ORBacus

For C++ and Java

Version 4.0.5

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" [4], "C++ Language Mapping" [5], "IDL/Java Language Mapping" [6], and "Portable Interceptors" [7].

These are some of the highlights of ORBACUS:

- Full CORBA IDL support
- C++ and Java language mappings
- Simple configuration and bootstrapping
- Portable Object Adapter
- Objects by Value
- Portable Interceptors
- Single- and Multi-threaded
- Active Connection Management
- Fault Tolerant Extensions
- Dynamic Invocation and Dynamic Skeleton Interface
- Dynamic Any

- Interface and Implementation Repository
- Pluggable Protocols
- IDL-to-HTML and IDL-RTF documentation tools
- Includes Naming, Event, Property and Time Services

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

1.2 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 [4], [6], and [6] is absolutely necessary to effectively use this manual. In particular, the chapters in [4], covering CORBA IDL and the IDL-to-C++ mapping, should be studied thoroughly.

For C++ users, we also highly recommend [3]. This book contains by far the best treatment of CORBA programming with C++ to date.

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.

1.3 Getting Help

Should you need any assistance with ORBACUS, please visit our Support Frequently Asked Questions (FAQ) list at `http://www.orbacus.com/faq/support.html`.

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:

```
// C++
#include <iostream.h>

int main(int, char*[])
{
    cout << "Hello World!" << endl;
    return 0;
}
```

Or in introductory Java books:

```
// Java
public class Greeter
{
    public static void main(String args[])
    {
        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 `say_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 interface Hello
3 {
4     void say_hello();
5 };
```

2 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.

4 The only operation defined is `say_hello`, which neither takes any parameters nor returns any result.

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 `POA_Hello`, defined in the file `Hello_skel.h`. The definition for `Hello_impl` looks like this:

```
1  // C++
2  #include <Hello_skel.h>
3
4  class Hello_impl : public POA_Hello,
5                     public PortableServer::RefCountServantBase
6  {
7  public:
8
9      virtual void say_hello() throw(CORBA::SystemException);
10 };
```

2 Since our implementation class derives from the skeleton class `POA_Hello`, we must include the file `Hello_skel.h`.

4-5 Here we define `Hello_impl` as a class derived from `POA_Hello` and `RefCountServantBase`. `RefCountServantBase` is part of the `PortableServer` namespace and provides reference counting.

9 Our implementation class must implement all operations from the IDL interface. In this case, this is just the operation `say_hello`.

The implementation for `Hello_impl` looks as follows:

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

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`.

6-9 The `say_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 program. To simplify exception handling and ORB destruction, we split our server into two functions: `main()` and `run()`. `main()` only creates the ORB, and calls `run()`:

```
1  // C++
2  #include <OB/CORBA.h>
3  #include <Hello_impl.h>
4
5  #include <fstream.h>
6
7  int run(CORBA::ORB_ptr);
8
9  int main(int argc, char* argv[])
10 {
11    int status = EXIT_SUCCESS;
12    CORBA::ORB_var orb;
13
14    try
15    {
16       orb = CORBA::ORB_init(argc, argv);
17       status = run(orb);
18    }
19    catch(const CORBA::Exception&)
20    {
21       status = EXIT_FAILURE;
22    }
23
24    if(!CORBA::is_nil(orb))
25    {
26       try
27       {
28          orb -> destroy();
29       }
30       catch(const CORBA::Exception&)
31       {
32          status = EXIT_FAILURE;
33       }
34    }
35
36    return status;
37 }
```

2-5 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.

7 A forward declaration for the `run()` function.

16 The first thing a CORBA program must do is initialize the ORB. This operation expects the parameters with which the program was started. These parameters may or may not be used by the ORB, depending on the CORBA implementation. ORBACUS recognizes certain options that will be explained later.

17 The `run()` helper function is called.

19-22 This code catches and prints all CORBA exceptions raised by `ORB_init()` or `run()`.

24-34 If the ORB was successfully created, it is destroyed. This releases the resources used by the ORB. If `destroy()` raises a CORBA exception, this exception is caught and printed.

36 The exit status is returned. If there was no error, `EXIT_SUCCESS` is returned, or `EXIT_FAILURE` otherwise.

Now we write the `run()` function:

```
1  // C++
2  int run(CORBA::ORB_ptr orb)
3  {
4      CORBA::Object_var poaObj =
5          orb -> resolve_initial_references("RootPOA");
6      PortableServer::POA_var rootPoa =
7          PortableServer::POA::_narrow(poaObj);
8
9      PortableServer::POAManager_var manager =
10         rootPoa -> the_POAManager();
11
12     Hello_impl* helloImpl = new Hello_impl();
13     PortableServer::ServantBase_var servant = helloImpl;
14     Hello_var hello = helloImpl -> _this();
15
16     CORBA::String_var s = orb -> object_to_string(hello);
17     const char* refFile = "Hello.ref";
18     ofstream out(refFile);
19     out << s << endl;
20     out.close();
21
22     manager -> activate();
23     orb -> run();
24
25     return EXIT_SUCCESS;
26 }
```

4-7 Using the ORB reference, `resolve_initial_references()` is invoked to obtain a reference to the Root POA.

9-10 The Root POA is used to obtain a reference to its POA Manager.

12-14 A servant of type `Hello_impl` is created and assigned to a `ServantBase_var` variable. The servant is then used to incarnate a CORBA object, using the `_this()` operation. `ServantBase_var` and `Hello_var`, like all `_var` types, are "smart" pointer, i.e., `servant` and `hello` will release their assigned object automatically when they go 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 then be read by the client and converted back to the actual object reference.[1] The operation `object_to_string()` converts a CORBA object reference into its string representation.

22-23 The server must activate the POA Manager to allow the Root POA to start processing requests, and then inform the ORB that it is ready to accept requests.

Save the code for `main()` and `run()` to a file with the name `Server.cpp`.

2.3.2 Implementing the Client

In several respects, the client program is similar to the server program. The code to initialize and destroy the ORB is the same:

```
1  // C++
2  #include <OB/CORBA.h>
3  #include <Hello.h>
4
5  #include <fstream.h>
6
7  int run(CORBA::ORB_ptr);
8
9  int main(int argc, char* argv[])
10 {
11     ... // Same as for the server
12 }
13
14 int run(CORBA::ORB_ptr orb)
```

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.

```
15 {
16     const char* refFile = "Hello.ref";
17     ifstream in(refFile);
18     char s[2048];
19     in >> s;
20     CORBA::Object_var obj = orb -> string_to_object(s);
21
22     Hello_var hello = Hello::_narrow(obj);
23
24     hello -> say_hello();
25
26     return 0;
27 }
```

3 In contrast to the server, the client does not need to include `Hello_impl.h`. Only the generated file `Hello.h` is needed.

7-12 This code is the same as for the server.

16-20 The "stringified" object reference written by the server is read and converted to a `CORBA::Object` object reference. It's not necessary to obtain a reference to the Root POA or its POA Manager, because they are only needed by server applications.

22 The `_narrow` operation generates a `Hello` object reference from the `CORBA::Object` object reference. Although `_narrow` for CORBA objects works similar to `dynamic_cast<>` for plain C++ objects, `dynamic_cast<>` must not be used for CORBA object references. That's because in contrast to `dynamic_cast<>`, `_narrow` might have to query the server for type information.

24 The `say_hello` operation on the `hello` object reference is invoked, causing the server to print "Hello World!".

Save this code 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 as `libsocket.a` and `libnsl.a` for Solaris or `wsock32.lib` for Windows.

The ORBACUS distribution includes 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.

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`, `HelloOperations.java`, `HelloPOA.java` and `_HelloStub.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:

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

4 The implementation class `Hello_impl` must inherit from the generated class `HelloPOA`.

6-8 As with the C++ implementation, the `say_hello` method simply prints “Hello World!” on standard output.

Save this class to the file `Hello_impl.java` in the directory `hello`.

We also have to write a class which holds the server's `main()` and `run()` methods. We call this class `Server`, saved as the file `Server.java` in the directory `hello`:

```
1  // Java
2  package hello;
3
4  public class Server
5  {
6      public static void main(String args[])
7      {
8          java.util.Properties props = System.getProperties();
9          props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
10         props.put("org.omg.CORBA.ORBSingletonClass",
11                   "com.ooc.CORBA.ORBSingleton");
12
13         int status = 0;
14         org.omg.CORBA.ORB orb = null;
15
16         try
17         {
18             orb = org.omg.CORBA.ORB.init(args, props);
19             status = run(orb);
20         }
21         catch(Exception ex)
22         {
23             ex.printStackTrace();
24             status = 1;
25         }
26
27         if(orb != null)
28         {
29             try
30             {
31                 ((com.ooc.CORBA.ORB)orb).destroy();
32             }
33             catch(Exception ex)
34             {
35                 ex.printStackTrace();
36                 status = 1;
37             }
```

```
38          }
39
40          System.exit(status);
41      }
```

8-11 These properties are necessary to use the ORBACUS ORB with JDK 1.2 or later.

18 The ORB must be initialized using `ORB.init`. The ORB class resides in the package `org.omg.CORBA`. You must either import this package, or, as shown in this example, you must use `org.omg.CORBA` explicitly.

19 The `run()` helper function is called.

21-25 This code catches and prints all CORBA exceptions raised by `ORB.init()` or `run()`.

27-38 If the ORB was successfully created, it is destroyed. This releases the resources used by the ORB. If `destroy()` raises a CORBA exception, this exception is caught and printed. The cast to `com.ooc.CORBA.ORB` is required when using JDK 1.2, but not when using JDK 1.1 or JDK 1.3.

41 The exit status is returned. If there was no error, `0` is returned, or `1` otherwise.

Now we write the `run()` method:

```
1  // Java
2      static int run(org.omg.CORBA.ORB orb)
3          throws org.omg.CORBA.UserException
4      {
5          org.omg.PortableServer.POA rootPOA =
6              org.omg.PortableServer.POAHelper.narrow(
7                  orb.resolve_initial_references("RootPOA"));
8
9          org.omg.PortableServer.POAManager manager =
10             rootPOA.the_POAManager();
11
12         Hello_impl helloImpl = new Hello_impl();
13         Hello hello = helloImpl._this(orb);
14
15         try
16         {
17             String ref = orb.object_to_string(hello);
18             String refFile = "Hello.ref";
19             java.io.PrintWriter out = new java.io.PrintWriter(
20                 new java.io.FileOutputStream(refFile));
21             out.println(ref);
22             out.close();
```

```
23          }
24          catch(java.io.IOException ex)
25          {
26              ex.printStackTrace();
27              return 1;
28          }
29
30          manager.activate();
31          orb.run();
32
33          return 0;
34      }
35  }
```

5-10 A reference to the Root POA is obtained using the ORB reference, and the Root POA is used to obtain a reference to its POA Manager.

12-23 A servant of type `Hello_impl` is created and is used to incarnate a CORBA object. The CORBA object is released automatically when it is not used anymore.

15-28 The object reference is "stringified" and written to a file.

30-31 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` in the directory `hello`:

```
1  // Java
2  package hello;
3
4  public class Client
5  {
6      public static void main(String args[])
7      {
8          ... // Same as for the server
9      }
10
11     static int run(org.omg.CORBA.ORB orb)
12     {
13         org.omg.CORBA.Object obj = null;
14         try
15         {
16             String refFile = "Hello.ref";
17             java.io.BufferedReader in = new java.io.BufferedReader(
```

```
18              new java.io.FileReader(refFile));
19          String ref = in.readLine();
20          obj = orb.string_to_object(ref);
21      }
22      catch(java.io.IOException ex)
23      {
24          ex.printStackTrace();
25          return 1;
26      }
27
28      Hello hello = HelloHelper.narrow(obj);
29
30      hello.say_hello();
31
32      return 0;
33  }
34  }
```

6-9 This code is the same as for the server.

14-26 The stringified object reference is read and converted to an object.

28 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`.

30 The `say_hello` operation is invoked, causing the server to print "Hello World!" on standard output.

2.4.3 Compiling

Ensure that your `CLASSPATH` environment variable includes the current working directory as well as the ORBACUS for 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.

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

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

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 to 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 another[1]. 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.

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.

2.6 Where to go from here

To understand the remaining chapters of this manual, you *must* have read the CORBA specifications in [4], [5], and [6]. You will not be able to understand the chapters that follow without a good understanding 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:

`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
`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...

`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 `MyFile.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 44 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 41 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 41 provides more information.

`irgen` generates C++ code directly from the contents of an Interface Repository. See "The IDL-to-C++ Translator and the Interface Repository" on page 40 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.

