport without intermediate copying into a single CDR buffer (hence zero copy). This function is best implemented where the underlying transport mechanism supports a similar write mechanism, such as the POSIX `writev` call.

7.9.5 Implementing the Connection::read Function

The `read` function is called on connected connections to read data sent from one to another. The data read on one connection is written from the other. The signature of the `read` function is as follows:

```
CORBA_unsigned_long ETF_Connection_read
(
   ETF_Connection obj,
   CORBA_octet * data,
   CORBA_unsigned_long offset,
   CORBA_unsigned_long min_length,
   CORBA_unsigned_long max_length,
   TimeBase_TimeT time_out,
   CORBA_Environment * ev
)
```

As a GIOP message contains its length in the GIOP message header which is twelve bytes long, each GIOP message is read with two read operations. The first is for the twelve bytes of the message header containing the total message length, the second for the remainder of the message. A transport implementation can distinguish between the two calls as the first will be made with the `offset` argument set at zero (offset to read into from `data`, the start of the buffer), and the second with `offset` equal to twelve. This double read can cause problems for non stream based transports where it may not be possible to only read the first twelve bytes of a message (for example. message queue based transports). Here the connection implementation needs to cache extra data from the first read so it can copied on the second. If this operation fails it should throw a `CORBA::COMM_FAILURE` system exception, to close the connection. On the server side the connection-handling thread will exit when this occurs.

7.9.6 Implementing the Connection::close Function

This function is called when a connection is closed. It should close the underlying transport connection between client and server connection. It may be called on either a client or server connection depending on how an application is coded and used. After a connection has been closed it will be freed by the ORB. The ORB infrastructure ensures that the `GIOP::CloseConnection` message is sent when a client-side connection is closed if the `m_close` flag has been set in the client-side connection (by the `Connection::connect` implementation). This flag should be set when

a server-side connection cannot detect that the connected client-side connection has been closed, typically the case with non node local transports.

7.10 Implementing a Profile

7.10.1 Implementing a Profile Plugin

To install a profile with the ORB a plug-in function is written that registers the profile and associated transport with the ORB runtime. The plug-in function is called in the program main body before the ORB is initialized, for example `ORB_IIOP_plugin` installs the standard IIOP profile and associated TCP ETF transport. All profile information and functionality is contained within a single static instance of a `EORB_IOP_ProfileInfo` structure, defined as:

```
typedef struct EORB_IOP_ProfileInfo
{
   struct EORB_IOP_ProfileInfo * m_next;
   const char * m_name;
   ETF_Profile (*m_profFN) (void);

   void (*m_getFN)
   (
      EORB_IOP_Profile * profile,
      CORBA_DataInputStream is
   );
   void (*m_putFN)
   (
      EORB_IOP_Profile * profile,
      CORBA_DataOutputStream os
   );
   void (*m_keyPutFN)
   (
      EORB_IOP_Profile * profile,
      CORBA_DataOutputStream os
   );
   void (*m_keyGetFN)
   (
      CORBA_DataInputStream is,
      CORBA_sequence_octet * key,
      CORBA_octet minor
   );
   void (*m_fromStrFN)
   (
      EORB_IOP_Profile * profile,
      char * str
   );
   char * (*m_toStrFN) (EORB_IOP_Profile * profile);
   RTCORBA_ProtocolProperties (*m_propsFN) (char * str);

   ETF_Factories m_factories;
   EORB_Protocol_Type m_type;
   EORB_Protocol_ConnType m_conn_type;
   CORBA_unsigned_long m_fragment_size;
}
EORB_IOP_ProfileInfo;
```

- `m_next`: list entry used to hold an internal list of the structure.
- `m_name`: string name for profile e.g. "`iiop`".
- `m_profFN`: function to create new transport profile.

- `m_getFN`: function to marshal a transport profile from a stream.
- `m_putFN`: function to marshal a transport profile to a stream.
- `m_keyPutFN`: function to marshal a key to a stream (multicast only).
- `m_keyGetFN`: function to marshal a key from a stream (multicast only).
- `m_fromStrFN`: function to parse a `corbaloc` format IOR string (optional).
- `m_toStrFN`: function to write a `corbaloc` format IOR string (optional).
- `m_propsFN`: function to create a `ProtocolProperties` object from an endpoint string (optional).
- `m_factories`: transport `Factories`.
- `m_type`: type of transport (one way, two way or multicast).
- `m_conn_type`: type of connection supported by transport (shared, private or both).
- `m_fragment_size`: maximum message size supported by transport. If a GIOP message exceeds this size GIOP fragmentation is used.

