ORBacus For C++ and Java

Version 3.1.2

Copyright (c) 1999 Object-Oriented Concepts, Inc. All Rights Reserved.

"Object-Oriented Concepts", "ORBacus" and "JThreads/C++" are trademarks or registered trademarks of Object-Oriented Concepts, Inc.

"OMG", "CORBA", and "Object Request Broker" are trademarks or registered trademarks of the Object Management Group.

"Java" is a trademark of Sun Microsystems, Inc.

"Netscape" is a registered trademark of Netscape Communications Corporation.

Other names, products, and services may be the trademarks or registered trademarks of their respective holders.

CHAPTER 1 Introduction 11

What is ORBacus? 11 How is it licensed? 12 About this Document 12 Getting Help 12

CHAPTER 2 Getting Started 15

The "Hello World" Application 15 The IDL Code 16 Implementing the Example in C++ 16 Implementing the Server 17 19 Implementing the Client Compiling and Linking 20 Running the Application 21 Implementing the Example in Java 21 Implementing the Server 21 Implementing the Client 23 Compiling 24 25 Running the Application Summary 25 25 Where to go from here

CHAPTER 3 The ORBacus Code Generators 27

Overview 27 Synopsis 27 Description 28 Options for idl 29 Options for jidl 31 Options for ridl 32 Options for ridl 32 Options for irserv 33 Options for irfeed 34 Options for irdel 34 Options for irgen 34 The IDL-to-C++ Translator and the Interface Repository Include Statements Documenting IDL Files Using javadoc

CHAPTER 4

4 ORB and BOA Initialization 43

ORB Initialization 43 Initializing the C++ ORB 43 Initializing the Java ORB for Applications 43 Initializing the Java ORB in JDK 1.2 44 Initializing the Java ORB for Applets 44 BOA Initialization 45 Initializing the C++ BOA 45 Initializing the Java BOA 45 Initializing the Java BOA in JDK 1.2 45 Configuring the ORB and BOA 46 Properties 46 Command-line Options 49 Filtering Command-line Options 50 Using a Configuration File 51 Using the Windows NT Registry 52 Defining Properties in Java 53 Precedence of Properties 53 Advanced Property Usage 54 Server Event Loop 58 Mixed Client/Server Applications 58 Deactivating the Server 59 Applets 61 Adding ORBacus Applets to Web Pages 61 Defining ORB and BOA Options for an Applet 62 Defining the ORB Class Parameters 62 Security Issues 62

CHAPTER 5

CORBA Objects 65

Overview **65** Implementing Servants **66**

Implementing Servants using Inheritance	67
Implementing Servants using Delegation	69
Creating Servants 73	
Creating Servants using C++ 73	
Creating Servants using Java 74	
Connecting Servants 75	
Connecting Servants using C++ 75	
Connecting Servants using Java 75	
Named Servants 76	
Factory Objects 77	
Factory Objects using C++ 78	
Factory Objects using Java 79	
Caveats 80	
Getting a Servant from a Reference 81	
Getting a Servant using C++ 81	
Getting a Servant using Java 83	

CHAPTER 6 Locating Objects 85

Obtaining Object References 85 Lifetime of Object References 87 Hostname 88 Port Number 88 Object Key 88 Stringified Object References 89 Using a File 89 Using a URL 91 Using Applet Parameters 92 Connecting to Named Objects 92 Using the iiop:// Notation 93 Using get_inet_object 93 Initial Services 94 Resolving an Initial Service 94 Providing IORs of Initial Services 96

CHAPTER 7 Reference Counting 99

What is Reference Counting? 99

Reference Counting in Java 99 Reference Counting in C++ 100 Marshalling Issues 100 Releasing Proxies and Servants 102 Global Object References 103 Cyclic Object Dependencies 104

109

CHAPTER 8 C++Mapping Notes

Reserved Names 109 Mapping of Modules 109 Extensions 110 Extensions to the String Type 110 Extensions to _var Types 110 Extensions to Sequence Types 111 C++ Mapping Tips & Tricks 112 CORBA Strings 112 Object References 116

CHAPTER 9 Concurrency Models 121

Introduction 121 What is a Concurrency Model? 121 Why different Concurrency Models? 121 ORBacus Concurrency Models Overview 122 Single-Threaded Concurrency Models 122 Blocking Clients and Servers 122 Reactive Clients and Servers 123 Multi-Threaded Concurrency Models 126 Threaded Clients and Servers 126 Thread-per-Client Server 127 Thread-per-Request Server 128 Thread Pool Server 128 Performance Comparisons 129 Sample Application 129 Regular Method Invocations 130 Nested Method Invocations 131 Selecting Concurrency Models 132

CHAPTER 10 The Reactor 135

What is a Reactor? 135
Available Reactors 135 *The X11 Reactor* 136 *The Windows Reactor* 137
Writing a Custom Event Handler 137
Using Timers 139

CHAPTER 11 The Open Communications Interface 141

What is the Open Communications Interface? 141
Interface Summary 141
Buffer 141
Transport 142
Acceptor and Connector 142
Connector Factory 142
The Registries 142
The Info Objects 142
Class Diagram 143
OCI Reference 143
OCI for the Application Programmer 144
A "Converter" Class for Java 144
Getting Hostnames and Port Numbers 145
Finding out a Client's IP Address 148

CHAPTER 12 Using Policies 151

Overview 151 Supported Policies 152 Examples 152 Connection Reuse Policy at ORB Level 153 Connection Reuse Policy at Object Level 154

CHAPTER 13 ORBacus Basic Services 155

Configuring and Using a Basic Service 155 Starting the Service 156 Connecting to the Service 156 Object Names for the Basic Services 157 The Naming Service 158 Properties 158 Command-line Options 158 Creating Bindings 159 Name Resolution 160 Persistence 161 A Simple Example 161 Windows NT Native Service 166 The Property Service 168 Command-line Options 168 Creating Properties 168 Querying for Properties 169 Deleting Properties 171 A Simple Example 171 The Event Service 173 Properties 173 Command-line Options 174 Diagnostics 174 NT Native Service 175 The Event Channel 176 Event Suppliers and Consumers 176 Event Channel Policies 177 A Simple Example 177

CHAPTER 14 Exceptions and Error Messages 183

CORBA System Exceptions 183 Non-Compliant Application Asserts 186

APPENDIX A ORBacus Policy Reference 191

Module SSL 191

Interface SSL::ConnectPolicy 192 Module OB 193 Interface OB::ProtocolPolicy 194 Interface OB::ConnectionReusePolicy 195 Interface OB::ReconnectPolicy 196 Interface OB::TimeoutPolicy 197

APPENDIX B Open Communications Interface Reference 199

Module OCI 199 Interface OCI::Buffer 201 Interface OCI::Transport 203 Interface OCI::TransportInfo 207 Interface OCI::CloseCB 209 Interface OCI::Connector 210 Interface OCI::ConnectorInfo 212 Interface OCI::ConnectCB 213 Interface OCI::Acceptor 214 Interface OCI::AcceptorInfo 216 Interface OCI::AcceptCB 217 Interface OCI::ConFactory 218 Interface OCI::ConFactoryInfo 221 Interface OCI::ConFactoryRegistry 222 Interface OCI::AccRegistry 224 Interface OCI::Current 226 Module OCI::IIOP 227 Interface OCI::IIOP::TransportInfo 228 Interface OCI::IIOP::ConnectorInfo 229 Interface OCI::IIOP::AcceptorInfo 230 Interface OCI::IIOP::ConFactoryInfo 231

APPENDIX C Royalty-Free Public License Agreement 233

References 239

CHAPTER 1 Introduction

1.1 What is ORBacus?

ORBACUS is an Object Request Broker (ORB) that is compliant with the Common Object Request Broker Architecture (CORBA) specification as defined in "The Common Object Request Broker: Architecture and Specification" [2] and "IDL/Java Language Mapping" [3].

These are some of the highlights of the ORBACUS 3.1 release:

- Full CORBA IDL support
- Complete CORBA IDL-to-C++ mapping
- Complete CORBA IDL-to-Java mapping
- Includes the Basic CORBA Services Naming, Event and Property
- Pluggable Protocols with IIOP as the default plug-in
- Single- and Multi-Threaded support with several concurrency models: *Blocking*, *Reactive*, *Threaded*, *Thread-per-Client*, *Thread-per-Request* and *Thread Pool*
- Nested method invocations, even in the single-threaded version
- Support for timeouts
- Seamless event loop integration with X11 and Windows
- Full support for dynamic programming: Dynamic Invocation Interface, Dynamic Skeleton Interface, Interface Repository and DynAny

• IDL-to-HTML and IDL-to-RTF translators for generating "javadoc"-style documentation

This version of ORBACUS has the following limitations:

• Only persistent (i.e., manually launched) servers are currently supported.

For platform availability, please refer to the ORBACUS home page at http://www.ooc.com/ob/.

1.2 How is it licensed?

ORBACUS is licensed as "free for non-commercial use." See the license agreement in Appendix C for details. For information on commercial licenses, please see the pricing information on our Web site, or contact support@ooc.com.

1.3 About this Document

This manual is - except for the "Getting Started" chapter - no replacement for a good CORBA book. This manual also does not contain the precise specifications of the CORBA standard, which are freely available on-line. A good grasp of the CORBA specifications in [2] and [3] is absolutely necessary to effectively use this manual. In particular, the chapters in [2] covering CORBA IDL and the IDL-to-C++ mapping should be studied thoroughly.

Do not expect any of the introductory CORBA books to provide a reference for the IDL-to-C++ mapping. The books that are currently available only give an overview and are neither complete nor up-to-date. *There is no substitute for the official CORBA specifica-tion as defined in [2].*

What this manual does contain, however, is information on *how* ORBACUS implements the CORBA standard. A shortcoming of the current CORBA specification is that it leaves a high degree of freedom to the CORBA implementation. For example, the precise semantics of a oneway call are not specified by the standard.

To make it easier to get started with ORBACUS, this manual contains a "Getting Started" chapter, explaining some ORBACUS basics with a very simple example.

1.4 Getting Help

Should you need any assistance with ORBACUS, do not hesitate to contact us at support@ooc.com. You might also consider joining our ORBACUS mailing list. To do so, send a message to majordomo@ooc.com (not ob@ooc.com) with

subscribe ob

in the body (not in the Subject: field) of your message. To unsubscribe, use

unsubscribe ob

in the body of your message. To send a message to the list, mail to ob@ooc.com (not majordomo@ooc.com).

An archive of the ORBACUS mailing list can be found at http://www.ooc.com/ob/mailing-list.html.

Introduction

CHAPTER 2 Getting Started

2.1 The "Hello World" Application

The example described in this chapter is founded on a well-known application: A "Hello World!" program presented here in a special client-server version.

Many books on programming start with this tiny demo program. In introductory C++ books you'll probably find the following piece of code in the very first chapter:

```
#include <iostream.h>
int
main(int, char*[], char*[])
{
    cout << "Hello World!" << endl;
    return 0;
}
Or in introductory Java books:
// Java
public class Greeter
{
    public static void main(String args[])</pre>
```

// C++

```
{
    System.out.println("Hello World!");
}
```

These applications simply print "Hello World!" to standard output and that is exactly what this chapter is about: Printing "Hello World!" with a CORBA-based client-server application. In other words, we will develop a client program that invokes a hello operation on an object in a server program. The server responds by printing "Hello World!" on its standard output.

2.2 The IDL Code

How do we write a CORBA-based "Hello World!" application? The first step is to create a file containing our IDL definitions. Since our sample application isn't a complicated one, the IDL code needed for this example is simple:

```
1 // IDL
2
3 interface Hello
4 {
5 void hello();
6 };
```

- 3 An interface with the name Hello is defined. An IDL interface is conceptually equivalent to a pure abstract class in C++, or to an interface in Java.
- 5 The only operation defined is hello, which neither takes any parameters nor returns any value.

2.3 Implementing the Example in C++

The next step is to translate the IDL code to C++ using the IDL-to-C++ translator. Save the IDL code shown above to a file called Hello.idl. Now translate the code to C++ using the following command:

idl Hello.idl

This command will create the files Hello.h, Hello.cpp, Hello_skel.h and Hello_skel.cpp.

2.3.1 Implementing the Server

To implement the server, we need to define an implementation class for the Hello interface. To do this, we create a class Hello_impl that is derived from the "skeleton" class Hello_skel, defined in the file Hello_skel.h. The definition for Hello_impl looks like this:

```
1 // C++
2
3 #include <Hello_skel.h>
4
5 class Hello_impl : public Hello_skel
6 {
7 public:
8
9 virtual void hello();
10 };
```

- 3 Since our implementation class derives from the skeleton class Hello_skel, we must include the file Hello_skel.h.
- 5 Here we define Hello_impl as a class derived from Hello_skel.
- 9 Our implementation class must implement all operations from the IDL interface. In this case, this is just the operation hello.

The implementation for Hello_impl looks as follows:

```
1 // C++
2
3 #include <OB/CORBA.h>
4 #include <Hello_impl.h>
5
6 void
7 Hello_impl::hello()
8 {
9     cout << "Hello World!" << endl;
10 }</pre>
```

- 3 We must include OB/CORBA.h, which contains definitions for the standard CORBA classes, as well as for other useful things.
- 4 We must also include the Hello_impl class definition, contained in the header file Hello_impl.h.

6-10 The hello function simply prints "Hello World!" on standard output.

Save the class definition of Hello_impl in the file Hello_impl.h and the implementation of Hello_impl in the file Hello_impl.cpp.

Now we need to write the server's main program, which looks like this:

```
1 // C++
 2
 3 #include <OB/CORBA.h>
 4 #include <Hello_impl.h>
 5
 6 #include <fstream.h>
 7
 8 int
 9 main(int argc, char* argv[], char*[])
10 {
      CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
11
12
      CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
13
14
      Hello_var p = new Hello_impl;
15
16
      CORBA_String_var s = orb -> object_to_string(p);
      const char* refFile = "Hello.ref";
17
      ofstream out(refFile);
18
      out << s << endl;
19
      out.close();
20
21
22
      boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
23
   }
```

- 3-6 Several header files are included. Of these, OB/CORBA.h provides the standard CORBA definitions, and Hello_impl.h contains the definition of the Hello_impl class.
- 11,13 The first thing a CORBA program has to do is to initialize the ORB¹ and the BOA². This is done by CORBA_ORB_init and BOA_init. Both operations expect the parameters with which the program was started. These parameters may or may not be used by the ORB and BOA, depending on the CORBA implementation. ORBACUS recognizes certain options that will be explained later.
 - 1. Object Request Broker
 - 2. Basic Object Adapter

- 14 An instance of Hello_impl is created. Hello_var, like all _var types, is a "smart" pointer, i.e., p will release the object created by new Hello_impl automatically when p goes out of scope.
- 16-20 The client must be able to access the implementation object. This can be done by saving a "stringified" object reference to a file which can be read by the client and converted back to the actual object reference.¹ The operation <code>object_to_string()</code> converts a CORBA object reference into its string representation.
 - 22 Finally, in order to react to incoming requests, the server must enter its event loop. This is done by calling impl_is_ready. Since ORBACUS does not use the CORBA_ImplementationDef argument, CORBA_ImplementationDef::_nil() can be used as a dummy argument.

Save this to a file with the name Server.cpp.

2.3.2 Implementing the Client

Writing the client requires less work than writing the server, since the client, in this example, only consists of the main function. In several respects the client's main is similar to the server's main function:

```
// C++
 1
 2
 3 #include <OB/CORBA.h>
   #include <Hello.h>
 Δ
 5
   #include <fstream.h>
 6
 7
   int
 8
  main(int argc, char* argv[], char*[])
 9
10
   {
      CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
11
12
13
      const char* refFile = "Hello.ref";
14
      ifstream in(refFile);
15
      char s[1000];
      in >> s;
16
      CORBA_Object_var obj = orb -> string_to_object(s);
17
18
```

^{1.} If your application contains more than one object, you do not need to save object references for all objects. Usually you save the reference of one object which provides operations that can subsequently return references to other objects.

```
19 Hello_var hello = Hello::_narrow(obj);
20
21 hello -> hello();
22 }
```

- 4 In contrast to the server, the client does not need to include Hello_impl.h. Only the generated file Hello.h is needed.
- 11 Like the server's implementation of main, the client's main starts with the initialization of the ORB. It's not necessary to initialize the BOA, because the BOA is only needed by server applications.
- 13-17 The "stringified" object reference written by the server is read and converted to a CORBA_Object object reference.
 - 19 The _narrow operation generates a Hello object reference from the CORBA_Object object reference.¹
 - 21 Finally, the hello operation on the hello object reference is invoked, causing the server to print "Hello World!".

Save this into the file Client.cpp.

2.3.3 Compiling and Linking

Both the client and the server must be linked with the compiled Hello.cpp, which usually has the name Hello.o under Unix and Hello.obj under Windows. The compiled Hello_skel.cpp and Hello_impl.cpp are only needed by the server.

Compiling and linking is to a large degree compiler- and platform-dependent. Many compilers require unique options to generate correct code. To build ORBACUS programs, you must at least link with the ORBACUS library libOB.a (Unix) or ob.lib (Windows). Additional libraries are required on some systems, such aslibsocket.a and libnsl.a for Solaris or wsock32.lib for Windows.

The ORBACUS distribution comes with various README files for different platforms which give hints on the options needed for compiling and the libraries necessary for linking. Please consult these README files for details.

^{1.} Although CORBA's T::_narrow for an interface T works similar to dynamic_cast<T>() for a plain C++ class T, dynamic_cast<T>() must not be used for CORBA object references.

2.3.4 Running the Application

Our "Hello World!" application consists of two parts: the client program and the server program. The first program to be started is the server, because it must create the file Hello.ref that the client needs in order to connect to the server. As soon as the server is running, you can start the client. If all goes well, the "Hello World!" message will appear on the screen.

2.4 Implementing the Example in Java

In order to implement this application in Java, the interface specified in IDL is translated to Java classes similar to the way the C++ code was created. The ORBACUS IDL-to-Java translator jidl is used like this:

jidl --package hello Hello.idl

This command results in several Java source files on which the actual implementation will be based. The generated files are Hello.java, HelloHelper.java, HelloHolder.java, StubForHello.java and _HelloImplBase.java, all generated in a directory with the name hello.

2.4.1 Implementing the Server

The server's Hello implementation class looks as follows:

```
// Java
 1
 2
 3 package hello;
 4
 5 public class Hello_impl extends _HelloImplBase
 6
   {
 7
      public void hello()
 8
      {
          System.out.println("Hello World!");
 9
10
       }
11
```

- 5 The implementation class Hello_impl must inherit from the generated class _HelloImplBase.
- 7-9 As with the C++ implementation, the hello method simply prints "Hello World!" on standard output.

Save this class to the file Hello_impl.java.

We also have to write a class which holds the server's main method. We call this class Server, saved in the file Server.java:

```
1 // Java
 2
 3 package hello;
 4
 5 public class Server
 6 {
 7
      public static void main(String args[])
 8
      {
 9
          java.util.Properties props = System.getProperties();
          props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
10
11
          props.put("org.omg.CORBA.ORBSingletonClass",
12
                    "com.ooc.CORBA.ORBSingleton");
13
          System.setProperties(props);
14
15
          org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
          org.omg.CORBA.BOA boa =
16
17
              ((com.ooc.CORBA.ORB)orb).BOA_init(args, props);
18
19
          Hello_impl p = new Hello_impl();
20
21
          try
22
          {
             String ref = orb.object_to_string(p);
23
24
             String refFile = "Hello.ref";
             java.io.PrintWriter out = new java.io.PrintWriter(
25
26
                 new java.io.FileOutputStream(refFile));
27
             out.println(ref);
28
             out.flush();
29
          }
          catch(java.io.IOException ex)
30
31
          {
32
             System.err.println("Can't write to `" +
33
                                ex.getMessage() + "'");
34
             System.exit(1);
35
          }
36
          boa.impl_is_ready(null);
37
38
       }
39
```

9-13 These properties are necessary to use the ORBacus ORB with JDK 1.2.

- 15-17 The ORB and BOA must be initialized. This is done using ORB.init and ORB.BOA_init. Note that all standard CORBA definitions are in the package org.omg.CORBA. That is, you must either import this package, or, as shown in our example, you must use org.omg.CORBA explicitly.
 - 19 An instance of Hello_impl is created. This instance is released automatically when it is not used anymore.
- 21-35 The object reference is "stringified" and written to a file.
 - 37 Finally, the server enters its event loop to receive incoming requests.

2.4.2 Implementing the Client

Save this to a file with the name Client.java:

```
// Java
 1
 2
 3 package hello;
 4
 5 public class Client
 6 {
 7
      public static void main(String args[])
 8
       {
 9
          java.util.Properties props = System.getProperties();
          props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
10
          props.put("org.omg.CORBA.ORBSingletonClass",
11
12
                    "com.ooc.CORBA.ORBSingleton");
13
          System.setProperties(props);
14
15
          org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
16
17
          String ref = null;
18
          try
19
          {
20
             String refFile = "Hello.ref";
21
             java.io.BufferedReader in = new java.io.BufferedReader(
22
                 new java.io.FileReader(refFile));
             ref = in.readLine();
23
          }
24
          catch(java.io.IOException ex)
25
26
          {
             System.err.println("Can't read from `" +
27
                                ex.getMessage() + "'");
28
29
             System.exit(1);
```

```
30  }
31  org.omg.CORBA.Object obj = orb.string_to_object(ref);
32
33  Hello p = HelloHelper.narrow(obj);
34
35  p.hello();
36  }
37 }
```

9-13 These properties are necessary to use the ORBacus ORB with JDK 1.2.

- 15 The ORB is initialized. BOA initialization is not necessary for clients.
- 17-31 The stringified object reference is read and converted to an object.
 - 33 The object reference is "narrowed" to a reference to a Hello object. A simple Java cast is not allowed here, because it is possible that the client will need to ask the server whether the object is really of type Hello.
 - 35 Finally the hello operation is invoked, causing the server to print "Hello World!" on standard output.

2.4.3 Compiling

To compile the implementation classes and the classes generated by the ORBACUS IDLto-Java translator, use javac (or the Java compiler of your choice):

```
javac *.java hello/*.java
```

Ensure that your CLASSPATH environment variable includes the ORBACUS Java classes, i.e., the OB. jar file. If you are using the Unix Bourne shell or a compatible shell, you can do this with the following commands:

```
CLASSPATH=your_orbacus_directory/lib/OB.jar:$CLASSPATH export CLASSPATH
```

Replace your_orbacus_directory with the name of the directory where ORBACUS is installed.

If you are running ORBACUS on a Windows-based system, you can use the following command within the Windows command interpreter:

```
set CLASSPATH=your_orbacus_directory\lib\OB.jar;%CLASSPATH%
```

Note that for Windows you must use ";" and not ":" as the delimiter.

2.4.4 Running the Application

The "Hello World" Java server is started with:

java hello.Server

And the client with:

java hello.Client

Again, make sure that your CLASSPATH environment variable includes the OB. jar file.

You might also want to use a C++ server together with a Java client (or vice versa). This is one of the primary advantages of using CORBA: If something is defined in CORBA IDL, the programming language used for the implementation is irrelevant. CORBA applications can talk with each other, regardless of the language they are written in.

2.5 Summary

At this point, you might be inclined to think that this is the most complicated method of printing a string that you have ever encountered in your career as a programmer. At first glance, a CORBA-based approach may indeed seem complicated. On the other hand, think of the benefits this kind of approach has to offer. You can start the server and client applications on different machines with exactly the same results. Concerning the communication between the client and the server, you don't have to worry about platform-specific methods or protocols at all, provided there is a CORBA ORB available for the platform and programming language of your choice. If possible, get some hands-on experience and start the server on one machine, the client on another¹. As you will see, CORBA-based applications run interchangeably in both local and network environments.

One last point to note: you likely won't be using CORBA to develop systems as simple as our "Hello, World!" example. The more complex your applications become (and today's applications *are* complex), the more you will learn to appreciate having a high-level abstraction of your applications' key interfaces captured in CORBA IDL.

2.6 Where to go from here

To understand the remaining chapters of this manual, you *must* have read the CORBA specifications in [2] and [3]. You will not be able to understand the chapters that follow

^{1.} Note that after the startup of the server program, you have to copy the stringified object reference, i.e., the file Hello.ref, to the machine where the client program is to be run.

without a good knowledge of CORBA in general, CORBA IDL and the IDL-to-C++ and IDL-to-Java mappings.

CHAPTER 3

The ORBacus Code Generators

3.1 Overview

ORBACUS includes the following code generators and Interface Repository tools:

idl	The ORBACUS IDL-to-C++ Translator
jidl	The ORBACUS IDL-to-Java Translator
hidl	The ORBACUS IDL-to-HTML Translator
ridl	The ORBACUS IDL-to-RTF Translator
irserv	The ORBACUS Interface Repository Server
irfeed	The ORBACUS Interface Repository Feeder
irdel	The ORBACUS Interface Repository Deleter
irgen	The ORBACUS Interface Repository C++ Code Generator

3.2 Synopsis

idl [options] idl-files ...

jidl [options] idl-files ...

hidl [options] idl-files ...

ridl [options] idl-files ...

irserv [options] [idl-files...]

irfeed [options] idl-files...

irdel [options] scoped-name...

irgen name-base

3.3 Description

idl is the ORBACUS IDL-to-C++ translator. It translates IDL files into C++ files. For each IDL file, four C++ files are generated. For example,

idl MyFile.idl

produces the following files:

MyFile.h	Header file containing M_{y} File.idl's translated data types and interface stubs
MyFile.cpp	Source file containing MyFile.idl's translated data types and interface stubs
MyFile_skel.h	Header file containing skeletons for MyFile.idl's interfaces
MyFile_skel.cpp	Source file containing skeletons for MyFile.idl's interfaces

jidl translates IDL files into Java files. For every construct in the IDL file that maps to a Java class or interface, a separate class file is generated. Directories are automatically created for those IDL constructs that map to a Java package (e.g., a module).

jidl can also add comments from the IDL file starting with /** to the generated Java files. This allows you to use the javadoc tool to produce documentation from the generated Java files. See "Using javadoc" on page 39 for additional information.

hidl creates HTML files from IDL files. An HTML file is generated for each module and interface defined in an IDL file. Comments in the IDL file are preserved and javadoc style keywords are supported. The section "Documenting IDL Files" on page 36 provides more information.

ridl creates Rich Text Format (RTF) files from IDL files. An RTF file is generated for each module and interface defined in an IDL file. Comments in the IDL file are preserved

and javadoc style keywords are supported. The section "Documenting IDL Files" on page 36 provides more information.

irserv is the Interface Repository Server. Together with irfeed, a program that feeds the Interface Repository with IDL code, and irgen, the Interface Repository C++ Code Generator, it is possible to generate C++ code directly from the contents of an Interface Repository. See "The IDL-to-C++ Translator and the Interface Repository" on page 35 for an example.

3.4 Options for idl

-h, --help

Show a short help message.

-v, --version

Show the ORBACUS version number.

-e, --cpp NAME

Use NAME as the preprocessor program.

-d, --debug

Print diagnostic messages. This option is for ORBACUS internal debugging purposes only.

-DNAME

Defines NAME as 1. This option is directly passed to the preprocessor.

```
-DNAME=DEF
```

Defines NAME as DEF. This option is directly passed to the preprocessor.

-UNAME

Removes any definition for NAME. This option is directly passed to the preprocessor.

-IDIR

Adds DIR to the include file search path. This option is directly passed to the preprocessor.

--no-skeletons

Don't generate skeleton classes.

--no-type-codes

Don't generate type codes and insertion and extraction functions for the Any type. Use of this option will cause the translator to generate more compact code.

```
--locality-constrained
```

Generate locality-constrained objects.

```
--no-virtual-inheritance
```

Don't use virtual C++ inheritance. If you use this option, you cannot use multiple interface inheritance in your IDL code, and you also cannot use multiple C++ inheritance to implement your servant classes.

```
--tie
```

Generate tie classes for delegate-based interface implementations. Tie classes depend on the corresponding skeleton classes, i.e., you must not use --no-skeletons in combination with --tie.

```
--c-suffix SUFFIX
```

Use SUFFIX as the suffix for source files. The default value is . cpp.

```
--h-suffix SUFFIX
```

Use SUFFIX as the suffix for header files. The default value is . h.

--all

Generate code for included files instead of inserting #include statements. See "Include Statements" on page 35.

```
--no-relative
```

When generating code, idl assumes that the same -I options that are used with idl are also going to be used with the C++ compiler. Therefore idl will try to make all #include statements relative to the directories specified with -I. The option --no-relative suppresses this behavior, in which case idl will not make #include statements for include files relative to the paths specified with the -I option.

```
--header-dir DIR
```

This option can be used to make #include statements for header files relative to a specific directory.

```
--other-header-dir DIR
```

This option works like --header-dir, but it only applies to header files for included IDL files.

--output-dir DIR

Write generated files to directory DIR.

--dll-import DEF

Put DEF in front of every symbol that needs an explicit DLL import statement.

3.5 Options for jidl

```
-h, --help
-v, --version
-e, --cpp NAME
-d, --debug
-DNAME
-DNAME=DEF
-UNAME
-IDIR
--no-skeletons
--locality-constrained
--all
--tie
```

These options are the same as for the idl command.

--no-comments

The default behavior of jidl is to add any comments from the IDL file starting with /** to the generated Java files. Specify this option if you don't want these comments added to your Java files.

--package PKG

Specifies a package name for the generated Java classes. Each class will be generated relative to this package.

```
--prefix-package PRE PKG
```

Specifies a package name for a particular prefix¹. Each class with this prefix will be generated relative to the specified package.

--auto-package

^{1.} Prefix refers to the value of the #pragma prefix statement in an IDL file. For example, the statement #pragma prefix ooc.com defines "ooc.com" as the prefix. The prefix is included in the Interface Repository identifiers for all types defined in the IDL file.