`-E`

Runs the source files through the preprocessor without generating code.

`--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`.

`--fwd`

Generate separate header files for forward declarations.

`--impl`

Generate example servant implementation classes. An input file `Foo.idl` will generate the files `Foo_impl.h` and `Foo_impl.cpp`. These files will not be overwritten, therefore you must first remove the existing files before new ones can be generated. You must not use `--no-skeletons` in combination with this option.

`--impl-all`

Similar to `--impl`, but function signatures are generated for all inherited operations and attributes. You must not use `--no-skeletons` in combination with this option.

`--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`.

`--skel-suffix SUFFIX`

Use `SUFFIX` as the suffix for skeleton files. The default value is `_skel`.

`--all`

Generate code for included files instead of inserting `#include` statements. See “Include Statements” on page 40.

`--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 included 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 the specified directory.

`--this-header-dir DIR`

Like the `--header-dir` option, this option can be used to make `#include` statements for header files relative to the specified directory. However, this option only applies to `#include` statements for the header files of this IDL file.

`--other-header-dir DIR`

Like the `--header-dir` option, this option can be used to make `#include` statements for header files relative to the specified directory. However, this option only applies to `#include` statements for the header files corresponding to IDL files that were included in this IDL file.

`--output-dir DIR`

Write generated files to directory DIR.

`--file-list FILE`

Write a list of all generated files to file FILE.

`--dll-import DEF`

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

`--with-interceptor-args`

Generate code with support for arguments, result and exception list values for interceptors.

`--no-orb-mediation`

By default, invocations on collocated servants are mediated by the ORB. Specify this option to disable ORB mediation.

`--no-local-copy`

To ensure strict compliance with CORBA's location transparency semantics, the default behavior of the translator is to generate code that copies valuetype argument and result values for collocated invocations. Specify this option to disable strict compliance and generate more efficient code.

`--case-sensitive`

The semantics of OMG IDL forbid identifiers in the same scope to differ only in case. This option relaxes these semantics, but is only provided for backward compatibility with non-compliant IDL.

3.5 *Options for jidl*

```
-h, --help
-v, --version
-e, --cpp NAME
-d, --debug
-DNAME
-DNAME=DEF
-UNAME
-IDIR
-E
--no-skeletons
--locality-constrained
--all
--tie
--file-list FILE
--no-local-copy
--case-sensitive
```

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 prefix[1]. Each class with this prefix will be generated relative to the specified package.

`--auto-package`

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`

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.

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, exception, valuetype and abstract interface types. For valuetypes, only an abstract method is generated. The valuetype implementer must supply an implementation for `clone`.

`--impl`

Generates example servant implementation classes. For IDL interface types, a class is generated in the same package as the interface classes, having the same name as the interface with the suffix `_impl`. The generated class extends the POA class of the interface. For IDL valuetypes, a class is generated in the same package as the valuetype with the suffix `ValueFactory_impl`. You must not use `--no-skeletons` in combination with this option.

`--impl-tie`

Similar to `--impl`, but implementation classes for interfaces implement the `Operations` interface to facilitate the use of TIE classes. You must not use `--no-skeletons` in combination with this option.

`--with-interceptor-args`

Generate code with support for arguments, result and exception list values for interceptors. Note that use of this option will generate proprietary stubs and skeletons which are not compatible with ORBs from other vendors.

3.6 Options for hidl

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

These options are the same as for the `idl` command.

`--no-sort`

Don’t sort symbols alphabetically.

`--ignore-case`

Sort case-insensitive.

`--use-tables`

Use tables for indices.

`--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
-E
--all
--case-sensitive
```

These options are the same as for the `idl` command.

`--no-sort`

Don’t sort symbols alphabetically.

`--ignore-case`

Sort case-insensitive.

`--use-tables`

Use tables for indices.

`--output-dir DIR`

Write RTF files to the directory DIR.

`--single-file FILE`

Create a single file called FILE.rtf.

```
--with-index
```

Create index entries.

```
--font PARA NAME
--font-size PARA SIZE
```

Specify the font name or size for a particular paragraph type. The paragraph types and their default values are shown below.

Type	Font	Size
body	roman Times New Roman	12pt
entry	swiss Tahoma	12pt
extra	*same as body*	12pt
heading	swiss Arial	18pt
index	*same as heading*	15pt
literal	roman Courier New	10pt
symbol	roman Symbol	12pt

3.8 Options for irgen

```
-h, --help
-v, --version
--no-skeletons
--no-type-codes
--locality-contrained
--no-virtual-inheritance
--tie
--impl
--impl-all
--c-suffix SUFFIX
--h-suffix SUFFIX
--skel-suffix SUFFIX
--header-dir DIR
--other-header-dir DIR
--output-dir DIR
--file-list FILE
--dll-import DEF
--with-interceptors-args
--no-local-copy
```

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.

See Chapter 14 for more information on the Interface Repository.

3.9 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
```

See Chapter 14 for more information on the Interface Repository.

3.10 Include Statements

If you use the `#include` statement in your IDL code, the ORBACUS IDL-to-C++ translator `idl` does not create code for included IDL files. Instead, the translator inserts the appropriate `#include` statements in the generated header files. 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 does not place `#include` statements in the generated files, but rather generates code for all IDL definitions in the Interface Repository.

3.11 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 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`*

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`*

1. Preprocessor statements like `#define` or `#ifdef` may be placed before your `#include` statements.

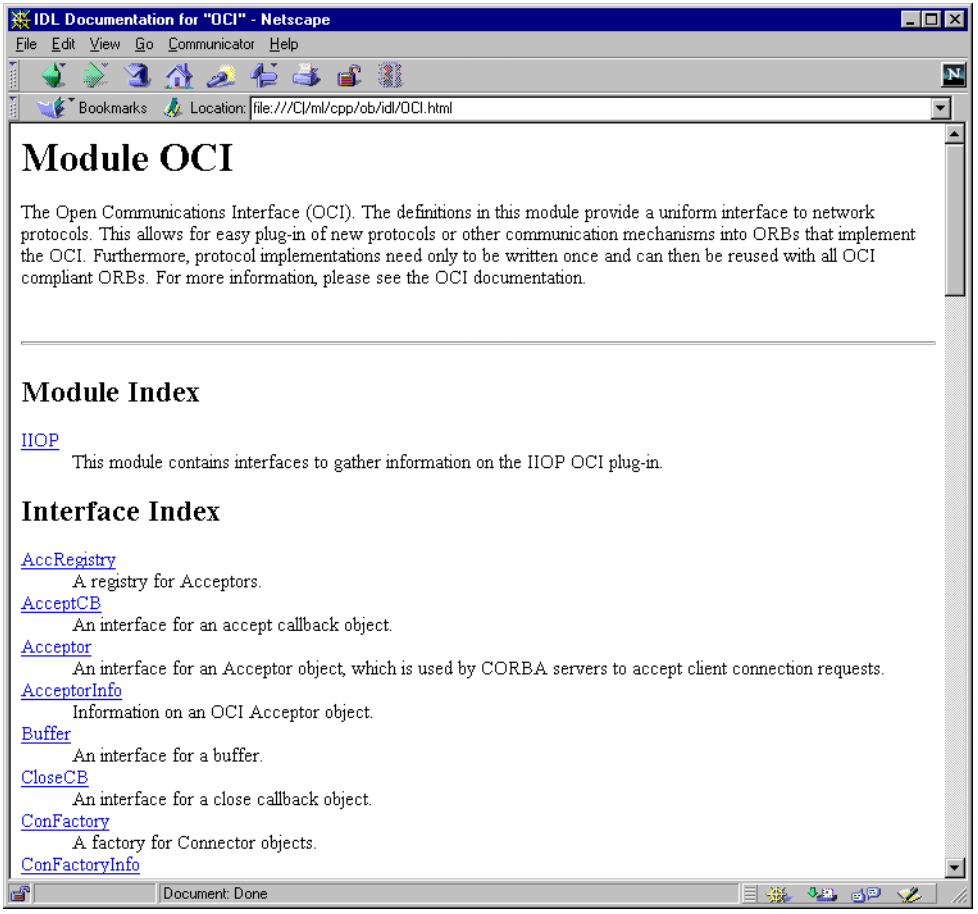


Figure 3.1: Documentation generated with the IDL-to-HTML translator

Adds descriptive text for the return value of an operation.

`@see` *`reference`*

Adds a “See also” note.

`@since` *`since-text`*

Comment related to the availability of new features.

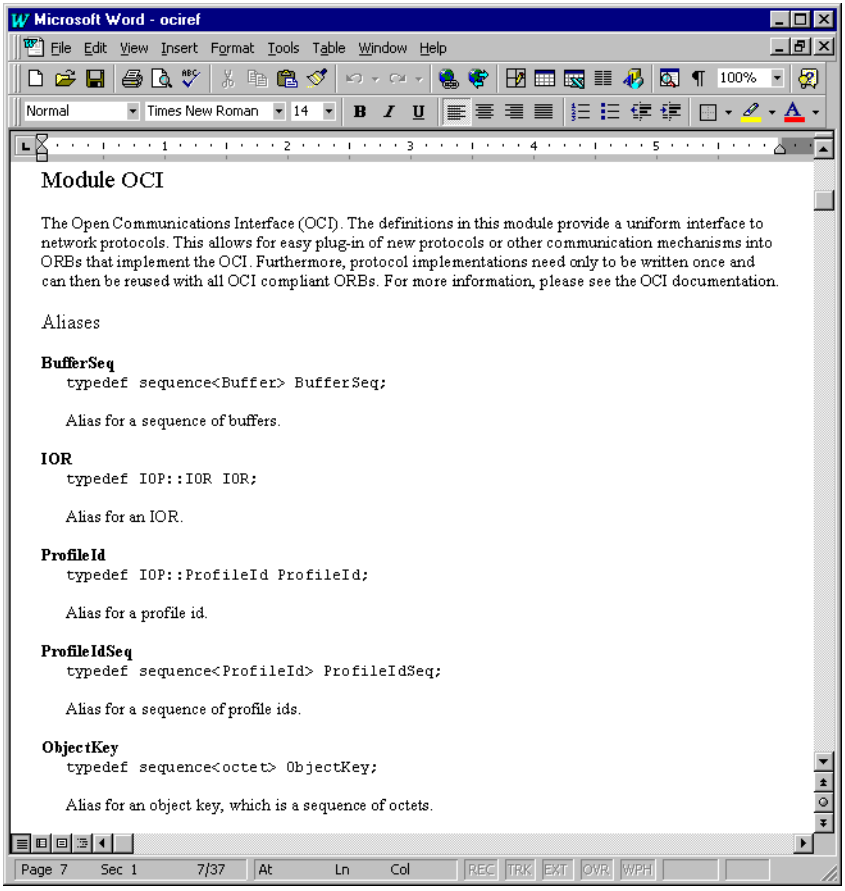


Figure 3.2: Documentation generated with the IDL-to-RTF translator

`@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 produces an equivalent format in the generated RTF files. The following HTML tags are supported:

```
<B> <BR> <CODE> <DD> <DL> <DT> <EM> <HR> <I> <LI> <OL> <P> <TABLE>
<TD> <TR> <U> <UL>
```

3.12 Using javadoc

If not explicitly suppressed with the `--no-comments` option, the ORBACUS IDL-to-Java translator `jidl` adds IDL comments starting with `/**` 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 that shows 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 are automatically 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;

//
// IDL:M/I:1.0
//
/**
 * This is a comment related to interface I.
 *
 * @author Uwe Seimet
 *
 * @version 1.0
 *
 **/
public interface I extends org.omg.CORBA.Object
{
    //
    // IDL:M/I/S:1.0
    //
    /**
     *
     * 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` (`M.IPackage.E` in this case).