Some of the function pointer fields in this structure are optional. The key marshal functions are not generally used and are only to support multicast transports (such as MIOP) where a key needs to be encapsulated within an IOR. The plug-in function should create a single static instance of this structure and register it with the ORB:

```
extern void EORB_SLIOP_plugin (void)
{
   static EORB_IOP_ProfileInfo info =
   {
      NULL,
      "sliop",
      EORB_SL_Profile__alloc,
      EORB_SLIOP_get,
      EORB_SLIOP_put,
      NULL,
      NULL,
      NULL,
      NULL,
      EORB_SLIOP_getProps,
      NULL,
      EORB_PROTOCOL_TWOWAY,
      EORB_PROTOCOL_CONN_ANY,
      0
   };

   if (info.m_factories == NULL)
   {
      info.m_factories = EORB_SL_Factories_create ();
      EORB_IOP_ProfileInfo_register (&info);
   }
}
```

Note that the `m_factories` field is set from the associated transport factories object and is also used to ensure that the function can be called multiple times but will only register once with the ORB runtime. The `EORB_IOP_ProfileInfo_register` function registers the profile with the ORB runtime. The two transport functions for creating the factory and creating a profile, make the dependency between the installed profile and the associated transport.

7.11 Implementing Profile Marshal Functions

Two functions are used to read and write a profile to and from a stream. These functions support the encoding of a profile within an IOR. The transport profile component is the `m_etf_profile` field of the `EORB_IOP_Profile` argument passed to the functions.

7.11.1 Implementing the Profile Read Function

The members of the transport profile are read from a stream using the stream read functions:

```
static void EORB_SLIOP_get
(
   EORB_IOP_Profile * profile,
   CORBA_DataInputStream is
)
{
   EORB_SL_Profile * eprof = (EORB_SL_Profile*) profile->m_etf_profile;

   eprof->m_server_host = CORBA_DataInputStream_read_string (is);
   eprof->m_server_device = CORBA_DataInputStream_read_string (is);
   eprof->m_client_host = CORBA_DataInputStream_read_string (is);
   eprof->m_client_device = CORBA_DataInputStream_read_string (is);
}
```

7.11.2 Implementing the Profile Write Function

The members of the transport profile are written to a stream using the stream write functions:

```
static void EORB_SLIOP_put
(
   EORB_IOP_Profile * profile,
   CORBA_DataOutputStream os
)
{
   EORB_SL_Profile * eprof = (EORB_SL_Profile*) profile->m_etf_profile;

   /* Put client/server channel and host */

   CORBA_DataOutputStream_write_string (os, eprof->m_server_host);
   CORBA_DataOutputStream_write_string (os, eprof->m_server_device);
   CORBA_DataOutputStream_write_string (os, eprof->m_client_host);
   CORBA_DataOutputStream_write_string (os, eprof->m_client_device);
}
```

7.12 Implementing the Profile Protocol Properties Function

This function is used to parse a string representing a transport endpoint, create a `ProtocolProperties` transport component and set these fields in the new component. The created `ProtocolProperties` object is passed to the listener creation function associated with the transport `factories` component:

```
/* Parse host:device:host:device and set in protocol properties */

static RTCORBA_ProtocolProperties EORB_SLIOP_getProps (char * str)
{
   EORB_SL_ProtocolProperties * props = EORB_SL_ProtocolProperties__alloc ();
   char * ctx = NULL;

   str = strtok_r (str, ":", &ctx);
   if (str == NULL) goto error;
   props->m_server_host = CORBA_string_dup (str);
```

```
   str = strtok_r (NULL, ":", &ctx);
   if (str == NULL) goto error;
   props->m_server_device = CORBA_string_dup (str);
   str = strtok_r (NULL, ":", &ctx);
   if (str == NULL) goto error;
   props->m_client_host = CORBA_string_dup (str);
   str = strtok_r (NULL, ":", &ctx);
   if (str == NULL) goto error;
   props->m_client_device = CORBA_string_dup (str);

   return props;

error:

   CORBA_free (props);
   return NULL;
}
```

The string passed to the function comes from an `-ORBListenEndpoints` ORB initialization argument used to initialize the ORB. The general form of this argument is:

```
-ORBListenEndpoints <profile>:<endpoint>
```

For example "`-ORBListenEndpoints iiop:10.1.0.28:2324`". The `<endpoint>` part of the string associated with the profile is what gets passed to the function.

7.13 Implementing the corbaloc Support Functions

The `corbaloc` functions support the reading and writing of `corbaloc` format IOR strings for example:

```
-ORBInitRef server=corbaloc:iiop:10.1.0.28:2324/MyServer
```

The profile specific part of the `corbaloc` is what needs to be parsed or generated from the two support functions.