Derives the package names for generated Java classes from the IDL prefixes. The prefix ooc.com, for example, results in the package com.ooc.

```
--output-dir DIR
```

Specifies a directory where jidl will place the generated Java files. Without this option the current directory is used.

--clone

Generates a clone method for struct, union, enum and exception types.

3.6 Options for hidl

-h, --help -v, --version -e, --cpp NAME -d, --debug -DNAME -DNAME=DEF -UNAME -IDIR

These options are the same as for the idl command.

--no-sort

Don't sort symbols alphabetically.

```
--output-dir DIR
```

Write HTML files to the directory DIR.

3.7 Options for ridl

-h, --help -v, --version -e, --cpp NAME -d, --debug -DNAME -DNAME=DEF -UNAME -IDIR

These options are the same as for the idl command.

```
--no-sort
```

Don't sort symbols alphabetically.

--output-dir DIR

Write RTF files to the directory DIR.

--single-file FILE

Create a single .rtf file called FILE.

--with-index

Create index entries.

--font NAME

Use font NAME as the font for the text body.

--literal-font NAME

Use font NAME as the font for literals.

--title-font NAME

Use font NAME as the font for the title.

--heading-font NAME

Use font NAME as the font for headings.

--font-size SIZE

Text body font size in points.

--literal-font-size SIZE

Literal font size in points.

--title-font-size SIZE

Title font size in points.

--heading-font-size SIZE

Heading font size in points.

3.8 Options for irserv

-h, --help -v, --version -e, --cpp NAME -d, --debug -DNAME -DNAME=DEF -UNAME -IDIR

These options are the same as for the idl command.

-i, --ior

Print the stringified IOR of the Interface Repository on standard output.

The arguments to irserv are zero or more IDL files. If no IDL files are specified on the command line, the Interface Repository server can be populated dynamically using the irfeed command.

3.9 Options for irfeed

```
-h, --help
-v, --version
-e, --cpp NAME
-d, --debug
-DNAME
-DNAME=DEF
-UNAME
-IDIR
```

These options are the same as for the idl command.

The arguments to irfeed are one or more IDL files.

3.10 Options for irdel

-h, --help -v, --version

These options are the same as for the idl command.

The arguments to irdel are one or more scoped names. A scoped name has the form "X::Y::Z". For example, an interface I defined in a module M can be identified by the scoped name "M::I".

3.11 Options for irgen

-h, --help -v, --version --no-skeletons

```
--no-type-codes

--locality-contrained

--no-virtual-inheritance

--tie

--c-suffix SUFFIX

--h-suffix SUFFIX

--header-dir DIR

--other-header-dir DIR

--output-dir DIR
```

These options are the same as for the idl command.

The argument to irgen is the pathname to use as the base name of the output filenames. For example, if the pathname you supply is output/file, then irgen will produce output/file.cpp, output/file.h, output/file_skel.cpp and output/file_skel.h.

Note that irgen will generate code for *all* of the type definitions contained in the Interface Repository server.

3.12 The IDL-to-C++ Translator and the Interface Repository

The ORBACUS IDL-to-C++ and IDL-to-Java translators internally use the Interface Repository for generating code. That is, these programs have their own private Interface Repository that is fed with the specified IDL files. All code is generated from that private Interface Repository.

It is also possible to generate C++ code from a global Interface Repository. First, the command irserv must be used to start the Interface Repository. Then the Interface Repository must be fed with the IDL code, using the command irfeed. Finally, the irgen command can be used to generate the C++ code. For example:

```
irserv --ior > IntRep.ref &
irfeed -ORBrepository `cat IntRep.ref` file.idl
irgen -ORBrepository `cat IntRep.ref` file
```

The IDL-to-C++ translator idl performs all these steps at once, in a single process with a private Interface Repository. Thus, you only have to run a single command:

idl file.idl

3.13 Include Statements

If you use the #include statement in your IDL code, the ORBACUS IDL-to-C++ translator idl will not create code for included IDL files. The translator will insert the appropriate #include statements in the generated header files instead. Please note that there are several restrictions on where to place the #include statements in your IDL files for this feature to work properly:

- #include may only appear at the beginning of your IDL files. All #include statements must be placed before the rest of your IDL code.¹
- Type definitions, such as interface or struct definitions, may not be split among several IDL files. In other words, no #include statement may appear within such definitions.

If you don't want these restrictions to be applied, you can use the translator option --all with idl. With this option the IDL-to-C++ translator treats code from included files as if the code appeared in your IDL file at the position where it is included. This means that the compiler will not place #include statements in the automatically-generated header files, regardless of whether the code comes directly from your IDL file or from files included by your IDL file.

Note that when generating code from an Interface Repository using irgen, the translator behaves identically to idl with the --all option. In other words, the irgen command will not place #include statements in the generated files, but rather generates code for all IDL definitions in the Interface Repository.

3.14 Documenting IDL Files

With the ORBACUS IDL-to-HTML and IDL-to-RTF translators, hidl and ridl, you can easily generate HTML and RTF files containing IDL interface descriptions. The translators will generate a nicely-formatted file for each IDL module and interface. Figure 3.1 shows an HTML example and Figure 3.2 an RTF example.

The formatting syntax supported by hidl and ridl is similar to that used by javadoc. The following keywords are recognized:

@author author

Denotes the author of the interface.

@exception exception-name description

Adds an exception description to the exception list of an operation.

@member member-name description

^{1.} Preprocessor statements like #define or #ifdef may be placed before your #include statements.


Figure 3.1: Documentation generated with the IDL-to-HTML translator

Adds a member description to the member list of a struct, union, enum or exception type.

@param parameter-name description

Adds a parameter description to the parameter list of an operation.

@return description

Adds descriptive text for the return value of an operation.



Figure 3.2: Documentation generated with the IDL-to-RTF translator

@see reference
Adds a "See also" note.
@since since-text
Comment related to the availability of new features.
@version version
The interface's version number.

Like javadoc, hidl and ridl use the first sentence in the documentation comment as the summary sentence. This sentence ends at the first period that is followed by a blank, tab or line terminator, or at the first @.

ridl understands most basic HTML tags and will produce an equivalent format in the generated RTF files. The following HTML tags are supported:

 <CODE> <HR> <P> <U>

3.15 Using javadoc

If not explicitly suppressed with the --no-comments option, the ORBACUS IDL-to-Java translator jidl adds comments starting with /** from the IDL file to the generated Java files, so that javadoc can be used to generate documentation (as long as the comments are in a format compatible with javadoc).

Here is an example showing how to include documentation in an IDL interface description file. Let's assume we have an interface I in a module M:

```
// IDL
module M
{
/**
 *
 *
   This is a comment related to interface I.
 *
   @author Uwe Seimet
 *
 *
   @version 1.0
 *
 **/
interface I
{
    /**
     * This comment describes exception E.
     *
     **/
   exception E { };
    /**
     *
```

```
* The description for operation S.
*
*
@param arg A dummy argument.
*
* @return A dummy string.
*
* @exception E Raised under certain circumstances.
*
**/
string S(in long arg)
raises(E);
};
};
```

When running jidl on this file the comments will automatically be added to the generated Java files M/I.java and M/IPackage/E.java. For I.java the generated code looks as follows:

```
// Java
package M;
11
// IDL:M/I:1.0
11
/**
 * This is a comment related to interface I.
 *
 *
   @author Uwe Seimet
 * @version 1.0
 *
 **/
public interface I extends org.omg.CORBA.Object
{
   11
   // IDL:M/I/S:1.0
   11
    /**
    * The description for operation S.
    *
    * @param arg A dummy argument.
     *
```

}

```
* @return A dummy string.
*
* @exception M.IPackage.E Raised under certain circumstances.
*
**/
public String
S(int arg)
throws M.IPackage.E;
```

Note that jidl automatically inserts the fully-qualified Java name for the exception E, in this case M.IPackage.E.

These are the contents of IPackage/E.java:

```
// Java
package M.IPackage;
11
// IDL:M/I/E:1.0
11
/**
 * This comment describes exception E.
 *
 **/
final public class E extends org.omg.CORBA.UserException
{
   public
   E()
   {
   }
}
```

Now you can use javadoc to extract the comments from the generated Java files and produce nicely-formatted HTML documentation.

For additional information please refer to the javadoc documentation.

The ORBacus Code Generators

CHAPTER 4

ORB and BOA Initialization

4.1 ORB Initialization

4.1.1 Initializing the C++ ORB

In C++ the ORB is initialized with <code>CORBA_ORB_init()</code>. For example:

```
// C++
int main(int argc, char* argv[], char*[])
{
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
    // ...
}
```

The CORBA_ORB_init() call interprets arguments starting with -ORB. All of these arguments, passed through the argc and argv parameters, are automatically removed from the argument list.

4.1.2 Initializing the Java ORB for Applications

A Java application can initialize the ORB in the following manner:

```
// Java
import org.omg.CORBA.*;
public static void main(String args[])
{
```

```
ORB orb = ORB.init(args, new java.util.Properties());
    // ...
}
```

The ORB.init() call interprets arguments starting with -ORB. Unlike the C++ version, these arguments are not removed (see "Filtering Command-line Options" on page 50 for more information).

4.1.3 Initializing the Java ORB in JDK 1.2

The ORB included in JDK 1.2 is not complete and lacks some functions (see the JDK 1.2 documentation for more information). To use the ORBacus ORB instead of the ORB included with JDK 1.2, you must start your application with the following properties:

```
java -Dorg.omg.CORBA.ORBClass=com.ooc.CORBA.ORB \
    -Dorg.omg.CORBA.ORBSingletonClass=com.ooc.CORBA.ORBSingleton \
    MyApp
```

An alternative is to set these properties in your program before initializing the ORB. For example:

4.1.4 Initializing the Java ORB for Applets

A different overloading of ORB.init() is provided for use by applets:

```
// Java
import org.omg.CORBA.*;
public void init()
{
     ORB orb = ORB.init(this, new java.util.Properties());
     // ...
```

}

See "Applets" on page 61 for more information on using ORBACUS in an applet.

4.2 BOA Initialization

4.2.1 Initializing the C++ BOA

In C++ the BOA is initialized with <code>CORBA_ORB::BOA_init()</code>. For example:

```
// C++
int main(int argc, char* argv[], char*[])
{
    CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
    CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
    // ...
}
```

 ${\tt BOA_init()}$ removes all arguments starting with –OA passed through the argc and argv parameters.

4.2.2 Initializing the Java BOA

In Java the BOA initialization looks like this:

```
// Java
import org.omg.CORBA.*;
public static void main(String args[])
{
    ORB orb = ORB.init(args, new java.util.Properties());
    BOA boa = orb.BOA_init(args, new java.util.Properties());
    // ...
}
```

4.2.3 Initializing the Java BOA in JDK 1.2

The org.omg.CORBA.ORB class included with JDK1.2 does not support the BOA_init() method. Therefore, in order to use BOA_init() with JDK1.2, you must cast the ORB to com.ooc.CORBA.ORB, as shown below.

```
// Java
import org.omg.CORBA.*;
public static void main(String args[])
{
```

```
Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass",
            "com.ooc.CORBA.ORBSingleton");
System.setProperties(props);
ORB orb = ORB.init(args, props);
BOA boa = ((com.ooc.CORBA.ORB)orb).BOA_init(args, props);
// ...
}
```

4.3 Configuring the ORB and BOA

ORBACUS applications can tailor the behavior of the ORB and BOA objects using a collection of properties¹. These properties can be defined in a number ways:

- using a configuration file
- using system properties (Java)
- using command-line options
- programmatically at run-time

4.3.1 Properties

The ORBACUS configuration properties are described in the sections below. Unless otherwise noted, every property can be used in both C++ and Java applications.

ORB Properties

ooc.orb.add_iiop_connector

Value: true, false

Determines whether the ORB should register an IIOP connector during initialization. The default value is true.

ooc.orb.conc_model

Value: blocking, reactive, threaded

^{1.} Note that these properties have nothing to do with the Property Service as described in "The Property Service" on page 168.

Selects the client-side concurrency model. The reactive concurrency model is not currently available in ORBACUS for Java. The default value is blocking for both C++ and Java applications. See Chapter 9 for more information on concurrency models.

ooc.orb.id

Value: id

Specifies the identifier of the ORB to be used by the application. The only valid identifier is OB_ORB.

ooc.orb.trace_level

Value: *level* >= 0

Defines the output level for diagnostic messages printed by ORBACUS. A level of 1 produces information about connection events. The default level is 0, which produces no output.

ooc.service.name

Value: ior

Adds an initial service to the ORB's internal list. This list is consulted when the application invokes the ORB operation resolve_initial_references. *name* is the key that is associated with a stringified IOR created using object_to_string. For example, the property ooc.service.NameService adds "NameService" to the list of initial services. See "Stringified Object References" on page 89 and "Initial Services" on page 94 for more information.

BOA Properties

ooc.boa.add_iiop_acceptor

Value: true, false

Determines whether the BOA should register an IIOP acceptor during initialization. The default value is true.

ooc.boa.conc_model

Value: blocking, reactive, threaded, thread_per_client, thread_per_request, thread_pool

Selects the server-side concurrency model. The reactive concurrency model is not available in ORBACUS for Java. The default value is reactive for C++ applications and threaded for Java applications. See Chapter 9 for more information on concurrency models. If this property is set to thread_pool, then the property ooc.boa.thread_pool determines how many threads are in the pool.

ooc.boa.disable_iiop_acceptor

Value: true, false

Determines whether the BOA should disable the IIOP acceptor after registering it. The default value is false.

ooc.boa.host

Value: hostname

Explicitly defines the hostname to be used in object references generated by the BOA. The default value is the canonical hostname of the machine. This property is especially useful if a host has more than one name. Note that this property is ignored if ooc.boa.numeric is true.

ooc.boa.id

Value: id

Specifies the identifier of the BOA to be used by the application. The only valid identifier is OB_BOA.

ooc.boa.numeric

Value: true, false

If true, the BOA will generate object references that contain an internet (IP) address in dotted decimal notation instead of the canonical hostname. The default value is false.

ooc.boa.port

Value: 0 <= *port* <= 65535

Specifies the port number on which the server should listen for new connections. If no port is specified, one will be selected automatically by the BOA. Use this property if you plan to publish an IOR (e.g., in a file, a naming service, etc.) and you want that IOR to remain valid across executions of your server. Without this property, your server is likely to use a

different port number each time the server is executed. See Chapter 6 for more information.

ooc.boa.thread_pool

Value: n > 0

Determines the number of threads to reserve for servicing incoming requests. The default value is 10. This property is only effective when the ooc.boa.conc_model property has the value thread_pool.

4.3.2 Command-line Options

There are equivalent command-line options for many of the ORBACUS properties. The options and their equivalent property settings are shown in Table 4.1. Refer to "Properties" on page 46 for a description of the properties.

Option	Property
-OAblocking	ooc.boa.conc_model=blocking
-OAdisable_iiop_acceptor	ooc.boa.disable_iiop_acceptor=true
-OAhost host	ooc.boa.host=host
-OAid id	ooc.boa.id= <i>id</i>
-OAnumeric	ooc.boa.numeric=true
-OAport port	ooc.boa.port=port
-OAreactive	ooc.boa.conc_model=reactive
-OAthreaded	ooc.boa.conc_model=threaded
-OAthread_per_client	ooc.boa.conc_model=thread_per_client
-OAthread_per_request	ooc.boa.conc_model=thread_per_request
-OAthread_pool n	<pre>ooc.boa.conc_model=thread_pool ooc.boa.thread_pool=n</pre>
-ORBblocking	ooc.orb.conc_model=blocking
-ORBid id	ooc.orb.id=id
-ORBnaming ior	ooc.service.NameService=ior

Table 4.1: Command-line Options

Option	Property
-ORBreactive	ooc.orb.conc_model=reactive
-ORBrepository ior	ooc.service.InterfaceRepository=ior
-ORBservice name ior	ooc.service.name=ior
-ORBthreaded	ooc.orb.conc_model=threaded
-ORBtrace_level level	ooc.orb.trace_level=level

Table 4.1: Command-line Options

A few additional command-line options are supported that do not have equivalent properties. These options are described in Table 4.2.

Option	Description
-ORBversion	Causes the ORB to print its version to standard output.
-ORBlicense	Causes the ORB to print its license to standard output.

Table 4.2: Additional Command-line Options

4.3.3 Filtering Command-line Options

In C++, all command-line options recognized by ORBACUS are automatically removed from the argv array after initializing the ORB and BOA.

In Java, command-line options are not automatically removed by ORBACUS. If you would like to have ORBACUS-specific options removed from the argument list, you will need to do so using two additional methods.

The example below demonstrates how to remove the ORB and BOA options in Java:

```
1 // Java
2 org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
3 org.omg.CORBA.BOA boa = orb.BOA_init(args, null);
4 String[] noOrbArgs = ((com.ooc.CORBA.ORB)orb).filter_options(args);
5 String[] noBoaArgs =
6 ((com.ooc.CORBA.BOA)boa).filter_options(noOrbArgs);
```

2,3 Initialize the ORB and BOA.

- 4 Remove the ORB options (i.e., options starting with -ORB) from args. The array noOrbArgs contains the filtered options.
- 5,6 Remove the BOA options (i.e., options starting with -OA). By passing noOrbArgs to this method, we ensure that both ORB and BOA options have been removed.

Note that the casts for the ORB and BOA are necessary because filter_options is an ORBACUS-specific operation, which only exists in the ORB and BOA classes residing in the com.ooc.CORBA package, and not in the org.omg.CORBA package.

4.3.4 Using a Configuration File

A convenient way to define a group of properties is to use a configuration file. A sample configuration file is shown below:

```
# Concurrency models
ooc.orb.conc_model=threaded
ooc.boa.conc_model=thread_pool
ooc.boa.thread_pool=5
```

```
# Initial services
ooc.service.NameService=iiop://myhost:5000/DefaultNamingContext
ooc.service.EventService=iiop://myhost:5001/DefaultEventChannel
ooc.service.TradingService=iiop://myhost:5002/TradingService
```

You can define the name of the configuration file¹ using a command-line option, an environment variable (C++), or a system property (Java):

• Command-line option:

-ORBconfig*filename*

- Environment variable:
 ORBACUS_CONFIG=filename
- Java system property:

ooc.config=*filename*

The file is read once when the ORB is initialized, and is not read again for the lifetime of the application process.

^{1.} ORBACUS for Java also accepts a URL specification as the filename.

4.3.5 Using the Windows NT Registry

Another convenient mechanism for use with C++ applications under Windows NT is to use the system registry¹. Properties can be stored in the registry under the following registry keys:

HKEY_LOCAL_MACHINE\Software\OOC\Properties
HKEY_CURRENT_USER\Software\OOC\Properties

Individual properties are defined as sub-keys of the base. For example, the property ooc.orb.trace_level=5 is stored in the registry as the following key containing a value named trace_level with a REG_SZ data member equal to "5":

Software\OOC\Properties\ooc\orb

RegUpdate

The ORBACUS distribution includes a utility called RegUpdate. The tool first removes all sub-keys defined under the specified registry key. Next, all values defined in an ORBA-CUS configuration file are transferred to the registry.

Synopsis

RegUpdate [-ORBconfig file] HKEY_LOCAL_MACHINE | HKEY_CURRENT_USER

Options for RegUpdate

-ORBconfig file

Read the properties from this configuration file. If the -ORBconfig option is not specified the properties are read from the file defined by the ORBACUS_CONFIG environment variable.

Example:

RegUpdate -ORBconfig ob.conf HKEY_LOCAL_MACHINE

This command will read the properties defined in the file ob.conf and write the values under the following registry key:

```
HKEY_LOCAL_MACHINE\Software\OOC\Properties
```

^{1.} Use caution when defining ORBACUS properties in the registry, as they become global properties that will be used in every ORBACUS for C++ application. For example, subtle errors can occur if the ooc.boa.port property is defined on a global basis.

4.3.6 Defining Properties in Java

Java applications can use the standard Java mechanism for defining system properties, because ORBACUS will also search the system properties during ORB and BOA initialization.

For example:

```
1 // Java
2 java.util.Properties props = System.getProperties();
3 props.put("ooc.orb.conc_model", "threaded");
4 props.put("ooc.boa.port", "10000");
5 org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
```

- 2 Obtain the system properties.
- 3,4 Define ORBACUS properties.
 - 5 Initialize the ORB.

Java virtual machines typically allow you to define system properties on the command line. For example, using Sun's JVM you can do the following:

java -Dooc.boa.port=5000 MyServer

You can also use the java.util.Properties object that is passed to the org.omg.CORBA.ORB.init() and org.omg.CORBA.ORB.BOA_init() methods to provide ORBACUS property definitions:

```
1 // Java
2 java.util.Properties props = new java.util.Properties();
3 props.put("ooc.boa.numeric", "true");
4 org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
5 org.omg.CORBA.BOA boa = orb.BOA_init(args, props);
```

- 2 Create a java.util.Properties object to hold our properties.
- 3 Define ORBACUS properties.
- 4,5 Initialize the ORB and BOA using the java.util.Properties object.

4.3.7 Precedence of Properties

Given that properties can be defined in several ways, it's important to establish the order of precedence used by ORBACUS when collecting and processing the property definitions.

The order of precedence is listed below, from highest to lowest. Properties defined at a higher precedence override the same properties defined at a lower precedence.

- 1. Command-line options
- 2. System properties (Java only)
- 3. User-supplied properties (Java only)
- 4. Configuration file
- 5. HKEY_CURRENT_USER\SoftwareOOCProperties (Windows NT/C++ only)
- $6. \ {\tt HKEY_LOCAL_MACHINE\Software\OOC\Properties} \ (Windows \ NT/C++ \ only)$

For example, a property defined using a command-line option overrides the same property defined in a configuration file.

4.3.8 Advanced Property Usage

If you need explicit control of the properties from within your application, you may also elect to use ORBACUS-specific classes to create and retrieve property definitions.¹

In Java, this class is com.ooc.CORBA.Properties, and in C++ the class is OBProperties. These classes are used internally by ORBACUS, but you can also use them in your applications.

```
// Java
package com.ooc.CORBA;

class Properties
{
    public static Properties init(String[] args);
    public static Property(String key);
    public String getProperty(String key);
    public void setProperty(String key, String value);
    public String[] getKeys(String prefix);
    public String[] getKeys();
}
// C++
class OBProperties
{
```

^{1.} The Properties class is probably more useful for C++ applications, since Java applications can use system properties to achieve the same effect.

```
public:
    static OBProperties* init(int& argc, char** argv);
    static OBProperties* instance();
    typedef OBStrSeq KeySeq;
    void setProperty(const char* key, const char* value);
    const char* getProperty(const char* key);
    KeySeq getKeys(const char* prefix);
    KeySeq getKeys();
};
```

In the discussion below, these classes are referred to generically as the Properties class.

To use a Properties class correctly, you must be aware of the initialization steps taken by the ORB and BOA objects. The Properties class is a *Singleton* class, in that only one instance of the class is allowed. The ORB initializes the Properties object during its own initialization. However, if you need to use the Properties class before the ORB has been initialized (e.g., if you need to define an ORB property), then you will need to initialize the Properties class manually.

Defining ORB Properties

The code below demonstrates a situation where an application needs to define a property prior to initializing the ORB. First, we'll show the example in C++:

```
1 // C++
2 #include <OB/CORBA.h>
3 #include <OB/Properties.h>
4
5 // ...
6
7 OBProperties* properties = OBProperties::init(argc, argv);
8 properties -> setProperty("ooc.orb.conc_model", "reactive");
9 CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
```

- 2,3 Include the necessary header files.
 - 7 The call to OBProperties::init() creates the OBProperties object and initializes it with the contents of a configuration file (if necessary).
 - 8 Set the ORB concurrency model using a property.
 - 9 Initialize the ORB.

The code looks very similar in Java:

```
1 // Java
```

```
2 com.ooc.CORBA.Properties properties =
```

```
3 com.ooc.CORBA.Properties.init(args);
```

```
4 properties.setProperty("ooc.orb.conc_model", "threaded");
```

- 5 org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
- 2,3 Creates the com.ooc.CORBA.Properties object and initializes it with the contents of a configuration file (if necessary).
 - 4 Set the ORB concurrency model using a property.
 - 5 Initialize the ORB.

Defining BOA Properties

In a situation where you don't need access to the Properties object until after the ORB has been initialized, you can simply do the following:

```
1 // C++
2 #include <OB/CORBA.h>
3 #include <OB/Properties.h>
4
5 // ...
6
7 CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
8 OBProperties* properties = OBProperties::instance();
9 properties -> setProperty("ooc.boa.conc_model", "reactive");
10 CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
```

2,3 Include the necessary header files.

- 7 Initialize the ORB. The ORB will initialize the Properties object.
- 8 Obtain the OBProperties instance.
- 9,10 Set the BOA concurrency model using a property and initialize the BOA.

Note that in this example we are defining a BOA property prior to initializing the BOA. Also note that the Properties object has already been initialized by the ORB, so the application simply needs to obtain a pointer to the object using the instance method.

Here's the same example in Java:

```
1 // Java
2 org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
3 com.ooc.CORBA.Properties properties =
4     com.ooc.CORBA.Properties.instance();
5 properties.setProperty("ooc.boa.conc_model", "threaded");
6 org.omg.CORBA.BOA boa = orb.BOA_init(args, null);
```

- 2 Initialize the ORB. The ORB will initialize the Properties object.
- 3,4 Obtain the Properties instance.
- 5,6 Set the BOA concurrency model using a property and initialize the BOA.

Application-specific Properties

Another situation where the Properties class can be useful is if you'd like to obtain application-specific properties from the ORBACUS configuration file. Suppose your configuration file looks as follows:

```
# ORBacus configuration file
ooc.orb.conc_model=threaded
# Application-specific settings
acme.widget_count=20
```

The following C++ example demonstrates how to access your application-specific properties:

```
1 // C++
2 #include <OB/CORBA.h>
3 #include <OB/Properties.h>
4
5 // ...
6
7 CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
8 OBProperties* properties = OBProperties::instance();
9 const char* value = properties -> getProperty("acme.widget_count");
```

- 2,3 Include the necessary header files.
 - 7 The ORB must be initialized so that the configuration file is processed.
- 8,9 Obtain the OBProperties instance and then retrieve the value of the property.

And in Java:

1 // Java

```
2 org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
```

```
3 com.ooc.CORBA.Properties properties =
```

4 com.ooc.CORBA.Properties.instance();

```
5 String value = properties.getProperty("acme.widget_count");
```

- 2 The ORB must be initialized so that the configuration file is processed.
- 3-5 Obtain the Properties instance and then retrieve the value of the property.

Finally, it is important to remember the precedence rules for ORBACUS properties. Specifically, command-line options will *always* override any existing property definitions, including those you set within your application.

4.4 Server Event Loop

A server's event loop is entered by calling BOA:: impl_is_ready. For example, in Java:¹

```
// Java
org.omg.CORBA.BOA boa = ... // Get the BOA somehow
boa.impl_is_ready(null);
```

And in C++:

```
// C++
CORBA_BOA_var boa = ... // Get the BOA somehow
boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
```

impl_is_ready only returns, if:

- The blocking concurrency model (see Chapter 9) has been chosen for the server, and the client disconnects.
- deactivate_impl is called (see "Deactivating the Server" on page 59).

4.4.1 Mixed Client/Server Applications

In case the reactive or one of the threaded concurrency models has been chosen (see Chapter 9) it is possible to service requests without calling impl_is_ready. This is especially useful in mixed client/server applications. For example, consider a mixed client/server program that wants to invoke operations on a server in the program's main function, but still wants to be able to receive "callbacks" from this server. In order to receive these

The argument to impl_is_ready is currently unused by ORBACUS, therefore the "dummy" argument null (Java) or CORBA_ImplementationDef::_nil() (C++) is used.

"callback" requests, usually impl_is_ready would have to be called in main. However, this is not possible, since impl_is_ready blocks, which makes it impossible for the mixed client/server program to invoke operations on the server after the call to impl_is_ready.

To solve this problem, ORBACUS provides the operation init_servers. Here's how init_servers is called in Java:

```
// Java
org.omg.CORBA.BOA boa = ... // Get the BOA somehow
((com.ooc.CORBA.BOA)boa).init_servers();
```

This is similar to impl_is_ready, except that init_servers does not block. Note that the cast for the BOA is necessary because init_servers is an ORBACUS-specific operation, which only exists in com.ooc.CORBA.BOA, and not in org.omg.CORBA.BOA.

The C++ version look similar:

```
// C++
CORBA_BOA_var boa = ... // Get the BOA somehow
boa -> init_servers();
```

4.4.2 Deactivating the Server

A server can be deactivated with a call to BOA::deactivate_impl. This causes BOA::impl_is_ready to return. For example, consider a server which can be shut down by a client by calling a deactivate operation on one of the server's objects. First the IDL code:

```
// IDL
interface ShutdownObject
{
    void deactivate();
};
```

On the server side, ShutdownObject can be implemented like this:

```
1 // C++
2
3 class ShutdownObject_impl : public virtual ShutdownObject_skel
4 {
5      CORBA_BOA_var boa_;
6
7 public:
8
9      ShutdownObject_impl(CORBA_BOA_ptr boa)
```

```
10
          : boa_(CORBA_BOA::_duplicate(boa))
11
       {
       }
12
13
14
      virtual void deactivate()
15
       ł
16
          boa_ -> deactivate_impl(CORBA_ImplementationDef::_nil());
17
       }
18
   };
```

3 A servant class for ShutdownObject is defined. For more information on how to implement servant classes, see Chapter 5.

- 5 A BOA is needed to call deactivate_impl.
- 9-12 The constructor initializes the BOA member.
- 14-17 deactivate calls deactivate_impl on the BOA.

Here's the main code for this example:

```
1 // C++
 2
 3 int main(int argc, char* argv[], char*[])
 4 {
 5
      CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
      CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
 6
 7
 8
      ShutdownObject_var shutdownObj = new ShutdownObject_impl(boa);
 9
      boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
10
11
12
      return 0;
13
   ł
```

5,6 ORB and BOA initialization.