These are the contents of `IPackage/E.java`:

```
// Java

package M.IPackage;

//
// IDL:M/I/E:1.0
//
/**
 *
 * 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.

CHAPTER 4

ORB and Object Adapter Initialization

4.1 ORB Initialization

4.1.1 Initializing the C++ ORB

In C++, the ORB is initialized with `CORBA::ORB_init()`. For example:

```
// C++
int main(int argc, char* argv[])
{
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    // ...
}
```

The `CORBA::ORB_init()` call interprets arguments starting with `-ORB` and `-OA`. 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` and `-OA`. Unlike the C++ version, these arguments are not removed (see "Advanced Property Usage" on page 63 for more information).

4.1.3 Initializing the Java ORB in JDK 1.2/1.3

The ORB implementation included in JDK 1.2 and beyond can be considered a "minimal" ORB, suitable primarily for use in basic client-oriented tasks. In order to use the ORBACUS ORB instead of the JDK's default ORB, 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:

```
// Java
import org.omg.CORBA.*;
public static void main(String args[])
{
    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
              "com.ooc.CORBA.ORBSingleton");

    ORB orb = ORB.init(args, props);
    // ...
}
```

4.2 Object Adapter Initialization

4.2.1 Initialization of the Object Adapter

In ORBACUS, the object adapter is not initialized until the Root POA is first resolved. For example:

```
// C++
CORBA::Object_var poaObj =
    orb -> resolve_initial_references("RootPOA");
```

```
// Java
org.omg.CORBA.Object poaObj =
   orb.resolve_initial_references("RootPOA");
```

Upon completion, the ORB will have created the Root POA and its POA Manager, and will have initialized the ORB's server-side functionality.

4.3 Configuring the ORB and Object Adapter

ORBACUS applications can tailor the behavior of the ORB and object adapters using a collection of properties[1]. These properties can be defined in a number ways:

- using the Windows Registry (Windows NT/C++)
- using a configuration file
- using system properties (Java)
- using command-line options
- programmatically at run-time

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

4.3.1 ORB Properties

ooc.config

Value: *filename*

Selects the default configuration file. This property is only available in Java applications and is equivalent to the `ORBACUS_CONFIG` environment variable in C++. See "Using a Configuration File" on page 59 for more information on configuration files.

ooc.orb.client_shutdown_timeout

Value: *timeout >= 0*

If the client is not able to gracefully disconnect from the server in *timeout* seconds, a connection shutdown is forced. If this property is set to zero, then the client will not force a

1. Note that these properties have nothing to do with the Property Service as described in Chapter 11.

connection shutdown. If the property is not set, a default timeout value of two seconds is used.

ooc.orb.client_timeout

Value: *timeout >= 0*

The client actively closes a connection that has been idle for *timeout* seconds once that connection has no more outstanding replies. Note that the application must use the threaded client-side concurrency model if connection timeouts are desired. If this property is set to zero, or not set at all, then the client does not close idle connections. Note that a policy can also be set on the ORB or on individual object references. See "OB::ACMTimeoutPolicy" on page 202 for more information.

ooc.orb.conc_model

Value: `blocking`, `reactive`, `threaded`

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 16 for more information on concurrency models.

ooc.orb.default_init_ref

Value: *URL*

Specifies a partial URL. If an application calls the ORB operation `resolve_initial_references` and no match is found, the ORB appends a slash ('/') character and the service identifier to the specified URL and invokes `string_to_object` to obtain the initial reference.

ooc.orb.default_wcs

Value: *string*

Specifies the default wide character code set for the ORB. Note that the CORBA specification states that a default wide character code set does not exist. Therefore, this option should only be used when communicating with a broken ORB that expects a particular wide character code set and does not correctly support the negotiation of wide character code sets.

ooc.orb.giop.max_message_size

Value: *max >= 0*

Specifies the maximum GIOP message size in bytes. If set to 0, no maximum message size will be used. If a message is sent or received that exceeds the maximum size, the ORB will raise the IMP_LIMIT system exception.

ooc.orb.id

Value: *id*

Specifies the identifier of the ORB to be used by the application.

ooc.orb.init_iiop

Value: `none`, `client`, `server`, `both`

Specifies to what extent the ORB should initialize the IIOP protocol plug-in. The default value is `both`. For security reasons, applications may wish to disable the client and/or server capabilities of the plug-in. Note that the values `server` and `both` do not require that the application must be a server. This property simply determines which protocol services are installed. A value of `none` will typically be used when a different protocol plug-in is being used and the IIOP plug-in is unnecessary.

ooc.orb.native_cs

Value: *string*

Specifies the native character code set for the ORB. The default is ISO 8859-1.

ooc.orb.native_wcs

Value: *string*

Specifies the native wide character code set for the ORB. The default is UTF-16.

ooc.orb.raise_dii_exceptions

Value: `true`, `false`

Determines whether system exceptions that occur during Dynamic Invocation Interface (DII) operations are raised immediately or are stored only in the `CORBA::Environment` object. This property is only available for Java applications. The default value is `false` for JDK 1.1.x, and `true` for later JDK versions. Note that specifying a value of `false` when using JDK 1.2 or later may result in unexpected behavior.

ooc.orb.server_name

Value: *string*

Specifies the name of the server, as registered with the Implementation Repository (IMR). Note that you should not put this property in a configuration file that is shared by several IMR-enabled servers. Furthermore, this property should not be specified for servers that are not registered with the IMR.

ooc.orb.server_shutdown_timeout

Value: *timeout >= 0*

If the server is not able to gracefully disconnect from the client in *timeout* seconds, a connection shutdown is forced. If this property is set to zero, then the server will not force a connection shutdown. If the property is not set, a default timeout value of two seconds is used.

ooc.orb.server_timeout

Value: *timeout >= 0*

The server actively closes a connection that has been idle for *timeout* seconds once that connection has no more outstanding replies. Note that the application must use one of the threaded server-side concurrency model if connection timeouts are desired. If this property is set to zero, or not set at all, then the server does not close idle connections.

ooc.orb.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 an IOR or URL. For example, the property `ooc.orb.service.NameService` adds "NameService" to the list of initial services. See "Initial Services" on page 108 for more information.

ooc.orb.trace.connections

Value: *level >= 0*

Defines the output level for diagnostic messages printed by ORBACUS that are related to connection establishment and closure. A level of 1 or higher produces information about

connection events, and a level of 2 or higher produces code set exchange information. The default level is 0, which produces no output.

ooc.orb.trace.retry

Value: *level >= 0*

Defines the output level for diagnostic messages printed by ORBACUS that are related to transparent re-sending of failed messages. A level of 1 or higher produces information about re-sending of messages, and a level of 2 or higher also produces information about use of individual IOR profiles. The default level is 0, which produces no output.

4.3.2 OA Properties

Configuring an object adapter is achieved by setting properties on POA Managers. These properties are grouped into two categories: global properties, and properties specific to a particular POA Manager. Global properties have the prefix `ooc.orb.oa`, while properties specific to a particular POA Manager have the prefix `ooc.orb.poamanager.`*`name`*, where *name* is the name of the POA Manager (see "Using POA Managers" on page 65).

Unless otherwise noted, a POA Manager will search for configuration properties using the following algorithm:

- First, use properties defined specifically for that POA Manager
- Next, use global properties
- Finally, use default settings.

See "Using POA Managers" on page 65 for more information on POA Managers.

ooc.orb.oa.conc_model

Value: `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 16 for more information on concurrency models. If this property is set to `thread_pool`, then the property `ooc.orb.oa.thread_pool` determines how many threads are in the pool.

This property is also used to determine the default value of the communications concurrency model for POA Managers (see `ooc.orb.poamanager.`*`manager`*`.conc_model` below). If the value of `ooc.orb.oa.conc_model` is `reactive`, the default value for the

communications concurrency model is `reactive`, otherwise the default value is `threaded`.

ooc.orb.oa.host

Deprecated. See `ooc.iiop.host`.

ooc.orb.oa.numeric

Deprecated. See `ooc.iiop.numeric`.

ooc.orb.oa.port

Deprecated. See `ooc.iiop.port`.

ooc.orb.oa.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.orb.oa.conc_model` property has the value `thread_pool`.

ooc.orb.oa.version

Value: `1.0, 1.1 or 1.2`

Specifies the GIOP version to be used in object references. The default value is `1.2`. This option is useful for backward compatibility with older ORBs that reject object references using a newer version of the protocol.

ooc.orb.poamanager.*manager*.conc_model

Value: `reactive`, `threaded`

Specifies the communications concurrency model used by the POA Manager with name *manager*. The default value is determined by `ooc.orb.oa.conc_model`. See Chapter 16 for more information on concurrency models.

ooc.orb.poamanager.*manager*.host

Deprecated. See `ooc.iiop.acceptor.`*`manager`*`.host`.

ooc.orb.poamanager.*manager*.numeric

Deprecated. See `ooc.iiop.acceptor.`*`manager`*`.numeric`.

ooc.orb.poamanager.*manager*.port

Deprecated. See `ooc.iiop.acceptor.`*`manager`*`.port`.

ooc.orb.poamanager.*manager*.version

Value: `1.0, 1.1 or 1.2`

Specifies the GIOP version to be used in object references generated by a particular POA Manager. This option is useful for backward compatibility with older ORBs that reject object references using a newer version of the protocol. The default value is determined by the value of `ooc.orb.oa.version`.

4.3.3 IIOP Properties

An application can configure IIOP endpoints with properties having the prefix `ooc.iiop`. As with the object adapter properties described in the previous section, IIOP properties are also grouped into global and POA Manager-specific categories. See "Using POA Managers" on page 65 for more information on POA Managers and their relationship to IIOP endpoints.

ooc.iiop.backlog

Value: *0 <= backlog <= 65535*

Specifies the length of the queue for incoming connection requests. Note that the operating system may override this setting if the value exceeds the maximum allowed by the operating system.

ooc.iiop.bind

Value: *hostname or IP address*

Specifies that the server should bind its socket to a specific network interface If not specified, the server will bind its socket to all available network interfaces. This property is useful in situations where a host has several network interfaces, but the server should only listen for connections on a particular interface.

ooc.iiop.host

Value: *host[,host,...]*

Explicitly defines the hostname(s) and/or IP address(es) to be used in generated object references. The default value is the canonical hostname of the machine. If multiple hostnames are specified, the value of `ooc.iiop.multi_profile` determines how those hostnames are represented in object references.

ooc.iiop.multi_profile

Value: `true`, `false`

Specifies how multiple addresses should be represented in an object reference. If true, each address is represented as a separate "tagged profile" in the object reference. If false, there will be only one tagged profile, with the first address used as the primary address of the profile, and all other addresses represented as alternate addresses in that profile. The default representation depends on the protocol version in use. If the protocol version is `1.0`, the default value is `true`, otherwise the default value is `false`. As far as ORBACUS is concerned, the ORB behavior is identical for both representations, but other ORBs may require a particular representation.

ooc.iiop.numeric

Value: `true`, `false`

If true, object references are generated using the internet (IP) address in dotted decimal notation instead of the canonical hostname. The default value is `false`. This property is ignored if `ooc.iiop.host` is specified.

ooc.iiop.port

Value: *0 < port <= 65535*

Specifies the port number on which the Root POA Manager should listen for new connections. If no port is specified, one will be selected automatically by the server. 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.iiop.acceptor.*manager*.backlog

Value: *0 <= backlog <= 65535*

Specifies the length of the queue for incoming connection requests on the POA Manager with name *manager*. Note that the operating system may override this setting if the value exceeds the maximum allowed by the operating system.

ooc.iiop.acceptor.*manager*.bind

Value: *hostname or IP address*

Specifies that the POA Manager with name *manager* should bind its socket to a specific network interface If not specified, the POA Manager will bind its socket to all available network interfaces. This property is useful in situations where a host has several network interfaces, but the POA Manager should only listen for connections on a particular interface.

ooc.iiop.acceptor.*manager*.host

Value: *hostname(s) and/or IP address(es)*

Explicitly defines the hostname(s) to be used in object references generated by the POA Manager with name *manager*. The default value is determined by the value of `ooc.iiop.host`.

ooc.iiop.acceptor.*manager*.multi_profile

Value: `true`, `false`

Specifies how multiple addresses should be represented in an object reference generated by the POA Manager with name *manager*. If true, each address is represented as a separate "tagged profile" in the object reference. If false, there will be only one tagged profile, with the first address used as the primary address of the profile, and all other addresses represented as alternate addresses in that profile. The default representation depends on the protocol version in use. If the protocol version is `1.0`, the default value is `true`, otherwise the default value is `false`. As far as ORBACUS is concerned, the ORB behavior is identical for both representations, but other ORBs may require a particular representation.

ooc.iiop.acceptor.*manager*.numeric

Value: `true`, `false`

If true, the POA Manager with name *manager* will generate object references that contain an internet (IP) address in dotted decimal notation instead of the canonical hostname. The default value is determined by the value of `ooc.iiop.numeric`. This property is ignored if either `ooc.iiop.host` or `ooc.iiop.acceptor.`*`manager`*`.host` is specified.

ooc.iiop.acceptor.*manager*.port

Value: *0 < port <= 65535*

Specifies the port number on which the POA Manager with name *manager* should listen for new connections. If no port is specified, one will be selected automatically by the server.

4.3.4 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 "ORB Properties" on page 49 for a description of the properties.

Option	Property
`-OAbacklog` *backlog*	`ooc.iiop.backlog=`*backlog*
`-OAbind` *host*	`ooc.iiop.bind=`*host*
`-OAhost` *host[,host,...]*	`ooc.iiop.host=`*host[,host,...]*
`-OAnumeric`	`ooc.iiop.numeric=true`
`-OAport` *port*	`ooc.iiop.port=`*port*
`-OAreactive`	`ooc.orb.oa.conc_model=reactive`
`-OAthreaded`	`ooc.orb.oa.conc_model=threaded`
`-OAthread_per_client`	`ooc.orb.oa.conc_model=thread_per_client`
`-OAthread_per_request`	`ooc.orb.oa.conc_model=thread_per_request`
`-OAthread_pool` *n*	`ooc.orb.oa.conc_model=thread_pool` `ooc.orb.oa.thread_pool=`*n*
`-OAversion` *version*	`ooc.orb.oa.version=`*version*
`-ORBblocking`	`ooc.orb.conc_model=blocking`
`-ORBDefaultInitRef` *URL*	`ooc.orb.default_init_ref=`*URL*
`-ORBid` *id*	`ooc.orb.id=`*id*
`-ORBInitRef` *name=ior*	`ooc.orb.service.`*name*`=`*ior*
`-ORBnative_cs` *name*	`ooc.orb.native_cs=`*name*

Table 4.1: Command-line Options

Option	Property
`-ORBnative_wcs` *name*	`ooc.orb.native_wcs=`*name*
`-ORBnaming` *ior*	`ooc.orb.service.NameService=`*ior*
`-ORBproperty` *name=value*	*name=value*
`-ORBreactive`	`ooc.orb.conc_model=reactive`
`-ORBrepository` *ior*	`ooc.orb.service.InterfaceRepository=`*ior*
`-ORBserver_name` *string*	`ooc.orb.server_name=`*string*
`-ORBservice` *name ior*	`ooc.orb.service.`*name*`=`*ior*
`-ORBthreaded`	`ooc.orb.conc_model=threaded`
`-ORBtrace_connections` *level*	`ooc.orb.trace.connections=`*level*
`-ORBtrace_retry` *level*	`ooc.orb.trace.retry=`*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
`-ORBconfig` *filename*	Causes the ORB to load the configuration file specified by *filename*.
`-ORBversion`	Causes the ORB to print its version to standard output.

Table 4.2: Additional Command-line Options

4.3.5 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.orb.oa.conc_model=thread_pool
ooc.orb.oa.thread_pool=5

# Initial services
```

```
ooc.orb.service.NameService=corbaloc::myhost:5000/NameService
ooc.orb.service.EventService=corbaloc::myhost:5001/DefaultEventChannel
ooc.orb.service.TradingService=corbaloc::myhost:5002/TradingService
```