7.13.1 Implementing the corbaloc Read Function

This function parses a `corbaloc` string endpoint and sets the appropriate fields in the transport profile for IIOP:

```
/* Parse host[:port] */

static void EORB_IIOP_fromString
(
   EORB_IOP_Profile * profile,
   char * str
)
{
   EORB_TCP_Profile * tcp = (EORB_TCP_Profile*) profile->m_etf_profile;
   char * separator = (char*) strchr (str, ':');

   if (separator)
   {
      *separator = '\0';
      tcp->m_port = atoi (separator + 1);
   }
   else
   {
```

```
      tcp->m_port = 2809;
   }
   tcp->m_host = CORBA_string_dup (str);
}
```

7.13.2 Implementing the corbaloc Write Function

This function outputs the endpoint fields of a transport profile (IIOP) as a string:

```
static char * EORB_IIOP_toString (EORB_IOP_Profile * profile)
{
   EORB_TCP_Profile * tcp = (EORB_TCP_Profile*) profile->m_etf_profile;
   char * str = CORBA_string_alloc (strlen (tcp->m_host) + 6);

   sprintf (str, "%s:%d", tcp->m_host, tcp->m_port);

   return str;
}
```

7.14 Implementation Notes

- All transport and profile components are reference counted by the ORB. To support this all components must be allocated with the internal allocator function `EORB_allocVar`. This allocation function also supports the registration of a destructor function which is called when the reference count of the allocated object is zero. The destructor function should be used to free any transport specific resource associated with the component and call the base class finalizer. This allocator function also zeros all members of the allocated type.
- No threads are used by client-side connections. The thread used to make a client-side request is used to establish the client-side connection when an object reference is first used (bound).
- By default for each transport installed in the ORB, when the root POA is activated a listener thread will be created. Objects created in the POA will have an endpoint for each active transport.
- When a server listener receives a connection request (in accept) a new thread is assigned from the POA thread pool to handle read/write operations on this connection. The listener thread re-enters the accept function to handle new connection requests. The listener thread will do a single initial read from the new connection to manage any initial connection semantics (such as thread pool allocation for real time requests or GIOP location requests).
- There is no concurrent thread access to any of the transport implementation functions. The connection read and write functions may be called concurrently from different POA request handling threads unless server side private connections are enabled.
- If a profile is flagged as only supporting one way calls (the `m_type` field in the profile registration structure), then any attempt to make a two way call using this transport will cause a `CORBA::COMM_FAILURE` system exception to be raised.
- For host local transports it can be useful to encode the host name into a `Profile` as well as the real transport endpoint, this ensures that if the same transport endpoint is used on different hosts then they are not considered to be the same.
- A transport can have the ORB send the `GIOP::CloseConnection` message from client to server or server to client when a connection is closed (by setting the `m_close` connection member). Enabling this behaviour for client connections should be used with care as the sending of this message in this direction was not supported until GIOP 1.2.

8 Using the ORB

8.1 Introduction

The major steps of creating a CORBA-based client-server application using Spectra ORB are:

Step 1: Declare the application's classes and/or interfaces in IDL. (Refer to the *Using the IDL Compiler* section of the *IDL Guide* for *Step 1:* through *Step 4:*.)

Step 2: Compile the IDL in order to create C sources for the stubs, skeletons and/or tie classes and interfaces.

Step 3: Write the application's server modules (if they are not being implemented by third parties). These are derived from the generated IDL *server implementation base* header files (containing the *skeleton* and/or *tie* declarations) and must include (using the `#include` directive) the servant implementation base file (an IDL generated file with an `_s` suffix).

Step 4: Write the application's client modules (if they are not being implemented by third parties). These modules must include (using the `#include` directive) the generated IDL client header file (containing the *stub* declarations).

Step 5: Compile the developer-written source code and IDL-generated source code with a C compiler for the required platform, ensuring that the compiled code is linked to the ORB libraries for that specific platform.

Step 6: Deploy the application's server and client components on the required platforms.

Step 7: Start the server and run the client.

This section describes the procedures, requirements and practical details needed to create CORBA-based applications with Spectra ORB. (Please note that this section is ***not*** intended as a tutorial of how to write CORBA-based applications.) The topics covered in this section include:

The topics covered in this section are:

- a general description of how to use the IDL compiler, `idlc`
- the requirements, procedures and settings for creating, compiling, linking, deploying and running applications which are common to all platforms supported by Spectra ORB
- the particular information needed to compile, link, deploy and run the developer-written and IDL-generated source code for each, specific platform supported by Spectra ORB

8.2 Conventions

The following conventions are used in this section:

`$EORBHOME` or `%EORBHOME%` - the directory where the ORB is installed

`$EORBENV` or `%EORBENV%` - the target platform architecture (e.g. *`win32-msdev-x86`*)

8.3 Using the IDL Compiler

Basic instructions for using the Spectra ORB C Edition's IDL to C compiler, `idlc`, are given here.

Detailed information, including descriptions of all of the compiler's command line options are provided in the *IDL Guide*. Special instructions for specific platforms, such as environment variable settings, are provided in the *Product and Installation Guide*.