- 8 The shutdown object is created.
- 10 The impl_is_ready main event loop is entered. This call only returns if deactivate is called.
- 12 The server was deactivated, so main can now return.

The client can use the deactivate call as shown below:

```
// C++
 1
 2
   int main(int argc, char* argv[], char*[])
 3
 4
   {
 5
       CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
 6
 7
       ShutdownObject_var shutdownObj = ... // Get a reference somehow
 8
 9
       try
10
       {
11
          shutdownObj -> deactivate();
12
13
       catch(const CORBA_COMM_FAILURE& ex)
14
       ł
       }
15
16
17
       return 0;
18
```

- 5 Initialize the ORB.
- 7 Get a reference to the server's shutdown object somehow, for example by reading in a "stringified" object reference (see "Stringified Object References" on page 89).
- *9-15* Call deactivate on the shutdown object. COMM_FAILURE exceptions must be ignored, since the server may shut down immediately, without any chance for a proper reply message to be delivered back to the client. Therefore, the client will usually get a COMM_FAILURE exception at this point.

4.5 Applets

4.5.1 Adding ORBacus Applets to Web Pages

Like any other applet, ORBACUS applets can be added to HTML pages with the APPLET tag:

```
<APPLET CODE="Client.class" ARCHIVE="OB.jar" WIDTH=500 HEIGHT=300>
</APPLET>
```

It is necessary to tell the Web browser where to find the ORBACUS Java classes. This is best done with the ARCHIVE attribute as shown above. An alternative is to use the CODEBASE attribute and to extract the OB.jar archive in the directory defined by CODEBASE. For more information, please consult your Java Development Kit documentation.

4.5.2 Defining ORB and BOA Options for an Applet

The PARAM tag is used in HTML to define parameters for an applet. When initialized by an applet, the ORB looks for the parameters ORBparams and BOAparams, whose values should be command-line options separated by spaces.

For example, the HTML code below uses the -ORBconfig option to specify the URL of the ORB configuration file:

Your applet can also define ORBACUS configuration properties using Java system properties, or using the java.util.Properties object passed to

org.omg.CORBA.ORB.init(). See "Initializing the Java BOA in JDK 1.2" on page 45 for more information.

4.5.3 Defining the ORB Class Parameters

Some Web browsers¹ have a built-in ORB. In order to use ORBACUS instead of this builtin ORB, you must set the following applet parameters:

4.5.4 Security Issues

Web browsers generally place several security restrictions on applets that you need to be aware of when developing an applet using ORBACUS:

- Applets can only communicate with the host from which the applet was downloaded.
- Applets cannot accept connections from any host.

The first limitation forces you to run any CORBA server applications that your applet communicates with on your Web server host.² The second limitation prevents your applet from acting as a CORBA server, which is often necessary when a client wishes to receive callbacks from a server.

^{1.} For example, Netscape v4 has a built-in ORB.

These limitations are the most common causes of security exceptions in an applet. You must ensure that any object references used by your applet refer to objects on the Web server host. Furthermore, you must not attempt to enable CORBA server functionality in your applet by initializing the BOA.

^{2.} Netscape v4 also does not normally allow CORBA applets to be loaded from a local (i.e., filesystem) HTML file, causing a SecurityException when the applet attempts to connect to the CORBA server. To work around this problem, CORBA applets must be downloaded from a Web server.

ORB and BOA Initialization

CHAPTER 5 CORBA Objects

5.1 Overview

A *CORBA object* is an object with an interface defined in CORBA IDL. CORBA objects have different representations in clients and servers.

- A *server* implements a CORBA object in a concrete programming language, for example in C++ or Java. This is done by writing an *implementation class* for the CORBA object and by instantiating this class. The resulting object is called a *servant*.
- A *client* that wants to make use of a servant implemented by a server creates an object that delegates all operation calls to the servant via the ORB. Such an object is called a *proxy*.

When a client invokes a method on the local proxy object, the ORB packs the input parameters and sends them to the server, which in turn unpacks these parameters and invokes the actual method on the servant. Output parameters and return values, if any, follow the reverse path back to the client. From the client's perspective, the proxy acts just like the servant since it hides all the communication details within itself.

A servant must somehow be connected to the ORB, so that the ORB can invoke a method on the servant when a request is received from a client. This connection is handled by the object adapter, as shown in Figure 5.1.

ORBACUS comes with an object adapter called the "Basic Object Adapter" (BOA). Unfortunately, the specification for the BOA [2] is quite incomplete, leaving a lot of freedom to



Figure 5.1: Servants, Proxies and the Object Adapter

ORB implementors.¹ Therefore all BOAs are in fact more or less vendor specific. It is therefore necessary to have a chapter explaining how servants are implemented in ORBA-CUS and how they are connected to the ORBACUS BOA implementation.

5.2 Implementing Servants

In this chapter, we will implement servant classes (or "implementation classes") for the IDL interfaces defined below:

```
1 // IDL
 2
 3 interface A
 4
   {
 5
      void op_a();
 6
   };
 7
 8
  interface B
 9
   {
10
      void op_b();
11 };
12
13 interface I : A, B
14 {
```

^{1.} Because of these problems, the OMG is currently defining a new object adapter, the so-called "Portable Object Adapter" (POA). Future versions of ORBACUS will implement the POA.

15 void op_i();
16 };

- 3-6 An interface A is defined with the operation op_a.
- 8-11 An interface B is defined with the operation op_b.
- 13-16 Interface I is defined, which is derived from A and B. It also defines a new operation op_i.

5.2.1 Implementing Servants using Inheritance

ORBACUS for C++ and ORBACUS for Java both support the use of inheritance for interface implementation. To implement an interface using inheritance, you write a servant class that inherits from a skeleton class generated by the IDL translator. By convention, the name of the servant class should be the name of the interface with the suffix _impl, e.g., for an interface I, the implementation class is named I_impl.¹

```
Inheritance using C++
```

In C++, I_impl must inherit from the skeleton class I_skel that was generated by the IDL-to-C++ translator. If I inherits from other interfaces, for example from the interfaces A and B, then I_impl must also inherit from the corresponding implementation classes A_impl and B_impl.

```
1
  // C++
 2
 3 class A_impl : virtual public A_skel
   {
 4
 5
  public:
 6
      virtual void op_a();
 7
   };
 8
 9
   class B_impl : virtual public B_skel
10 {
11 public:
12
      virtual void op_b();
13 };
14
  class I_impl : virtual public I_skel,
15
                   virtual public A_impl,
16
```

^{1.} These naming rules are not mandatory, they are just a recommendation.

```
17 virtual public B_impl
18 {
19 public:
20 virtual void op_i();
21 };
```

- 3-7 The servant class A_impl is defined, inheriting from the skeleton class A_skel. If op_a had any parameters, these parameters would be mapped according to the standard IDL-to-C++ mapping rules [2].
- 9-14 This is the servant class for B_impl.
- 15-21 The servant class for I_impl is not only derived from I_skel, but also from the servant classes A_impl and B_impl.

Note that virtual public inheritance must be used. The only situation in which the keyword virtual is not necessary is for an interface I which does not inherit from any other interface and from which no other interface inherits. This means that the implementation class I_impl only inherits from the skeleton class I_skel and no implementation class inherits from I_impl.

It is not strictly necessary to have an implementation class for every interface. For example, it is sufficient to only have the class I_impl as long as I_impl implements all interface operations, including the operations of the base interfaces:

```
1 // C++
2
3 class I_impl : virtual public I_skel
4 {
5 public:
6 virtual void op_a();
7 virtual void op_b();
8 virtual void op_i();
9 };
```

- 3 Now I_impl is only derived from I_skel, but not from the other servant classes.
- 6-8 I_impl must implement all operations from the interface I as well as the operations of all interfaces from which I is derived.

Inheritance using Java

Several files are generated by the ORBACUS IDL-to-Java translator for an interface I, including:

- I. java, which defines a Java interface I containing public methods for the operations and attributes of I, and
- _IImplBase.java, which is an abstract skeleton class that serves as the base class for servant classes.

In contrast to C++, Java's lack of multiple inheritance currently makes it impossible for a servant class to inherit operation implementations from other servant classes. For our interface I it is therefore necessary to implement all operations in a single servant class I_impl , regardless of whether those operations are defined in I or in an interface from which I is derived.

```
1
   // Java
 2
 3 public class I_impl extends _IImplBase
   {
 4
 5
        public void op_a()
 6
        {
 7
        }
 8
        public void op_b()
 9
10
        {
11
        }
12
        public void op_i()
13
14
        {
15
        }
16
```

3-16 The servant class I_impl is defined, which implements op_i, as well as the inherited operations op_a and op_b.

5.2.2 Implementing Servants using Delegation

Sometimes it is not desirable to use an inheritance-based approach for implementing an interface. This is especially true if the use of inheritance would result in an implementation being incompatible with existing legacy code. Therefore, another alternative is available for implementing servants which does not use inheritance. A special class, known as a *tie class*, can be used to delegate the implementation of an interface to another class.

CORBA Objects

Delegation using C++

The ORBACUS IDL-to-C++ translator can automatically generate a tie class for an interface in the form of a template class. A tie template class is derived from the corresponding skeleton class and has the same name as the skeleton, with the suffix _tie appended.

For the interface I from the C++ example above, the template I_skel_tie is generated and must be instantiated with a class that implements all operations of I. By convention, the name of this class should be the name of the interface with _impl_tie appended.¹

In contrast to the inheritance-based approach, it is not necessary that the class implementing I's operations, i.e., I_impl_tie, be derived from any skeleton class. Instead, an instance of I_skel_tie delegates all operation calls to I_impl_tie, as shown in Figure 5.2.



Figure 5.2: Class Hierarchy for Inheritance and Delegation Implementation in C++

^{1.} Again, you are free to choose whatever name you like. This is just a recommendation.

Here is our definition of I_impl_tie:

```
1 // C++
2
3 class I_impl_tie
4 {
5 public:
6 virtual void op_a();
7 virtual void op_b();
8 virtual void op_i();
9 };
```

3 I_impl_tie is defined, which is not derived from any other class.

6-8 I_impl_tie must implement all of I's operations, including inherited operations.

A servant class for I can then be defined using the I_skel_tie template:

```
1 // C++
2 typedef I_skel_tie< I_impl_tie > I_impl;
```

2 The servant class I_impl is defined as a template instance of I_skel_tie, parameterized with I_impl_tie.

Delegation using Java

The ORBACUS IDL-to-Java translator generates two additional files to support delegationbased servant implementation for an interface I:

- IOperations.java, an interface that defines public methods for all attributes and operations of I, and
- __IImplBase_tie.java, the tie class that inherits from __IImplBase and delegates all requests to an instance of IOperations.

To implement our servant class using delegation, we need to write a class that implements the IOperations interface:

```
1 // Java
2
3 public class I_impl_tie implements IOperations
4 {
5     public void op_a()
6     {
7     }
8
```

```
9    public void op_b()
10    {
11    }
12    
13    public void op_i()
14    {
15    }
16 }
```

3 The servant class I_impl_tie is defined to implement the IOperations interface.

5-15 I_impl_tie must implement all of I's operations, including inherited operations.

Figure 5.3 illustrates the relationship between the classes generated by the IDL-to-Java translator and the servant implementation classes.



Figure 5.3: Class Hierarchy for Inheritance and Delegation Implementation in Java
5.3 Creating Servants

Servants are created the same way in both C++ and Java: once your servant class is written, you simply instantiate a servant with new.

5.3.1 Creating Servants using C++

Here is how to create servants using C++:

```
1 // C++
2 I_var impl = new I_impl;
3 I_var anotherImpl = new I_impl;
```

2,3 Two servants, impl and another Impl, are created with new.

In case the servant class was written using the delegation approach, an object of the class implementing I's operations must be passed to the servant's constructor:

```
1 // C++
2 I_impl_tie* impl = new I_impl_tie;
3 I_var tie = new I_skel_tie< I_impl_tie >(impl, CORBA_TRUE);
```

- 2 A new I_impl_tie is created with new.
- 3 An instance of I_skel_tie parameterized with I_impl_tie is created, taking impl as a parameter. All operation calls to tie will then be delegated to impl.

In this example, the lifetime of impl is coupled to the lifetime of the servant tie. That is, when tie is destroyed, delete impl is called. In case you don't want the lifetime of impl to be coupled to the lifetime of tie, for example because you want to create impl on the stack and not on the heap (making it illegal to call delete on impl), use the following code:

```
1 // C++
2 I_impl_tie impl;
3 I_var tie = new I_skel_tie< I_impl_tie >(&impl, CORBA_FALSE);
```

- 2 A new I_impl_tie is created, this time on the stack, not on the heap.
- 3 An instance of I_skel_tie is created. The CORBA_FALSE parameter tells tie not to call delete on impl.

5.3.2 Creating Servants using Java

This example demonstrates how to create servants using Java:

```
1 // Java
2 I impl = new I_impl();
3 I anotherImpl = new I_impl();
```

2,3 Two servants, impl and another Impl, are created with new.

In case the servant class was written using the delegation approach, an object implementing the IOperations interface must be passed to the servant's constructor:

```
1 // Java
2 I_impl_tie impl = new I_impl_tie();
3 _IImplBase_tie tie = new _IImplBase_tie(impl);
```

- 2 A new I_impl_tie is created.
- 3 An instance of _IImplBase_tie is created, taking impl as a parameter. All operation calls to tie will then be delegated to impl.

Every tie class generated by the IDL-to-Java translator includes methods for accessing and changing the implementation object:

```
1 // Java
 2
 3 public class _IImplBase_tie extends _IImplBase
 4 {
 5
       . . .
 6
       public IOperations _delegate() { ... }
 7
 8
 9
       public void _delegate(IOperations delegate) { ... }
10
11
       . . .
12
```

3 The tie class for interface I is defined.

7 This method returns the current delegate (i.e., implementation) object.

9 This method changes the delegate object.

5.4 Connecting Servants

Servants must be connected to the object adapter in order to receive requests from clients. Usually this is done automatically whenever an object reference to a servant is passed to a client as a parameter or return value. Servants are also connected implicitly when used in calls to operations like object_to_string. However, it is also possible to connect a servant explicitly.

5.4.1 Connecting Servants using C++

The following code shows how to explicitly connect a servant:

```
1 // C++
2 CORBA_ORB_var orb = ... // Get a reference to the ORB somehow
3 I_var impl = new I_impl;
4 orb -> connect(impl);
```

- 2 To connect a servant, we need the ORB.
- 3 A new servant impl is created.
- 4 The new servant is connected to the object adapter.

A servant can also be disconnected from the object adapter. This is done with the disconnect call:

```
1 // C++
2 orb -> disconnect(impl);
```

2 The servant impl is disconnected from the object adapter. From now on, requests from clients to this servant will cause an OBJECT_NOT_EXIST exception to be raised.

5.4.2 Connecting Servants using Java

This is how Java servants are explicitly connected to the object adapter:

```
1 // Java
2 org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
3 I impl = new I_impl();
4 orb.connect(impl);
```

- 2 To connect a servant, we need the ORB.
- 3 A new servant impl is created.

4 The new servant is connected to the object adapter.

A servant can also be disconnected from the object adapter. This is done with the disconnect call:

1 // Java

2 orb.disconnect(impl);

2 The servant impl is disconnected from the object adapter. From now on, requests from clients to this servant will cause an OBJECT_NOT_EXIST exception to be raised.

5.4.3 Named Servants

ORBACUS for C++ and ORBACUS for Java support the notion of named servants, in which a name is assigned to a servant when it is connected to the object adapter, allowing a client to identify a servant by its name. The ORB operation get_inet_object is used on the client side to resolve a named servant within a specific server (see "Connecting to Named Objects" on page 92).

For named servants, a parameter for the servant's name must be provided to connect. For example, in C++:

```
// C++
CORBA_ORB_var orb = ... // Get a reference to the ORB somehow
orb -> connect(impl, "MyName");
```

And in Java;

```
// Java
org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
((com.ooc.CORBA.ORB)orb).connect(impl, "MyName");
```

In both examples, the servant impl is connected to the object adapter, using the name "MyName".

The cast to com.ooc.CORBA.ORB is necessary because the Java overloading of connect in support of named servants is an ORBACUS-specific extension and is not available in org.omg.CORBA.ORB.

The name assigned to a servant must be unique among all servants in a server. In case the name is already in use, the INV_IDENT exception is raised.

5.5 Factory Objects

It is quite common to use the Factory [10] design pattern in CORBA applications. In short, a factory object provides access to one or more additional objects. In CORBA applications, a factory object can represent a focal point for clients. In other words, the object reference of the factory object can be published in a well-known location, and clients know that they only need to obtain this object reference in order to gain access to other objects in the system, thereby minimizing the number of object references that need to be published.

The Factory pattern can be applied in a wide variety of situations, including the following:

- **Security** A client is required to provide security information before the factory object will allow the client to have access to another object.
- **Load-balancing** The factory object manages a pool of objects, often representing some limited resource, and assigns them to clients based on some utilization algorithm.
- **Polymorphism** A factory object enables the use of polymorphism by returning object references to different implementations depending on the criteria specified by a client.

These are only a few examples of the potential applications of the Factory pattern. The examples listed above can also be used in any combination, depending on the requirements of the system being designed.

A simple application of the Factory pattern, in which a new object is created for each client, is illustrated below. The implementation uses the following interface definitions:

```
// IDL
 1
 2 interface Product
 3
  {
      void destroy();
 4
   };
 5
 6
 7 interface Factory
 8
   {
 9
       Product createProduct();
10
   };
```

- 2-5 The Product interface is defined. The destroy operation allows a client to destroy the object when it is no longer needed.
- 7-10 The Factory interface is defined. The createProduct operation returns the object reference of a new Product.

5.5.1 Factory Objects using C++

First, we'll implement the Product interface:

```
1 // C++
 2 class Product_impl : public virtual Product_skel
 3 {
 4
       CORBA_ORB_var orb_;
 5
 6 public:
 7
      void Product_impl(CORBA_ORB_ptr orb)
          : orb_(CORBA_ORB::_duplicate(orb))
 8
 9
       {
       }
10
11
      virtual void destroy()
12
13
       {
          orb_ -> disconnect(this);
14
15
       }
16 };
```

2 Servant class Product_impl is defined as an implementation of the Product interface.

7-8 The constructor takes an ORB parameter and saves it for later use.

14 The destroy operation disconnects the object from the object adapter. A side-effect of disconnecting the object is that the object adapter no longer holds a reference to the servant. If there are no other references to this servant in the server, then the servant will be destroyed. See "Releasing Proxies and Servants" on page 102 for more information.

Next, we'll implement the factory:

```
1 // C++
 2 class Factory_impl : public virtual Factory_skel
 3 {
 4
      CORBA_ORB_var orb_;
 5
 6 public:
 7
      void Factory_impl(CORBA_ORB_ptr orb)
          : orb_(CORBA_ORB::_duplicate(orb))
 8
 9
       {
       }
10
11
      virtual Product_ptr createProduct()
12
      {
13
```

```
14 Product_ptr result = new Product_impl(orb_);
15 orb_ -> connect(result);
16 return result;
17 }
18 };
```

- 2 Servant class Factory_impl is defined as an implementation of the Factory interface.
- 7-8 The constructor takes an ORB parameter and saves it for later use.
- 14-16 The createProduct operation instantiates a new Product servant, connects it to the object adapter, and returns an object reference to the client. Use of the connect operation is optional; an object will be connected automatically if it has not already been connected at the time a reference to the object is transmitted to a client.

Users familiar with other CORBA implementations may think there is an error in the createProduct method because _duplicate is not being used. However, the code is correct. See Chapter 7 for a complete discussion of reference counts.

5.5.2 Factory Objects using Java

Here is our Java implementation of the Product interface:

```
// Java
 7
 2 public class Product_impl extends _ProductImplBase
   {
 3
 4
       org.omg.CORBA.ORB orb_;
 5
       public Product_impl(org.omg.CORBA.ORB orb)
 6
 7
       ł
 8
          orb_ = orb;
 9
       }
10
11
       public void destroy()
12
       {
13
          orb_.disconnect(this);
       }
14
15
```

2 Servant class Product_impl is defined as an implementation of the Product interface.

13 The destroy operation disconnects the object from the object adapter. As long as no other references to the servant are held in the server, the object will be eligible for garbage collection. See "Reference Counting in Java" on page 99 for more information on garbage collection of servant objects.

Here's our implementation of the factory:

```
// Java
 1
   public class Factory_impl extends _FactoryImplBase
 2
 3
   {
       org.omg.CORBA.ORB orb_;
 4
 5
       public Factory_impl(org.omg.CORBA.ORB orb)
 6
 7
       {
 8
          orb_ = orb;
 9
       }
10
11
       public Product createProduct()
12
       {
13
          Product result = new Product impl(orb );
14
          orb_.connect(result);
15
          return result;
       }
16
17
   }
```

- 2 Servant class Factory_impl is defined as an implementation of the Factory interface.
- 13-16 The createProduct operation instantiates a new Product servant, connects it to the object adapter, and returns an object reference to the client. Like in the C++ version, the explicit call to connect is optional.

5.5.3 Caveats

In these simple examples, the factory objects do not maintain any references to the Product servants they create; it is the responsibility of the client to ensure that it destroys a Product object when it is no longer needed. This design has a significant potential for resource leaks in the server, as it is quite possible that a client will not destroy its Product objects, either because the programmer who wrote the client forgot to invoke destroy, or because the client program crashed before it had a chance to clean up. You should keep these issues in mind when designing your own factory objects.¹

^{1.} Two possible strategies for handling this issue include: time-outs, in which a servant that has not been used for some length of time is automatically released; and expiration, in which an object reference is only valid for a certain length of time, after which a client must obtain a new reference. The implementation of these solutions is beyond the scope of this manual.

5.6 Getting a Servant from a Reference

In some situations it may be necessary to obtain the servant implementation object of an object reference (typically because you need to invoke a method on the servant implementation object that is not available via its IDL interface).

In ORBACUS, servant classes are derived from skeleton classes, which are derived from proxy classes (so-called "stub" classes). Therefore, you can simply cast an object reference to its servant class.

5.6.1 Getting a Servant using C++

In C++, dynamic_cast<> can be used to obtain a pointer to the servant, as shown below:

```
1 // C++
 2
 3 class I_impl : virtual public I_skel
 4 {
 5 };
 6
 7 void foo(I_ptr ref)
   {
 8
      I_impl* p = dynamic_cast<I_impl*>(ref);
 9
10
      if(p)
11
12
       {
13
          // The implementation for ref is in the same process
14
       }
15
       else
16
       {
          // The implementation for ref is not in the same process
17
       }
18
19
```

3 A servant class for an interface I is defined.

7 The operation foo takes an object reference ref to an object I as a parameter.

9 dynamic_cast<> is used on ref to get a pointer to an I_impl.

11-18 The call to dynamic_cast<> returns a pointer to the servant if the object referred to by ref was local, or a null pointer otherwise.

In case your compiler does not support RTTI¹, you can use the OB_MAKE_NARROW_IMPL macros from the ORBACUS header file Narrow_impl.h to obtain a pointer to a servant class:

```
1 // C++
 2
 3 #include <OB/Narrow_impl.h>
 4
 5 class I_impl : virtual public I_skel
 6
   {
 7
      OB_MAKE_NARROW_IMPL(I_impl)
 8 };
 9
   OB_MAKE_NARROW_IMPL_1(I_impl, I_skel)
10
11 void foo(I_ptr ref)
12 {
13
      I_impl* p = I_impl::_narrow_impl(ref);
14
15
      if(p)
16
       {
          // The implementation for ref is local
17
       }
18
19
      else
20
       {
          // The implementation for ref is not local
21
       }
22
23
```

3 The file <OB/Narrow_impl.h> must be included for the definitions of the OB_MAKE_NARROW_IMPL macros.

- 5-9 A servant class for I is defined with OB_MAKE_NARROW_IMPL as shown.
- 13 The only other difference is that now I_impl::_narrow_impl must be used instead of dynamic_cast<>.

The macro OB_MAKE_NARROW_IMPL_1 can only be used if the servant class has exactly one super class (the skeleton class). If the servant class has two or more super classes, use the macro OB_MAKE_NARROW_IMPL_n, where n is the number of super classes. For example:

^{1.} RunTime Type Identification.

```
// C+++
 1
 2
 3 class C_impl : virtual public C_skel,
                   virtual public A_impl,
 4
 5
                   virtual public B_impl
   {
 6
 7
      OB_MAKE_NARROW_IMPL(C_impl)
   };
 8
 9
10 OB_MAKE_NARROW_IMPL_3(C_impl, C_skel, A_impl, B_impl)
```

- 3-5 C_impl is derived from three classes, C_skel, A_impl and B_impl.
- 10 Now OB_MAKE_NARROW_IMPL_3 must be used, with the names of all super classes as arguments.

If you are using ORBACUS on multiple platforms, where some support RTTI and others don't, it might be best to always use OB/Narrow_impl.h, since _narrow_impl will automatically use dynamic_cast<> on those platforms where it is available.

5.6.2 Getting a Servant using Java

This example demonstrates how to cast an object reference to the servant class in Java:

```
1 // Java
 2
 3 public class I_impl extends _IImplBase
   {
 4
 5 }
 6
 7 public void foo(I ref)
 8
   {
 9
       try
10
       {
          I_impl impl = (I_impl)ref;
11
          // The implementation for ref is local
12
13
       }
      catch(ClassCastException ex)
14
15
       ł
16
          // The implementation for ref is not local
17
       }
18
```

3-5 Servant class I_impl is defined.

- 7 The method foo takes an object reference ref to an I object as a parameter.
- 11 An attempt is made to cast ref to I_impl. If this cast succeeded, then the servant is local (i.e., the servant is in the same address space as the program).
- 16 If the cast failed, then ClassCastException will be thrown, indicating that the servant is not in the same address space as the program. In other words, the reference ref is really the proxy for a remote object, therefore you cannot obtain a reference to the servant.

CHAPTER 6 Locating Objects

6.1 Obtaining Object References

Using CORBA, an object can obtain a reference to another object in a multitude of ways. One of the most common ways is by receiving an object reference as the result of an operation, as demonstrated by the following example:

```
1 // IDL
2 interface A
3 {
4 };
5
6 interface B
7 {
8 A getA();
9 };
```

3-5 An interface A is defined.

7-10 An interface B is defined with an operation returning an object reference to an A.

On the server side, A and B can be implemented in C++ as follows:

```
1 // C++
2 class A_impl : virtual public A_skel
3 {
```

```
4 };
 5
 6 class B_impl : virtual public B_skel
 7
   {
 8
      A_var a_;
 9
10 public:
11
12
       void B_impl()
13
       {
14
          a_ = new A_impl;
15
       }
16
17
      virtual A_ptr getA()
18
       {
19
          return A::_duplicate(a_);
       }
20
21
   };
```

2-4 The servant class A_impl is defined, which inherits from the skeleton class A_skel.

6-21 The servant class B_impl is defined, which inherits from the skeleton class B_skel.

12-15 B_impl's constructor creates a new A_impl servant.

17-20 getA returns an object reference to the A_impl servant.

In Java, the interfaces can be implemented like this:

```
1 // Java
 2 public class A_impl extends _AImplBase
 3 {
 4 }
 5
 6 public class B_impl extends _BImplBase
 7
   {
 8
      A a_;
 9
      public B_impl()
10
11
       {
12
          a_ = new A_impl();
       }
13
14
15
      A getA()
       {
16
17
          return a_;
```

- 18 } 19 }
- 2-4 The servant class A_impl is defined, which inherits from the skeleton class _AImplBase.
- 6-19 The servant class B_impl is defined, which inherits from the skeleton class _BImplBase.
- 10-13 B_impl's constructor creates a new A_impl servant.
- 15-18 getA returns an object reference to the A_impl servant.

A client written in C++ could use code like the following to get references to A:

```
1 // C++
2 B_var b = ... // Get a B object reference somehow
3 A_var a = b -> getA();
```

3 Invoke getA to obtain an object reference for an A.

And in Java:

```
1 // Java
2 B b = ... // Get a B object reference somehow
3 A a = b.getA();
```

3 Invoke getA to obtain an object reference for an A.

In this example, once your application has a reference to a B object, it can obtain a reference to an A object using getA. The question that arises, however, is How do I obtain a reference to a B object? This chapter answers that question by describing a number of ways an application can *bootstrap* its first object reference.

6.2 Lifetime of Object References

All of the strategies described in this chapter involve the publication of an object reference in some form. A common source of problems for newcomers to CORBA is the lifetime and validity of object references. Using IIOP, an object reference can be thought of as encapsulating several pieces of information:

- hostname
- port number
- object key

If any of these items were to change, any published object references containing the old information would likely become invalid and their use might result in an INV_OBJREF

exception being raised. The sections below discuss each of these components and describe the steps you can take to ensure that a published object reference remains valid.

6.2.1 Hostname

By default, the hostname in an object reference is the canonical hostname of the host on which the server is running. Therefore, running the server on a new host invalidates any previously published object references for the old host.

ORBACUS provides the -OAhost option to allow you to override the hostname in any object references published by the server. This option can be especially helpful when used in conjunction with the Domain Name System (DNS), in which the -OAhost option specifies a hostname alias that is mapped by DNS to the canonical hostname.

See "Initializing the Java BOA in JDK 1.2" on page 45 for more information on the – OAhost option.

6.2.2 Port Number

Each time a server is executed, the BOA selects a new port number on which to listen for incoming requests. Since the port number is included in published object references, subsequent executions of the server could invalidate existing object references.

To overcome this problem, ORBACUS provides the -OAport option that causes the BOA to use the specified port number. You will need to select an unused port number on your host, and use that port number every time the server is started.

See "Initializing the Java BOA in JDK 1.2" on page 45 for more information on the – OAport option.

6.2.3 Object Key

Each object created by a server is assigned a unique key that is included in object references published for the object. Furthermore, the order in which your server creates its objects affects the keys assigned to those objects.