Note that trailing blanks are *not* ignored but are a part of the property.

You can define the name of the configuration file[1] 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*

When an ORB is initialized, it first checks for the presence of the environment variable or system property. If present, the ORB loads the configuration file. Next, the ORB loads the configuration file specified by the `-ORBconfig` option. Therefore, the properties loaded from the file specified by `-ORBconfig` will override any existing properties, including those loaded by a configuration file specified in the environment variable or system property. See “Precedence of Properties” on page 63 for more information.

Configuration files are only loaded during ORB initialization. Changes made to a configuration file after an ORB has been initialized have no effect on that ORB.

4.3.6 Using the Windows NT Registry

Another convenient mechanism for use with C++ applications under Windows NT is to use the system registry[2]. Properties can be stored in the registry under the following registry keys:

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

1. ORBACUS for Java also accepts a URL specification as the filename.
2. 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.iiop.port` property is defined on a global basis.

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

```
Software\OOC\Properties\ooc\orb\trace
```

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 ORBACUS configuration file are transferred to the registry.

Synopsis

```
RegUpdate HKEY_LOCAL_MACHINE|HKEY_CURRENT_USER config-file
```

Example:

```
RegUpdate HKEY_LOCAL_MACHINE ob.conf
```

This command reads the properties defined in the file `ob.conf` and writes the values under the following registry key:

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

4.3.7 Defining Properties

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 initialization.

For example:

```
1 // Java
2 java.util.Properties props = System.getProperties();
3 props.put("ooc.orb.conc_model", "threaded");
4 props.put("ooc.iiop.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.iiop.port=5000 MyServer
```

You can also use the `java.util.Properties` object that is passed to the `ORB.init()` method to provide ORBACUS property definitions:

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

2 Create a `java.util.Properties` object to hold our properties.

3 Define ORBACUS properties.

4,5 Initialize the ORB using the `java.util.Properties` object.

Properties in C++

In C++, the ORBACUS-specific class `OB::Properties` can be used to define properties:

```
// C++
class Properties
{
   // ...
public:
   Properties();
   Properties(Properties_ptr p);
   ~Properties();

   static Properties_ptr _duplicate(Properties_ptr p);
   static Properties_ptr _nil();

   static Properties_ptr getDefaultProperties();

   void setProperty(const char* key, const char* value);
   const char* getProperty(const char* key) const;

   // ...
};
```

For example, to add the threaded concurrency model to a property set that is used to initialize the ORB:

```
1 // C++
2 OB::Properties_var dflt = OB::Properties::getDefaultProperties();
3 OB::Properties_var props = new OB::Properties(dflt);
4 props -> setProperty("ooc.orb.conc_model", "threaded");
5 CORBA::ORB_var orb = OBCORBA::ORB_init(argc, argv, props);
```

2-3 Create an `OB::Properties` object that is based on the default properties. This is important because, unlike `org.omg.CORBA.ORB.init`, `OBCORBA::ORB_init` does not read the default properties if the property parameter is not null.

4 Define ORBACUS properties.

5 Initialize the ORB using the ORBACUS-specific `OBCORBA::ORB_init` operation.

4.3.8 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. Configuration file specified at the command-line
3. User-supplied properties
4. Configuration file specified by the `ORBACUS_CONFIG` environment variable (C++) or the `ooc.config` system property (Java)
5. System properties (Java only)
6. `HKEY_CURRENT_USER\Software\OOC\Properties` (Windows NT/C++ only)
7. `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.

4.3.9 Advanced Property Usage

With the methods for ORB initialization discussed in the previous sections, the command-line arguments are not processed until a call to `CORBA::ORB_init` (C++), `OBCORBA::ORB_init` (C++), or `org.omg.CORBA.ORB.init` (Java). Hence, the set of properties that will be used by the ORB is not available until after the ORB is initialized. This poses a problem if the properties need to be validated prior to ORB initialization.

If you need access to an ORB's property set before it is initialized, then you may elect to use the ORBACUS-specific operations `OB::ParseArgs` (C++) or `com.ooc.CORBA.ORB.ParseArgs` (Java). The following examples check the value of the `ooc.orb.conc_model` property to ensure that it is set to `reactive` or `threaded`. If not, the code chooses the `threaded` concurrency model.

```
1  // C++
2  OB::Properties_var dflt = OB::Properties::getDefaultProperties();
3  OB::Properties_var props = new OB::Properties(dflt);
4  OB::ParseArgs(argc, argv, props, OB::Logger::_nil());
5  const char* orbModel = props -> getProperty("ooc.orb.conc_model");
6  if(strcmp(orbModel, "threaded") != 0 &&
7     strcmp(orbModel, "reactive") != 0)
8  {
9     props -> setProperty("ooc.orb.conc_model", "threaded");
10 }
11 CORBA::ORB_var orb = OBCORBA::ORB_init(argc, argv, props);
```

2-3 Create an `OB::Properties` object that is based on the default properties.

4 Initialize the properties for the ORB. After invoking `OB::ParseArgs`, `props` contains the ORB properties and `argv` no longer contains any `-ORB` or `-OA` command-line arguments. The `OB::ParseArgs` operation takes an optional `Logger` object, which `ParseArgs` will use to display any warning or error messages. In this example, a custom `Logger` object is not used, so the code passes a nil value.

5-10 Retrieve the `ooc.orb.conc_model` property and set it to `threaded` if its value is not valid.

11 Initialize the ORB.

```
1  // Java
2  java.util.Properties props = System.getProperties();
3  args = com.ooc.CORBA.ORB.ParseArgs(args, props, null);
4  String orbModel = props.get("ooc.orb.conc_model");
5  if(!orbModel.equals("threaded"))
6  {
7     props.put("ooc.orb.conc_model", "threaded");
8  }
9  org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(arg, props);
```

2 Create a `java.util.Properties` object.

3 Initialize the properties for the ORB. After invoking `com.ooc.CORBA.ORB.ParseArgs`, `props` contains the ORB properties. The return value of `ParseArgs` is a string array with

all `-ORB` and `-OA` arguments removed. As in the C++ example, a `Logger` object is not used.

4-8 Retrieve the `ooc.orb.conc_model` property and set it to `threaded` if its value is not valid.

9 Initialize the ORB.

4.4 Using POA Managers

The CORBA specification states that a POA Manager is used to control the flow of requests to one or more POAs. In ORBACUS, each POA Manager also encapsulates a set of network endpoints on which a server listens for new connections. This design provides applications with a great deal of flexibility:

- endpoints can be activated and deactivated on demand
- a group of endpoints can be controlled using a single POA Manager and serviced by one or more POAs

4.4.1 Creating POA Managers

When the Root POA is first resolved using `resolve_initial_references`, a POA Manager is created to manage the Root POA. Additional POA Managers are created implicitly when a `nil` value for the POA Manager argument is passed to the `create_POA` operation. In this text, we'll refer to POA Managers created in this way as "anonymous" POA Managers.

By default, each new POA Manager creates a single IIOP endpoint (i.e., the POA Manager will listen for new connections on a port). Applications should therefore use caution to ensure that POA Managers are not created unnecessarily.

An application can control the endpoint configuration using configuration properties (see "ORB Properties" on page 49). However, properties cannot be specified for anonymous POA Managers, therefore these POA Managers are created using the default settings. For IIOP, the default settings will cause an anonymous POA Manager to listen for new connections on a port chosen by the operating system. To override this behavior and allow an application to easily configure POA Managers, ORBACUS provides a proprietary factory interface for creating named POA Managers:

```
// IDL
module OBPortableServer
{
interface POAManagerFactory
{
```

```
    struct AcceptorConfig
    {
        OCI::ProtocolId id;
        OCI::ParamSeq params;
    };

    typedef sequence< AcceptorConfig > AcceptorConfigSeq;

    exception POAManagerAlreadyExists
    {
    };

    POAManager create_poa_manager(in string name)
        raises(POAManagerAlreadyExists,
               OCI::InvalidParam);

    POAManager create_poa_manager_with_config(
        in string name,
        in AcceptorConfigSeq config)
        raises(POAManagerAlreadyExists,
               OCI::NoSuchFactory,
               OCI::InvalidParam);

    POAManagerSeq get_poa_managers();

    void destroy();
};
};
```

The example below illustrates how to create and configure a new POA Manager using the ORBACUS POA Manager Factory.

Here is an example in C++:

```
1 // C++
2 CORBA::Object_var obj =
3     orb -> resolve_initial_references("POAManagerFactory");
4 OBPortableServer::POAManagerFactory_var factory =
5     OBPortableServer::POAManagerFactory::_narrow(obj);
6 PortableServer::POAManager_var myPOAManager =
7     factory -> create_poa_manager("MyPOAManager");
```

2-5 Resolve the POA Manager Factory.

6-7 Create a new POA Manager with the name "MyPOAManager".

And in Java:

```
1 // Java
2 org.omg.CORBA.Object obj =
3    orb.resolve_initial_references("POAManagerFactory");
4 com.ooc.OBPortableServer.POAManagerFactory factory =
5    com.ooc.OBPortableServer.POAManagerFactoryHelper.narrow(obj);
6 org.omg.PortableServer.POAManager myPOAManager =
7    factory.create_poa_manager("MyPOAManager");
```

2-5 Resolve the POA Manager Factory.

6-7 Create a new POA Manager with the name "MyPOAManager".

Now we can configure the IIOP port used by our new POA Manager with the following property:

```
ooc.iiop.acceptor.MyPOAManager.port=5001
```

When experimenting with various endpoint configurations, it can be very useful to enable connection tracing diagnostics. With diagnostics enabled, the ORB will display its endpoint information, allowing you to confirm that the application's endpoints are configured correctly. Diagnostics can be enabled using the `-ORBtrace_connections` command-line option, or using the equivalent property `ooc.orb.trace.connections`. See "Configuring the ORB and Object Adapter" on page 49 for more information on the supported configuration properties and command-line options.