! You should read the instructions provided in the *IDL Guide* and *Product Guide* before attempting to use the IDL compiler, *idlc*.

The idlc compiler must be in the system's PATH in order to run. The idlc compiler is run from the command line as:

```
% idlc [options] <source_files>
```

where

[options] is a list of zero or more command-line parameters.

<source_files> is a list of one or more developer-written IDL source files

The source files *must* have *idl* as the filename's extension, for example myfile.*idl*.

Using idlc with no parameters or with the *-help* option displays usage information. The complete list of command-line parameters are described in the *IDL Guide*: please refer to the instructions in the *IDL Guide* before using the idlc compiler.

The IDL compiler will create a number of C source files for each IDL file. The number, type and names of the generated output files depend on the command-line options used.

The standard generated source files used by clients are:

- a header file which has a *C* suffix and *.h* extension, e.g. *myfile.h*; this file contains stub declarations
- an implementation file which has a *C* suffix and *.c* extension, e.g. *myfile.c*; this file contains stub definitions

The standard generated source files used by servers are:

- a header file which has an *S* suffix and *.h* extension, e.g. *myfileS.h*; this file contains skeleton declarations
- an implementation file which has an *S* suffix and *.c* extension, e.g. *myfileS.c*; this file contains skeleton definitions.
- an optional implementation file which has an *S_i* suffix and *.c* extension, e.g. *myfile_i.c*; this file contains generated EPV structures and implementation skeletons.

Refer to the *IDL Guide* for the appropriate command line options to use to generate specific files, such as just the client files, for example.

Example

To generate the client and server stub and skeleton files from an IDL source file called *myapp.idl* use:

```
% idlc -both myapp.idl
```

This will generate:

- *myappC.h* and *myappC.c*, the C client header and implementation files (containing the stub)
- *myappS.h* and *myappS.c*, the C server header and implementing files (containing the skeleton)
- myappB.c, common (client and server) implementation functions.

8.4 Requirements Common to All Platforms

This section describes the requirements, procedures and settings for creating, compiling, linking, deploying and running applications common to all platforms supported by Spectra ORB . I*nformation specific to each supported platform is provided in the Product and Installation Guide*

Pre-built libraries for each supported platform are in the `$EORBHOME/lib/$EORBENV` directory. The Spectra ORB product includes examples which can be used to verify that the ORB and your host or target platform is correctly installed and configured.

An application's source file modules (i.e. the C coded files) may need to include the ORB's libraries that are listed in Section 6.3.1, *Libraries*

Developers must determine which methodologies their application clients will use to find or resolve servers, including their own application servers and other, third party servers (such as the *Naming Service*). Refer to Section 8.6.1 *Resolving Servers* for a list of alternative resolution approaches.

Spectra ORB licensing information pertaining to all platforms is provided in the Product and Installation Guide.

8.5 Compiling and Linking Applications

The procedures which are common to all platforms and compilers for compiling and linking the source files (including both the IDL generated C files and developer-written C implementations) are described in this section.

The following requirements are common to all platforms, unless otherwise stated.

- `$EORBHOME/bin/$EORBENV` must be added to the executable search path (generally `PATH`).
- `$EORBHOME/include` must be added to the compiler's include search path.
- `$EORBHOME/lib/$EORBENV` must be added to the linker's library search path

The developer-written client and server application files must be linked with the correct object files. The object files will have been compiled from the application's IDL-generated sources.

It is natural and correct to assume that the *client*-side files must be linked with object files compiled from the IDL generated *client* files. However, the CORBA architecture and specification requires that the server-side files must also be linked with the client files, not only with the equivalent server object files. Refer to *The Common Object Request Broker: Architecture and Specification* by the OMG.

Example

An IDL specification file, `myInterface.idl`, is used to generate C client and server files, including: `myInterfaceC.h` and `myInterface.c` for the client `myInterfaceS.h`, `myInterfacesS_i.c` and `myInterfaceS.c` for the server.

The C compiler will create the following object files from these files: `myInterface.o` for the client `myInterfaceS.o` and `myInterfacesS_i.o` for the server.

The developer written client-side files must be linked with `myInterface.o`. The developer written server-side files must be linked with `myInterfaceS.o` *and with* `myInterfaceC.o`.

8.5.1 Required Libraries

The Spectra ORB ibraries are described under Section 6.3.1, *Libraries*. Spectra ORB C Edition based client or server applications must be linked to:

- the ORB core (`ec_orb` and `ec_core`)
- a protocol profile (e.g. `ec_iiop` or `ec_diop`)
- a transport plug-in (e.g. `ec_tcp` or `ec_udp`)

- the platform abstraction library (`ec_os`)

Server applications, i.e. applications which are used as servers, must also be linked to the POA library, `ec_poa`.