To ensure that your objects always have the same keys, ORBACUS allows you to specify a unique name to be used as the key for an object. See "Named Servants" on page 76 for more information.

6.3 Stringified Object References

The CORBA specification defines two operations on the ORB interface for converting object references to and from strings.

```
// IDL
module CORBA
{
    interface ORB
    {
        string object_to_string(in Object obj);
        Object string_to_object(in string ref);
    };
};
```

Using "stringified" object references is the simplest way of bootstrapping your first object reference. In short, the server must create a stringified object reference for an object and make the string available to clients. A client obtains the string and converts it back into an object reference, and can then invoke on the object.

The examples discussed in the sections below are based on the IDL definitions presented at the beginning of this chapter.

6.3.1 Using a File

One way to publish a stringified object reference is for the server to create the string using <code>object_to_string</code> and then write it to a well-known file. Subsequently, the client can read the string from the file and use it as the argument to <code>string_to_object</code>. This method is shown in the following C++ and Java examples.

First, we'll look at the relevant server code:

```
1 // C++
2 CORBA_ORB_var orb = ... // Get a reference to the ORB somehow
3 B_var impl = new B_impl;
4 CORBA_String_var s = orb -> object_to_string(impl);
5 ofstream out("object.ref")
6 out << s << endl;
7 out.close();</pre>
```

- 3 A servant for the interface B is created.
- 4 The object reference of the servant is "stringified".
- 5-7 The stringified object reference is written to a file.

In Java, the server code looks like this:

```
1 // Java
2 org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
3 B impl = new B_impl();
4 String ref = orb.object_to_string(impl);
5 java.io.PrintWriter out = new java.io.PrintWriter(
6     new java.io.FileOutputStream("object.ref"));
7 out.println(ref);
8 out.flush();
```

3 A servant for the interface B is created.

- 4 The object reference of the servant is "stringified".
- 5-8 The stringified object reference is written to a file.

Now that the stringified object reference resides in a file, our clients can read the file and convert the string to an object reference:

```
1 // C++
2 CORBA_ORB_var orb = ... // Get a reference to the ORB somehow
3 ifstream in("object.ref");
4 char s[1000];
5 in >> s;
6 CORBA_Object_var obj = orb -> string_to_object(s);
7 B_var b = B::_narrow(obj);
```

3-5 The stringified object reference is read.

- 6 string_to_object creates an object reference from the string.
- 7 Since the return value of string_to_object is of type CORBA_Object_ptr, B::_narrow must be used to get a B_ptr (which is assigned to a self-managed B_var, in this example).

```
1 // Java
2 org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
3 java.io.BufferedReader in = new java.io.BufferedReader(
4 new java.io.FileReader("object.ref"));
5 String ref = in.readLine();
6 org.omg.CORBA.Object obj = orb.string_to_object(ref);
7 B b = BHelper.narrow(obj);
```

3-5 The stringified object reference is read.

- 6 string_to_object creates an object reference from the string.
- 7 Use BHelper.narrow to narrow the return value of string_to_object to B.

6.3.2 Using a URL

It is sometimes inconvenient or impossible for clients to have access to the same filesystem as the server in order to read a stringified object reference from a file. A more flexible method is to publish the reference in a file that is accessible by clients as a URL. Your clients can then use HTTP or FTP to obtain the contents of the file, freeing them from any local filesystem requirements. This strategy only requires that your clients know the appropriate URL, and is especially suited for use in applets.

Note: This example will only be shown in Java, because of its built-in support for URLs, but the strategy can also be used in C++.

```
1 // Java
 2 import java.io.*;
 3 import java.net.*;
 4
 5 String location = "http://www.mywebserver/object.ref";
   org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
 6
 7
 8 URL url = new URL(location);
 9 URLConnection conn = url.openConnection();
10 BufferedReader in = new BufferedReader(
      new InputStreamReader(conn.getInputStream()));
11
12 String ref = in.readLine();
13 in.close();
14
15 org.omg.CORBA.Object object = orb.string_to_object(ref);
16 B b = BHelper.narrow(object);
```

5 location is the URL of the file containing the stringified object reference.

8-14 Read the string from the URL connection.

16 Convert the string to an object reference.

17 Narrow the reference to a B object.

6.3.3 Using Applet Parameters

In addition to using the URL method described in the previous section, an applet can also use an applet parameter to obtain a stringified object reference. The following HTML illustrates this concept:

The stringified object reference is inserted directly into the HTML file and passed to the applet as a parameter. The applet can retrieve this parameter and convert it to an object reference as shown below:

```
1 // Java
2 org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
3 String ref = getParameter("ref");
4 org.omg.CORBA.Object object = orb.string_to_object(ref);
5 B b = BHelper.narrow(object);
```

- 3 Obtain the applet parameter ref.
- 4 Convert the string to an object reference.
- 5 Narrow the object reference to a B object.

The presence of the stringified object reference in the HTML file could present a maintenance problem. One solution is for the server to write the entire HTML file, thereby ensuring that the object reference is always up to date. You can find an example of this approach in the demo/hello subdirectory.

See "Applets" on page 61 for more information on using ORBACUS in applets.

6.4 Connecting to Named Objects

In some applications, it may be necessary for the client to have no resource dependencies (e.g., files, URLs, etc.) in order to bootstrap an object reference. In this case, you can use the ORBACUS-specific iiop:// notation for IORs or the ORB operation get_inet_object. The only prerequisites are that the object must have been assigned a name by the server (see "Named Servants" on page 76), and the client must be able to determine the hostname and port number of the server and the name of the desired object.

The services included with ORBACUS all use named objects that can be accessed using get_inet_object. The names for these objects can be found in "Object Names for the Basic Services" on page 157.

6.4.1 Using the iiop:// Notation

The standard string representation of an object reference is completely opaque and can be quite long, making it difficult to use. ORBACUS also supports a non-standard but more human-friendly string representation of an object reference that uses URL notation:

```
iiop://hostname:port/object-name
```

This notation is only suitable for referring to named objects, but it can be used anywhere a normal stringified object reference is expected.

6.4.2 Using get_inet_object

The ORB operation get_inet_object is defined as follows:

Here's an example of using get_inet_object in C++:

```
1 // C++
2 CORBA_ORB_var orb = ... // Get a reference to the ORB somehow
3 CORBA_Object_var obj = orb -> get_inet_object(host, port, "MyName");
4 B_var b = B::_narrow(obj);
```

- 3 get_inet_object is called with the hostname, the port number and the object name, which in this case is "MyName".
- 4 As with string_to_object, the reference returned by get_inet_object must be narrowed to a B reference.

Here is an identical implementation in Java:

```
1 // Java
```

```
2 org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
3 org.omg.CORBA.Object obj =
```

```
4 ((com.ooc.CORBA.ORB)orb).get_inet_object(host, port, "MyName");
5 B b = BHelper.narrow(obj);
```

- 3-4 The operation get_inet_object is only defined in com.ooc.CORBA.ORB (because it is ORBACUS-specific), therefore the cast is necessary.
 - 5 Again, we must narrow to the derived type B.

6.5 Initial Services

The CORBA specification provides another standard way to bootstrap an object reference through the use of *initial services*, which denote a set of unique services whose object references, if available, can be obtained using the ORB operation resolve_initial_references, which is defined as follows:

```
// IDL
module CORBA
{
    interface ORB
    {
        typedef string ObjectId;
        exception InvalidName {};
        Object resolve_initial_references(in ObjectId identifier)
        raises(InvalidName);
    };
};
```

Initial services are intended to have well-known names, and the OMG has standardized the names for some of the CORBAservices [4]. For example, the Naming Service has the name "NameService", and the Trading Service has the name "TradingService".

6.5.1 Resolving an Initial Service

An example in which the ORB is queried for a Naming Service object reference will demonstrate how to use resolve_initial_references. The example assumes that the ORB has already been initialized as usual. First the Java version:

```
1 // Java
2 org.omg.CORBA.Object obj = null;
3 org.omg.CosNaming.NamingContext ctx = null;
4
5 try
6 {
```

```
obj = orb.resolve_initial_references("NameService");
7
8 }
9 catch(org.omg.CORBA.ORBPackage.InvalidName ex)
10 {
11
      // An error occured, service is not available
12 }
13
14 if(obj == null)
15 {
     // The object reference is invalid
16
17 }
18
19 ctx = org.omg.CosNaming.NamingContextHelper.narrow(obj);
20 if(ctx == null)
21 {
22
     // This object does not implement a NamingContext
23 }
```

And here's the C++ version:

```
1 // C++
2 CORBA_Object_var obj;
3 CosNaming_NamingContext_var ctx;
4
5 try
6 {
7
      obj = orb -> resolve_initial_references("NameService");
8 }
9 catch(CORBA_InvalidName&)
10 {
      // An error occured, service is not available
11
12 }
13
14 if(CORBA_is_nil(obj))
15 {
     // The object reference is invalid
16
17 }
18
19 ctx = CosNaming_NamingContext::_narrow(ctx);
20 if(CORBA_is_nil(ctx))
21 {
      // This object does not implement NamingContext
22
23 }
```

- 5-12 Try to resolve the name of a particular service. If a service of the specified name is not known to the ORB, an InvalidName exception is thrown.
- 19-23 The service type was known. Now the object reference has to be narrowed to the particular service type. If this fails, the service is not available.

ORBACUS allows you to define your own initial services, as described in the next section. However, these are the recommended names for the services included with ORBACUS:

```
NameService
PropertyService
EventService
```

6.5.2 Providing IORs of Initial Services

When starting a program that makes use of an initial service, the object references of the objects implementing these services have to be registered with the ORB. ORBACUS supports the -ORBservice command-line option for adding an initial service:

```
-ORBservice name IOR
```

The -ORBconfig option is an alternative method for defining a list of initial services, and is often preferable when a number of services must be defined. See "Initializing the Java BOA in JDK 1.2" on page 45 for more information on the -ORBservice and -ORBconfig options.

In addition to using command-line parameters, a program can also add to the list of initial services using the ORBACUS-specific ORB operation add_initial_reference:

For example, in C++:

```
1 // C++
2 CORBA_ORB_var orb = ... // Get a reference to the ORB somehow
3 CORBA_Object_var obj = ... // Get a name service reference somehow
4 orb -> add_initial_reference("NameService", obj);
```

Or in Java:

```
1 // Java
```

- 2 org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
- 3 org.omg.CORBA.Object obj = ...// Get a name service reference somehow
- 4 ((com.ooc.CORBA.ORB)orb).add_initial_reference("NameService", obj);
- 3 Get a reference to the naming service, for example by reading a stringified object reference and converting it with string_to_object, or by using get_inet_object, or by any other means.
- 4 Add the reference to the ORB's list of initial references. In Java, it's necessary to cast the ORB to com.ooc.CORBA.ORB, since add_initial_reference is an ORBACUS-specific extension and thus is not supported with org.omg.CORBA.ORB.

Locating Objects

CHAPTER 7 Reference Counting

7.1 What is Reference Counting?

Reference counting is a commonly-used technique to manage CORBA servant and proxy objects. In general, a reference count is an integer value associated with an object. The counter is initialized to 1, and will be incremented and decremented during the life of the object. When the counter reaches zero, the object is destroyed.

Unlike some distributed object technologies, most notably Microsoft's Distributed Component Object Model (DCOM), CORBA reference counting mechanisms typically are not distributed. In other words, the reference count of a proxy is independent of the reference count of its corresponding servant. Therefore, if the reference count of a proxy reaches zero, the proxy object is destroyed, but the servant is unaffected. Similarly, the reference counts of any proxy objects for a servant are not affected when that servant's reference count reaches zero and the servant is subsequently destroyed.

7.2 Reference Counting in Java

ORBACUS for Java does not need to use reference counting because the standard Java garbage collector performs this activity automatically. However, there is one issue that should be mentioned regarding garbage collection of servant objects.

In Java, the garbage collector does not reclaim an object until there are no more references to that object held by the program. When you use the ORB's connect method to connect

a servant to the object adapter, the ORB will keep a reference to your servant. Therefore, in order for your servant to be eligible for garbage collection, you must eliminate all references to the servant in your server code, and you must use the ORB's disconnect method to ensure that the ORB no longer holds a reference to the servant. Although use of connect is optional, because the ORB will automatically connect objects when necessary, use of disconnect is always required.

7.3 Reference Counting in C++

ORBACUS for C++ implements servants and proxies as reference-counted objects. The reference-counting semantics used by ORBACUS for C++ are outlined in Table 7.1.

new Servant_impl	Reference count of new servant is initialized to 1
ORB::string_to_object	Reference count of proxy is initialized to 1
ORB::get_inet_object	Like string_to_object, reference count of proxy is initialized to 1
ORB::connect(servant)	Reference count of servant is incremented by 1, since a reference to the servant is added to the object adapter. ^a
ORB::disconnect(servant)	Reference count of servant is decremented by 1, since the object adapter's reference to the servant is removed. ^b
_duplicate(obj)	Reference count of servant or proxy is incremented by 1
CORBA_release(obj)	Reference count of servant or proxy is decremented by 1

Table 7.1: C++ Reference Counting Semantics

- a. The reference count is only incremented by 1 after the first (implicit or explicit) call to connect. Subsequent calls to connect do not affect the reference count.
- b. If the servant is already disconnected, calling disconnect again does not change the reference count.

7.3.1 Marshalling Issues

When a server returns the object reference of a servant to a client, either as a return value or as an out or inout parameter, the marshalling code automatically decrements the ser-

vant's reference count by 1. Therefore, you will need to use _duplicate if you wish to preserve the existing reference count of your servant, as shown in the following example.

```
1 // IDL
2
3 interface A
4 {
5 };
6
7 interface B
8 {
9         A getA();
10 };
```

3–10 Interfaces A and B are defined.

9 The operation getA returns a reference to an object of A.

Here is our implementation:

```
1 // C++
 2
 3 class A_impl : public virtual A_skel
 4 {
 5 };
 6
 7 class B_impl : public virtual B_skel
   {
 8
 9
      A_var a_;
10
      CORBA_ORB_var orb_;
11
12 public:
13
      void B_impl(CORBA_ORB_ptr orb)
14
15
          : orb_(CORBA_ORB::_duplicate(orb))
16
       {
17
          a_ = new A_impl;
          orb_ -> connect(a_);
18
       }
19
20
21
      virtual A_ptr getA()
22
       {
23
          return A::_duplicate(a_);
24
       }
25
   };
```

- 3-5 Servant class A_impl is defined.
- 7-25 Servant class B_impl is defined.
- 14-19 The B_impl constructor saves a reference to the ORB, instantiates A_impl and connects it to the object adapter. It is not strictly necessary to invoke connect, because the object will be connected automatically when the object's reference is returned to a client.
- 21-24 Upon entry to getA, the reference count of a_ is 2 (the initial value is 1 upon construction, and is incremented to 2 when connected). To maintain this value, getA duplicates a_, which increments the reference count to 3. The marshalling code that returns the reference will decrement the reference count back to 2.

For more information on using object references as in, inout, out and return values, see "Object References" on page 116.

7.3.2 Releasing Proxies and Servants

The reference count of a servant is incremented by 1 when the servant is (implicitly or explicitly) connected to the object adapter (see "Connecting Servants using C++" on page 75). Therefore, **you must disconnect a servant from the object adapter prior to releasing it** with CORBA_release in order to ensure that its reference count reaches zero. See "Factory Objects using C++" on page 78 for an example that properly manages the reference count of a servant.

It is important to remember to never use delete to destroy proxies or servants. Use only CORBA_release. For example, the following code calling delete on a proxy obtained with string_to_object is wrong:

```
1 const char* s = ... // Obtain a stringified reference somehow
```

```
2 CORBA_Object_ptr p = orb -> string_to_object(s);
```

```
3 delete p; // Wrong!
```

3 This line is wrong. Instead of delete, CORBA_release must be used.

This is the correct version:

```
1 const char* s = ... // Obtain a stringified reference somehow
2 CORBA_Object_ptr p = orb -> string_to_object(s);
3 CORBA_release(p);
```

3 OK, CORBA_release is used.

You should use self-managed types whenever possible:

- 1 const char* s = ... // Obtain a stringified reference somehow
 2 CORBA_Object_var p = orb -> string_to_object(s);
- 2 No CORBA_release is necessary, since the _var will automatically call CORBA_release upon destruction.

You should also avoid allocating servants on the stack. If you do so, the servant will be destroyed if the stack unwinds, without any calls to CORBA_release. The following code demonstrates the problem:

```
1 // C++
2
3 void f()
4 {
5    I_impl impl; // Wrong!
6 }
```

5 Upon return from f, impl is destroyed without the proper call to CORBA_release.

7.3.3 Global Object References

You should never have global _var type object references, because you can never tell exactly when and in which order they will be destroyed. For example, it is possible that a _var reference could be destroyed *after* the ORB was destroyed. Here's an example.

```
1 I_var impl; // Don't do this!
2
3 int
4 main(int argc, char* argv[], char*[])
5 {
6 CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
7 impl = new I_impl;
8 return 0;
9 }
```

- 1 A global object reference _var type is created.
- 6 The ORB is initialized.
- 7 The I_var object reference is initialized with a new servant.
- 8 Upon return, the ORB is destroyed (since orb is destroyed, causing CORBA_release to be called for the ORB). However, impl is still alive, and therefore the servant is not

destroyed, meaning that there is still a servant, but no ORB anymore. This will most likely result in a crash.

The ORB must be the last object to be destroyed! In addition to the technical justification for avoiding global object references, it is generally a bad programming style to have global object references.

7.3.4 Cyclic Object Dependencies

Consider the following code:

```
1 class X_impl : virtual public X_skel
 2 {
 3
      Y_var y_;
 4
 5 public:
 6
 7
      void setY(Y_ptr y) { y_ = Y::_duplicate(y); }
 8 };
 9
10 class Y_impl : public Y_skel
11 {
12
      X_var x_;
13
14 public:
15
      void setX(X_ptr x) { x_ = X::_duplicate(x); }
16
17 };
18
19 void f()
20 {
21
      X_var x = new X_impl;
      Y_var y = new Y_impl;
22
23
      x \rightarrow setY(y);
24
      y \rightarrow setX(x);
25
   }
```

- 1-8 A servant class X_impl is defined, which has a Y_var data member that can be set with setY.
- 10-17 Ditto, but a servant class Y_impl with a data member X_var is defined.
- 19-25 The function f creates new x and y servants. It stores the reference of the x servant in the y servant and vice versa.

Here the x_impl has a reference to the Y_impl and the Y_impl has a reference to the x_impl, what is known as a "cyclic object dependency." This means that when f returns, even though x and y get destroyed, the objects they are referring to are *not* destroyed since the reference count never becomes zero. Why? Let's take a deeper look into what happens in the example program:

```
X_var x = new X_impl
```

The initial reference count of the X_impl after the new is 1.

```
Y_var y = new Y_impl
```

Same as above, the initial reference count of the Y_impl is 1.

x -> setY(y)

After setY, the reference count of the Y_impl is 2.

y -> setX(x)

After setX, the reference count of the X_impl is 2.

return

x and y get destroyed and therefore call CORBA_release on their contents, so the reference count of the x_impl and the y_impl is 1. This means that after the return of f the x_impl and the y_impl will live forever.

This problem can be solved by adding a releaseInternal function¹ to at least one of the two interface implementations. For example:

```
1 class X_impl : public X_skel
   {
 2
 3
      Y_var y_;
 4
 5 public:
 б
 7
      void setY(Y_ptr y) { y_ = Y::_duplicate(y); }
 8
   };
 9
10 class Y_impl : public Y_skel // Implements interface Y
11 {
12
      X_var x_;
13
14 public:
```

1. Of course you are free to choose whatever name you like.

```
15
       void setX(X_ptr x) { x_ = X::_duplicate(x); }
16
       void releaseInternal() { x_ = X::_nil(); }
17
18 };
19
20 void f()
21 {
22
       X_var x = new X_impl;
23
       Y_var y = new Y_impl;
24
       x \rightarrow setY(y);
       y \rightarrow setX(x);
25
26
       y -> releaseInternal();
27 }
```

1-8 Same as before

10-18 The releaseInternal operation has been added.

26 releaseInternal is called before f returns.

Now both the X_impl and the Y_impl get destroyed at the return of f:

X_var x = new X_impl

The initial reference count of the x_impl after the new is 1.

Y_var y = new Y_impl

Same as above, the initial reference count of the Y_impl is 1.

x -> setY(y)

After setY, the reference count of the Y_impl is 2.

y -> setX(x)

After setX, the reference count of the X_impl is 2.

```
y -> releaseInternal()
```

The releaseInternal function sets the x_value of the Y_impl to X::_nil. Assignment to a _var object reference causes CORBA_release to be called on its contents. So now the reference count of the X_impl is 1.

return

x and y are destroyed and therefore call CORBA_release on their contents. That means that the reference count of the x_impl becomes zero, resulting in x_impl being destroyed. This of course also eliminates x_impl's y_ data member, causing

Reference Counting in C++

CORBA_release to be called on the Y_impl. So the Y_impl's reference count also becomes zero and the Y_impl is also destroyed.

Reference Counting
CHAPTER 8 C++Mapping Notes

ORBACUS implements the IDL-to-C++ mapping as described in [2]. The standard IDL-to-C++ mapping is not a topic of this manual. Please refer to [2] for the exact specifications.

8.1 Reserved Names

All names starting with OB, $_OB_$ or $_ob_$ are reserved by ORBACUS for internal use and must not be used as identifiers.¹

8.2 Mapping of Modules

Generally, IDL modules are mapped to C++ namespaces. However, since most C++ compilers currently do not support namespaces, the IDL-to-C++ mapping defines two alternatives. The first one maps modules to C++ classes, implying that nested classes are needed for interfaces or other modules defined within a module. The second alternative is to map modules to name prefixes, e.g., the name of an interface I in a module M is mapped to M_I.

ORBACUS uses the name prefix mapping alternative for the following reasons:

• As mentioned earlier, C++ namespaces are not widely available yet. ORBACUS was designed to be portable among a variety of C++ compilers. Therefore using namespaces was not possible.

^{1.} Who wants to use such ugly names anyway?

• Although nested classes are available with most C++ compilers, this mapping alternative has the disadvantage that modules cannot be "reopened" (since classes cannot be reopened). That is, it is not possible to define in one IDL file one part of a module and in another IDL file another part of the same module.

8.3 Extensions

ORBACUS provides several extensions to the standard IDL-to-C++ mapping. If you are concerned about source code compatibility with CORBA-compliant ORBs from other vendors, you should not use these extensions. However, if you plan to use your source code exclusively with ORBACUS these extensions will reduce programming overhead.

8.3.1 Extensions to the String Type

The ORBACUS CORBA_String_var type provides the operator+= for appending to the string. The argument to operator+= can be of type const char*, char and unsigned char as well as short, unsigned short, int, unsigned int, long and unsigned long. For example:

```
1 CORBA_String_var s;
2 s += "abc";
3 s += 'x';
4 s += 'y';
5 s += 'z';
6 s += 12345;
```

- 1 s is empty.
- 2 s is "abc".
- 3 s is "abcx".
- 4 s is "abcxy".
- 5 s is "abcxyz".
- 6 s is "abcxyz12345".

8.3.2 Extensions to _var Types

All _var types have the following additional member functions:

- in: This function converts the _var type to a type suitable for in parameters.
- inout: This function converts the _var type to a type suitable for inout parameters.

- out: This function converts the _var type to a type suitable for out parameters. As a side effect, this function ensures that the value held by the _var is released or freed, by either calling CORBA_string_free (in case of a string), CORBA_release (in case of an object reference) or delete (in case of types like sequences, variable-length structs etc.).
- _retn: This function converts the _var type to a type suitable for function return values. The _retn function also removes the value that is held by the _var type without destroying it, i.e., without calling delete, CORBA_string_free or CORBA_release on its value. For example consider a function f that returns its three in string arguments as a single string:

```
char*
f(const char* s1, const char* s2, const char* s3)
{
    CORBA_String_var s = s1;
    s += s2;
    s += s3;
    return s._retn();
}
```

Please note that these functions are not covered by the CORBA 2.0 version of the IDL-to-C++ mapping, but it is likely that they will become a part of the standard for the next major mapping revision.

8.3.3 Extensions to Sequence Types

All unbounded non-array sequences (for example unbounded string, struct and object reference sequences) have an additional insert, append and remove member function. For a sequence s and a value v, the s.insert(v) and s.append(v) behave as follows:

```
s.length(s.length() + 1);
... // Somehow shift sequence contents one to the right
s[0] = v;
and
s.length(s.length() + 1)
s[s.length() - 1] = v;
```

respectively.

Please note that ORBACUS's sequence implementation does not really shift the contents of the sequence. It is rather implemented as a "double ended queue" (like the Standard Template Library's "dequeue"), and therefore needs no value shifting. That is, the insert function is as efficient as the append function.

8.4 C++ Mapping Tips & Tricks

Unfortunately, the official CORBA IDL-to-C++ mapping is a little complicated.¹ The traps & pitfalls justify devoting a section of the ORBACUS manual to how to avoid the most common mistakes.

Note that compared to the IDL-to-C++ mapping, the IDL-to-Java mapping is nice, clean and easy to understand, so it's not really necessary to have a "Java Mapping Tips & Tricks". The official mapping specification [3] is completely sufficient.

8.4.1 CORBA Strings

When using CORBA strings, always remember the following rules.

CORBA-Specific String Functions

Use the CORBA-specific string functions CORBA_string_alloc, CORBA_string_free and CORBA_string_dup if you're dealing with CORBA strings. Never use new, delete, malloc, free, strdup or similar functions.

For example, the following code is incorrect:

```
1 char* s1 = strdup("Hello!"); // Wrong!
2
3 // Allocate a string for 10 characters + trailing '\0' ...
4 String_var s2 = malloc(11); // Wrong!
```

1 Error, CORBA_string_dup must be used instead of strdup.

4 No! CORBA_string_alloc must be used!

This is the correct version:

```
1 char* s1 = CORBA_string_dup("Hello!");
2
3 // Allocate a string for 10 characters + trailing '\0' ...
4 CORBA_String_var s2 = CORBA_string_alloc(10);
```

1 OK, CORBA_string_dup is fine.

^{1.} Note that OOC did not invent this mapping. We just had to implement it exactly as specified to be CORBA compliant.

4 OK. Note that CORBA_string_alloc (unlike malloc) adds an additional character for the trailing "\0" automatically.

This code is wrong, too:

```
1 free(s2); // Wrong!
2
```

1 No! Use CORBA_string_free!

And again, the corrected version:

```
1 CORBA_string_free(s1);
```

- 2
- 1 This is OK. Note that there is no need to free s2 explicitly since CORBA_String_var types release the string they manage automatically when the CORBA_String_var type is destroyed.

Initialization and Assignment from char* and const char*

Initialization of a CORBA_String_var type or assignment to a CORBA_String_var type from a char* type value *consumes* that value. That means that if the CORBA_String_var is destroyed, the value from which the CORBA_String_var was initialized or that was assigned to the CORBA_String_var will *also be destroyed*.

Initialization of a CORBA_String_var type or assignment to a CORBA_String_var type from a const char* type value *duplicates* that value. This means that if the CORBA_String_var is destroyed, the value from which the CORBA_String_var was initialized or that was assigned to the CORBA_String_var is *not destroyed*.

Note that for compatibility reasons with C the type of string literals in C++ is char*, not const char*. So the following code is wrong:

```
1 CORBA_String_var s = "Hello!"; // Wrong!
2
```

1 Error, since "Hello!" is char*, not const char*.

The following code is OK:

```
1 CORBA_String_var s1 = CORBA_string_dup("Hello!");
2 CORBA_String_var s2 = (const char*)"Hello!";
```

1 OK, s1 consumes the value returned by CORBA_string_dup.

2 OK, s2 will implicitly duplicate "Hello!".

Initialization and Assignment from CORBA_String_var

Initialization of a CORBA_String_var type or assignment to a CORBA_String_var type from another CORBA_String_var type value automatically duplicates that value. This means that it is not necessary to use explicit calls to CORBA_string_dup. The following examples are correct:

```
1 CORBA_String_var s1 = CORBA_string_dup("ABC");
2 CORBA_String_var s2 = s1;
3 CORBA_String_var s3 = CORBA_string_dup(s1);
```

- 2 OK, s2 will implicitly duplicate "ABC".
- 3 Also OK, explicit duplication.

Note that string elements of a structure, elements of a string array and elements of a string sequence behave exactly like the CORBA_String_var type¹, i.e., you can deliberately assign between these types or use one of these types to initialize any other of these types. There is no need to call CORBA_string_dup explicitly.

Strings as Parameters and Return Values

If a function is called returning a string value via an out or inout parameter or as a return value, the callee must *duplicate* and the caller must *release* this value. The duplication can be done using CORBA_string_dup and the release by either explicitly calling CORBA_string_free or by assigning the value to a CORBA_String_var. For example:

```
1 // IDL
2 interface I
3 {
4 string op(out string os, inout string ios);
5 };
```

4 An operation op is defined with an out string argument, an inout string argument and a string return value.

The following implementation of I's op operation is wrong:

^{1.} In code generated by the ORBACUS IDL-to-C++ translator, array and structure string elements *are* actually of type CORBA_String_var. String sequence elements are not of type CORBA_String_var (for technical reasons), but the type used for string sequence elements behaves exactly like the CORBA_String_var type.

```
1 // C++
 2 class I_impl : virtual public I_skel
 3 {
 4 public:
 5
      virtual char* op(char*& os, char*& ios)
 6
      {
 7
          // Wrong, ios is not freed
          ios = "abc"; // Wrong!
 8
          os = "def"; // Wrong!
 9
10
          return "ghi"; // Wrong!
      }
11
12 };
```

7 Forgot to free the inout string parameter ios.

8,9,10 Wrong. Strings must be duplicated.

Here is the correct version:

```
1 // C++
 2 class I_impl : virtual public I_skel
 3 {
 4 public:
      virtual char* op(char*& os, char*& ios)
 5
 6
      {
 7
          CORBA_string_free(ios);
          ios = CORBA_string_dup("abc");
 8
 9
          os = CORBA_string_dup("def");
          return CORBA_string_dup("ghi");
10
11
      }
12
   };
```

7 Now ios is freed.

8-10 All String values are now duplicated.