4.4.2 The Root POA Manager

As its name suggests, the Root POA Manager is the POA Manager of the Root POA. The Root POA Manager is usually created by the ORB when `resolve_initial_references` is first called for the Root POA. However, an application can also manually create the Root POA Manager if a special configuration is desired. This capability is useful in that it allows the application to use the Root POA with customized endpoints.

Note that an application must create the Root POA Manager before resolving the Root POA. Otherwise, the ORB will have already created the Root POA Manager and the application will receive an exception when it attempts to create the Root POA Manager.

When using the POA Manager Factory, the name of the Root POA Manager is always "RootPOAManager".

4.4.3 Dispatching Requests

As explained in [4], a POA Manager is initially in the "holding" state, where incoming requests on the POA Manager's endpoints are queued. To dispatch requests, the POA Manager must be activated using the `activate()` operation.

4.4.4 Callbacks

In mixed client/server applications in which callbacks occur, it is important to remember that callbacks will not be dispatched until the POA Manager has been activated. If the POA Manager has not been activated, the application will likely hang. In general, applications should activate the POA Manager prior to making any request that might result in a callback.

4.4.5 Advanced Configuration Example

The `create_poa_manager` operation demonstrated in section 4.4.1 is a simple way to create a POA Manager with a single endpoint. In some architectures, however, it may be more appropriate to configure a POA Manager with multiple endpoints.

Using the `create_poa_manager_with_config` operation, the application can instruct the POA Manager Factory to create a POA Manager with a specific endpoint configuration. In terms of the ORBACUS Open Communications Interface (OCI), the application is configuring the POA Manager's *acceptors* (see Chapter 17 for more information on the OCI). The example below illustrates how to configure the Root POA Manager with two acceptors.

```
1  // C++
2  #include <OB/CORBA.h>
3  #include <OB/OCI_IIOP.h>
4  ...
5  OBPortableServer::POAManagerFactory_var factory = ... // resolve
6  OBPortableServer::POAManagerFactory::AcceptorConfigSeq config;
7  config.length(2);
8  config[0].id = OCI::IIOP::TAG_IIOP;
9  config[0].params.length(1);
10 config[0].params[0].name = CORBA::string_dup("port");
11 config[0].params[0].value <<= (CORBA::UShort)3000;
12 config[1].id = OCI::IIOP::TAG_IIOP;
13 config[1].params.length(1);
14 config[1].params[0].name = CORBA::string_dup("port");
15 config[1].params[0].value <<= (CORBA::UShort)3001;
16 PortableServer::POAManager_var manager =
```

```
17     factory -> create_poa_manager_with_config("RootPOAManager",
18                                               config);
19 PortableServer::RootPOA_var rootPOA =
20     orb -> resolve_initial_references("RootPOA");
```

5 Resolve the ORBACUS POA Manager Factory. See section 4.4.1 for an example.

6-7 Initialize a configuration sequence with two elements.

8-11 Configure an IIOP acceptor to listen on port 3000.

12-15 Configure an IIOP acceptor to listen on port 3001.

16-18 Create the Root POA Manager.

19-20 The Root POA Manager must be created before resolving the Root POA.

```
1  // Java
2  import com.ooc.OBPortableServer.POAManagerFactory;
3  import com.ooc.OBPortableServer.POAManagerFactoryPackage.*;
4  ...
5  POAManagerFactory factory = ... // resolve
6  AcceptorConfig[] config = new AcceptorConfig[2];
7  config[0] = new AcceptorConfig();
8  config[0].id = com.ooc.OCI.IIOP.TAG_IIOP.value;
9  config[0].params = new com.ooc.OCI.Param[1];
10 config[0].params[0] = new com.ooc.OCI.Param();
11 config[0].params[0].name = "port";
12 config[0].params[0].value = orb.create_any();
13 config[0].params[0].value.insert_ushort((short)3000);
14 config[1] = new AcceptorConfig();
15 config[1].id = com.ooc.OCI.IIOP.TAG_IIOP.value;
16 config[1].params = new com.ooc.OCI.Param[1];
17 config[1].params[0] = new com.ooc.OCI.Param();
18 config[1].params[0].name = "port";
19 config[1].params[0].value = orb.create_any();
20 config[1].params[0].value.insert_ushort((short)3001);
21 org.omg.PortableServer.POAManager manager =
22     factory.create_poa_manager_with_config("RootPOAManager",
23                                            config);
24 org.omg.PortableServer.POA rootPOA =
25     orb.resolve_initial_references("RootPOA");
```

5 Resolve the ORBACUS POA Manager Factory. See section 4.4.1 for an example.

6 Initialize a configuration sequence with two elements.

7-13 Configure an IIOP acceptor to listen on port 3000.

14-20 Configure an IIOP acceptor to listen on port 3001.

21-23 Create the Root POA Manager.

24-25 The Root POA Manager must be created before resolving the Root POA.

4.5 ORB Destruction

4.5.1 Destroying the C++ ORB

Applications must destroy the ORB before returning from `main` so that resources used by the ORB are properly released. To destroy the ORB, invoke `destroy` on the ORB:

```
// C++
CORBA::ORB_var orb = // Initialize the orb
// ...
orb -> destroy();
```

4.5.2 Destroying the Java ORB

As in C++, Java must destroy the ORB so that resources are properly released. In Java, the ORB is destroyed as follows:

```
// Java
org.omg.CORBA.ORB orb = // Initialize the orb
// ...
orb.destroy();
```

In JDK 1.2, the ORB interface does not define the standard `destroy` method. To work around this problem, an application can be compiled using the `-Xbootclasspath` option to ensure that the ORBACUS classes (i.e., `OB.jar`) are located before the JDK's runtime classes. Alternatively, an application can cast the ORB reference to the type `com.ooc.CORBA.ORB` when calling `destroy`, as shown below:

```
// Java
org.omg.CORBA.ORB orb = // Initialize the orb
// ...
((com.ooc.CORBA.ORB)orb).destroy();
```

Note that using this cast is ORBACUS-specific, therefore an application wishing to remain strictly portable must use a workaround such as `-Xbootclasspath`. For example:

```
javac -Xbootclasspath/p:/path/to/OB.jar MyApp.java
```

The `/p` suffix indicates that the ORBACUS classes should be prepended to the JVM's boot classpath. See the JDK documentation for more information.

4.6 Server Event Loop

A server's event loop is entered by calling `POAManager::activate` on each POA Manager, and then calling `ORB::run`. For example, in Java:

```
// Java
org.omg.CORBA.ORB orb = ... // Initialize the orb
org.omg.PortableServer.POAManager manager = ... // Get Root POA manager
manager.activate();
orb.run();
```

And in C++:

```
// C++
CORBA::ORB_var orb = ... // Initialize the orb
PortableServer::POAManager_var manager = ... // Get the Root POA manager
manager -> activate();
orb -> run();
```

You can deactivate a server by calling `ORB::shutdown`, which causes `ORB::run` to return. For example, consider a server that 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  class ShutdownObject_impl :
3      public POA_ShutdownObject,
4      public PortableServer::RefCountServantBase
5  {
6      CORBA::ORB_var orb_;
7
8  public:
9
10     ShutdownObject_impl(CORBA::ORB_ptr orb)
11         : orb_(CORBA::ORB::_duplicate(orb))
12     {
```

```
13      }
14
15      virtual void deactivate() throw(CORBA::SystemException)
16      {
17          orb_ -> shutdown(false);
18      }
19  };
```

2-3 A servant class for `ShutdownObject` is defined. For more information on how to implement servant classes, see Chapter 5.

5 An ORB is needed to call `shutdown`.

9-12 The constructor initializes the ORB member.

14-17 `deactivate` calls `shutdown` on the ORB. Note that `shutdown` is called with the argument `false` to avoid a deadlock. A `false` argument instructs `shutdown` to terminate request processing without waiting for executing operations to complete. A `true` argument instructs `shutdown` to return only once all operations have completed. If `shutdown` were to be called with a `true` argument in this example, it would deadlock. That is because `shutdown(true)` would be invoked from within an operation and, therefore, could not ever return.

The client can use the `deactivate` call as shown below:

```
// C++
ShutdownObject_var shutdownObj = ... // Get a reference somehow
shutdownObj -> deactivate();
```

4.7 Applets

4.7.1 Compatibility with Netscape

The Java mapping in ORBACUS 4 uses the portable stream-based stubs as specified in [6]. These stubs are not compatible with Netscape's built-in ORB. Therefore, in order to write applets using ORBACUS 4, you will need to disable or replace Netscape's built-in ORB.

The Netscape ORB classes are contained in the file `iiop10.jar`, located in the `java/classes` subdirectory of the Netscape installation. To disable the ORB, rename this file (e.g., to `iiop10_old.jar`). With the ORB disabled, applets can download the ORBACUS classes along with the applet.

To replace the built-in ORB with ORBACUS, rename the `iiop10.jar` file as described above, and copy the `OB.jar` file to the same directory, giving it the name `iiop10.jar`. For example:

```
cd <netscape-home>/java/classes
mv iiop10.jar iiop10_old.jar
cp <orbacus-home>/lib/OB.jar iiop10.jar
```

4.7.2 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());
   // ...
}
```

4.7.3 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.

4.7.4 Defining ORB 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 `ORBparams` parameter, whose value 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:

```
<APPLET CODE="Client.class" ARCHIVE="OB.jar" WIDTH=500 HEIGHT=300>
   <PARAM NAME="ORBparams" VALUE="-ORBconfig http://www/orb.cfg">
```

```
</APPLET>
```

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 "ORB Properties" on page 49 for more information.

4.7.5 Defining the ORB Class Parameters

Some Web browsers[1] have a built-in ORB. In order to use ORBACUS instead of this built-in ORB, you must set the following applet parameters:

```
<APPLET CODE="Client.class" ARCHIVE="OB.jar" WIDTH=500 HEIGHT=300>
    <PARAM NAME="org.omg.CORBA.ORBClass"
           VALUE="com.ooc.CORBA.ORB">
    <PARAM NAME="org.omg.CORBA.ORBSingletonClass"
           VALUE="com.ooc.CORBA.ORBSingleton">
</APPLET>
```

4.7.6 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.

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 using the POA.

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

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 implementation object is called a *servant*.
- A *client* that wants to make use of an object 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 remote object 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 *Portable Object Adapter (POA)*, as shown in Figure 5.1.

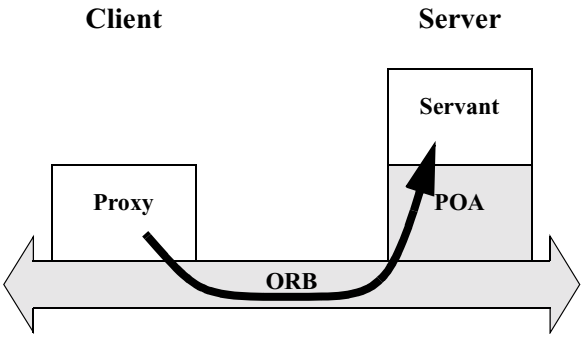


Figure 5.1: Servants, Proxies and the Object Adapter

The Portable Object Adapter in ORBACUS replaces the deprecated "Basic Object Adapter" (BOA). (The BOA was deprecated by the OMG because it had a number of serious deficiencies and was under-specified.) The POA is a far more flexible and powerful object adapter than the BOA. The POA not only allows you to write code that is portable among ORBs from different vendors, it also provides a number of features that are essential for building high-performance and scalable servers.

5.2 Implementing Servants

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

```
1  // IDL
2  interface A
3  {
4      void op_a();
5  };
6
7  interface B
8  {
9      void op_b();
10 };
11
12 interface I : A, B
13 {
14     void op_i();
15 };
```

2-5 An interface `A` is defined with the operation `op_a`.

7-10 An interface `B` is defined with the operation `op_b`.