8.5.2 Build Version Options

Spectra ORB based applications can be built with or without debug information.

Each Spectra ORB distribution comes with a set of debug libraries and a set of release libraries. The debug library names have an *_g* suffix in order to distinguish them from the standard release libraries. For example, standard release version of the `ec_orb` library is called *ec_orb.dll*; the debug version is *ec_orb_g.dll*.

The *-DE_DEBUG* switch must be used with the compiler if the compiled code is to be linked against the debug version of the ORB libraries since there is inline code that is conditional on the *E_DEBUG* symbol.

On some compilers and/or platforms, the compiler and linker switches that they must use are dependant on the configuration of their particular ORB distribution.

8.6 Deploying and Running Applications

Information about running applications which is common to all Spectra ORB supported platforms are described here. The aspects and procedures specific to particular platforms are described in Section 8.7 *Platform-Specific Requirements*.

Regardless of whether clients and servers are run from the same or different machines or any particular platform, they always:

- Run as separate processes. Clients and servers are started in their own, separate shells, windows or processes.
- Must be able to locate each other. Clients locate servers using one of the methods described in Section 8.6.1 *Resolving Servers* below. Servers locate clients using the internal mechanisms provided by the ORB.

8.6.1 Resolving Servers

An application's clients and server are run as separate processes. Subject to the limitations of particular platforms, developers can implement their client(s) so that they can find or *resolve* their server by either:

- reading the server's IOR from a file created by the server,
- using a corbaloc URL (this URL is output to the screen when the server is first invoked) or
- using the Naming Service.

8.7 Platform-Specific Requirements

For details on how to build and run Spectra ORB applications on specific platforms such as VxWorks or Integrity please refer to the *Spectra ORB Product and Installation Guide.*.

8.8 Application Creation Example

The following example demonstrates the procedures as described in the sections above for creating, compiling and deploying an application. We do this using the simple hello example.

Step 1: Declare the application's classes and/or interfaces in IDL.

The following IDL code declares the `GreetingService` interface in the file `hello.idl`

```
interface GreetingService
{
   string greeting (in string str);
};
```

Step 2: Create C sources by compiling the IDL.

The -both command line option is needed to generate both client-side and server-side code.

```
% idlc -both hello.idl
```

Step 3: Write the application's server modules.

The example server implementation file is called `server.c`, this implements the IDL-generated server interfaces and methods. This file must include the generated server-side header.

```
#include "helloS.h"
```

Step 4: Write the application's client modules.

The example client implementation file is called `client.c`, this makes calls to the gerneric CORBA and generated client-side functions. The file must include the generated client-side header.

```
#include "hello.h"
```

Step 5: Compile and link the C source code.

8.9 Build Directives

The example is being built as a *release* (not *debug*) build.

UNIX No additional directives are required other than using the libraries as listed under *Linking* below.

WIN Add the */MD*, */GX* and */GR* switches to the Microsoft Visual C++ Project's *Project options* box located on *C/C++* tab of the *Project Setting* dialogue pane before compiling.

8.10 Compiling

The example server and client modules are compiled with the following ORB-specific include and library directories:

```
$EORBHOME/include
$EORBHOME/lib/$EORBENV
```

Examples

UNIX A Linux Intel x86 host/target build using the *gcc* compiler would use:

```
% gcc -c -I$EORBHOME/include -L$EORBHOME/lib/linux-gcc-x86 server.c
```

WIN Using the Microsoft Visual C++ compiler for a Windows host/target, add *win32_msdev_x86* to the *Preprocessor definitions* box located on *C/C++* tab of the *Project Setting* dialogue pane before compiling.

8.11 Linking

The example server and client modules are linked with the following ORB-specific libraries:

```
ec_orb and ec_core
ec_iiop or ec_diop
ec_poa
ec_tcp or ec_udp
ec_os
```

The server must also be linked with the server *and* client object files, `helloS.o` and `hello.o` in this example.

UNIX A Linux Intel x86 host/target build using the *gcc* compiler would use:

```
% gcc -o server hello.o helloS.o -L$EORBHOME/lib/linux-gcc-x86 \
     -lec_orb -lec_core -lec_iiop -lec_poa -lec_tcp -lec_os
```

WIN Using the Microsoft Visual C++ compiler for a Windows host/target, add the libraries listed above (`ec_orb.lib`, etc.) to the *Object/libraries modules* box located on *Link* tab of the *Project Setting* dialogue pane before compiling.

Step 6: Deploy the application.

Copy the compiled executable files to the directories where they are intended to be run from. The server and client executables for this example have been designed to be run from the same directory.

Step 7: Start the server and run the client.

For this example, create separate shells or windows for the server and client: first run the server, then the client. The client obtains planet objects from the server, then prints each planet's name.