Here is an example showing how to use string out, inout or return values on the calling side if CORBA_string_free is used:

```
1 // C++
2 I_ptr i = ... // Get a reference to an I somehow
3
4 char* out;
5 char* inOut = CORBA_string_dup("This is my inout arg");
6 char* result;
```

```
7
8 result = i -> op(out, inOut);
9
10 CORBA_string_free(out);
11 CORBA_string_free(inOut);
12 CORBA_string_free(result);
```

- 4-6 The parameters are defined. A value must be assigned to the inout parameter. Of course values to in parameters must also be assigned, but our example does not have any in parameters.
 - 8 op is called.
- 10-12 All out and inout parameters, as well as the return value, must be freed.

Here is the same example, but with self-managed CORBA_String_var types instead of explicitly calls to CORBA_string_free:

```
1 // C++
2 I_ptr i = ... // Get a reference to an I somehow
3
4 CORBA_String_var out;
5 CORBA_String_var inOut = CORBA_string_dup("This is my inout arg");
6 CORBA_String_var result;
7
8 result = i -> op(out, ios);
```

- 4-6 CORBA_String_var is used instead of char*.
 - 8 After the call to op, no explicit calls to CORBA_string_free are necessary, since the CORBA_String_var type destroys its contents automatically.

Since method two in this example is much less error prone, you should always use the self-managed type CORBA_String_var in such a case.

8.4.2 Object References

If you use CORBA object references, i.e., _ptr and _var types for specific interfaces, keep the following in mind.

Object References as Parameters and Return Values

If a function returning an object reference via an out or inout parameter or as a return value is called, the callee must *duplicate* and the caller must *release* the reference. As described above, an object reference to an object of type I (i.e., an object with the inter-

face I) is duplicated with I::_duplicate and released with CORBA_release. This is quite similar to strings as parameters and return values. For example:

```
1 // IDL
2 interface I
3 {
4 };
5
6 interface A
7 {
8 I op(out I oref, inout I ioref);
9 };
```

- 2-4 An interface I is defined.
- 6-9 An interface A is defined, having an operation op, which returns an I and has an I in and inout parameter.

This implementation of the op operation is wrong:

```
1 // C++
 2 class A_impl : virtual public A_skel
 3 {
 4
      I_var myref;
 5
  public:
 6
 7
      A_impl()
 8
 9
       {
10
          myref = ... // Initialize myref somehow
11
       }
12
13
      virtual I_ptr op(I_ptr& oref, I_ptr& ioref)
14
       {
15
          // Wrong, ioref is not released
          ioref = myref; // Wrong!
16
          oref = myref; // Ditto!
17
          return myref; // Ditto!
18
       }
19
20
   };
```

15 Forgot to free the inout object reference parameter ioref.

16-18 Wrong. Object references must be duplicated.

This version is correct:

```
1 // C++
 2 class A_impl : virtual public A_skel
 3 {
 4
      I_var myref;
 5
 6 public:
 7
 8
      A_impl()
 9
       {
10
          myref = ... // Initialize myref somehow
11
       }
12
      virtual I_ptr op(I_ptr& oref, I_ptr& ioref)
13
14
       {
15
          CORBA_release(ioref);
          ioref = I::_duplicate(myref);
16
          oref = I::_duplicate(myref);
17
18
          return I::_duplicate(myref);
19
       }
20
   };
```

15 Now ioref is released.

16-18 All object references are now duplicated.

The first example on how to use object reference out, inout or return values on the calling side uses explicit calls to CORBA_release:

```
1 // C++
2 A_ptr a = ... // Get a reference to an A somehow
3
4 I_ptr out;
5 I_ptr inOut = ... // Get a reference to an I somehow
6 I_ptr result;
7
8 result = a -> op(out, inOut);
9
10 CORBA_release(out);
11 CORBA_release(inOut);
12 CORBA_release(result);
```

4-6 The parameters are defined. A value must be assigned to the inout parameter.

8 op is called.

10-12 All out and inout parameters, as well as the return value, must be released.

The second example uses self-managed I_var types:

```
1 // C++
2 A_ptr a = ... // Get a reference to an A somehow
3
4 I_var out;
5 I_var inOut = ... // Get a reference to an I somehow
6 I_var result;
7
8 result = i -> op(out, ios);
```

- 4-6 I_var is used instead of I_ptr.
 - 8 After the call to op, no explicit calls to CORBA_release are necessary, since the I_var type destroys its contents automatically.

We recommend that you use method two with the self-managed types, since this method is much less error prone.

Differences between String_var and Object Reference _var Types

There is a slight but important difference between String_var and object reference _var types regarding their initialization or assignment from in parameters. Consider the following IDL code:

```
1 // IDL
2 interface Y
3 {
4 };
5
6 interface X
7 {
8 void init(in string s1, in string s2, in Y y1, in Y y2);
9 };
```

Here the init function is used to initialize an X with two strings and two Y object references. The following code shows the difference between _var type assignments from strings and from object references:

```
1 // C++
2 class X_impl : virtual public X_skel
```

```
3 {
      CORBA_String_var s1_;
 4
 5
      CORBA_String_var s2_;
 6
      Y_var y1 ;
 7
      Y_var y2_;
 8
 9 public:
10
      void init(const char* s1, const char* s2, Y_ptr y1, Y_ptr y2)
11
12
       {
13
          s1_{=} s1;
          s2_ = CORBA_string_dup(s2);
14
15
          y1_ = y1; // Wrong!
          y2_ = Y::_duplicate(y2);
16
       }
17
18
   }
```

- 13 OK, CORBA_String_var automatically duplicates const char*.
- 14 Explicit duplication is also OK, as the CORBA_String_var consumes the duplicated string returned from CORBA_string_dup, which returns a string of type char*.
- 15 This is wrong, Y_var consumes the value of type Y_ptr. Therefore Y::_duplicate must be used.
- 16 This is correct now, since Y::_duplicate was used.

The reason for this behavior is that there is no such thing as a constant object reference for in parameters. Therefore it is not possible for the object reference _var type to distinguish between assignments from regular object references and in object references.

CHAPTER 9 CONCURRENCY Models

9.1 Introduction

9.1.1 What is a Concurrency Model?

A concurrency model describes how an Object Request Broker (ORB) handles communication and request execution. There are two main categories of concurrency models, single-threaded concurrency models and multi-threaded concurrency models.

Single-threaded concurrency models describe how an ORB behaves while a request is sent or received in a single-threaded environment. For example, one model is to simply let the ORB block on sending and receiving messages. Another model is to let the ORB do some work while sending and receiving messages, for example to receive user input through a keyboard or a GUI, or to simply transfer buffered messages.

Multi-threaded concurrency models describe how the ORB makes use of multiple threads, for example to send and receive messages "in the background." Multi-threaded concurrency models also describe how several threads can be active in the user code and the strategy the ORB employs to create these threads.

9.1.2 Why different Concurrency Models?

There is no "one size fits all" approach with respect to concurrency models. Each concurrency model provides a unique set of properties, each having advantages and disadvan-

tages. For example, applications using callbacks must have a concurrency model that allows nested method invocations to avoid deadlocks. Other applications must be optimized for speed, in which case a concurrency model with the least overhead will be chosen.

Some ORBs are highly specialized, providing only the most frequently used concurrency models for a specific domain. ORBACUS takes a different approach by supporting several concurrency models.

9.1.3 ORBacus Concurrency Models Overview

ORBACUS allows different concurrency models to be established for the client and server activities of an application. The client-side concurrency models are *Blocking*, *Reactive* and *Threaded*. The server-side concurrency models are *Blocking*, *Reactive*, *Threaded*, *Thread-per-Client*, *Thread-per-Request* and *Thread Pool*.

9.2 Single-Threaded Concurrency Models

9.2.1 Blocking Clients and Servers

The blocking concurrency model is the simplest one. For the client, "blocking" means that the ORB blocks while sending requests to or receiving replies from a server.

A special case are oneway requests,¹ which do not block the ORB. If the ORB determines that sending the oneway request would cause blocking, it puts the oneway request into a request buffer. Whenever the client tries to send another request to the same server, this buffer's contents are sent first.

Blocking servers block the ORB while receiving a request or sending a reply. Additionally, since the ORB blocks on a connection after accepting it with a call like accept, the ORB cannot accept any new connections. Therefore a blocking server can only serve one client at a time. This is shown in Figure 9.1.

Because of its simplicity, the blocking concurrency models are the fastest models available. There is no overhead, neither for calls to operations like select² (because the ORB

^{1.} A oneway request is a request for which no reply is received. Therefore a oneway request cannot return any results and there is no guarantee that a oneway request was properly executed by a server.

^{2.} select is used for synchronous I/O multiplexing. For more information, see the select Unix manual page.



Figure 9.1: Blocking Server

is allowed to block on a single connection), nor for any thread creation or context switches.

9.2.2 Reactive Clients and Servers

Reactive servers use calls to operations like select in order to simultaneously accept incoming connection requests, to receive requests from multiple clients and to send back replies. This means that a reactive server can handle more than one client at a time. This is shown in Figure 9.2. Reactive servers are the most common server types for single-threaded client/server applications.

Reactive clients also use operations like select to avoid blocking. This means that while a request to a server is sent or a reply from that server is received, the client can simultaneously send buffered requests to other servers or receive and buffer replies. This is very useful for oneway operations or the Dynamic Invocation Interface (DII) operation send_deferred in combination with get_response or poll_response.¹

However, the main advantage of a reactive client becomes apparent if it is used together with a reactive server in mixed client/server applications. A mixed client/server applica-



Figure 9.2: Reactive Server

tion is a program that is both a client and server at the same time. Without the reactive concurrency model it is not possible to use nested method calls in single-threaded applications, which are absolutely necessary for most kinds of callbacks.

Consider two programs A and B, both mixed client/server applications. First A tries to call a method \pm on B. Before this method returns, B calls back A by invoking method g. This scenario is quite common, and for example is used in the popular Model-View-Controller pattern [7].

^{1.} For more information on send_deferred, get_response and poll_response, see the chapter "The Dynamic Invocation Interface" in [2].



For blocking client/servers this scenario is shown in Figure 9.3. As you can see, the call-

Figure 9.3: Blocking Client/Server

back g from B to A does not succeed, because A blocks while waiting for a reply for f from B. In contrast, if the reactive concurrency model for the client and the server is used, A can dispatch incoming requests while waiting for B's reply for f. This is shown in Figure 9.4.



Figure 9.4: Reactive Client/Server

The reactive concurrency models are also very fast. There is no overhead for thread creation or context switching. Only an additional call to an operation like select is needed before operations such as send, recv or accept can be used by the ORB.¹

9.3 Multi-Threaded Concurrency Models

9.3.1 Threaded Clients and Servers

A threaded client uses two separate threads for each connection to a server, one for sending requests and another for receiving replies. In contrast to a blocking server, this model has the advantage that oneway requests can be sent "in the background", i.e., without blocking the user thread execution. The separate receiver thread allows messages to be received and buffered for later retrieval by the user thread with DII operations such as get_response or poll_response.

Like a threaded client, a threaded server uses separate threads for receiving requests from clients and sending replies. Additionally, there is a separate thread dedicated to accepting incoming connection requests, so that a threaded server can serve more than one client at a time.

ORBACUS's threaded server concurrency model allows only one active thread in the user code. This means that even though many requests can be received simultaneously, the execution of these requests is serialized. This is shown in Figure 9.5. (For simplicity, the



Figure 9.5: Threaded Server

"dispatch" arrows and the corresponding return arrows are omitted in this and all follow-

^{1.} Instead of directly using operations like select, ORBACUS uses a *Reactor* to provide for flexible integration with existing event loops and to allow the installation of user supplied event handlers. See Chapter 10 for more information.

ing diagrams.) In the example, the threaded server has two clients connected to it and thus two receiver threads (sender threads not shown). First A calls f on the server. If, before f returns, B tries to call another operation g, this request is delayed until f returns. The same is true for A's call to h, which must wait until g returns.

Allowing only one active thread in user code has the advantage of the user code not having to take care of any kind of thread synchronization. This means that the user code can be written as if for a single threaded system, but without losing the advantage of the ORB optimizing its operation by using multiple threads internally.

The threaded concurrency model is still fast. No calls to operations like select are required. Time consuming thread creation is only necessary when a new client is connecting, but not for each request. However, thread context switching makes this approach slower than the blocking concurrency model, at least on a single-processor computer.

9.3.2 Thread-per-Client Server

The thread-per-client server concurrency model is very similar to the threaded server concurrency model, except that the ORB allows one active thread-per-client in the user code. This is shown in Figure 9.6. A's call to f and B's call to g are carried out simultaneously,



Figure 9.6: Thread-per-Client Server

each in its own thread. However, if A tries to call another operation h (for example by sending requests from different threads in a multi-threaded client or by using the DII operation send_deferred in a single-threaded client) as long as f has not finished yet, the execution of h is delayed until f returns.

The thread-per-client model is still efficient. Like with the threaded concurrency model, no threads need to be created, except when new connections are accepted.

9.3.3 Thread-per-Request Server

If the thread-per-request server concurrency model is chosen, the ORB creates a new thread for each request. This is shown in Figure 9.7. (For simplicity there are no separate



Figure 9.7: Thread-per-Request Server

arrows for dispatch and thread creation in the diagram.) With the thread-per-request model, requests are never delayed. When they come in, a new thread is created and the request is executed in the user code using this thread. On return, the thread is destroyed.

Besides using a reactive client together with a reactive server, the thread-per-request server in combination with a threaded client is the only other model that allows nested method calls with an unlimited nesting level. The thread pool model also allows nested method calls, but the nesting level is limited by the number of threads in the pool.

The thread-per-request concurrency model is inefficient. The main problem results from the overhead involved in creating new threads, namely one for each request.

9.3.4 Thread Pool Server

The thread pool model uses threads from a pool to carry out requests, so that threads have to be created only once and can then be reused for other requests. Figure 9.8 shows an



Figure 9.8: Thread Pool Server

example with one client and a thread pool server with three threads in the pool. (Sender and receiver threads are not shown.) The first three operation calls f, g and h can be carried out immediately, since there are three threads in the pool. However, the fourth request i is delayed until at least one of the other requests returns.

Since there is no time-consuming thread creation, the thread pool concurrency model performs better than the thread-per-request model. The thread pool is a good trade-off if on the one hand frequent thread creation and destruction result in unacceptable performance, but on the other hand delaying the execution of concurrent method calls is also not desired.

9.4 Performance Comparisons

9.4.1 Sample Application

In order to measure the performance overhead introduced by a given concurrency model, it is important to keep all other overhead not directly related to the concurrency model minimal. Therefore the sample application for performance measurements only consists of a single interface with a single operation with no parameters and return values:

```
// IDL
interface I
{
    void f()
```

}

This ensures that any additional overhead for parameter marshalling or request dispatching is minimal.

All tests have been performed with ORBACUS for C++ version 3.1.1 on a Linux 2.0.35 based machine, libc 5.4.33, PII 400 MHz, 128 MB memory, egcs 1.0.3a C++ compiler, with optimization (compiled with -O2 -DNDEBUG), shared libraries, and no debug code.

9.4.2 Regular Method Invocations

	Blocking	Reactive	Threaded
Blocking	0.20 ms	0.25 ms	0.28 ms
Reactive	0.25 ms	0.29 ms	0.33 ms
Threaded	0.26 ms	0.31 ms	0.37 ms
Thread-per-Client	0.25 ms	0.30 ms	0.36 ms
Thread-per-Request	0.63 ms	0.68 ms	0.71 ms
Thread Pool	0.31 ms	0.39 ms	0.42 ms

The first test scenario is a server that is used by a single client. Table 9.1 shows the time

 Table 9.1: Regular Method Invocations

needed for a single call to f. In this and all following tables, the different columns correspond to the client side concurrency models and the different rows to the server side concurrency models.

The clear winners are the blocking concurrency models, which are fastest. Second fastest are the reactive concurrency models, followed by the different threaded concurrency models.

Note that Table 9.1 shows the performance results for a thread safe version of ORBACUS. In case no threads are used at all, i.e., if no multi-threaded concurrency model is chosen and if multiple threads are not used in application code, then it's also possible to use a non-thread-safe version of ORBACUS. Table 9.2 shows that such a version is much faster

	Blocking	Reactive
Blocking	0.16 ms	0.20 ms
Reactive	0.20 ms	0.23 ms

Table 9.2: Non-Thread-Safe Version

than a thread-safe one, because there is no additional overhead for any thread synchronization.

9.4.3 Nested Method Invocations

As already pointed out, nested methods invocations are only possible with the following concurrency model combinations:

- reactive client / reactive server
- threaded client / thread-per-request server
- threaded client / thread pool server

Table 9.3 shows the performance results for a nesting level of 100. That is, in the test

	Reactive	Threaded
Reactive	2.78 ms	n/a
Thread-per-Request	n/a	3.39 ms
Thread Pool	n/a	3.23 ms

Table 9.3: Nested Method Invocations

applications there are two mixed client/servers, each of them implementing the IDL code of the test application. The first client/server calls f on the second, and *before* f returns, the second client/server calls f on the first client/server, then the first client/server f on the second again and so on. This is repeated until each client/server called f on the other client/server 50 times, which corresponds to a total nesting level of 100.

Again, the clear winner is a single-threaded concurrency model, namely the reactive concurrency model. Here the difference between single-threaded and multi-threaded concurrency models is very significant, because there is a huge overhead for creating threads and thread context switches in the multi-threaded concurrency models.

The maximum nesting level for the reactive concurrency model is usually much higher than for the thread-per-request and thread pool concurrency models. The reason is that the maximum nesting level for thread-per-request and thread pool is determined by the maximum number of threads allowed per process, whereas the reactive concurrency model is only limited by the maximum stack size per process.

9.5 Selecting Concurrency Models

Concurrency models can be selected either by command-line parameters (see Chapter 4), or with the operations ORB::conc_model and BOA::conc_model. The default concurrency models are shown in Table 9.4.

	Client	Server
Java	Blocking	Threaded
C++	Blocking	Reactive

Table 9.4: Default Concurrency Models

For example, here is how to establish the concurrency models in C++:

```
// C++
CORBA_ORB_var orb = ... // Get a reference to the ORB somehow
CORBA_BOA_var boa = ... // Get a reference to the BOA somehow
orb -> conc_model(CORBA_ORB::ConcModelThreaded)
boa -> conc_model(CORBA_BOA::ConcModelThreadPerRequest)
```

Other possible parameters for ORB::conc_model are:

```
ConcModelBlocking
ConcModelReactive
ConcModelThreaded
```

And for BOA::conc_model:

ConcModelBlocking ConcModelReactive ConcModelThreaded ConcModelThreadPerClient ConcModelThreadPerRequest ConcModelThreadPool

In Java, the example looks like this:

```
// Java
org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
org.omg.CORBA.BOA boa = ... // Get a reference to the BOA somehow
((com.ooc.CORBA.ORB)orb).conc_model(
    com.ooc.CORBA.ORB.ConcModel.ConcModelThreaded)
((com.ooc.CORBA.BOA)boa).conc_model(
    com.ooc.CORBA.BOA)boa).conc_model(
```

The casts to com.ooc.CORBA.ORB and com.ooc.CORBA.BOA are necessary because the conc_model operations are ORBACUS-specific and are not available in the classes org.omg.CORBA.ORB and org.omg.CORBA.BOA, respectively.

In case the thread pool concurrency model has been selected, it's also necessary to specify the number of threads in the thread pool. This can be done with the operation BOA::conc_model_thread_pool:

```
// C++
CORBA_BOA_var boa = ... // Get a reference to the BOA somehow
boa -> conc_model_thread_pool(10);
```

This allocates 10 threads for the thread pool. Here is the same example in Java:

```
// Java
org.omg.CORBA.BOA boa = ... // Get a reference to the BOA somehow
((com.ooc.CORBA.BOA)boa).conc_model_thread_pool(10);
```

Concurrency Models

CHAPTER 10 The Reactor

10.1 What is a Reactor?

In "reactive" mode (see "Reactive Clients and Servers" on page 123), ORBACUS uses a so-called "Reactor" for event dispatching [6]. Simply speaking, the Reactor is an instance in ORBACUS (a singleton) where special objects — so-called event handlers — can register if they are interested in specific events. These events can be network events, such as an event signaling that data are ready to be read from a network connection.

Again, this chapter only applies to ORBACUS when used with reactive concurrency models. If you use ORBACUS with any other concurrency model, for example "blocking" or any of the multi-threaded models, the following examples are not applicable. Also, since ORBACUS for Java currently doesn't support the reactive model at all, the following only applies to ORBACUS for C++.

10.2 Available Reactors

Currently there are three Reactors supported by ORBACUS:

- The standard "select" Reactor which relies on the Berkeley Sockets select function.
- A special Reactor for use with the X11 Window System. This Reactor handles X11 events (which for example can trigger X11 callbacks) and CORBA network events simultaneously.

 A special Reactor for use with Microsoft Windows 95 or Windows NT. This Reactor handles Windows messages and CORBA network events simultaneously.

The "default" Reactor is the "select" Reactor. If one of the other Reactors is to be used, it must be initialized explicitly.

10.2.1 The X11 Reactor

An application that wants to use the X11 Reactor simply has to call the function OBX11Init *before* the ORB is initialized with CORBA_ORB_init. For example:

```
1 #include <X11/Intrinsic.h>
 2
 3 #include <OB/CORBA.h>
 4 #include <OB/X11.h>
 5
 6 int
 7 main(int argc, char* argv[], char*[])
 8 {
 9
      XtAppContext appContext;
      Widget topLevel = XtAppInitialize(&appContext, "MyApplication",
10
                                        0, 0, &argc, argv, 0, 0, 0);
11
12
13
      OBX11Init(appContext);
14
      CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
15
      CORBA_BOA_var boa = orb -> BOA_init(argc, argv);
16
17
18
      // More application code ...
19
20
      boa -> impl_is_ready();
21
22
      return 0;
23
   }
```

1-4 Include header files.

6-7 Define the main function.

9–11 Initialize the X11 application.

- 13 Use the X11 application context to initialize the X11 Reactor.
- 15-16 Initialize ORB and BOA as usual.

20 Enter the CORBA event loop. This loop will also dispatch X11 events. Alternatively, the standard X11 event loop may be called, which will also dispatch CORBA events.

10.2.2 The Windows Reactor

For the Windows Reactor, the function OBWindowsInit must be called, *before* the ORB is initialized. For example:

```
1 #include <Windows.h>
 2
 3 #include <OB/CORBA.h>
 4 #include <OB/Windows.h>
 5
 6 int WINAPI
 7
   WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
 8
           LPSTR lpszArgs, int nWinMode)
 9
   {
10
      OBWindowsInit(hInstance);
11
      int dummy = 0;
12
13
      CORBA_ORB_var orb = CORBA_ORB_init(dummy, 0);
14
      CORBA_BOA_var boa = orb -> BOA_init(dummy, 0);
15
16
      // More application code ...
17
18
      boa -> impl_is_ready();
19
20
      return 0;
21
```

- 1-4 Include header files.
- 6-8 Define the WinMain function.
- 10 Use the Windows application instance to initialize the Windows Reactor.
- 12-14 Initialize ORB and BOA as usual.
 - 18 Enter the CORBA event loop, which now also dispatches Windows events. The standard Windows event loop may also be called, which will then also dispatch CORBA events.

10.3 Writing a Custom Event Handler

ORBACUS in reactive mode includes support for customized event handlers. This means that while your application is running, it can react to events like keyboard events. In order

to implement your own ORBACUS event handler, you must derive a class from OBEventHandler and overload the handleEvent member function. The constructor of the derived class must ensure that objects of this class are registered with the Reactor. This is an example for an event handler that listens to keyboard events:

```
1 #include <OB/Reactor.h>
 2
 3 class MyEventHandler : public OBEventHandler
 4 {
 5 public:
 6
 7
      MyEventHandler();
 8
      virtual ~MyEventHandler();
 9
10
      virtual void handleEvent(CORBA_ULong);
11 };
12
13 MyEventHandler::MyEventHandler()
14 {
15
      OBReactor* Reactor = OBReactor::instance();
      Reactor -> registerHandler(this, OBEventRead, 0);
16
17 }
18
19 MyEventHandler::~MyEventHandler()
20 {
      OBReactor* Reactor = OBReactor::instance();
21
22
      Reactor -> unregisterHandler(this);
23 }
24
25 void
26 MyEventHandler::handleEvent(OBMask mask)
27 {
28
      assert(mask == OBEventRead);
29
30
      char c;
      cin.read(&c, 1);
31
32
33
      // Handle character input here ...
34
   }
```

¹ The header file for the reactor is included. This file also contains the definition of OBEventHandler.

- 3-11 An event handler MyEventHandler is defined, which has a constructor, a destructor, and a handleEvent operation.
- 13-17 The constructor registers the event handler with the Reactor singleton. Only "read" events are requested.
- 19-23 The destructor unregisters the event handler with the Reactor singleton.
- 25-34 The ORBACUS Reactor calls the handleEvent function each time a read event from standard input is pending.

10.4 Using Timers

Often an application may wish to perform tasks on a regular timed basis. The reactor provides an API for the integration of timed tasks into an ORBACUS application.

In order to have a timed task a class must be created that inherits from the class OBTimer. The class OBTimer provides a pure-virtual method notify that is called when the timer expires.

For the timer to be notified it must first be enabled. To do this the activate method is called. This method takes two parameters. A timeval, and a boolean flag. The boolean flag indicates whether the timeval is a relative or absolute - true indicates a relative value. The timeval contains two members, tv_sec and tv_usec. Note that the timer is only as accurate as the underlying operating system, so microsecond accuracy is not necessarily offered.

The activate method causes the timer to be notified once. If a recurring timer is desired the activate method should be invoked before the notify method is completed.

The stop method is provided to cancel the next notification.

```
#include <OB/Timer.h>
 1
 2
  class CustomTimer : public OBTimer
 3
   {
 4
      void _register()
 5
       {
 6
          struct timeval tv;
 7
          tv.tv\_sec = 10;
 8
          tv.tv\_usec = 0;
 9
          activate(tv, true);
       }
10
11
12 public:
13
```

```
14
       CustomTimer()
15
       {
16
           _register();
17
       }
18
       virtual void notify()
19
20
       {
           cout << "Notify called" << endl;</pre>
21
           _register();
22
23
       }
24
   };
```

1 The file OB/Timer.h must be included to use the timer classes.

2 A class CustomTimer is defined that inherits from OBTimer.

- 4-10 The private method _register causes the timer to be notified every 10 seconds.
- 14-17 The constructor calls the _register method.
- 19-23 The notify method is called every 10 seconds. A string is displayed, and the timer is reregistered.

CHAPTER 11

The Open Communications Interface

11.1 What is the Open Communications Interface?

The Open Communications Interface (OCI) defines common interfaces for pluggable protocols. It supports connection-oriented, reliable "byte-stream" protocols. That is, protocols which allow the transmission of a continuous stream of bytes (octets) from the sender to the receiver.

TCP/IP is one possible candidate for an OCI plug-in. Since ORBACUS uses GIOP, such a plug-in then implements the IIOP protocol. Other candidates are SCCP (Signaling Connection Control Part, part of SS.7) or SAAL (Signaling ATM Adaptation Layer).

Non-reliable or non-connection-oriented protocols can also be used if the protocol plug-in itself takes care of reliability and connection management. For example, UDP/IP can be used if the protocol plug-in provides for packet ordering and packet repetition in case of a packet loss.

11.2 Interface Summary

11.2.1 Buffer

An interface for a buffer. A buffer can be viewed as an object holding an array of octets and a position counter, which determines how many octets have already been sent or received.

11.2.2 Transport

The Transport interface allows the sending and receiving of octet streams in the form of Buffer objects. There are blocking and non-blocking send/receive operations available, as well as operations that handle time-outs and detection of connection loss.

11.2.3 Acceptor and Connector

Acceptors and Connectors are Factories [10] for Transport objects. A Connector is used to connect clients to servers. An Acceptor is used by a server to accept client connection requests.

Acceptors and Connectors also provide operations to manage protocol-specific IOR profiles. This includes operations for comparing profiles, adding profiles to IORs or extracting object keys from profiles.

11.2.4 Connector Factory

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

11.2.5 The Registries

The ORB provides a Connector Factory Registry and the Object Adapter provides an Acceptor Registry. These registries allow the plugging-in of new protocols. Transport, Connector, Connector Factory and Acceptor must be written by the plug-in implementors. The Connector Factory must then be registered with the ORB's Connector Factory Registry and the Acceptor must be registered with the Object Adapter's Acceptor Registry.

11.2.6 The Info Objects

Info objects provide information on Transports, Acceptors and Connectors. A Transport Info provides information on a Transport, an Acceptor Info on an Acceptor and a Connector Info on a Connector. To get information for a concrete protocol, these info objects must be narrow'd to an info object for this protocol, for example, in the case of an IIOP plug-in, a OCI::TransportInfo must be narrow'd to OCI::IIOP::TransportInfo.

11.2.7 Class Diagram

Figure 11.1 shows the classes and interfaces of the OCI (except for the Buffer and Info



Figure 11.1: OCI Class Diagram

interfaces). ORBACUS provides abstract base classes for the interfaces Connector Factory, Connector, Transport and Acceptor. The protocol plug-in must inherit from these classes in order to provide concrete implementations for a specific protocol. ORBACUS also provides concrete classes for the interfaces Buffer, Connector Factory Registry and Acceptor Registry. Instances of Connector Factory Registry and Acceptor Registry are provided by the ORB and Object Adapter, respectively. Concrete implementations of the Connector Factory must be registered with the ORB's Connector Factory Registry, and concrete implementations of the Acceptor must be registered with the Acceptor Registry.