12-15 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 `POA_I` 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  class A_impl : virtual public POA_A
3  {
4  public:
5      virtual void op_a() throw(CORBA::SystemException);
6  };
7
8  class B_impl : virtual public POA_B
9  {
10 public:
11     virtual void op_b() throw(CORBA::SystemException);
12 };
13
14 class I_impl : virtual public POA_I,
15                virtual public A_impl,
16                virtual public B_impl
17 {
18 public:
19     virtual void op_i() throw(CORBA::SystemException);
20 };
```

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

2-6 The servant class `A_impl` is defined, inheriting from the skeleton class `POA_A`. If `op_a` had any parameters, these parameters would be mapped according to the standard IDL-to-C++ mapping rules [4].

8-13 This is the servant class for `B_impl`.

14-20 The servant class for `I_impl` is not only derived from `POA_I`, 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 `POA_I` 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 class I_impl : virtual public POA_I
3 {
4 public:
5    virtual void op_a() throw(CORBA::SystemException);
6    virtual void op_b() throw(CORBA::SystemException);
7    virtual void op_i() throw(CORBA::SystemException);
8 };
```

2 Now `I_impl` is only derived from `POA_I`, but not from the other servant classes.

5-7 `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
- `IPOA.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, except when

using delegation-based implementation. 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  public class I_impl extends IPOA
3  {
4      public void op_a()
5     {
6     }
7
8      public void op_b()
9     {
10    }
11
12     public void op_i()
13    {
14    }
15 }
```

2-15 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 overly complex inheritance hierarchies (for example, because of use of an existing class library that requires extensive use of inheritance). 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.[1]

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.

1. Note that tie classes are rarely necessary. Not only is the inheritance implementation less complex, but it also avoids a number of problems that arise with the life cycle of objects, particularly in threaded servers. We suggest that you use the tie approach only if you have no other option.

For the interface `I` from the C++ example above, the template `POA_I_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 for the class implementing `I`'s operations, i.e., `I_impl_tie`, to be derived from a skeleton class. Instead, an instance of `POA_I_tie` delegates all operation calls to `I_impl_tie`, as shown in Figure 5.2.

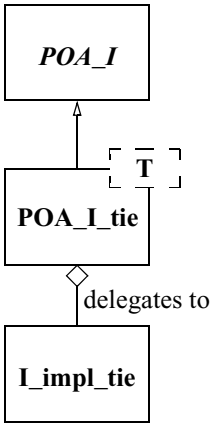


Figure 5.2: Class Hierarchy for Delegation Implementation in C++

Here is our definition of `I_impl_tie`:

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

2 `I_impl_tie` is defined and not derived from any other class.

5-7 `I_impl_tie` must implement all of `I`'s operations, including inherited operations.

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

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

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

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

The tie template generated by the IDL compiler contains functions that permit you change the instance denoted by the tie:

```
1  // C++
2  template<class T>
3  class POA_I_tie : public POA_I
4  {
5  public:
6      // ...
7      T* _tied_object();
8      void _tied_object(T& obj);
9      void _tied_object(T* obj, CORBA::Boolean release = true);
10     // ...
11 }
```

7-9 The `_tied_object` function permits you to retrieve and change the implementation instance that is currently associated with the tie. The first modifier function calls `delete` on the current tied instance before accepting the new tied instance if the `release` flag is currently true; the `release` flag for the new tied instance is set to false. The second modifier function also calls `delete` on the current tied instance before accepting the new instance but sets the `release` flag to the passed value.

Delegation using Java

For every IDL interface, the IDL-to-Java mapping generates an “operations” interface containing methods for the IDL attributes and operations. This operations interface is also used to support delegation-based servant implementation. For an interface `I`, the following additional class is generated:

- `IPOATie.java`, the tie class that inherits from `IPOA` 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  public class I_impl_tie implements IOperations
3  {
4      public void op_a()
5      {
6      }
7
8      public void op_b()
9      {
10     }
11
12     public void op_i()
13     {
14     }
15 }
```

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

4-14 `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.

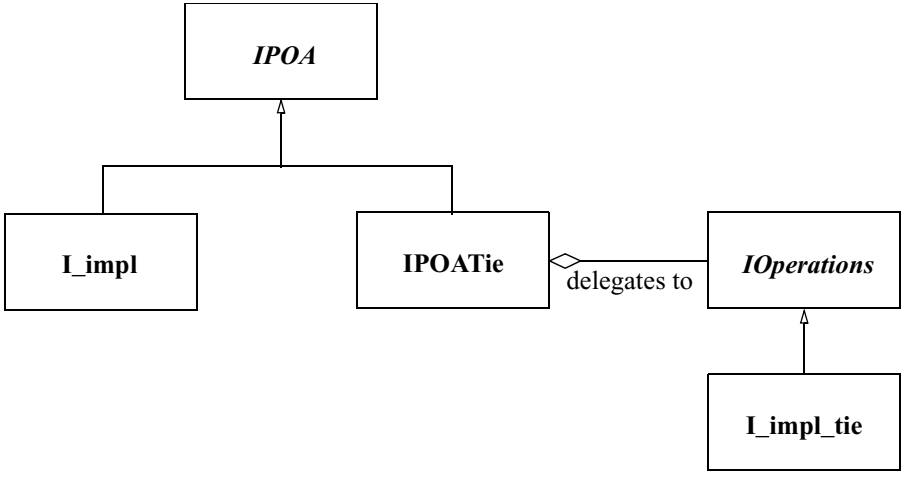


Figure 5.3: Class Hierarchy for Inheritance and Delegation Implementation in Java

As noted earlier, Java's lack of multiple inheritance makes it impossible to inherit an implementation from another servant class. Using tie classes, however, does allow implementation inheritance, but only in certain situations.

For example, let's implement each of our sample interfaces using delegation.

```
1  // Java
2  public class A_impl implements AOperations
3  {
4      public void op_a()
5      {
6      }
7  }
8
9  public class B_impl implements BOperations
10 {
11     public void op_b()
12     {
13     }
14 }
15
16 public class I_impl extends B_impl implements IOperations
17 {
18     public void op_a()
19     {
20     }
21
22     public void op_i()
23     {
24     }
25 }
```

2-7 Class `A_impl` is defined as implementing `AOperations`.

9-14 Class `B_impl` is defined as implementing `BOperations`.

16-21 Class `I_impl` inherits the implementation of `op_b` from `B_impl`, and provides an implementation of `op_a` and `op_i`. Since a Java class can only extend one class, it's not possible for `I_impl` to inherit the implementations of both `op_a` and `op_b`.

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`.[1]

5.3.1 Creating Servants using C++

Here is how to create servants using C++:

```
1 // C++
2 I_impl* servant_pointer = new I_impl;
3 I_impl* another_servant_pointer = new I_impl;
```

2,3 Two servants are created with `new`. Note that this merely instantiates the servants but does not inform the ORB that these servants exist yet. The ORB server-side run time only learns of the existence of the servants once you activate them.

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 POA_I_tie< I_impl_tie >* tie_pointer =
4     new POA_I_tie< I_impl_tie >(impl);
```

2 A new `I_impl_tie` is created with `new`.

3,4 An instance of `POA_I_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 the tie is destroyed, `delete impl` is called by the tie's destructor. In case you don't want the lifetime of `impl` to be coupled to the lifetime of the tie, for example, because you want to create a servant on the stack and not on the heap (making it illegal to call `delete` on the tie), use the following code:

```
1 // C++
2 I_impl_tie impl;
3 POA_I_tie< I_impl_tie >* tie =
4     new POA_I_tie< I_impl_tie >(&impl, false);
```

2 A new `I_impl_tie` is created, this time on the stack, not on the heap.

3,4 An instance of `POA_I_tie` is created. The `false` parameter tells `tie` not to call `delete` on `impl`.

1. You can also instantiate servants on the stack. However, this only works only for some POA policies, so servants are usually instantiated on the heap.

5.3.2 Creating Servants using Java

Every tie class generated by the IDL-to-Java translator has two constructors:

```
// Java
public class IPOATie extends IPOA
{
    public IPOATie(IOperations delegate) { ... }
    public IPOATie(IOperations delegate, POA poa) { ... }
    ...
}
```

The second constructor allows a POA instance to be supplied, which will be used as the return value for the tie's `_default_POA` method. If the POA instance is not supplied, the `_default_POA` method will return the root POA of the ORB with which the tie has been associated.

This example demonstrates how to create servants using Java:

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

2,3 Two servants, `impl` and `anotherImpl`, 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 tie's constructor:

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

2 A new `I_impl_tie` is created.

3 An instance of `IPOATie` is created, taking `impl` as a parameter. All operation calls to `tie` will then be delegated to `impl`.

The tie class also provides methods for accessing and changing the implementation object:

```
1 // Java
2 public class IPOATie extends IPOA
3 {
4     ...
5     public IOperations _delegate() { ... }
6     public void _delegate(IOperations delegate) { ... }
```

```
7     ...
8  }
```

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

6 This method changes the delegate object.

5.4 Activating Servants

Servants must be activated in order to receive requests from clients. Servant activation informs the ORB run time which particular servant represents (or *incarnates*) a particular CORBA object. Activation of a servant assigns an *object identifier* to the servant. That object identifier is also embedded in every object reference that is created for an object and serves to link the object reference with its servant.

The POA's `IdAssignmentPolicy` value controls whether object IDs are assigned by the POA or the server application code. The `SYSTEM_ID` policy value directs the ORB to assign a unique object identifier to the CORBA object represented by the servant; the `USER_ID` policy value requires the server application code to supply an ID that must be unique within the servant's POA.

Servants can be activated implicitly or explicitly. Implicit activation takes place when you create the first object reference for a servant. Explicit activation requires a separate API call. Typically, you will use implicit activation for transient objects and explicit activation for persistent objects. The `ImplicitActivationPolicy` controls whether explicit or implicit is in effect. Explicit activation requires the `NO_IMPLICIT_ACTIVATION` policy value on the servant's POA, whereas implicit activation requires the `IMPLICIT_ACTIVATION` policy value.

5.4.1 Implicit Activation of Servants using C++

The following code shows how to implicitly activate a servant:

```
1  // C++
2  I_impl impl;
3  I_var iv = impl -> _this();
```

2 A new servant `impl` is created.

3 The new servant is activated implicitly by calling `_this`.

Note that implicit activation as shown requires the `RETAIN, IMPLICIT_ACTIVATION`, and `SYSTEM_ID` policies on the servant's POA. The servant is activated with the POA that is returned by the servant's `_default_POA` member function. (The default implementa-

tion of `_default_POA` returns the Root POA; if you want servants activated on a different POA, you must override `_default_POA` in the implementation class to return the POA you want to use.)

5.4.2 Implicit Activation of Servants using Java

This is how Java servants are implicitly activated:

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

2 To activate a servant, we need the ORB.

3 A new servant `impl` is created.

4 The new servant is activated (using the POA returned by the servant's `_default_POA` operation).

As shown above, a servant in Java must be associated with an ORB, and cannot be associated with multiple ORBs. The first call to `_this()` must supply the ORB reference; subsequent calls to `_this()` for the same servant can omit the ORB reference.

An alternative way to associate a servant with an ORB is to call the `set_delegate` method defined in `org.omg.CORBA_2_3.ORB`.

```
// Java
org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
((org.omg.CORBA_2_3.ORB)orb).set_delegate(impl);
```

5.4.3 Explicit Activation of Servants using C++

If `NO_IMPLICIT_ACTIVATION` and `SYSTEM_ID` are in effect for a servant's POA, you activate the servant by calling `activate_object`:

```
1 I_impl impl;
2 PortableServer::POA_var poa = impl._default_POA();
3 poa -> activate_object(&impl);
```

1 The code instantiates a servant.

2 To activate a servant, we need the servant's POA.

3 `activate_object` creates a unique ID for the servant.

Once a servant is activated, calls to `_this` on the servant return an object reference that contains the ORB-assigned ID for the object.

If `NO_IMPLICIT_ACTIVATION` and `USER_ID` are in effect for servant's POA, you activate the servant by supplying the ID value as an octet sequence to `activate_object_with_id`:

```
1 I_impl impl;
2 PortableServer::POA_var poa = impl._default_POA();
3 PortableServer::ObjectId_var oid =
4     PortableServer::string_to_ObjectId("MyObjectName");
5 poa -> activate_object_with_id(oid, &impl);
```

3,4 The `string_to_ObjectId` helper function converts a string into an octet sequence.

5 `activate_object_with_id` uses the octet sequence as the object ID for the servant.

You can use any suitable key value as an object ID. Typically, the key will be part of the object's state, such as a social security number. However, you can also use keys that are not directly related to object state, such as database record identifiers. Once the servant is activated, calls to `_this` on the servant return an object reference that contains the ID you assigned to the object.

5.4.4 Explicit Activation of Servants using Java

Servant activation in Java also uses `activate_object` (for `SYSTEM_ID`) and `activate_object_with_id` (for `USER_ID`). With `SYSTEM_ID`, the code looks as follows:

```
1 I_impl impl = new I_impl();
2 orb.omg.PortableServer.POA poa = impl._default_POA();
3 poa.activate_object(impl);
```

For `USER_ID`, you must provide the Object ID:

```
1 I_impl impl = new I_impl();
2 org.omg.PortableServer.POA poa = impl._default_POA();
3 byte[] id = "MyObjectName".getBytes();
4 poa.activate_object_with_id(id, impl);
```

5.5 Deactivating Servants

5.5.1 Deactivation of Servants using C++

A servant can be deactivated. Deactivating a servant breaks the association between the CORBA object and the servant; requests that arrive from clients thereafter result in an `OBJECT_NOT_EXIST` exception (or a `TRANSIENT` exception, if the server is down at the time a request is made).

To deactivate a servant, call the `deactivate_object` member function on the servant's POA:

```
1 // C++
2 PortableServer::POA_var poa = impl._default_POA();
3 PortableServer::ObjectId_var id = poa -> servant_to_id(&impl);
4 poa -> deactivate_object(id);
```

2 The code obtains a reference to the servant's POA by calling `_default_POA`. (This assumes that `_default_POA` is correctly overridden to return the appropriate POA if the servant is not activated with the Root POA.)

3 The call to `servant_to_id` on the servant's POA returns the object ID with which the servant is activated.

4 The call to `deactivate_object` breaks the association between the CORBA object and the servant.

Note that `deactivate_object` returns immediately, even though the servant may still be executing requests, possibly in a number of different threads.

5.5.2 Deactivation of Servants using Java

Deactivation of a servant in Java is analogous to C++:

```
1 // Java
2 org.omg.PortableServer.POA poa = impl._default_POA();
3 byte[] id = poa.servant_to_id(impl);
4 poa.deactivate_object(id);
```

5.5.3 Transient and Persistent Objects

A POA has either the `TRANSIENT` or the `PERSISTENT` policy value. A transient POA generates transient object references. A transient object reference remains functional only for

as long as its POA remains in existence. Once the POA for a transient reference is destroyed, the reference becomes permanently non-functional and client requests on such a reference raise either `OBJECT_NOT_EXIST` or `TRANSIENT` (depending on whether or not the server is running at the time the request is sent). Transient references remain non-functional even if you restart the server and re-create a transient POA with the same name as was used previously. Transient POAs almost always use the `SYSTEM_ID` policy as a matter of convenience (although the combination of `TRANSIENT` and `USER_ID` is legal).

Object references created on a persistent POA continue to be valid beyond the POA's life time. That is, if you create a persistent reference on a POA, destroy the POA, and then recreate that POA again (with the same POA name), the original reference continues to denote the same CORBA object (even if the server was shut down and restarted). Persistent references require the same POA name and object ID to be used to denote the same object. This means that persistent references rely on the combination of `PERSISTENT` and `USER_ID`. `USER_ID` must be used in conjunction with `NO_IMPLICIT_ACTIVATION`, so servants for persistent references are always activated explicitly.

5.6 Factory Objects

It is quite common to use the Factory [2] 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. Note that the factory pattern applies equally to persistent and transient objects.

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:

```
1  // IDL
2  interface Product
3  {
4      void destroy();
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`.

5.6.1 Factory Objects using C++

First, we'll implement the `Product` interface:

```
1  // C++
2  class Product_impl :
3      public virtual POA_Product,
4      public virtual PortableServer::RefCountServantBase
5  {
6  public:
7
8      virtual void destroy() throw(CORBA::SystemException)
9      {
10         PortableServer::POA_var poa = _default_POA();
11         PortableServer::ObjectId_var id = poa -> servant_to_id(this);
12         poa -> deactivate_object(id);
13     }
14 };
```

2-4 The servant class `Product_impl` is defined as an implementation of the `Product` interface. In addition, `Product_impl` inherits from `RefCountServantBase`, which makes the servant reference counted.

8-13 The `destroy()` operation deactivates the servant with the POA. As a result, the POA will release all references it maintains to the servant. Since there are no other references to the servant left, the servant's reference count will drop to zero, and thus the servant is destroyed.

Next, we'll implement the factory:

```
1  // C++
2  class Factory_impl : public virtual POA_Factory
3  {
4  public:
5
6      virtual Product_ptr
7      createProduct() throw(CORBA::SystemException)
8      {
9          Product_impl* impl = new Product_impl(orb_);
10         PortableServer::ServantBase_var servant = impl;
11         PortableServer::POA_var poa = ... // Get servant's POA
12         PortableServer::ObjectId_var id = ... // Assign an ID
13         poa -> activate_object_with_id(id, impl);
14         return impl -> _this();
15     }
16 };
```

2 The servant class `Factory_impl` is defined as an implementation of the `Factory` interface.

9-10 A new reference counted `Product` servant is instantiated. The servant is assigned to a `ServantBase_var`, which decrements the servant's reference count when it goes out of scope.

11-14 Activates the servant and returns an object reference to the client.

It is important to understand how the servant is eventually destroyed. The `RefCountServantBase` class from which the servant inherits implements a reference count. When the servant is instantiated, the `RefCountServantBase` constructor sets this reference count to 1. When the servant is activated with the POA, the POA increases the reference count by at least 1. When the `ServantBase_var` we assigned the servant to goes out of scope, the reference count is decremented by 1. This means that when `createProduct()` returns, only the POA is "holding" a reference to the servant. Later, when the servant is deactivated in `destroy()`, the POA decrements the reference count by exactly the same number it used to increment the reference count upon activation. This causes the reference count to drop to zero, in which case the servant will be implicitly deleted.

Note that whenever the ORB starts to dispatch a request on the servant, the reference count is increased by 1. After request dispatching is finished, the count is decremented by 1. This ensures that a reference counted servant cannot be deleted while a request is executing.

5.6.2 Factory Objects using Java

Here is our Java implementation of the `Product` interface:

```
1 // Java
2 public class Product_impl extends ProductPOA
3 {
4     public void destroy()
5     {
6         byte[] id = _default_POA().servant_to_id(this);
7         _default_POA().deactivate_object(id);
8     }
9 }
```

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

6,7 The `destroy` operation deactivates the servant with the POA. As long as no other references to the servant are held in the server, the object will be eligible for garbage collection.

Here's our implementation of the factory:

```
1  // Java
2  public class Factory_impl extends FactoryPOA
3  {
4      public Product createProduct()
5      {
6          Product_impl result = new Product_impl(orb_);
7          org.omg.PortableServer.POA poa = ... // Get servant's POA
8          byte[] id = ... // Assign an ID
9          poa.activate_object_with_id(id, result);
10         return result._this(orb_);
11     }
12 }
```

2 Servant class `Factory_impl` is defined as an implementation of the `Factory` interface.

4-11 The `createProduct` operation instantiates a new `Product` servant, activates it with the POA, and returns an object reference to the client.

5.6.3 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]

5.6.4 Obtaining the POA for a Servant

As mentioned in the previous sections, every servant inherits a `_default_POA` function from its skeleton class. The default implementation of this function returns the Root POA. If you instantiate servants on anything but the Root POA, you must override the function in the servant; otherwise, calls to `_this` will create incorrect object references. Usually, this involves remembering the POA reference for a servant in a private member variable and returning that reference from a call to `_default_POA`. (If all servants for objects of a particular interface type use the same POA, you can use a static member variable.)

In C++, you can use an approach similar to the following:

```
1  // C++
2  class Product_impl :
3      public virtual POA_Product,
4      public virtual PortableServer::RefCountServantBase
5  {
6      PortableServer::POA_var poa_;
7
8  public:
9      void Product_impl(PortableServer::POA_ptr poa)
10         : poa_(PortableServer::POA::_duplicate(poa))
11     {
12     }
13
14     virtual PortableServer::POA_ptr _default_POA()
15     {
16         return PortableServer::POA::_duplicate(poa_)
```

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.

```
17      }
18  };
```

9-12 The constructor accepts a POA reference and remembers that reference in a private member variable.

14-17 The `_default_POA` function returns the servant's POA.

In Java, the approach is very similar:

```
// Java
public class Product_impl extends ProductPOA
{
    private org.omg.PortableServer.POA poa_;

    public Product_impl(org.omg.PortableServer.POA poa)
    {
        poa_ = poa;
    }

    public org.omg.PortableServer.POA
    _default_POA()
     {
         return poa_;
     }
}
```

5.6.5 Getting the POA for a Currently Executing Request

The ORB provides access to an object of type `PortableServer::Current`:

```
// IDL
module PortableServer
{
    interface Current : CORBA::Current
    {
        exception NoContext { };
        POA get_POA() raises(NoContext);
        ObjectId get_object_id() raises(NoContext);
    };
};
```

This interface provides access to the POA and the object ID for an executing request. Note that these operations must be invoked only from within the context of an executing operation inside a servant; otherwise, they raise `NoContext`. The `Current` object provides a useful way to obtain access to a servant's POA and object ID without having to store the

POA reference in a member variable, at the cost of being accessible only from within an operation implementation. You can obtain a reference to the `Current` object from `resolve_initial_references`. In C++, the code looks something like this:

```
// C++
CORBA::ORB_var orb = ... // Get the ORB somehow
CORBA::Object_var obj =
   orb -> resolve_initial_references("POACurrent");
PortableServer::Current_var current =
   PortableServer::Current::_narrow(obj);
if(!CORBA::is_nil(current))
   ... // Got Current object OK
```

You can keep the reference to the `Current` object in a variable and use it from within any executing operation in a servant. There is no need to "refresh" the `Current` reference for the current operation, not even for threaded servers. The ORB takes care of ensuring that operation invocations on the `Current` object return the correct data.

In Java, the code to obtain the `Current` reference looks like this:

```
// Java
org.omg.CORBA.ORB orb = ... // Get the ORB somehow
org.omg.CORBA.Object obj =
   orb.resolve_initial_references("POACurrent");
org.omg.PortableServer.Current current =
   org.omg.PortableServer.CurrentHelper.narrow(obj);
if(current != null)
   ... // Got Current object OK
```

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 };
```

2-4 An interface `A` is defined.

6-9 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 : public POA_A,
3                public PortableServer::RefCountServantBase
```

```
4  {
5  };
6
7  class B_impl : public POA_B,
8                 public PortableServer::RefCountServantBase
9  {
10    A_impl* a_;
11
12 public:
13
14    B_impl()
15    {
16       a_ = new A_impl();
17    }
18
19    ~B_impl()
20    {
21       a_ -> _remove_ref();
22    }
23
24    virtual A_ptr getA() throw(CORBA::SystemException)
25    {
26       return a_ -> _this();
27    }
28 };
```

2-5 The servant class `A_impl` is defined, which inherits from the skeleton class `POA_A` and the class `RefCountServantBase` which provides a reference counting implementation.

7-28 The servant class `B_impl` inherits from the skeleton class `POA_B` and the reference counting class `RefCountServantBase`.

14-17 An instance of the servant class `A_impl` is created in the constructor for `B_impl`.

19-22 In the destructor for `B_impl`, the reference count for the servant `A_impl` is decremented, which leads to the destruction of the servant.

24-27 `getA` returns an object reference to the `A_impl` servant (implicitly creating and activating the CORBA object if necessary).

In Java, the interfaces can be implemented like this:

```
1  // Java
2  public class A_impl extends APOA
3  {
4  }
```

```
 5
 6 public class B_impl extends BPOA
 7 {
 8     org.omg.CORBA.ORB orb_;
 9     A_impl a_;
10
11     public B_impl(org.omg.CORBA.ORB orb)
12     {
13         orb_ = orb;
14         a_ = new A_impl();
15     }
16
17     A getA()
18     {
19         return a_._this(orb_);
20     }
21 }
```

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

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

11-15 `B_impl`'s constructor stores a reference to the orb and creates a new `A_impl` servant.

17-20 `getA` returns an object reference to the `A_impl` servant (implicitly creating and activating the CORBA object if necessary).

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

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

And in Java:

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

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 its use might result in a `TRANSIENT` or `OBJECT_NOT_EXIST` exception. The sections that follow 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 “Command-line Options” on page 58 for more information on the `-OAhost` option.

6.2.2 Port Number

Each time a server is executed, the Root POA manager 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 Root POA manager 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 “Command-line Options” on page 58 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 may affect the keys assigned to those objects.

To ensure that your objects always have the same keys, activate your objects using POAs with the `PERSISTENT` life span policy and the `USER_ID` object identification policy.