UNIX Run *server* then *client* in separate windows.

WIN Run *server.exe* then *client.exe* in separate Command Prompt windows. The following command line example uses the *start* command open a separate window for the server, then runs the client. The the names of each planet retrieved by the clients from the server is displayed on the command line.

```
./server/server
Hello example starting
-ORBInitRef
hello=IOR:010000001800000049444c3a4772656574696e67536572766963653a312e300002000
000004f41545000000010102001a0000002f746d702f75696f702e617a6174686f74682e633435
434b420000002400000001921800654f52425690ad2b0000000008656f72622d636500000000040
000000000000000000000000000040000000010102000a00000031302e312e302e32380060b524
00000001921800654f52425690ad2b0000000008656f72622d6365000000000400000000000000000
0000000
```

8.12 Threading Model

8.12.1 General

The *CORBA_Environment* object is used to support thread specific ORB data.

Each thread should use its own instance of this object. For servers this is done automatically by the POA, however for clients this needs to be done by the application programmer. An example of client-side Environment management can be seen in the threads client example code.

8.12.2 Client Side

It is the responsibility of the application programmer to implement the client-side threading model. Object references are not thread safe, so where multiple threads need to make calls to the same servant using the same object reference, then the use of the reference should be guarded with a mutex. It is possible to make concurrent calls to the same servant by using a duplicated object reference.

8.12.3 Server Side

The server-side concurrency model is managed by the POA with support for both private and shared connection models. For a private connection a single thread is used, whereas for shared connections a thread per request model is supported.

For shared connections, thread per request is the default threading model with threads being taken from the shared ORB thread pool.

By default, the POA will always use a shared connection model. However a private connection model can be used when both client and server are Spectra ORB implementations. The `-ORBPrivateServerConn` ORB initialization argument should be provided to the ORB initialization function for the server to enable this behaviour.

8.12.4 Thread Pools

All POA server threads are taken from a global thread pool which is created on ORB initialization. The pool is created with an initial number of threads and can create additional threads on demand up to a configurable maximum. When the thread pool is empty as all threads are handling requests and the maximum number of threads has been reached, then additional requests are queued up to a configurable maximum queue size. The following ORB initialization arguments are used to configure the global thread pool:

`ORBThreadPoolSize <n> :` sets the number of threads initially created when the pool is initialized. The default value for this policy is one.

`ORBThreadPoolMax <n> :` sets the maximum number of threads the pool can hold. The default value for this policy is zero, meaning that there is no limit to the number of threads in the pool.

`ORBThreadPoolQueue <n>` : sets the maximum number of jobs that can be queued for processing when the thread pool is exhausted. If this limit is exceeded then a `NO_RESOURCES` system exception is thrown. The default value for this policy is zero, meaning that there is no limit to the number of queued jobs.

`ORBThreadStack <n>` : sets the stack size for threads created in the pool. The default value for this policy is zero meaning that the operating system determines the stack size for created threads.

`ORBThreadPriority <n>` : sets the native priority for threads created in the pool. The default value for this policy is zero meaning that the operating system determines the priority for created threads.

If the maximum number of threads is set too low then the ORB may not be able to handle requests since all available threads will be waiting on connections. As a minimum, there should be one thread per transport listener, plus at least one thread per connection. The ORB thread pool should not be configured to have a maximum of less than two threads for this reason.

When using the real-time ORB, the threading model of a POA can be explicitly controlled via thread pool policies.

9 Bibliography

The documents listed here may provide useful information or help for CORBA users and developers.

[1] *The Common Object Request Broker: Architecture and Specification Revision 2.3*, June 1999, Object Management Group (OMG), *http://www.omg.org*.

[2] *The Common Object Request Broker: Architecture and Specification*, *Revision 3.03*, March 2004, Object Management Group (OMG) , *http://www.omg.org*.

[3] *Minimum CORBA Specification*, *Version 1.0*, August 2002, Object Management Group (OMG), *http://www.omg.org*.

[4] *C Language Mapping Specification*, June 1999, Object Management Group (OMG), *http://www.omg.org*.

[5] *Software Communications Architecture Specification, JTRS-5000 SCA V2.2.1* April 2004, *http://jtrs.army.mil*

10 Appendices

Appendix A – Spectra ORB CORBA Profiles

Table 6 Provides a more detailed breakdown of the features and APIs defined in each of the CORBA Profiles supported by Spectra ORB C Edition. In a small number of cases the Profile may indicate that a feature is mandated but at the current time is not supported by Spectra ORB. These instances are indicated with a red tick.

Please note that the Minimum CORBA Profile does not make any explicit references to the inclusion of CORBA Services such as the Naming Service and these are listed as optional in *Table 6* . Spectra ORB C Edition includes implementations of Naming (Full and Lightweight), Even and Lightweight Log Services.