11.3 OCI Reference

This chapter does not contain a complete reference of the OCI. It only explains OCI basics and, in the remainder of this chapter, how it is used from the application programmer's

point of view for the most common tasks. For more information on how to use the OCI to write your own protocol plug-ins, and for a complete reference, please refer to Appendix B.

11.4 OCI for the Application Programmer

The following information only applies to the standard ORBACUS IIOP plug-in. For other plug-ins, like the ORBACUS SSL plug-in, please refer to the plug-in's documentation.

11.4.1 A "Converter" Class for Java

As you will see in the following examples, the OCI info objects return port numbers as IDL unsigned short values and IP addresses as an array of 4 IDL unsigned octet values. This works fine for C++, but in Java this causes a problem, because there are no unsigned types in Java. The Java mapping simply maps unsigned types to signed types. Consider for example the IP address 126.127.128.129. In Java, the OCI will return this as 126.127.-128.-127, because 128 and 129, if bit-wise mapped to the Java byte type, are -128 and -127.

To avoid this problem, we will use a helper class which converts port numbers and IP addresses to Java int types. This helper class looks as follows:

```
// Java
 1
 2
 3 final class Converter
 4
   {
 5
      static int port(short s)
       {
 6
 7
          if(s < 0)
 8
              return 0xffff + (int)s + 1;
 9
          else
10
              return (int)s;
       }
11
12
13
       static int[] addr(byte[] bArray)
       {
14
          int[] iArray = new int[4];
15
          for(int i = 0; i < 4; i++)
16
              if(bArray[i] < 0)
17
                  iArray[i] = 0xff + (int)bArray[i] + 1;
18
19
              else
20
                  iArray[i] = (int)bArray[i];
21
```
```
22 return iArray;
23 }
24 };
```

5-11 Converts short port numbers to int.

13-23 Converts byte[] IP addresses to int[].

The converter class is used throughout the examples in the sections below.

11.4.2 Getting Hostnames and Port Numbers

The following code fragments show how it is possible to find out on what hostnames and port numbers a server is listening. First the C++ version:

```
1 // C++
 2
 3 OCI_AccRegistry_var registry = boa -> get_acc_registry();
 4 OCI_AcceptorSeq_var acceptors = registry -> get_acceptors();
 5
 6 for(CORBA_ULong i = 0 ; i < acceptors -> length() ; i++)
 7 {
      OCI_AcceptorInfo_var info = acceptors[i] -> get_info();
 8
      OCI_IIOP_AcceptorInfo_var iiopInfo =
 9
          OCI_IIOP_AcceptorInfo::_narrow(info);
10
11
12
      if(!CORBA_is_nil(iiopInfo))
13
      {
14
          CORBA_String_var host = iiopInfo -> host();
          CORBA_UShort port = iiopInfo -> port();
15
16
17
          cout << "host: " << host << endl;</pre>
18
          cout << "port: " << port << endl;</pre>
19
      }
20
```

3 The Acceptor Registry is requested from the object adapter.

4 From the Acceptor Registry, the list of registered acceptors is requested.

6 The for loop iterates over all acceptors.

s-10 The info object for the acceptor is requested and narrowed to an IIOP acceptor info object.

12 The if block is only entered in case the info object really belongs to an IIOP plug-in.

14-18 The hostname and port number are requested from the IIOP acceptor info object and printed on standard output.

The Java version is basically equivalent to the C++ code and looks as follows:

```
1 // Java
 2
 3 com.ooc.OCI.AccRegistry registry =
       ((com.ooc.CORBA.BOA)boa).get_acc_registry();
 4
 5 com.ooc.OCI.Acceptor[] acceptors = registry.get_acceptors();
 6
 7 for(int i = 0 ; i < acceptors.length ; i++)</pre>
 8 {
 9
      com.ooc.OCI.AcceptorInfo info = acceptors[i].get_info();
      com.ooc.OCI.IIOP.AcceptorInfo iiopInfo =
10
11
          com.ooc.OCI.IIOP.AcceptorInfoHelper.narrow(info);
12
13
      if(iiopInfo != null)
14
      {
          String host = iiopInfo.host();
15
16
          short port = Converter.port(iiopInfo.port());
17
          System.out.println("host: " + host);
18
19
          System.out.println("port: " + port);
20
       }
21
```

³ The acceptor registry is requested from the BOA. Since the standard BOA org.omg.CORBA.BOA does not provide a method for this, there must be a cast to com.ooc.CORBA.BOA.

- 5-15 This is equivalent to the C++ version.
 - 16 The converter class is used to get a port number in int format.
- 18-19 Like in the C++ version, the hostname and port number are printed on standard output.

11.4.3 Finding out a Client's IP Address

To find out the IP address of a client within a server method, the following code can be used in a servant class method implementation:

```
1 // C++
```

```
2
```

3 CORBA_Object_var baseCurrent =

```
4
      orb -> resolve_initial_references("OCICurrent");
 5 OCI_Current_var current = OCI_Current::_narrow(baseCurrent);
 6
 7 OCI_TransportInfo_var info = current -> get_oci_transport_info();
  OCI_IIOP_TransportInfo_var iiopInfo =
 8
 9
      OCI_IIOP_TransportInfo::_narrow(info);
10
11 if(!CORBA_is_nil(iiopInfo))
12 {
13
      OCI_IIOP_InetAddr remoteAddr = iiopInfo -> remote_addr();
      CORBA_UShort remotePort = iiopInfo -> remote_port();
14
15
      cout << "Call from: "
16
           << remoteAddr[0] << '.' << remoteAddr[1] << '.'
17
18
           << remoteAddr[2] << '.' << remoteAddr[3]
19
           << ":" << remotePort << endl;
20
```

- 3-5 The OCI current object is requested and narrow'd to the correct OCI::Current type.
- 7-9 The info object for the transport is requested and narrow'd to an IIOP transport info object.
- 11 The remainder of the example code is only executed if this was really an IIOP transport info object.
- 13-19 The address and the port of the client calling this operation are obtained and printed on standard output.

The Java version looks as follows:

```
1 org.omg.CORBA.Object baseCurrent =
      orb.resolve_initial_references("OCICurrent");
 2
 3 com.ooc.OCI.Current current =
      com.ooc.OCI.CurrentHelper.narrow(baseCurrent);
 4
 5
 6 com.ooc.OCI.TransportInfo info = current.get_oci_transport_info();
   com.ooc.OCI.IIOP.TransportInfo iiopInfo =
 7
 8
      com.ooc.OCI.IIOP.TransportInfoHelper.narrow(baseInfo);
 9
10 if(iiopInfo != null)
11 {
12
      int[] remoteAddr = Converter.addr(iiopInfo.remote_addr());
13
      int remotePort = Converter.port(iiopInfo.remote_port());
14
15
      System.out.println("Call from: " +
```

1-10 This code is equivalent to the C++ version.

12-13 Again, the port number must be converted from short to int.

15-19 This is also equivalent to the C++ version.

11.4.4 Finding out a Server's IP Address

To find out the server's IP address and port that an object will attempt to connect to, the following code can be used:

```
1 // C++
 2
 3 CORBA_Object_var obj = ... // Get an object reference somehow
 4
 5 OCI_ConnectorInfo_var info = obj -> get_oci_connector_info();
 6 OCI_IIOP_ConnectorInfo_var iiopInfo =
 7
      OCI_IIOP_ConnectorInfo::_narrow(info);
 8
 9 if(!CORBA_is_nil(iiopInfo))
10 {
11
      OCI_IIOP_InetAddr_var remoteAddr = iiopInfo -> remoteAddr();
12
      CORBA_UShort remotePort = iiopInfo -> remote_port();
13
      cout << "Will connect to: "
14
15
           << remoteAddr[0] << '.' << remoteAddr[2] << '.'
           << remoteAddr[2] << '.' << remoteAddr[3]
16
17
           << ":" << remotePort << endl;
18
   }
```

5-7 Get the OCI connector info and narrow to an IIOP connector info

9 The if block is only executed if this really was an IIOP connector info.

11-17 The address and port are obtained and displayed on standard output.

The Java version looks as follows:

1 // Java

2

```
3 org.omg.CORBA.Object obj = ... // Get an object reference somehow
 4
 5 org.omg.CORBA.portable.ObjectImpl objImpl =
      (org.omg.CORBA.portable.ObjectImpl)obj;
 6
 7 com.ooc.CORBA.Delegate objDelegate =
      (com.ooc.CORBA.Delegate)objImpl._get_delegate();
 8
 9
10 com.ooc.OCI.ConnectorInfo info =
      objDelegate.get_oci_connector_info();
11
12 com.ooc.OCI.IIOP.ConnectorInfo iiopInfo =
      com.ooc.OCI.IIOP.ConnectorInfoHelper.narrow(info);
13
14
15 if(iiopInfo != null)
16 {
17
      int[] remoteAddr = Converter.addr(iiopInfo.remote_addr());
18
      int remotePort = Converter.port(iiopInfo.remote_port());
19
      System.out.println("Will connect to: " +
20
                      remoteAddr[0] + "." +
21
                      remoteAddr[1] + "." +
22
23
                      remoteAddr[2] + "." +
24
                      remoteAddr[3] + ":" + remotePort);
25
   }
```

5-8 We need to retrieve the ORBACUS-specific Delegate object so that we can get the connector info.

10-13 Get the OCI connector info and narrow to an IIOP connector info.

15 The if block is only entered if this really was an IIOP connector info.

17-24 The address and port are obtained and displayed on standard output.

The Open Communications Interface

CHAPTER 12 Using Policies

12.1 Overview

The ORB and its services may allow the application developer to configure the semantics of its operations. This configuration is accomplished in a structured manner through interfaces derived from the interface CORBA::Policy. For instance, the ORBACUS SSL plug-in [13] allows the configuration of the cipher suites used for peer communications through the interface SSL::CipherSuitePolicy.

The configuration of these policy objects is accomplished at three levels:

- **ORB Level**: These policies override the system defaults. The ORB has an initial reference ORBPolicyManager. A PolicyManager has a set of operations through which the current set of overriding policies can be obtained, and new policies can be applied.
- **Thread Level**: A standard PolicyCurrent is defined with operations that allow the querying and retrieval of policies that affect the current thread. These policies override the policies set at the ORB level.
- **Object Level**: The object interface contains operations to retrieve and set policies for itself. Policies applied at the object level override those applied at the thread level, or the ORB level.

At present ORBACUS does not support thread level policies.

For more information on Policies, the PolicyManager interface and the CORBA::Object policy operations see [11] and [12].

12.2 Supported Policies

The following is a brief description of the policies that are currently supported. For a detailed description, please refer to Appendix A.

SSL::ConnectPolicy

This policy determines whether the ORB is permitted to establish an insecure communications channel between peers. The default for this policy is true if the SSL plug-in is not installed. If the SSL plug-in is installed, the default is false. For more information on this policy, see [13].

OB::ConnectionReusePolicy

This policy determines whether the ORB is permitted to reuse a communications channel between peers. If this policy is false then each object will have a new communications channel to its peer. The default for this policy is true.

OB::ProtocolPolicy

This policy is used to force the selection of a particular protocol. If this policy is set, then the protocol with the identified tag will be used, if possible. If it is not possible to use this protocol, a CORBA::NO_RESOURCES exception will be raised.

OB::ReconnectPolicy

If an object possesses this policy and the value flag of this policy is true, then upon a communications failure a reconnection will automatically be attempted. If this reconnection attempt fails a CORBA::COMM_FAILURE exception is raised.

OB::TimeoutPolicy

If an object has this policy and no response is available for a request after value milliseconds, a CORBA: :NO_RESPONSE exception is raised.

12.3 Examples

The following examples demonstrate how to set OB::ConnectionReusePolicy at both the ORB level and the object level in C++ and Java. Setting a policy at the ORB level means that the ORB will honor this policy for all newly created objects. Existing objects

maintain their current set of policies. Setting a policy at the object level overrides any ORB level policies applied to that object.

Setting the connection reuse policy to false at the ORB level means that the ORB will create a new connection from the client to the server for each new proxy object instead of reusing existing ones. Setting the connection reuse policy to false at the object level means that the client does not reuse connections to the server only for a particular proxy object.

If the connection reuse policy is set to true at some later point, communications channels that were previously created with a connection reuse policy set to false will not be reused. That is, the connection reuse policy is sticky, in the sense that the reuse policy that was in effect at the time that a communications channel is created stays with it. Setting the reuse policy at the object level means that for a client the ORB will not reuse the communications channel that is associated with the proxy object.

12.3.1 Connection Reuse Policy at ORB Level

Our first example shows how the connection reuse policy can be set at the ORB level. First in C++:

```
1 // C++
2 CORBA_Any boolAny;
3 boolAny <<= CORBA_Any::from_boolean(CORBA_FALSE);
4 CORBA_PolicyList policies;
5 policies.length(1);
6 policies[0] = orb -> create_policy(OB_CONNECTION_REUSE, boolAny);
7 CORBA_Object_var pmObj =
8 orb -> resolve_initial_references("ORBPolicyManager");
9 CORBA_PolicyManager_var pm = CORBA_PolicyManager::_narrow(pmObj);
10 pm -> add_policy_overrides(policies);
```

2-3 Create an any and insert the value CORBA_FALSE.

4-5 Create a sequence containing one policy object.

- *ϵ* Ask the ORB to create a connection reuse policy. Pass the any that contains the value for this policy.
- 7-9 Obtain the ORB level policy manager object.
- 10 Add the policies to the ORB level policy manager.

And here is the same example in Java:

```
1 // Java
2 org.omg.CORBA.Any boolAny = orb.create_any();
3 boolAny.insert_boolean(false);
4 org.omg.CORBA.Policy[] policies = new org.omg.CORBA.Policy[1];
5 policies[0] =
6 orb.create_policy(com.ooc.OB.CONNECTION_REUSE.value, boolAny);
7 org.omg.CORBA.PolicyManager pm =
8 org.omg.CORBA.PolicyManagerHelper.narrow(
9 orb.resolve_initial_references("ORBPolicyManager"));
10 pm.add_policy_overrides(policies);
```

1-10 This is equivalent to the C++ version.

12.3.2 Connection Reuse Policy at Object Level

And now the same example, but at the object level. C++ first:

```
1 // C++
2 CORBA_Any boolAny;
3 boolAny <<= CORBA_Any::from_boolean(CORBA_FALSE);
4 CORBA_PolicyList policies(1);
5 policies.length(1);
6 policies[0] = orb -> create_policy(OB_CONNECTION_REUSE, boolAny);
7 CORBA_Object_var newObj =
8 _____obj -> _set_policy_overrides(policies, CORBA_ADD_OVERRIDES);
```

- 2-6 This is the same as in the example for the ORB level.
- 7-8 Set these policies on the object by using the set_policy_overrides method. This method returns a new object that has the set of policies applied.

And here is the same example in Java:

```
1 // Java
2 org.omg.CORBA.Any boolAny = orb.create_any();
3 boolAny.insert_boolean(false);
4 org.omg.CORBA.Policy[] policies = new org.omg.CORBA.Policy[1];
5 policies[0] =
6 orb.create_policy(com.ooc.OB.CONNECTION_REUSE.value, boolAny);
7 org.omg.CORBA.Object newObj =
8 obj._set_policy_overrides(policies,
9 org.omg.CORBA.SetOverrideType.ADD_OVERRIDE);
```

1-9 This is equivalent to the C++ version.

CHAPTER 13 ORBACUS Basic Services

This chapter describes the standard services included with the ORBACUS distribution:

- The Naming Service
- The Property Service
- The Event Service

These services are implemented compliant to [4] and available in C++ and Java versions.

Other services, such as the Trading Service "ORBACUS Trader", are *not* included in the standard ORBACUS distribution. For more information on other services available from Object-Oriented Concepts, please see our Web site.

This chapter does not provide a complete description of the naming, property and event services. It only provides an overview, suitable to get you started. For more information, please refer to the service specifications.

13.1 Configuring and Using a Basic Service

This section describes the steps necessary to start a service, publish its IOR, and connect to the service from a client. We will use the Naming Service as an example, but the steps outlined below are applicable to all of the services.

13.1.1 Starting the Service

To start the C++ version of the Naming Service, type the following:

```
nameserv -i -OAport 10000 > nameserv.ref
```

The Java version can be started like this:

java com.ooc.CosNaming.Server -i -OAport 10000 > nameserv.ref

Notice that we have specified a unique port number for the service, in order to ensure that the object reference of the service remains valid across executions of the service (see "Lifetime of Object References" on page 87).

The -i argument causes the service to dump its IOR to standard output, which we have redirected to the file nameserv.ref.

13.1.2 Connecting to the Service

Chapter 6 describes different strategies for locating objects, and these strategies can also be used to locate services. For example, it's possible to read the stringified IOR from the file nameserv.ref, convert it to an object using string_to_object and then narrow this object reference to the CosNaming::NamingContext interface.

A more common way is to use resolve_initial_references as shown in "Resolving an Initial Service" on page 94. The references for the initial services can be defined using the -ORBservice option. Here's a Unix example which uses "Bourne" shell command substitution ('command') to obtain an IOR from a file:

java MyClient -ORBservice NameService 'cat nameserv.ref'

On non-Unix operating systems, however, it can be inconvenient to handle IORs on the command line, therefore it's often easier to use the -ORBconfig option:

java MyClient -ORBconfig orb.cfg

The configuration file orb.cfg could be written as follows:

```
# ORB configuration file
ooc.service.NameService=iiop://myhost:10000/DefaultNamingContext
```

Notice that we are using the ORBACUS-specific iiop:// notation for specifying the IOR of the Naming Service, but we also could have pasted the contents of nameserv.ref. See "Using the iiop:// Notation" on page 93 for more information.

The IOR contains the name of the host where the naming service was started ("myhost"), the port number that we specified when starting the service, and the name assigned to the service's primary object: DefaultNamingContext.

13.1.3 Object Names for the Basic Services

Each of the Basic Services has a named primary object, which allows you to use the iiop:// notation or the ORB operation get_inet_object to obtain a reference to the service (see "Connecting to Named Objects" on page 92). The name and interface type of each service's primary object is shown in Table 13.1.

	Object Name	Interface Type
Naming Service	DefaultNamingContext	CosNaming::NamingContext
Event Service	DefaultEventChannel	CosEventChannelAdmin:: EventChannel
Typed Event Ser- vice	DefaultTypedEventChannel	CosTypedEventChannelAdmin:: TypedEventChannel
Property Service	DefaultPropertySetDefFactory	CosPropertyService:: PropertySetDefFactory
Interface Repository ^a	DefaultRepository	CORBA::Repository

Table 13.1: Primary Object Names and Interface Types

a. The Interface Repository is not a CORBA Service and therefore not described in this chapter. However, the object name of the Interface Repository is shown here for completeness.

The examples below illustrate how to connect to the Naming Service using get_inet_object. Here's the C++ version:

```
// C++
CORBA_ORB_var orb = ... // Get a reference to the ORB somehow
CORBA_Object_var obj =
    orb -> get_inet_object("myhost", 10000, "DefaultNamingContext");
CosNaming_NamingContext_var ctx =
    CosNaming_NamingContext::_narrow(obj);
```

And in Java:

For these examples to work, the Naming Service must have been started on the host "myhost" using the port number 10000.

13.2 The Naming Service

A CORBA object is often represented by an object reference in the form of a "stringified" IOR, a lengthy string that is difficult to read and cumbersome to use. It is much more natural to think of an object in terms of its name, which is a core feature of the CORBA Naming Service. In the Naming Service, objects are registered with a unique name, which can later be used to resolve its associated object reference.

13.2.1 Properties

The ORBACUS Naming Service supports the following properties:

ooc.naming.database=FILE	Enables persistence for the server. All of the bindings cre- ated by the server will be saved to the specified file. If you are starting the server for the first time using this database, you must also use the $-s$ command-line option.
ooc.naming.timeout=MINS	Specifies the timeout in minutes after which a persistent server automatically compacts its database. The default tim- eout is five minutes.

13.2.2 Command-line Options

The ORBACUS Naming Service supports the following command-line options:

-h Display the command-line options supported by the server.

-v version	Display the version of the server.
-i ior	Print the interoperable object reference (IOR) of the server to standard output.
-s start	Use this option only when starting a persistent server using a new database.
-d FILE database FILE	Equivalent to the ooc.naming.database property.
-t MINS timeout MINS	Equivalent to the ooc.naming.timeout property.

13.2.3 Creating Bindings

Object references registered with the Naming Service are maintained in a hierarchical structure similar to a filesystem. A file in a filesystem is analogous to an object binding in the Naming Service. The equivalent for a folder in a filesystem is a naming context in Naming Service terms. The pieces of information stored in a Naming Service are called *bindings*. A binding consists of an object's name and its type, as defined in the CosNaming module:

```
// IDL
typedef string Istring;
struct NameComponent
{
   Istring id;
   Istring kind;
};
typedef sequence<NameComponent> Name;
enum BindingType
{
   nobject,
   ncontext
};
struct Binding
{
   Name binding_name;
   BindingType binding_type;
```

};

As you can see, each name consists of one or more components, like a file is fully specified by its path in a filesystem. Each name component consists of two strings, id and kind, which could be likened to a file's name and its extension. Generally, the filesystem analogy works very well when describing the Naming Service structures.

A new Naming Service entry, i.e., a binding, is created with the following operations:

```
// IDL
void bind(in Name n, in Object obj)
    raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void bind_context(in Name n, in NamingContext nc)
    raises(notFound, CannotProceed, InvalidName);
NamingContext new_context();
NamingContext bind_new_context(in Name n)
    raise(NotFound, AlreadyBound, CannotProceed, InvalidName);
```

bind registers a new object with the Naming Service, whereas a new context is registered with bind_context. For each operation, an object reference and a Name are expected as parameters. If no exception was thrown, the bind operation was successful. New naming context objects are created with new_context or bind_new_context.

Use the unbind operation to delete a particular binding:

```
// IDL
void unbind(in Name n)
    raises(NotFound, CannotProceed, InvalidName);
```

13.2.4 Name Resolution

Besides registering objects, an equally important task of the Naming Service is name resolution. A name is passed to the resolve operation and an object reference is returned if the name exists.

```
// IDL
Object resolve(in Name n)
   raises(NotFound, CannotProceed, InvalidName);
```

The resolve operation is only useful when a particular name is known in advance. Sometimes it is necessary to ask for a list of all bindings registered with a particular naming context. The list operation returns a list of bindings.

```
// IDL
typedef sequence<Binding> BindingList;
void list(in unsigned long how_many,
    out BindingList bl, out BindingIterator bi);
```

If the number of bindings is especially large, the BindingIterator interface is provided so that you don't have to query for all available bindings at once. Simply get a certain number of bindings specified with how_many, and get the rest, if any, using the BindingIterator.

```
// IDL
interface BindingIterator
{
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
        out BindingList bl);
    void destroy();
};
```

Make sure that you destroy the iterator object when it is no longer needed.

13.2.5 Persistence

The ORBACUS Naming Service can optionally be used in a persistent mode in which all bindings managed by the service are saved in a file. If you do not run the service in its persistent mode, all of the bindings will be lost when the service terminates.

It is also important to note that **when using the service in its persistent mode, you should always start the service on the same port** (see "BOA Properties" on page 47 for more information).

13.2.6 A Simple Example

ORBACUS includes simple C++ and Java examples that demonstrate how to use the CORBA Naming Service. These examples are located in the folder naming/demo. We will concentrate on the Java example, but the C++ example works similarly. The example expects a Naming Service server to be already running and that the server's initial reference can be resolved by the ORB. Because of its volume we have split the code into several parts for the discussion below.

Initialization

The first code fragment deals with initializing the ORB and the BOA.

```
// Java
 1
 2
 3 try
 4 {
      ORB orb = ORB.init(args, new java.util.Properties());
 5
 6 }
 7 catch(SystemException ex)
 8
   {
 9
      // The ORB initialization failed
10 }
11
12 org.omg.CORBA.Object obj = null;
13 try
14 {
      obj = orb.resolve_initial_references("NameService");
15
16 }
17 catch(org.omg.CORBA.ORBPackage.InvalidName ex)
18 {
19
      // There is no Naming Service available
20 }
21
22 if(obj == null)
23 {
24
      // Something is wrong with the Naming Service reference
25 }
26
27 NamingContext nc = NamingContextHelper.narrow(obj);
28
29 if (nc == null)
30 {
31
      // This is not a Naming Service reference at all
32 }
33
34 BOA boa = orb.BOA_init(args, new java.util.Properties());
```

- 3-10 Usually the application is initialized in the main method. In order to initialize the ORB, its init operation is called.
- 12-20 In the next step we try to connect to the Naming Service by supplying "NameService" to resolve_initial_references. If InvalidName is thrown, there is no Naming Service available because the ORB doesn't know anything about this service.

- 22-32 If calling resolve_initial_references was successful, the object reference is checked and narrowed in order to verify that it's a Naming Service instance. If the narrow operation returns a null reference, the object returned is not a Naming Service instance but something else. This is considered to be an error because we explicitly asked for a Naming Service instance.
 - 34 Finally the BOA is initialized.

Binding

In the next step some sample bindings are created and bound to the Naming Service.

```
1 // Java
 2
 3 Named a = new Named_impl();
 4 Named a1 = new Named_impl();
 5 Named a2 = new Named_impl();
 6 Named a3 = new Named_impl();
 7 Named b = new Named_impl();
 8 Named c = new Named_impl();
 9
10 try
11 {
12
      NameComponent[] nclName = new NameComponent[1];
13
      nclName[0] = new NameComponent();
14
      nclName[0].id = "ncl";
      nclName[0].kind = "";
15
      NamingContext nc1 = nc.bind_new_context(nc1Name);
16
17
      NameComponent[] nc2Name = new NameComponent[2];
18
      nc2Name[0] = new NameComponent();
19
20
      nc2Name[0].id = "nc1";
      nc2Name[0].kind = "";
21
      nc2Name[1] = new NameComponent();
22
23
      nc2Name[1].id = "nc2";
24
      nc2Name[1].kind = "";
25
      NamingContext nc2 = nc.bind_new_context(nc2Name);
26
27
      NameComponent[] aName = new NameComponent[1];
28
      aName[0] = new NameComponent();
      aName[0].id = "a";
29
30
      aName[0].kind = "";
31
      nc.bind(aName, a);
32
```

```
33
      NameComponent[] alName = new NameComponent[1];
      alName[0] = new NameComponent();
34
35
      alName[0].id = "al";
36
      alName[0].kind = "";
37
      nc.bind(alName, al);
38
39
      NameComponent[] a2Name = new NameComponent[1];
40
      a2Name[0] = new NameComponent();
      a2Name[0].id = "a2";
41
      a2Name[0].kind = "";
42
43
      nc.bind(a2Name, a2);
44
45
      NameComponent[] a3Name = new NameComponent[1];
46
      a3Name[0] = new NameComponent();
      a3Name[0].id = "a3";
47
      a3Name[0].kind = "";
48
49
      nc.bind(a3Name, a3);
50
51
      NameComponent[] bName = new NameComponent[2];
52
      bName[0] = new NameComponent();
53
      bName[0].id = "nc1";
54
      bName[0].kind = "";
55
      bName[1] = new NameComponent();
56
      bName[1].id = "b";
57
      bName[1].kind = "";
58
      nc.bind(bName, b);
59
60
      NameComponent[] cName = new NameComponent[3];
      cName[0] = new NameComponent();
61
62
      cName[0].id = "nc1";
      cName[0].kind = "";
63
      cName[1] = new NameComponent();
64
65
      cName[1].id = "nc2";
66
      cName[1].kind = "";
67
      cName[2] = new NameComponent();
68
      cName[2].id = "c";
      cName[2].kind = "";
69
70
      nc.bind(cName, c);
71
72
      boa.impl_is_ready(null);
73
```

3-8 Several sample objects are created that will later be bound to our Naming Service. These objects implement an interface called Named. In this example, the details of this interface are not important. Named might even be an interface without any operations defined in it.

- 10-70 Create and bind some new contexts and bind the sample objects to these contexts. Each binding name consists of several name components NameComponent that are similar to the path components of a file located somewhere in a filesystem. Objects are bound with the Naming Service's bind operation; for contexts, the corresponding operation bind_context is used. In addition to the object's IOR, both calls expect a unique bind-ing name. If a name already exists, an AlreadyBound exception is thrown. There are also other exceptions you might encounter at this stage, e.g., IllegalName if an empty string was provided as part of a NameComponent.
 - 72 Everything is prepared now, so we can listen for requests by calling impl_is_ready on the BOA.

Unbinding

Some cleanup work should be done before exiting the program. Every binding is properly unbound here.

```
1 // Java
2
3 nc.unbind(cName);
4 nc.unbind(bName);
5 nc.unbind(aName);
6 nc.unbind(nc2Name);
7 nc.unbind(nc1Name);
```

Exceptions

The final code fragment deals with exception handling.

```
// Java
 1
 2
 3 catch(NotFound ex)
 4
  {
 5
      System.err.print("Got a 'NotFound' exception (");
 6
       switch(ex.why.value())
 7
       {
          case NotFoundReason._missing_nod:
 8
          System.err.print("missing node");
 9
          break;
10
11
          case NotFoundReason._not_context:
12
13
          System.err.print("not context");
14
          break;
15
```

```
16
          case NotFoundReason._not_object:
          System.err.print("not object");
17
          break;
18
19
      }
20
21
      System.err.println(")");
      ex.printStackTrace();
22
23
      return 1;
24 }
25 catch(CannotProceed ex)
26 {
      System.err.println("Got a `CannotProceed' exception");
27
28
      ex.printStackTrace();
      return 1;
29
30 }
31 catch(InvalidName ex)
32 {
      System.err.println("Got an 'InvalidName' exception");
33
      ex.printStackTrace();
34
      return 1;
35
36 }
37 catch(AlreadyBound ex)
38 {
39
      System.err.println("Got an 'AlreadyBound' exception");
40
      ex.printStackTrace();
      return 1;
41
42
   }
```

3-42 Catch exceptions. Don't ever forget to do this. It can be useful to call printStackTrace on the exception object in order to get detailed information about the program flow causing the exception.