Scope	Operation/ Feature	Minimum CORBA	CORBA/e Compact	CORBA/e Micro	SCA 4 Full Profile	SCA 4 Lightweight Profile
IDL	Abstract Interfaces	✓	×	×	×	×
IDL	Value Type	✓	✓	✓	×	×
IDL	any	✓	✓	×	✓	
IDL	operation context clauses	✓	×	×	×	×
IDL	boolean, octet, short, unsigned short, long, unsigned long, enum, float, double, long double, long long, unsigned long long, char, string	✓	✓	✓	✓	✓
IDL	wide character string	✓	✓	✓	×	×
IDL	unions	✓	✓	✓	✓	✓
IDL	arrays	✓	✓	✓	✓	✓
IDL	structs	✓	✓	✓	✓	✓
IDL	sequence	✓	✓	✓	✓	✓
IDL	import	✓	×	×	×	×
CORBA::ORB	orb_init	✓	✓	✓	✓	✓
CORBA::ORB	id	✓	✓	✓	×	×
CORBA::ORB	object_to_string	✓	✓	✓	✓	✓
CORBA::ORB	string_to_object	✓	✓	✓	✓	✓
CORBA::ORB	get_service_information	✓	✓	✓	×	×
CORBA::ORB	list_initial_services	×	✓	✓	×	×
CORBA::ORB	resolve_initial_references	✓	✓	✓	✓	×
CORBA::ORB	work_pending	×	✓	✓	✓	✓
CORBA::ORB	perform_work	×	✓	✓	✓	✓
CORBA::ORB	run	×	✓	✓	✓	✓
CORBA::ORB	shutdown	✓	✓	✓	✓	✓
CORBA::ORB	destroy	✓	✓	✓	✓	✓
CORBA::ORB	create_policy	✓	✓	✓	✓	×
CORBA::ORB	register_value_factory	✓	✓	✓	×	×
CORBA::ORB	unregister_value_factory	✓	✓	✓	×	×
CORBA::ORB	lookup_value_factory	✓	✓	✓	×	×
CORBA::ORB	register_initial_reference	✓	✓	✓	×	×
CORBA::OBJECT	get_interface	✓	✓	✓	×	×
CORBA::OBJECT	is_nil	✓	✓	✓	✓	×
CORBA::OBJECT	duplicate	✓	✓	✓	✓	×
CORBA::OBJECT	release	✓	✓	✓	✓	×
CORBA::OBJECT	is_a	×	✓	✓	✓	×
CORBA::OBJECT	non_existent	×	✓	✓	✓	×
CORBA::OBJECT	is_equivalent	×	✓	✓	✓	×
CORBA::OBJECT	hash	✓	✓	✓	×	×
CORBA::OBJECT	get_policy	✓	✓	✓	✓	×
CORBA::OBJECT	set_policy_overrides	✓	✓	✓	✓	×
CORBA::OBJECT	get_client_policy	✓	✓	✓	✓	×
CORBA::OBJECT	get_policy_overrides	✓	✓	✓	✓	×
CORBA::OBJECT	validate_connection	✓	✓	✓	✓	✓