Now you should have a look at the complete example as it is provided in the folder demo/ naming as a part of the ORBACUS distribution.

13.2.7 Windows NT Native Service

The C++ version of the ORBACUS Naming Service is also available as a native Windows NT service.

Command-line Options

The ORBACUS Naming Service supports the following command-line options:

-h help	Display the command-line options supported by the server.
-v version	Display the version of the server.
-i install	Install the service.
-u uninstall	Uninstall the service.
-d debug	Run the service in debug mode.

In order to use the Naming Service as a native Windows NT service, it is first necessary to add the NameService initial reference to the HKEY_LOCAL_MACHINE NT registry key (see "Using the Windows NT Registry" on page 52 for more details). If the service is to be persistent, the path to the database file must be stored in the following property:¹

HKEY_LOCAL_MACHINE\Software\OOC\Properties\ooc\naming\database

Next the service should be installed with:

```
ntnameservice -i
```

This adds the ORBacus Naming Service entry to the Services dialog in the Control Panel. To start the naming service, select the ORBacus Naming Service entry, and press start. If the service is to be started automatically when the machine is booted, select the ORBacus Naming Service entry, then click Startup. Next select Startup Type - Automatic, and press OK.

If you want to remove the service, run:

```
ntnameservice -u
```

If the executable for the Naming Service is moved it must be uninstalled and re-installed.

^{1.} You should be aware that services do not have access to network drives, so the path to the database must be on a local hard drive.

Any trace information provided by the service will be placed in the Windows NT Event Viewer with the title NamingService. To enable tracing information (such as connection tracing, etc.) set the following property to a REG_SZ value of at least 1:

HKEY_LOCAL_MACHINE\Software\OOC\Properties\ooc\orb\trace_level

13.3 The Property Service

The CORBA Property Service¹ is another important CORBA service. With it, you can annotate an object with extra attributes (called *properties*) that were not defined by the object's IDL interface. Properties can represent any value because they make use of the powerful CORBA Any data type.

13.3.1 Command-line Options

The ORBACUS Property Service supports the following command-line options:

-h help	Display the command-line options supported by the server.
-v version	Display the version of the server.
-i ior	Print the interoperable object reference (IOR) of the server to standard output.

13.3.2 Creating Properties

A property handled by the CORBA Property Service consists of two components, namely the property's name and its value. The name is simply a CORBA string and the associated value is represented by a CORBA Any:

```
// IDL
typedef string PropertyName;
struct Property
{
    PropertyName property_name;
    any property_value;
};
```

^{1.} Note that the Property Service has nothing to do with the properties used for configuration purposes, as described in "Properties" on page 46.

New properties are created using a factory object implementing the PropertySet interface. A new property is created using the define_property operation:

```
// IDL
void define_property(in PropertyName, in any property_value)
   raises(InvalidPropertyName, ConflictingProperty,
   UnsupportedTypeCode, UnsupportedProperty,
   ReadOnlyProperty);
```

As a property consists of a name-value pair, both the name and the value are the parameters to this operation.

13.3.3 Querying for Properties

As soon as a property is defined, the PropertySet can be queried for the property's value with the get_property_value operation:

```
// IDL
any get_property_value(in PropertyName property_name)
    raises(PropertyNotFound, InvalidPropertyName);
```

For a particular property name this call either returns the Any associated with this name or throws an exception if a property with the name does not exist.

You can not only query for a particular property value, but also for a list of all the properties defined within a PropertySet. The get_all_properties operation serves this purpose:

```
// IDL
void get_all_properties(in unsigned long how_many,
    out Properties nproperties, out PropertiesIterator rest);
```

This operation works similar to the list call offered by the Naming Service. In both cases the maximum number of items to be returned at once is specified. An iterator implementing the PropertiesIterator interface gives access to the remaining items, if any.

```
// IDL
interface PropertiesIterator
{
    void reset();
    boolean next_one(out Property aproperty);
    boolean next_n(in unsigned long how_many,
        out Properties nproperties);
```

```
void destroy();
};
```

If you are only interested in a list of property names you can get this list by calling get_all_property_names:

```
// IDL
void get_all_property_names(in unsigned long how_many,
    out PropertyNames property_names,
    out PropertyNamesIterator rest);
```

As with get_all_properties a list of names as well as an iterator is returned. This iterator implements the PropertyNamesIterator interface:

```
// IDL
interface PropertyNamesIterator
{
    void reset();
    boolean next_one(out PropertyName property_name);
    boolean next_n(in unsigned long how_many,
        out PropertyNames property_names);
    void destroy();
};
```

The iterators should always be destroyed when they are no longer needed.

Sometimes it is useful to know of how many properties a PropertySet consists of. This information is provided by get_number_of_properties:

```
// IDL
unsigned long get_number_of_properties();
```

Note that you have to be careful if you intend to use the return value of get_number_of_properties as the input value for the how_many parameter of get_all_properties in order to get a complete property list. You always have to check the PropertiesIterator for properties that were not returned as part of the Properties sequence returned by get_all_properties, otherwise you might miss a property that was defined by another process between your calls to get_number_of_properties and get_all_properties.

13.3.4 Deleting Properties

If a property has become obsolete it can be deleted from the PropertySet with delete_property:

```
// IDL
void delete_property(in PropertyName property_name)
    raises(PropertyNotFound, InvalidProperty, FixedProperty);
```

As you might have guessed by this operation's signature, there are properties that cannot be deleted at all. This kind of property is called a FixedProperty. The Property Service defines several other special property types, such as read-only properties. Please refer to the OMG's Property Service [4] specification for details.

13.3.5 A Simple Example

The Property Service test suite, which is part of the ORBACUS distribution, provides a good example of how to create properties and query for their values. The code below is based on excerpts of this test suite, which is located in the directory property/test. We will concentrate on an example in Java here. As with the previous examples, the Java code is very similar to what is necessary in C++. The example demonstrates how to create properties and how to get a list of all the properties defined within a PropertySet.

```
1
   // Java
 2
 3 org.omg.CORBA.Object obj = null;
 4
 5 try
 6
   {
 7
      obj = orb.resolve_initial_references("PropertyService");
 8
   }
 9
   catch(org.omg.CORBA.ORBPackage.InvalidName ex)
10 {
11
      // An error occurred, Property Service is not available
12 }
13
14 if(obj == null)
15 {
      // The object reference is invalid
16
17 }
18
19 PropertySetFactory factory = PropertySetFactoryHelper.narrow(obj);
20 if(factory == null)
21 {
```

```
22
      // This object does not implement the Property Service
23 }
24
25 PropertySet set = factory.create_propertyset();
26
27 Any anyLong = orb.create_any();
28 Any AnyInt = orb.create_any();
29 Any anyShort = orb.create_any();
30 anyLong.insert_long(12345L);
31 anyInt.insert_int(6789);
32 anyShort.insert_short(0);
33
34 try
35 {
36
      set.define_property("LongProperty", anyLong);
      set.define_property("IntProperty", anyInt);
37
      set.define_property("ShortProperty", anyShort);
38
39 }
40 catch(ReadOnlyProperty ex)
41 {
42
      // An error occurred
43 }
44 catch(ConflictingProperty ex)
45 {
46
      // An error occurred
47 }
48 catch(UnsupportedProperty ex)
49 {
50 // An error occurred
51 }
52 catch(UnsupportedTypeCode ex)
53 {
54
     // An error occurred
55 }
56 catch(InvalidPropertyName ex)
57 {
58
      // An error occurred
59 }
60
61 PropertiesHolder ph = new PropertiesHolder();
62 PropertiesIteratorHolder ih = new PropertiesIteratorHolder();
63 set.get_all_properties(0, ph, ih);
64
65 PropertyHolder h = new PropertyHolder();
66 while(ih.value.next_one(h))
```

```
67 {
68 // The next property is now stored in h.value
69 }
70
71 ih.value.destroy();
```

- 5-23 Get a Property Service reference and check for errors.
 - 25 The PropertySetFactory object is used to create a PropertySet instance.
- 27-32 Each property consists of a name and a value in the form of a CORBA Any.
- 34-59 Three properties are defined. The first has the name "LongProperty" and stores a long value. The second one is called "IntProperty" and stores an int. The remaining property represents a short value. If for some reason a property cannot be created, an exception is thrown.
- 61-69 Now we try to get a list of all the properties that were previously defined. With get_all_properties the PropertySet returns its properties. As we have set the how_many parameter to 0, we have to use the PropertiesIterator for each item. Usually you provide a positive integer for how_many.
 - 71 The iterator has fulfilled its duty and can now be destroyed.

13.4 The Event Service

Sometimes applications have to exchange information without explicitly knowing about each other. Often a server isn't even aware of the nature and number of clients that are interested in the data the server has to offer. A special mechanism is required that provides decoupled data-transfer between servers and clients. This issue is addressed by the CORBA Event Service [4].

13.4.1 Properties

The ORBACUS C++ Event Service supports the following properties:

ooc.event.retry_timeout=msec	Specifies the initial amount of time in milliseconds that the service will wait between successive retries.The default value is 1000.
ooc.event.retry_multiplier=n	A double that defines the factor by which the retry_timeout property should be multiplied for each successive retry.

ooc.event.pull_interval=msec	This specifies the number of milliseconds between successive calls to pull on PullSupplier. Default value is 0.
ooc.event.max_events	The maximum number of events in each event queue. If this limit is reached and another event is received, the oldest event is discarded. The default value is 10.
ooc.event.max_retries	The maximum number of times to retry before giving up and disconnecting the proxy. The default value is 10.

13.4.2 Command-line Options

The ORBACUS Event Service supports the following command-line options:

-h help	Display the command-line options supported by the server.
-v version	Display the version of the server.
-i	Print the interoperable object reference (IOR) of the server to standard
ior	output.

The C++ implementation of the Event Service supports both typed and untyped event channels, therefore the following additional command-line options are provided to allow you to select which kind of channel the server should create:

```
    -t
    -typed-service
    -u
    -untyped-service
    Run an untyped event service. This is the default behavior.
```

13.4.3 Diagnostics

The C++ Event Service uses the ORBACUS OBMessageViewer class to generate diagnostic messages. You can activate these messages by setting the ooc.orb.trace_level property to 2. Note that you must have compiled the ORBACUS distribution with the OB_TRACE preprocessor macro defined in order to enable diagnostic messages. This macro is defined by default.

13.4.4 NT Native Service

The C++ version of the ORBACUS Event Service is also available as a native Windows NT service.

Command-line Options

The ORBACUS Event Service supports the following command-line options:

-h help	Display the command-line options supported by the server.
-v version	Display the version of the server.
-i install	Install the service.
-u uninstall	Uninstall the service.
-d debug	Run the service in debug mode.

In order to use the Event Service as a native Windows NT service, it is first necessary to add the EventService initial reference to the HKEY_LOCAL_MACHINE NT registry key (see "Using the Windows NT Registry" on page 52 for more details).

Next the service should be installed with:

```
nteventservice -i
```

This adds the ORBacus Event Service entry to the Services dialog in the Control Panel. To start the event service, select the ORBacus Event Service entry, and press start. If the service is to be started automatically when the machine is booted, select the ORBacus Event Service entry, then click Startup. Next select Startup Type-Automatic, and press OK.

If you want to remove the service, run:

```
nteventservice -u
```

If the executable for the Event Service is moved it must be uninstalled and re-installed.

Any trace information provided by the service will be placed in the Windows NT Event Viewer with the title EventService. To enable tracing information (such as connection tracing, etc.) set the following property to a REG_SZ value of at least 1:

HKEY_LOCAL_MACHINE\Software\OOC\Properties\ooc\orb\trace_level

13.4.5 The Event Channel

The Event Service distributes data in the form of events. The term *event* in this context refers to a piece of information that is contributed by an event source. An event channel instance accepts this information and distributes it to a list of objects that previously have connected to the channel and are listening for events.

The Event Service specification defines two distinct kinds of event channels: untyped and typed. Whereas an untyped event channel forwards every event to each of the registered clients in the form of a CORBA Any, a typed event channel works more selectively by supporting strongly-typed events which allow for data filtering. We will only discuss the untyped event channel here. For information on typed event channels, and more details on the Event Service in general, please refer to the official Event Service specification [4].

13.4.6 Event Suppliers and Consumers

Applications participating in generating and accepting events are called *suppliers* and *consumers*, respectively. To be more precise, there are two kinds of suppliers, namely *push suppliers* and *pull suppliers*. The situation is similar with event consumers, in that there are *push consumers* and *pull consumers*.

What's the difference between pushing events and pulling events? Let's have a look at the consumer side first. There are consumers that have to be immediately informed when any new events become available on the event channel. These consumers usually act as push consumers. They implement the PushConsumer interface which ensures that the event channel actively forwards events to them using the push operation:.

```
// IDL
interface PushConsumer
{
    void push(in any data)
        raises(Disconnected);
    void disconnect_push_consumer();
};
```

The push consumer has a more or less passive role, only waiting for something to happen. This is different than pull consumers, which (optionally) implement the PullConsumer interface. A pull consumer has a more active role and (usually periodically) polls the event channel for new events. As these events may occur more frequently than they are polled for by the pull consumer, some events might get lost. The buffering policy implemented by the event channel determines whether events are buffered and what happens in case of an event queue overflow. A client is typically implemented as a pull consumer when it is not concerned about the possibility of lost events, e.g., if the client is only interested in the most recent events.

Like consumers, suppliers can also use push or pull behavior. Push suppliers are probably the more common type, in which the supplier directly forwards data to the event channel and thus plays the active role in the link to the channel. Pull suppliers, on the other hand, are polled by the event channel and supply an event in response, if a new event is available. Polling is done by the try_pull operation if it is to be non-blocking or by the blocking pull call:

```
// IDL
interface PullSupplier
{
    any pull()
    raises(Disconnected);
    any try_pull(out boolean has_event)
    raises(Disconnected);
    void disconnect_pull_supplier();
};
```

13.4.7 Event Channel Policies

The untyped event channel implementation included in the ORBACUS distribution features a simple event queue policy. Events are buffered in the form of a FIFO stack, i.e., a certain number of events are stored and, in case of a buffer overflow, the oldest events are discarded.

13.4.8 A Simple Example

In the Event Service example that comes with ORBACUS, two supplier and two consumer clients demonstrate how to use an untyped event channel to propagate information. The pieces of information transferred by this example are strings containing the current date and time. After starting the Event Service server, you can start these clients in any order. The demo applications obtain the initial Event Service reference as already demonstrated, i.e., by calling resolve_initial_references. When started, each supplier will provide information about the current date and time and each client displays the event data in its console window.

This is the push supplier's main loop:

```
// Java
 1
 2
 3 while(consumer_ != null)
 4
   {
 5
       java.util.Date date = new java.util.Date();
       String s = "PushSupplier says: " + date.toString();
 6
 7
 8
       Any any = orb_.create_any();
       any.insert_string(s);
 9
10
11
       try
12
       {
13
          consumer_.push(any);
14
       }
15
       catch(Disconnected ex)
16
       {
17
          // Supplier was disconnected from event channel
       }
18
19
20
       Thread.yield();
21
       try
22
       {
          Thread.sleep(1000);
23
24
       }
       catch(InterruptedException ex)
25
26
       {
27
       }
28
```

5-9 The current date and time is inserted into the Any.

11-18 The event data, in this example date and time, are pushed to the event channel. From the push supplier's view the event channel is just a consumer implementing the PushConsumer interface.

20-27 After sleeping for one second, the steps above are repeated.

The example's pull supplier works similarly to the push supplier, except that the event channel explicitly polls the supplier for new events. This is done by either pull or

try_pull. The pull supplier doesn't see anything from the event channel but an object implementing the PullConsumer interface. The following example shows the basic layout of a pull supplier:

```
// Java
 1
 2
 3 public Any
 4 pull()
 5
   {
      ORB orb = ORB.init();
 6
 7
 8
       java.util.Date date = new java.util.Date();
 9
       String s = "PullSupplier says: " + date.toString();
10
      Any any = orb.create_any();
11
      any.insert_string(s);
12
13
14
      return any;
15 }
16
17 public Any
18 try_pull(BooleanHolder has_event)
19
   {
20
      has_event.value = true;
21
22
      return pull();
23
```

8-12 Date and time are inserted into the Any.

17-23 In this example new event data can be provided at any time, so try_pull always sets has_event to true in order to signal that an event is available. It then returns the actual event data.

After examining the most important aspects of the event suppliers' code, we are now going to analyze the consumers' code. The push consumer with its push operation is shown first:

```
1 // Java
2
3 public void
4 push(Any any)
5 {
6 try
7 {
```

```
8
           String s = any.extract_string();
 9
           System.out.println(s);
10
       }
11
       catch(MARSHAL ex)
12
       {
13
          // Ignore unknown event data
14
       }
15
   }
```

6-14 The push consumer's push operation is called with the event wrapped in a CORBA Any. In this code fragment it is assumed that the Any contains a string with date and time information. In case the Any contains another data type a MARSHAL exception is thrown. This exception can be ignored here because other events aren't of interest. After extracting the string it is displayed in the console window.

In contrast to the push consumer, the pull consumer has to actively query the event channel for new events. This is how the pull consumer loop looks:

```
1 // Java
 2
 3 while(supplier_ != null)
 4
   {
       Any any = null;
 5
 6
 7
       try
 8
       {
 9
          any = supplier_.pull();
10
       }
11
       catch(Disconnected ex)
12
       {
13
          // Supplier was diconnected from event channel
14
       }
15
16
       try
17
       {
          String s = any.extract_string();
18
19
          System.out.println(s);
20
       }
21
       catch(MARSHAL ex)
22
       {
23
           // Ignore unknown event data
       }
24
25
       Thread.yield();
26
```
```
27
28
       try
29
        ł
30
           Thread.sleep(1000);
31
        }
32
       catch(InterruptedException ex)
33
        {
34
        }
35
    }
```

5 A CORBA Any is prepared for later use.

- 7-14 Using pull, the consumer polls the event channel for new events. The event channel acts as a pull supplier in this case. The pull operation blocks until a new event is available.
- 16-24 The consumer expects a string wrapped in a CORBA Any. The string value is extracted and displayed. If an exception is raised the Any contained some other data type which is simply ignored.
- 26-34 After sleeping for one second the event channel is polled for the next event.

In all of these examples the event channel acts either as a consumer (if the clients are suppliers) or a supplier (if the clients are consumers) of events. Actually each client is not directly connected to the event channel but to a proxy that receives or sends events on behalf of the channel. For more information on the Event Service and for the complete definitions of the IDL interfaces, please refer to the official Event Service specification.

ORBacus Basic Services

CHAPTER 14

Exceptions and Error Messages

14.1 CORBA System Exceptions

The CORBA specification defines the standard system exceptions shown in Table 14.1.

UNKNOWN	Unknown exception type
BAD_PARAM	An invalid parameter was passed
NO_MEMORY	Failure to allocate dynamic memory
IMP_LIMIT	Implementation limit was violated
COMM_FAILURE	Communication failure
INV_OBJREF	Invalid object reference
NO_PERMISSION	The attempted operation was not permitted
INTERNAL	Internal error in ORB
MARSHAL	Error marshalling a parameter or result
INITIALIZE	Failure when initializing ORB
NO_IMPLEMENT	Operation implementation unavailable

Table 14.1: Standard CORBA System Exceptions

UNKNOWN	Unknown exception type
BAD_TYPECODE	Bad typecode
BAD_OPERATION	Invalid operation
NO_RESOURCES	Insufficient resources for a request
NO_RESPONSE	Response to a request is not yet available
PERSIST_STORE	Persistent storage failure
BAD_INV_ORDER	Routine invocation out of order
TRANSIENT	Transient failure, request can be reissued
FREE_MEM	Cannot free memory
INV_IDENT	Invalid identifier syntax
INV_FLAG	Invalid flag was specified
INTF_REPOS	Error accessing interface repository
BAD_CONTEXT	Error processing context object
OBJ_ADAPTER	Failure detected by object adapter
DATA_CONVERSION	Error in data conversion
OBJECT_NOT_EXIST	Non-existent object, references should be discarded
INV_POLICY	Invalid Policy

Table 14.1: Standard CORBA System Exceptions

OBMinorRecv	recv() failed
OBMinorSend	send() failed
OBMinorRecvZero	recv() returned zero
OBMinorSendZero	send() returned zero
OBMinorSocket	socket() failed
OBMinorSetsockopt	setsockopt() failed
OBMinorGetsockopt	getsockopt() failed
OBMinorBind	bind() failed
OBMinorListen	bind() failed
OBMinorConnect	connect() failed
OBMinorAccept	accept() failed
OBMinorSelect	select() failed
OBMinorGethostname	gethostname() failed
OBMinorGethostbyname	gethostbyname()
OBMinorWSAStartup	WSAStartup() failed
OBMinorWSACleanup	WSACleanup() failed
OBMinorNoGIOP	Not a GIOP message
OBMinorUnknownMessage	Unknown GIOP message
OBMinorWrongMessage	Wrong GIOP message
OBMinorCloseConnection	Got a close connection message
OBMinorMessageError	Got a message error message

Table 14.2 shows the minor codes for the COMM_FAILURE exception, and Table 14.3 the

Table 14.2: Minor Exception Codes for COMM_FAILURE

minor codes for the INTF_REPOS exception. No other minor codes are currently defined by ORBACUS.

OBMinorNoIntfRepos	Interface repository is not available
OBMinorIdExists	Repository id already exists
OBMinorNameExists	Name already exists
OBMinorRepositoryDestroy	destroy() invoked on Repository object
OBMinorPrimitiveDefDestroy	destroy() invoked on PrimitiveDef object
OBMinorAttrExists	Attribute is already defined in a base interface
OBMinorOperExists	Operation is already defined in a base interface
OBMinorLookupAmbiguous	Search name for lookup() is ambiguous
OBMinorAttrAmbiguous	Attribute name collisions in base interfaces
OBMinorOperAmbiguous	Operation name collisions in base interfaces

Table 14.3: Minor Exception Codes for INTF_REPOS

14.2 Non-Compliant Application Asserts

If the ORBACUS library was compiled without the preprocessor definition -DNDEBUG defined, ORBACUS tries to detect common programming mistakes that lead to non-compliant CORBA applications. If such a mistake is found an error messages like this will appear:

Non-compliant application error detected: Application used wrong memory allocation function

After detecting such an error, the ORBACUS library dumps a core (Unix only) and prints the file and line number where the error was detected. You can use the core dump in order to track down the problem with a debugger.

The following error messages can appear:

Application requested a feature that has not yet been implemented

This is not an application error. This error message appears if an application attempts to use a feature that has not yet been implemented in ORBACUS. In this case the only thing that can be done is to wait for the next ORBACUS version that has this particular feature implemented.

Application used wrong memory allocation function

If this message appears, an incorrect memory allocation function has been used. A common mistake that leads to this error is to use malloc, strdup and free (or the new and delete operator) instead of CORBA_string_alloc and CORBA_string_dup and CORBA_string_free for string memory management.

Memory that was already deallocated was deallocated again

This message indicates multiple memory deallocations. For example, if CORBA_string_free is called twice on the same string, this message will be displayed.

Object was deleted without an object reference count of zero

This message appears if an object was deleted by calling delete on its object reference. Never use the delete operator for that. Use CORBA_release instead.

Object was already deleted (object reference count was already zero)

This message appears if the number of release operations on an object reference is higher than the number of _duplicate operations.

Sequence length was greater than maximum sequence length

This message indicates that the application tried to set the length of a bounded sequence to a value greater than its maximum length.

Index for sequence operator[]() or remove() function was out of range

This message appears if the argument to the sequence member functions <code>operator[]</code> or <code>remove</code> exceeds the sequence length.

Null pointer was used to initialize T_var type

This message indicates an attempt to initialize a _var type with a null pointer.

operator->() was used on null pointer or nil object reference

This message indicates an attempt to use operator-> on an uninitialized _var type.

Application tried to dereference a null pointer

Some CORBA _var types have built-in conversion operators to a C++ reference type, i.e., some _var types for type T have a conversion operator to T&. This message appears if an application uses this conversion operator on an uninitialized _var type.

Null pointer was passed as string parameter or return value

According to the IDL-to-C++ mapping specification, no null pointers may be passed as string parameters or return values. This message appears if an application tries to do so.

Self assignment caused a dangling pointer

This message appears if the content of a _var type is assigned to itself. For example, the following code will lead to this error message:

```
1 // Somehow get a pointer to a variable struct
2 AVariableStruct_var var = ...
3 AVariableStruct* ptr = var;
4 var = ptr;
```

3,4 This will result in a dangling pointer, because var will free its own content on assignment.

Replacement of Any content by its own value caused a dangling pointer

This message appears if there is an attempt to replace the content of an Any by its own value. For example:

```
1 char* s = CORBA_string_dup("Hello, world!");
```

```
2 CORBA_Any any;
```

```
3 any <<= s;
```

```
4 any <<= s;
```

3,4 Inserting s into any twice will result in a dangling pointer, because any will free its own value (which is s) on assignment.

Invalid union discriminator type used

This message appears if the discriminator type argument to CORBA_ORB::create_union_tc denotes a type invalid for union discriminators. Valid types have a CORBA_TCKind that is one of CORBA_tk_short, CORBA_tk_ushort, CORBA_tk_long, CORBA_tk_ulong, CORBA_tk_char, CORBA_tk_boolean or CORBA_tk_enum.

Union discriminator mismatch

This message either indicates an attempt to set a union discriminator to an invalid value with the _d modifier function or the use of a wrong accessor function, i.e., an accessor function that does not correspond to the type of the union's actual value.

Uninitialized union used

If this message appears, an uninitialized union (i.e., a union that was created with the default constructor and that was not set to any legal value) was used.

Dynamic implementation object cannot be used as static implementation object

This message appears if an attempt is made to use a DSI object implementation as a regular (i.e., static) implementation object.

Exceptions and Error Messages

APPENDIX A ORBACUS Policy Reference

A.1 Module SSL

Constants

CONNECT_POLICY

const CORBA::PolicyType CONNECT_POLICY = 1;

This policy type identifies the connection policy.

Enums

ConnectPolicyType enum ConnectPolicyType { ConnectSecure,

```
ConnectSecure,
ConnectInsecure
};
```

This enumeration is used to specify whether connection attempts should be secure or insecure.

A.2 Interface SSL::ConnectPolicy

interface **ConnectPolicy**

inherits from CORBA::Policy

The connection policy. This policy is used to specify whether secure or insecure connections are used.

Attributes

value

```
readonly attribute ConnectPolicyType value;
```

If an object has a ConnectPolicy set with value set to ConnectSecure, then only secure connections will be used for that object.

A.3 Module OB

Constants

PROTOCOL_POLICY

const CORBA::PolicyType PROTOCOL_POLICY = 2;

This policy type identifies the protocol policy.

CONNECTION_REUSE_POLICY

const CORBA::PolicyType CONNECTION_REUSE_POLICY = 3;

This policy type identifies the connection reuse policy.

RECONNECT_POLICY

const CORBA::PolicyType RECONNECT_POLICY = 4;

This policy type identifies the reconnect policy.

TIMEOUT_POLICY

const CORBA::PolicyType TIMEOUT_POLICY = 5;

This policy type identifies the timeout policy.

A.4 Interface OB::ProtocolPolicy

interface **ProtocolPolicy**

inherits from CORBA::Policy

The protocol policy. This policy is used to force the selection of a specific protocol.

Attributes

value

readonly attribute IOP::ProfileId value;

If a ProtocolPolicy is set, then the protocol with the identified tag will be used, if possible. If it is not possible to use this protocol, a CORBA: :NO_RESOURCES exception will be raised.

A.5 Interface OB::ConnectionReusePolicy

interface **ConnectionReusePolicy** inherits from CORBA::Policy

The connection reuse policy. This policy determines whether connections may be reused or are private to specific objects.

Attributes

value

```
readonly attribute boolean value;
```

If an object has a ConnectionReusePolicy set with value set to FALSE, then other objects will not be permitted to also use any connection made on behalf of this object.

A.6 Interface OB::ReconnectPolicy

interface **ReconnectPolicy**

inherits from CORBA::Policy

The reconnect policy. This policy determines if an object will automatically try to reconnect to a server upon a communication failure.

Attributes

value

readonly attribute boolean value;

If an object has a ReconnectPolicy set with value set to TRUE, then upon a CORBA::COMM_FAILURE a reconnection will automatically be attempted.

A.7 Interface OB::TimeoutPolicy

interface **TimeoutPolicy** inherits from CORBA::Policy

The timeout policy. This policy can be used to specify communication timeouts.

Attributes

value

readonly attribute unsigned long value;

If an object has a TimeoutPolicy set and no response to a request is available after value milliseconds, a CORBA::NO_RESOURCE exception is raised.

ORBacus Policy Reference

APPENDIX B

Open Communications Interface Reference

B.1 Module OCI

The Open Communications Interface (OCI). The definitions in this module provide a uniform interface to network protocols. This allows for easy plug-in of new protocols or other communication mechanisms into ORBs that implement the OCI. Furthermore, protocol implementations need only to be written once and can then be reused with all OCI compliant ORBs. For more information, please see the OCI documentation.

Aliases

BufferSeq

typedef sequence<Buffer> BufferSeq;

Alias for a sequence of buffers.

IOR

typedef IOP::IOR IOR;

Alias for an IOR.

ProfileId

typedef IOP::ProfileId ProfileId;

Alias for a profile id.

ProfileIdSeq

typedef sequence<ProfileId> ProfileIdSeq;

Alias for a sequence of profile ids.

ObjectKey

typedef sequence<octet> ObjectKey;

Alias for an object key, which is a sequence of octets.

Handle

typedef long Handle;

Alias for a system-specific handle type.

CloseCBSeq

typedef sequence<CloseCB> CloseCBSeq;

Alias for a sequence of close callback objects.

ConnectCBSeq

typedef sequence<ConnectCB> ConnectCBSeq;

Alias for a sequence of connect callback objects.

AcceptorSeq

typedef sequence<Acceptor> AcceptorSeq;

Alias for a sequence of Acceptors.

AcceptCBSeq

typedef sequence<AcceptCB> AcceptCBSeq;

Alias for a sequence of accept callback objects.

ConFactorySeq

typedef sequence<ConFactory> ConFactorySeq;

Alias for a sequence of Connector factories.

B.2 Interface OCI::Buffer

interface Buffer

An interface for a buffer. A buffer can be viewed as an object holding an array of octets and a position counter, which determines how many octets have already been sent or received. The IDL interface definition for Buffer is incomplete and must be extended by the specific language mappings. For example, the C++ mapping defines the following additional functions:

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

Attributes

length

readonly attribute unsigned long length;

The buffer length.

pos

attribute unsigned long pos;

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

Operations

advance

void advance(in unsigned long delta);

Increment the position counter.

Parameters:

delta - The value to add to the position counter.

rest_length

unsigned long rest_length();

Returns the rest length of the buffer. The rest length is the length minus the position counter's

value. If the value of the position counter exceeds the buffer's length, the return value is undefined.

Returns:

The rest length.

is_full

boolean is_full();

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

Returns:

TRUE if the buffer is full, FALSE otherwise.

202

B.3 Interface OCI::Transport

interface Transport

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

See Also:

Connector Acceptor

Attributes

tag

readonly attribute ProfileId tag;

The profile id tag.

handle

readonly attribute Handle handle;

The "handle" for this Transport. The handle may *only* be used to determine whether the Transport object is ready to send or to receive data, e.g., with select() on Unix-based operating systems. All other uses (e.g., calls to read(), write(), close()) are strictly non-compliant. A handle value of -1 indicates that the protocol plug-in does not support "selectable" Transports.

fragmentation

readonly attribute unsigned long fragmentation;

The Transport's maximum packet size. The send and receive operations must not be used to send packets larger than this size within a single call. A value of 0 means that there is no upper limit for the packet size.

Operations

close

```
void close();
```

Closes the Transport. send and receive must not be called after close has been called.

shutdown

void shutdown();

Shuts down the Transport. After calling shutdown, all calls to the send and receive operations result in an appropriate CORBA::COMM_FAILURE exception being raised.

receive

Receives a buffer's contents.

Parameters:

buf - The buffer to fill.

block - If set to TRUE, the operation blocks until the buffer is full. If set to FALSE, the operation fills as much of the buffer as possible without blocking.

receive_detect

Similar to receive, but it signals a connection loss by returning FALSE instead of raising CORBA: :COMM_FAILURE.

Parameters:

buf - The buffer to fill.

block - If set to TRUE, the operation blocks until the buffer is full. If set to FALSE, the operation fills as much of the buffer as possible without blocking.

Returns:

FALSE if a connection loss is detected, TRUE otherwise.

receive_timeout

Similar to receive, but it is possible to specify a timeout. On return the caller can test whether there was a timeout by checking if the buffer has been filled completely.

Parameters:

buf - The buffer to fill. timeout - The timeout value in milliseconds. A zero timeout is equivalent to calling receive(buf, FALSE).

send

Sends a buffer's contents.

Parameters:

buf - The buffer to send.

block - If set to TRUE, the operation blocks until the buffer has completely been sent. If set to FALSE, the operation sends as much of the buffer's data as possible without blocking.

send_detect

Similar to send, but it signals a connection loss by returning FALSE instead of raising CORBA: :COMM_FAILURE.

Parameters:

buf - The buffer to fill.

block - If set to TRUE, the operation blocks until the entire buffer has been sent. If set to FALSE, the operation sends as much of the buffer's data as possible without blocking.

Returns:

FALSE if a connection loss is detected, TRUE otherwise.

send_timeout

Similar to send, but it is possible to specify a timeout. On return the caller can test whether there was a timeout by checking if the buffer has been sent completely.

Parameters:

buf - The buffer to send.

timeout - The timeout value in milliseconds. A zero timeout is equivalent to calling send(buf, FALSE).

get_info

```
TransportInfo get_info();
```

Open Communications Interface Reference

Returns the information object associated with the Transport.

Returns:

The Transport information object.

B.4 Interface OCI::TransportInfo

interface TransportInfo

Information on an OCI Transport object. Objects of this type must be narrowed to a Transport information object for a concrete protocol implementation, for example to OCI::IIOP::TransportInfo in case the plug-in implements IIOP.

See Also:

Transport

Attributes

tag

readonly attribute ProfileId tag;

The profile id tag.

connector_info

readonly attribute ConnectorInfo connector_info;

The ConnectorInfo object for the Connector that created the Transport object that this TransportInfo object belongs to. If the Transport for this TransportInfo was not created by a Connector, this attribute is set to the nil object reference.

acceptor_info

readonly attribute AcceptorInfo acceptor_info;

The AcceptorInfo object for the Acceptor that created the Transport object that this Transport-Info object belongs to. If the Transport for this TransportInfo was not created by an Acceptor, this attribute is set to the nil object reference.

Operations

add_close_cb

void add_close_cb(in CloseCB cb);

Add a callback that is called before a connection is closed. If the callback has already been registered, this method has no effect.

Parameters:

cb - The callback to add.

remove_close_cb

void remove_close_cb(in CloseCB cb);

Remove a close callback. If the callback was not registered, this method has no effect.

Parameters:

cb - The callback to remove.

B.5 Interface OCI::CloseCB

interface CloseCB

An interface for a close callback object.

See Also: TransportInfo

Operations

close_cb

```
void close_cb(in TransportInfo transport_info);
```

Called before a connection is closed.

Parameters:

transport_info - The TransportInfo for the new closeion.

B.6 Interface OCI::Connector

interface Connector

An interface for Connector objects. A Connector is used by CORBA clients to initiate a connection to a server. It also provides operations for the management of IOR profiles.

See Also:

ConFactory Transport

Attributes

tag

readonly attribute ProfileId tag;

The profile id tag.

Operations

connect

Transport connect();

Used by CORBA clients to establish a connection to a CORBA server. It returns a Transport object, which can be used for sending and receiving octet streams to and from the server.

Returns:

The new Transport object.

connect_timeout

Transport connect_timeout(in unsigned long timeout);

Similar to connect, but it is possible to specify a timeout. On return the caller can test whether there was a timeout by checking whether a nil object reference was returned.

Parameters:

timeout - The timeout value in milliseconds.

Returns:

The new Transport object.

is_usable

ObjectKey is_usable(in IOR ior);

Checks whether this Connector can be used for a specific IOR. That is, the IOR must contain at least one profile that matches this Connector.

Parameters:

ior - The IOR to check for.

Returns:

The object key of the matching profile if the Connector can be used for the given IOR, or an empty object key otherwise.

is_usable_with_policies

Checks whether this Connector can be used for a specific IOR with a given set of polcies. That is, the IOR must contain at least one profile that matches this Connector and the Connector must also satisfy the provided list of policies for the given IOR.

Parameters:

ior - The IOR to check for. policies - The policies that must be satisfied.

Returns:

The object key of the matching profile if the Connector can be used for the given IOR and policies, or an empty object key otherwise.

get_info

```
ConnectorInfo get_info();
```

Returns the information object associated with the Connector.

Returns:

The Connector information object.

B.7 Interface OCI::ConnectorInfo

interface ConnectorInfo

Information on a OCI Connector object. Objects of this type must be narrowed to a Connector information object for a concrete protocol implementation, for example to OCI::IIOP::ConnectorInfo in case the plug-in implements IIOP.

See Also:

Connector

Attributes

tag

readonly attribute ProfileId tag;

The profile id tag.

Operations

add_connect_cb

void add_connect_cb(in ConnectCB cb);

Add a callback that is called whenever a new connection is established. If the callback has already been registered, this method has no effect.

Parameters:

cb - The callback to add.

remove_connect_cb

void remove_connect_cb(in ConnectCB cb);

Remove a connect callback. If the callback was not registered, this method has no effect.

Parameters:

cb - The callback to remove.

B.8 Interface OCI::ConnectCB

interface ConnectCB

An interface for a connect callback object.

See Also: ConnectorInfo

Operations

connect_cb

```
void connect_cb(in TransportInfo transport_info);
```

Called after a new connection has been established. If the application wishes to reject the connection CORBA::NO_PERMISSION may be raised.

Parameters:

transport_info - The TransportInfo for the new connection.

B.9 Interface OCI::Acceptor

interface Acceptor

An interface for an Acceptor object, which is used by CORBA servers to accept client connection requests. It also provides operations for the management of IOR profiles.

See Also:

AccRegistry Transport

Attributes

tag

readonly attribute ProfileId tag;

The profile id tag.

handle

readonly attribute Handle handle;

The "handle" for this Acceptor. Like with the handle for Transports, the handle may *only* be used with operations like select(). A handle value of -1 indicates that the protocol plug-in does not support "selectable" Transports.

Operations

close

void close();

Closes the Transport. accept or listen must not be called after close has been called.

shutdown

```
void shutdown();
```

Shuts down the Transport. After calling shutdown, calls to accept or listen result in an appropriate CORBA::COMM_FAILURE exception being raised.

listen

void listen();

Sets the acceptor up to listen for incoming connections. Until this method is called on the acceptor, new connection requests should result in a connection request failure.

accept

Transport accept();

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

Returns:

The new Transport object.

add_profile

Adds a new profile that matches this Acceptor to an IOR.

Parameters:

key - The object key to use for the new profile. ior - The IOR.

is_local

ObjectKey is_local(in IOR ior);

Checks whether an IOR is for a local object, taking only profiles into account matching this Acceptor.

Parameters:

ior - The IOR to check for.

Returns:

If the IOR is for a local object, the object key for that local object, or an empty object key otherwise.

get_info

AcceptorInfo get_info();

Returns the information object associated with the Acceptor.

Returns:

The Acceptor information object.

B.10 Interface OCI::AcceptorInfo

interface AcceptorInfo

Information on an OCI Acceptor object. Objects of this type must be narrowed to an Acceptor information object for a concrete protocol implementation, for example to OCI::IIOP::AcceptorInfo in case the plug-in implements IIOP.

See Also:

Acceptor

Attributes

tag

readonly attribute ProfileId tag;

The profile id tag.

Operations

add_accept_cb

void add_accept_cb(in AcceptCB cb);

Add a callback that is called whenever a new connection is accepted. If the callback has already been registered, this method has no effect.

Parameters:

cb - The callback to add.

remove_accept_cb

void remove_accept_cb(in AcceptCB cb);

Remove an accept callback. If the callback was not registered, this method has no effect.

Parameters:

cb - The callback to remove.
B.11 Interface OCI::AcceptCB

interface AcceptCB

An interface for an accept callback object.

See Also: AcceptorInfo

Operations

accept_cb

```
void accept_cb(in TransportInfo transport_info);
```

Called after a new connection has been accepted. If the application wishes to reject the connection CORBA: :NO_PERMISSION may be raised.

Parameters:

transport_info - The TransportInfo for the new connection.

B.12 Interface OCI:: ConFactory

interface ConFactory

A factory for Connector objects.

See Also: Connector ConFactoryRegistry

Attributes

tag

readonly attribute ProfileId tag;

The profile id tag.

Operations

create

Connector create(in IOR ior);

Creates a new Connector for a given IOR. All connection specific data is taken from an IOR profile that matches this Connector factory. If more than one profile matches, then which of these profiles is used is implementation specific.

Parameters:

ior - The IOR from which the profile and connection data are extracted.

Returns:

The new Connector. A nil object reference is returned if the IOR does not contain a profile which matches this Connector factory.

create_with_policies

```
Connector create_with_policies(in IOR ior,
in CORBA::PolicyList policies);
```

Creates a new Connector for a given IOR, satisfing a list of policies. Like create, all connection specific data is taken from an IOR profile that matches this Connector factory, and if more than one profile matches, then which of these profiles is used is implementation specific.

Parameters:

ior - The IOR from which the profile and connection data are extracted. policies - The policies that must be satisfied.

Returns:

The new Connector. A nil object reference is returned if the IOR does not contain a profile which matches this Connector factory or if the policies cannot be satisfied.

consider_with_policies

Determines whether this Connector factory can create a Connector for a given IOR and a given list of policies.

Parameters:

ior - The IOR to consider. policies - The policies that must be satisfied.

Returns:

TRUE if a Connector can be created for the IOR and the policies can be satisfied, FALSE otherwise.

equivalent

Checks whether two IORs are equivalent, taking only profiles into account matching this Connector factory.

Parameters:

ior1 - The first IOR to check for equivalence. ior2 - The second IOR to check for equivalence.

Returns:

TRUE if the IORs are equivalent, FALSE otherwise.

hash

Calculates a hash value for an IOR.

Parameters:

ior - The IOR to calculate a hash value for. maximum - The maximum value of the hash value.

Returns:

The hash value.

get_info

ConFactoryInfo get_info();

Returns the information object associated with the Connector factory.

Returns:

The Connnector factory information object.

220

B.13 Interface OCI:: ConFactoryInfo

interface ConFactoryInfo

Information on an OCI ConFactory object.

See Also: ConFactory

Attributes

tag

readonly attribute ProfileId tag;

The profile id tag.

Operations

add_connect_cb

void add_connect_cb(in ConnectCB cb);

Add a callback that is called whenever a new connection is established. If the callback has already been registered, this method has no effect.

Parameters:

cb - The callback to add.

remove_connect_cb

void remove_connect_cb(in ConnectCB cb);

Remove a connect callback. If the callback was not registered, this method has no effect.

Parameters:

cb - The callback to remove.

B.14 Interface OCI:: ConFactoryRegistry

interface ConFactoryRegistry

A registry for Connector factories.

See Also: Connector ConFactory

Operations

add_factory

void add_factory(in ConFactory factory);

Adds a Connector factory to the registry.

Parameters:

factory - The Connector factory to add.

get_factory

ConFactory get_factory(in IOR ior);

Returns a suitable Connector factory for an IOR.

Parameters:

ior - The IOR to for which a Connector factory is requested.

Returns:

The Connector factory. A nil object reference is returned if no Connector factory is registered which is able to create a Connector for the given IOR.

get_factory_with_policies

```
ConFactory get_factory_with_policies(in IOR ior,
in CORBA::PolicyList policies);
```

Returns a suitable Connector factory for an IOR. The Connector factory returned must satisfy a list of policies.

Parameters:

ior - The IOR for which a Connector factory is requested. policies - The list of policies which have to be satisfied.

Returns:

The Connector factory. A nil object reference is returned if no Connector factory is registered which is able to create a Connector for the given IOR with the given list of policies.

get_factories

ConFactorySeq get_factories();

Returns a sequence of all registered Connector factories.

Returns:

A sequence with all registered Connector factories.

equivalent

Checks whether two IORs are equivalent. It calls the equivalent operation of all registered Connector factories. Two IORs are considered equivalent, if all these calls return TRUE.

Parameters:

ior1 - The first IOR to check for equivalence. ior2 - The second IOR to check for equivalence.

Returns:

TRUE if the IORs are equivalent, FALSE otherwise.

hash

Calculates an hash value for an IOR. This hash value is based on the return values of the hash operations of all registered Connector factories.

Parameters:

ior - The IOR to calculate an hash value for. maximum - The maximum hash value that is allowed.

Returns:

The hash value.

B.15 Interface OCI::AccRegistry

interface AccRegistry

A registry for Acceptors.

See Also:

Acceptor

Operations

add_acceptor

void add_acceptor(in Acceptor Acceptor);

Adds an Acceptor to the registry.

Parameters:

Acceptor - The Acceptor to add.

get_acceptors

AcceptorSeq get_acceptors();

Returns a sequence of all registered Acceptors.

Returns:

A sequence of all registered Acceptors.

add_profiles

Adds new profiles to an IOR. For each registered Acceptor a new profile is added by calling the Acceptor's add_profile operation.

Parameters:

key - The object key to use for the new profiles. ior - The IOR.

is_local

ObjectKey is_local(in IOR ior);

Checks whether an IOR is for a local object. It calls the is_local operation of all registered

Acceptors. An IOR is considered local, if at least one of these calls returns a non-empty object key.

Parameters:

ior - The IOR to check for.

Returns:

If the IOR is for a local object, the object key for that local object, or an empty object key otherwise.

B.16 Interface OCI::Current

interface **Current** inherits from CORBA::Current

Interface to access Transport and Acceptor information objects related to the current request.

Operations

get_oci_transport_info

TransportInfo get_oci_transport_info();

This method returns the Transport information object for the Transport used to invoke the current request.

get_oci_acceptor_info

AcceptorInfo get_oci_acceptor_info();

This method returns the Acceptor information object for the Acceptor which created the Transport used to invoke the current request.

B.17 Module OCI::IIOP

This module contains interfaces to gather information on the IIOP OCI plug-in.

Aliases

InetAddr

```
typedef octet InetAddr[4];
```

Alias for an array of four octets. This alias will be used for address information from the various information classes. The address will always be in network byte order.

B.18 Interface OCI::IIOP::TransportInfo

interface **TransportInfo**

inherits from OCI::TransportInfo

Information on an IIOP OCI Transport object.

See Also:

Transport TransportInfo

Attributes

addr

readonly attribute InetAddr addr;

The local 32 bit IP address.

port

readonly attribute unsigned short port;

The local port.

remote_addr

readonly attribute InetAddr remote_addr;

The remote 32 bit IP address.

remote_port

readonly attribute unsigned short remote_port;

The remote port.

B.19 Interface OCI::IIOP::ConnectorInfo

interface **ConnectorInfo** inherits from OCI::ConnectorInfo

Information on an IIOP OCI Connector object.

See Also: Connector ConnectorInfo

Attributes

remote_addr

readonly attribute InetAddr remote_addr;

The remote 32 bit IP address to which this connector connects.

remote_port

readonly attribute unsigned short remote_port;

The remote port to which this connector connects.

B.20 Interface OCI::IIOP::AcceptorInfo

interface AcceptorInfo

inherits from OCI::AcceptorInfo

Information on an IIOP OCI Acceptor object.

See Also:

Acceptor AcceptorInfo

Attributes

host

readonly attribute string host;

Hostname used for creation of IIOP object references.

addr

readonly attribute InetAddr addr;

The local 32 bit IP address on which this acceptor accepts.

port

readonly attribute unsigned short port;

The local port on which this acceptor accepts.

B.21 Interface OCI::IIOP::ConFactoryInfo

interface **ConFactoryInfo** inherits from OCI::ConFactoryInfo

Information on an IIOP OCI Connector Factory object.

See Also: ConFactory ConFactoryInfo

Open Communications Interface Reference

APPENDIX C

Royalty-Free Public License Agreement

ROYALTY-FREE PUBLIC LICENSE AGREEMENT FOR ORBACUS SOFTWARE

IMPORTANT-READ CAREFULLY: This Object-Oriented Concepts, Inc. Royalty-Free Public License Agreement for ORBacus Software ("License") is a legal agreement between you, the Licensee, (either an individual or a single entity) and Object-Oriented Concepts, Inc. ("OOC") for non-commercially using, copying, distributing and modifying the Software and any work derived from the Software, as defined hereinbelow. Any commercial use is subject to a different license.

By using, modifying or distributing the Software or any work derived from the Software, Licensee indicates acceptance of this License, and agrees to be bound by all its terms and conditions for using, copying, distributing or modifying the Software and works derived from the Software.

No rights are granted to the Software except as expressly set forth herein. Nothing other than this License grants Licensee permission to use, copy, distribute or modify the Software or any work derived from the Software. Licensee may not use, copy, distribute or modify the Software or any work derived from the Software except as expressly provided under this License. If Licensee does not accept the terms and conditions of this License, do not use, copy, distribute or modify the Software.

In consideration for Licensee's forbearance of commercial use of the Software, OOC grants Licensee non-exclusive, royalty-free rights as expressly provided herein.

DEFINITIONS.

The "Software" is the ORBacus software, including, but not limited to, the ORBacus Libraries and Class Files, the ORBacus IDL-to-C++ and IDL-to-Java translators, the ORBacus C++ and Java

Naming, Event, and Property Services, interface definitions, demo programs, associated media and printed materials, and any included "on-line" documentation.

A "work derived from the Software" is any derivative work, as defined in 17 U.S.C. paragraph 101, which is derived from the Software, for example, code generated by the ORBacus IDL-to-C++ or IDL-to-Java translators, a program which is linked with or otherwise incorporates the ORBacus Libraries or Class Files, or a translation, improvement, enhancement, extension or other modification of the Software.

To "use" means to execute (i.e. run) the Software.

To "copy" means to create one or more copies as defined in 17 U.S.C. paragraph 101.

To "distribute" means to broadcast, publish, transfer, post, upload, download or otherwise disseminate in any medium to any third party.

To "modify" means to create a work derived from the Software.

A "commercial use" is:

(1) the use of the Software or any work derived from the Software in connection with, for or in aid of the generation of revenue, such as in the conduct of Licensee's daily business operations; or

(2) any copying, distribution or modification of the Software or any work derived from the Software to any party where payment or other consideration is made in connection with such copying, distribution or modification, whether directly (as in payment for a copy of the Software) or indirectly (including but not limited to payment for some good or service related to the Software, or payment for some product or service that includes a copy of the Software "without charge"). However, the following actions which involve payment do not in and of themselves constitute a commercial use:

(a) posting the Software on a public access information storage and retrieval service for which a fee is received for retrieving information (such as an on-line service), provided that the fee is not content-dependent. Such fees which are not content dependent include, but are not limited to, fees which are based solely on the storage capacity required to store the information, and fees which are based solely on the time required to transfer the information from/to the public access information storage and retrieval service; and

(b) distributing the Software on a CD-ROM, provided that the Software is reproduced entirely and verbatim on such CD-ROM, and provided further that all information on such CD-ROM may be distributed in a manner which does not constitute a commercial use.

GRANT OF LICENSE.

LICENSE TO USE. Licensee may use the Software provided that such use does not constitute a commercial use.

LICENSE TO COPY AND DISTRIBUTE. Licensee may copy and distribute literal (i.e., verbatim) copies of the Software as Licensee receives it throughout the world, in any medium, provided that

Licensee distributes an unmodified, easily-readable copy of this License with the Software, and provided further that such distribution does not constitute a commercial use.

LICENSE TO CREATE WORKS DERIVED FROM THE SOFTWARE. Licensee may create works derived from the Software, provided that any such work derived from the Software carries prominent notices stating both the manner in which Licensee has created a work derived from the Software (for example, notices stating that the work derived from the Software is linked with or otherwise incorporates the ORBacus Libraries or Class Files or code generated by the ORBacus IDL-to-C++ or IDL-to-Java translators, or notices stating that the work derived from the Software is an enhancement to the Software which Licensee has created) and the date any such work derived from the Software was created.

LICENSE TO COPY AND DISTRIBUTE WORKS DERIVED FROM THE SOFTWARE. Licensee may copy and distribute works derived from the Software throughout the world, provided that Licensee distributes an unmodified, easily-readable copy of this License with such works derived from the Software, and provided further that such distribution does not constitute a commercial use. Licensee must cause any work derived from the Software that Licensee distributes to be licensed as a whole and at no charge to all third parties under the terms of this License.

Any work derived from the Software must be accompanied by the complete corresponding machinereadable source code of such work derived from the Software, delivered on a medium customarily used for software interchange. The source code for the work derived from the Software means the preferred form of the work derived from the Software for making modifications to it. For an executable work derived from the Software, complete source code means all of the source code for all modules of the work derived from the Software, all associated interface definition files and all scripts used to control compilation and installation of all or any part of the work derived from the Software. However, the source code delivered need not include anything that is normally distributed, in either source code or binary (object-code) form, with major components (including but not limited to compilers, linkers and kernels) of the operating system on which the executable work derived from the Software runs, unless that component itself accompanies the executable code of the work derived from the Software;

Furthermore, if the executable code or object code of the work derived from the Software may be copied from a designated place, and if the source code of the work derived from the Software may be copied from the same place, then the work derived from the Software shall be construed as accompanied by the complete corresponding machine-readable source code of such work derived from the Software, even though third parties are not compelled to copy the source code along with the executable code or object code.

If the work derived from the Software normally reads commands interactively when run, Licensee must cause the work derived from the Software, at each time it commences operation, to print or display an announcement including an appropriate copyright notice and either a notice consisting of the verbatim warranty and liability provisions of this License, or a notice that Licensee, and not OOC provides a warranty. Such notice must also state that users may distribute the Software and/or the work derived from the Software only under the conditions of this License, and must further state how to view the copy of this License included with the work derived from the Software.

Licensee may not impose any further restrictions on the exercise of the rights granted herein by any recipient of any work derived from the Software.

RESTRICTIONS.

Licensee acknowledges that the Software is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The Software is licensed, not sold. All title and copyrights in and to the Software, including but not limited to any images, photographs, databases, animations, video, text and "applets" incorporated into the Software, the accompanying printed materials, and any copies of the Software, are owned exclusively by OOC.

Licensee may not sublicense, assign or transfer this License, the Software or any work derived from the Software except as permitted by this License.

If Licensee distributes any written or printed material at all with the Software or any work derived from the Software, such material must include either (a) a written copy of this License, or (b) a prominent written indication that the Software or work derived from the Software is covered by this License, and also written instructions for printing and/or displaying the copy of this License which is provided on the distribution medium.

If using, copying, distributing and/or modifying the Software is restricted in certain countries for any reason, OOC may in the future add an explicit geographical distribution limitation excluding those countries, so that using, copying, distributing and/or modifying is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

LICENSE TO WORKS DERIVED FROM THE SOFTWARE.

Licensee hereby grants to OOC a non-exclusive, non-transferable, royalty-free right to use, copy, distribute and modify, with the right to sublicense at any tier, any and all works derived from the Software that Licensee creates, provided such works derived from the Software are distributed to OOC by Licensee, and further provided that, if such works derived from the Software comprise either code generated by the ORBacus IDL-to-C++ or IDL-to-Java translators or a program which is linked with or otherwise incorporates the ORBacus Libraries or Class Files, such works derived from the Software would constitute works derived from the Software independent of comprising code generated by the ORBacus IDL-to-C++ or IDL-to-Java translators or a program which is linked with or otherwise incorporates the ORBacus Libraries or Class Files, for example, a "bug fix" of the Software.

LIMITED WARRANTY.

NO WARRANTIES. OOC EXPRESSLY DISCLAIMS ANY WARRANTY FOR THE SOFT-WARE. THE SOFTWARE IS PROVIDED TO LICENSEE "AS IS," WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PUR-POSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE ENTIRE RISK AS TO THE USE, QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH LICENSEE. SHOULD THE SOFTWARE PROVE DEFECTIVE, LICENSEE ASSUMES THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

NO LIABILITY FOR DAMAGES. IN NO EVENT WILL OOC, OR ANY OTHER PARTY WHO MAY COPY, DISTRIBUTE OR MODIFY THE SOFTWARE AS PERMITTED HEREIN, BE LIABLE FOR ANY GENERAL, DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSE-QUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAM-AGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, INACCURATE INFORMATION, LOSS OF INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE, EVEN IF OOC OR SUCH OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT RESTRICTED RIGHTS.

The Software is provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software-Restricted Rights 48 C.F.R. paragraph 52.227-19, as applicable. Manufacturer is Object-Oriented Concepts, Inc./44 Manning Road/Billerica, MA 01821

TERMINATION.

Any violation or any attempt to violate any of the terms and conditions of this License will automatically terminate Licensee's rights under this License. Licensee further agrees upon such termination to cease any and all using, copying, distributing and modifying of the Software and any work derived from the Software, and further to destroy any and all of Licensee's copies of the Software and any work derived from the Software.

However, parties who have received copies of the Software or copies of any work derived from the Software, or rights, from Licensee under this License will not have their licenses terminated so long as such parties remain in full compliance with this License.

LICENSE SCOPE AND MODIFICATION.

This License sets forth the entire agreement between Licensee and OOC and supersedes all prior agreements and understandings between the parties relating to the subject matter hereof. None of the terms of this License may be waived or modified except as expressly agreed in writing by both Licensee and OOC.

SEVERABILITY.

Should any provision of this License be declared void or unenforceable, the validity of the remaining provisions shall not be affected thereby.

GOVERNING LAWS.

This License is governed by the laws of the Commonwealth of Massachusetts, U.S.A., and shall be interpreted in accordance with and governed by the laws thereof.

Licensee hereby waives any and all right to assert a defense based on jurisdiction and venue for any action stemming from this License brought in U.S. District Court for the District of Massachusetts.

Should Licensee have any questions concerning this License, or if Licensee desires to contact Object-Oriented Concepts, Inc. for any reason, please contact Object-Oriented Concepts, Inc. at:

Object-Oriented Concepts, Inc. 44 Manning Road Billerica, MA 01821

References

[1]	The ORBACUS Home Page, http://www.ooc.com/ob/, Object-Oriented Concepts, Inc.
[2]	<i>The Common Object Request Broker: Architecture and Specification</i> , Revision 2.0, OMG Document 97–02–25
[3]	IDL/Java Language Mapping, OMG document 97-03-01
[4]	CORBAservices: Common Object Services Specification, OMG document 97-12-02
[5]	Marc Laukien and Robert Resendes, <i>Introduction to CORBA Distributed Objects</i> , C/C++ Users Journal, April 1998
[6]	D. C. Schmidt, <i>Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplex-</i> <i>ing and Event Handler Dispatching</i> , in Pattern Languages of Program Design, Addison- Wesley, 1995
[7]	Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, A System of Patterns, John Wiley & Sons, Inc.
[8]	<i>The</i> JTHREADS/C++ <i>Home Page</i> , http://www.ooc.com/jtc/, Object-Oriented Concepts, Inc.
[9]	JTHREADS/C++ User's Manual, Object-Oriented Concepts, Inc.

- [10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, 1994
- [11] CORBA Messaging, OMG document 98-03-11
- [12] *The Common Object Request Broker: Architecture and Specification*, Revision 2.2, OMG document 98-02-33
- [13] ORBacus SSL User's Manual, Object Oriented Concepts, Inc.