Scope	Operation/ Feature	Minimum CORBA	CORBA/e Compact	CORBA/e Micro	SCA 4 Full Profile	SCA 4 Lightweight Profile
CORBA::OBJECT	get_orb	✓	✓	✓	✓	×
CORBA::OBJECT	get_component	✓	×	×	×	×
CORBA::Policy	policy_type	✓	✓	✓	✓	×
CORBA::Policy	copy	✓	✓	✓	✓	×
CORBA::Policy	destroy	✓	✓	✓	✓	×
CORBA::PolicyManager	get_policy_overrides	×	✓	✓	✓	×
CORBA::PolicyManager	set_policy_overrides	×	✓	✓	✓	×
CORBA::TypeCode		✓	✓	✓	✓	×
CORBA::PolicyCurrent		×	✓	✓	✓	×
Messaging::RebindPolicy	rebind_mode	×	✓	✓	×	×
Messaging::SyncScopePolicy	synchonization	×	✓	✓	✓	×
Messaging::RequestEndTimePolicy	end_time	×	✓	✓	×	×
Messaging::ReplyEndTimePolicy	end_time	×	✓	✓	×	×
Messaging::RelativeRequestTimeoutPolicy	relative_expiry	×	✓	✓	×	×
Messaging::RelativeRoundtripTimeoutPolicy	relative_expiry	×	✓	✓	×	×
PortableServer::LifespanPolicy	value	✓	✓	✓	✓	×
PortableServer::IdUniquenessPolicy	value	✓	✓	✓	×	×
PortableServer::IdAsssignmentPolicy	value	✓	✓	✓	✓	×
PortableServer::POAManager	activate	✓	✓	✓	✓	✓
PortableServer::POAManager	get_state	×	✓	✓	×	×
PortableServer::POAManager	get_id	×	✓	✓	✓	×
PortableServer::POA	create_poa	✓	✓	×	✓	×
PortableServer::POA	find_poa	✓	✓	×	✓	×
PortableServer::POA	destroy	✓	✓	✓	✓	×
PortableServer::POA	create_lifespan_policy	✓	✓	×	✓	×
PortableServer::POA	create_id_uniqueness_policy	✓	✓	×	✓	×
PortableServer::POA	create_id_assignment_policy	✓	✓	×	✓	×
PortableServer::POA	the_name	✓	✓	✓	✓	×
PortableServer::POA	the_parent	✓	✓	✓	✓	×
PortableServer::POA	the_POAManager	✓	✓	✓	✓	✓
PortableServer::POA	activate_object	✓	✓	✓	✓	✓
PortableServer::POA	activate_object_with_id	✓	✓	×	✓	✓
PortableServer::POA	deactivate_object	✓	✓	✓	✓	✓
PortableServer::POA	create_reference	✓	✓	✓	✓	×
PortableServer::POA	create_reference_with_id	✓	✓	✓	✓	×
PortableServer::POA	servant_to_id	✓	✓	✓	✓	×
PortableServer::POA	servant_to reference	✓	✓	✓	✓	×
PortableServer::POA	reference_to_servant	✓	✓	✓	✓	×
PortableServer::POA	reference_to_id	✓	✓	✓	✓	×
PortableServer::POA	id_to_servant	✓	✓	✓	✓	×
PortableServer::POA	id_to_reference	✓	✓	✓	✓	×
PortableServer::Current	get_POA	✓	✓	✓	✓	×
PortableServer::Current	get_object_id	✓	✓	✓	✓	×
CORBA::Current	get_reference	×	✓	✓	✓	×
CORBA::Current	get_servant	×	✓	✓	✓	×
RTCORBA::ServerProtocolPolicy		×	×	×	✓	×
RTCORBA::PriorityModelPolicy	CLIENT_PROPAGATED	×	✓	×	✓	×

Scope	Operation/ Feature	Minimum CORBA	CORBA/e Compact	CORBA/e Micro	SCA 4 Full Profile	SCA 4 Lightweight Profile
RTCORBA::PriorityModelPolicy	SERVER_DECLARED	×	✓	×	✓	×
RTCORBA::PriorityBandedConnectionPolicy	priority_bands	×	✓	×	✓	×
RTCORBA::Thread Pools	create_threadpool	×	✓	×	✓	×
RTCORBA::Thread Pools	create_threadpool_with_lanes	×	✓	×	✓	×
RTCORBA::Current	the_priority	×	✓	×	✓	×
RTCORBA::Mutex	lock	×	✓	✓	×	×
RTCORBA::Mutex	unlock	×	✓	✓	×	×
RTCORBA::Mutex	try_lock	×	✓	✓	×	×
RTCORBA::RTORB	create_mutex	×	✓	✓	×	×
RTCORBA::RTORB	destroy_mutex	×	✓	✓	×	×
RTCORBA::RTORB	create_priority_model_policy	×	✓	×	✓	×
RTCORBA::RTORB	create_priority_banded_connection_ policy	×	✓	×	✓	×
RTPortableServer::POA	activate_object_with_priority	×	✓	×	✓	×
CosNaming::NamingContext		o	✓	×	×	×
CosNaming::BindingIterator		o	✓	×	×	×
CosNaming::NamingContextExt		o	✓	×	×	×
CosEventComm::PushConsumer		o	✓	×	✓	×
CosEventComm::PushSupplier		o	✓	×	✓	×
CosEventComm::PullConsumer		o	✓	×	×	×
CosEventComm::PullSupplier		o	✓	×	×	×
CosEventChannelAdmin::ProxyPushConsumer		o	✓	×	×	×
CosEventChannelAdmin::ProxyPushSupplier		o	✓	×	×	×
CosEventChannelAdmin::ProxyPullConsumer		o	✓	×	×	×
CosEventChannelAdmin::ProxyPullSupplier		o	✓	×	×	×
CosEventChannelAdmin::ConsumerAdmin		o	✓	×	×	×
CosEventChannelAdmin::SupplierAdmin		o	✓	×	×	×
CosEventChannelAdmin::EventChannel		**o**	✓	×	×	×
CosLwLog::LogProducer		o	✓	×	✓	×
CosLwLog::LogConsumer		o	✓	×	✓	×
CosLwLog::LogStatus		o	✓	×	✓	×
CosLwLog::LogAdministrator		o	✓	×	✓	×
GIOP		✓	✓	✓	✓	✓
IIOP		✓	✓	✓	✓	✓

Table 6: Spectra ORB C Edition CORBA Profiles Mapping