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 `object_to_string` 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 `string_to_object`. 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_impl* bImlp = new B_impl();
```

```
4 PortableServer::ServantBase_var servant = bImpl;
5 B_var b = bImpl -> _this();
6 CORBA::String_var s = orb -> object_to_string(b);
7 ofstream out("object.ref")
8 out << s << endl;
9 out.close();
```

3-5 A servant for the interface `B` is created and is used to incarnate a CORBA object.

6 The object reference of the servant is “stringified”.

7-9 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 bImpl = new B_impl();
4 B b = bImpl._this(orb);
5 String ref = orb.object_to_string(b);
6 java.io.PrintWriter out = new java.io.PrintWriter(
7    new java.io.FileOutputStream("object.ref"));
8 out.println(ref);
9 out.close();
```

3-4 A servant for the interface `B` is created and is used to incarnate a CORBA object.

5 The object reference of the servant is “stringified”.

6-9 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 string s;
5 in >> s;
6 CORBA::Object_var obj = orb -> string_to_object(s.c_str());
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 is shown only in Java because of Java's 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";
6  org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
7
8  URL url = new URL(location);
9  URLConnection conn = url.openConnection();
10 BufferedReader in = new BufferedReader(
11    new InputStreamReader(conn.getInputStream()));
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.

8-13 Read the string from the URL connection.

15 Convert the string to an object reference.

16 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:

```
<APPLET CODE="Client.class" ARCHIVE="OB.jar" WIDTH=500 HEIGHT=300>
    <PARAM NAME="ref" VALUE="IOR:000012031...">
</APPLET>
```

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 72 for more information on using ORBACUS in applets.

6.4 Object Reference URLs

Prior to the adoption of the Interoperable Naming Service (INS) [10], the only standard format for stringified object references was the cumbersome `IOR:` format. The INS introduced two new, more readable formats for object references that use a URL-like syntax. Object reference URLs can be passed to `string_to_object`, just like `IOR:` references. The two new URL formats are described in detail in the specification, but will be briefly discussed here. The optional `file:` URL format is also discussed, as well as the proprietary `relfile:` URL format.

6.4.1 corbaloc: URLs

The `corbaloc:` URL supports any number of protocols; the format of the URL depends on the protocol in use. The general format of a corbaloc: URL is shown below:

```
corbaloc:[protocol]:<protocol-specific>
```

ORBACUS supports two standard protocols, `iiop` and `rir`.

The `corbaloc:` URL for the `iiop` protocol has the following structure:

```
corbaloc:[iiop]:[version@]host[:port]/object-key
```

The components of the URL are as follows:

- `iiop` - This is the default protocol for `corbaloc:` URLs, and therefore is optional.
- `version` - The IIOP version number in `major.minor` format. The default is `1.0`.
- `host` - The hostname of the server.
- `port` - The port on which the server is listening. The default is `2089`.
- `object-key` - A stringified object key.

The specification allows a URL to contain multiple addresses, but the semantics are vendor-specific. In ORBACUS, each address is used in turn until one is found that works or until the ORB has tried them all and failed to contact the object.

The `rir` protocol is a shortcut for the ORB operation `resolve_initial_references`. The `corbaloc:` URL for the `rir` protocol has the following structure:

```
corbaloc:rir:[/id]
```

The components of the URL are as follows:

- `rir` - The protocol.

- `id` - The identifier of the service to be resolved. The identifier `NameService` is used if `id` is not supplied.

Some examples of `corbaloc:` URLs are:

```
corbaloc::nshost:10000/NameService
corbaloc::myhost:10000/MyObjectId
corbaloc:rir:/NameService
```

In the above examples, `NameService` and `MyObjectId` are used as object keys. Normally, object keys contain the information necessary to uniquely identify a POA and a servant within the POA. However, the object keys used above do not contain information which identifies both the POA and the servant (unless some assumptions are made, e.g., a default POA name). To solve this problem, ORBACUS defines the interfaces `BootManager` and `BootLocator`. (See Appendix A for a detailed description.)

The `BootManager::add_binding` operation binds an object id to an object reference. The `BootManager::remove_binding` operation is used to remove a binding. A `BootLocator` object can be registered with the `BootManager` using the `set_locator` operation and is used to dynamically locate a reference for a given object id. The following example illustrates how a server can add a binding for the object id `MyObjectId`. First, in C++:

```
1 // C++
2 CORBA::Object_var obj // ... A reference to a persistent object
3 CORBA::Object_var bmgrObj =
4     orb -> resolve_initial_references("BootManager");
5 OB::BootManager_var bootManager =
6     OB::BootManager::_narrow(bmgrObj);
7 PortableServer::ObjectId_var objId =
8     PortableServer::string_to_ObjectId("MyObjectId");
9 bootManager -> add_binding(objId, obj);
```

3-6 Get a reference to the `BootManager` object by invoking `resolve_initial_references` (see 6.5.1 on page 108) on the ORB.

7-8 Create the object id.

9 Create the new binding.

Or in Java:

```
1 // Java
2 org.omg.CORBA.Object obj = ... // A reference to a persistent object
3 org.omg.CORBA.Object bmgrObj =
```

```
4     orb.resolve_initial_references("BootManager");
5  com.ooc.OB.BootManager bootManager =
6     com.ooc.OB.BootManagerHelper.narrow(bmgrObj);
7  byte[] objId = "MyObjectId".getBytes();
8  bootManager.add_binding(objId, obj);
```

3-6 Get a reference to the `BootManager` object by invoking `resolve_initial_references` (see 6.5.1 on page 108) on the ORB.

7 Create the object id.

8 Create the new binding.

6.4.2 corbaname: URLs

A `corbaname:` URL provides additional flexibility by incorporating use of the Naming Service in the `string_to_object` operation. The `corbaname:` URL extends the capabilities of the `corbaloc:` URL to allow the `object-key` to identify a binding in a Naming Service. For example, consider this URL:

```
corbaname::ns1:5001/NameService#ctx/MyObject
```

When the ORB interprets this URL, it attempts to resolve a naming context object located at host `ns1` on port `5001` and having the object key `NameService`. Once the naming context has been resolved, the ORB attempts to lookup the binding named `MyObject` in the naming context `ctx`. If successful, the result of `string_to_object` is the object reference associated with the binding.

6.4.3 file: URLs

A `file:` URL provides a convenient way to obtain object references using an IOR or URL reference that is in a file. The format of a `file:` URL is:

```
file:/<absolute file name>
```

Using the `file:` URL and given that the file `object.ref` is located in the `/tmp` directory, the client side example of 6.3.1 on page 101 may be simplified as follows:

```
// C++
CORBA::ORB_var orb = ... // Get a reference to the ORB somehow
CORBA::Object_var obj
   = orb -> string_to_object("file:/tmp/object.ref");
B_var b = B::_narrow(obj);

// Java
```

```
org.omg.CORBA.ORB orb = ... // Get a reference to the ORB somehow
org.omg.CORBA.Object obj =
   orb.string_to_object("file:/tmp/object.ref");
B b = BHelper.narrow(obj);
```

6.4.4 relfile: URLs

ORBACUS also provides the proprietary `relfile:` URL. This URL is the same as the `file:` URL except that it takes a relative file name instead of an absolute file name.

6.5 Initial Services

The CORBA specification provides a 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 [9]. 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 {
 7     obj = orb.resolve_initial_references("NameService");
 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 {
16     ... // The object reference is invalid
17 }
18
19 try
20 {
21     ctx = org.omg.CosNaming.NamingContextHelper.narrow(obj);
22 }
23 catch(org.omg.CORBA.BAD_PARAM ex)
24 {
25     ... // This object does not implement a NamingContext
26 }
```

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.

19-26 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.

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::ORB::InvalidName&)
10 {
11     ... // An error occured, service is not available
12 }
13
```

```
14 if(CORBA::is_nil(obj))
15 {
16     ... // The object reference is invalid
17 }
18
19 ctx = CosNaming::NamingContext::_narrow(obj);
20 if(CORBA::is_nil(ctx))
21 {
22     ... // This object does not implement NamingContext
23 }
```

1-23 This is the equivalent to the Java version above.

6.5.2 Configuring the Initial Services

When an application uses initial services that are not locality-constrained, the application must register the object references for these objects with the ORB. ORBACUS supports the standard `-ORBInitRef` and `-ORBDefaultInitRef` command-line options for registering initial service object references:

```
-ORBInitRef name=URL
-ORBDefaultInitRef URL
```

For example, starting an application as shown below will enable the client to resolve the `NameService` initial reference:

```
myclient -ORBInitRef NameService=corbaloc::nshost:10000/NameService
```

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 “Configuring the ORB and Object Adapter” on page 49 for more information on these command-line options. Also refer to the INS specification [10] for detailed information on the standard options `-ORBInitRef` and `-ORBDefaultInitRef`.

In addition to using command-line parameters, a program can add to the list of initial services using the ORB operation `register_initial_reference`[1]:

```
// IDL
module CORBA
{
    interface ORB
```

1. This will become part of the ORB interface when the Portable Interceptor specification is adopted.

```
    {
        void register_initial_reference(in ObjectId id, in Object obj)
            raises(InvalidName);
    };
};
```

For example, in C++:

```
1 // C++
2 CORBA::Object_var obj = ... // Get a name service reference somehow
3 orb -> register_initial_reference("NameService", obj);
```

2 Get a reference to the naming service, for example by reading a stringified object reference and converting it with `string_to_object`, or by any other means.

3 Add the reference to the ORB's list of initial references.

Or in Java:

```
1 // Java
2 org.omg.CORBA.Object obj = ...// Get a name service reference somehow
3 orb.register_initial_reference("NameService", obj);
```

1-3 This is the same as the C++ version above.

6.5.3 The Initial Service Locator

In addition to providing the ORBACUS Implementation Repository, the IMR server (see Chapter 7) acts as an initial service locator. That is, assuming that the IMR server is properly configured, the name of the host running the IMR server is the only information needed to find a particular initial service.

To locate an initial service with name `foo`, the IMR server must first be configured with the initial reference of this service. This may be done with the `-ORBInitRef` command-line option or the `ooc.orb.service` configuration property (see Chapter 4 for details). Next, the client that wishes to connect to `foo` must be configured with the default initial reference specifying the host running the IMR server. The `-ORBDefaultInitRef` command-line option or the `ooc.orb.default_init_ref` configuration property may be used to configure the default initial reference. For example, given that the IMR server is running on *imr-host*, then the client can be started with the option:

`-ORBDefaultInitRef=corbaloc::`*`imr-host`*

When the client is configured with this default initial reference it may invoke `resolve_initial_references("foo")` on the ORB to obtain a reference to `foo`.

CHAPTER 7

The Implementation Repository

The ORBACUS Implementation Repository (IMR) provides support for the indirect binding[1] of persistent object references. The key advantage of indirect binding is that it loosens the coupling between clients and servers so that the location of the server can change without affecting the client. In practical terms, this is accomplished by providing the client with an IOR that actually refers to the IMR, rather than to the server itself. The IMR also provides the ability to start servers on demand using the Object Activation Daemon (OAD).

The CORBA specification does not standardize how servers and the IMR interact, it only suggests functionality for vendors to implement. Hence, the interface between servers and the IMR is strictly proprietary. Due to the proprietary interface between servers and the IMR, servers using the IMR must be developed using ORBACUS for C++ or Java. However, the interaction between clients and the IMR is strictly specified by the GIOP specification, so any client that is CORBA compliant may interact with the IMR.

1. Binding refers to the process of opening a connection and associating an object reference with its servant.

7.1 Background

7.1.1 How It All Works

When a server is using the IMR, object references created by one of its persistent POAs refer to the IMR rather than to the server itself. When the client makes a request using this reference, the IMR receives the request, activates the server (if necessary) using the OAD, and returns a new object reference to the client that identifies the server at its current host and port. The client then establishes a connection with the server using the new object reference and communicates directly with the server, without the intervention of the IMR. However, should the server fail, a well-behaved client will contact the IMR again, which may restart the server and allow the client to resume its activities.

7.1.2 Information Managed by the IMR

The IMR provides support for the indirect binding and automatic activation of servers within a given domain. In order to provide this support, the IMR manages three types of entities: OADs, servers, and POAs.

OADs

An OAD is responsible for the activation of servers on a given host. Each OAD is registered in the IMR using a host name. The IMR also maintains the status of each OAD. If the OAD is running and in a ready state it will have a status of `up`, otherwise, its status will be `down`.

Servers

Servers are registered with a name that is unique within the domain and the host corresponding to the OAD that is responsible for the server. Since the name is unique within the domain, it is not currently possible to register the same server with multiple OADs. The server name that is registered in the IMR can be any string, but it must be the same as the name used by the server (i.e., the name specified by the `-ORBserver_name` option, or equivalent property). The attributes of a server that are stored by the IMR are summarized below:

`host`	The host corresponding to the OAD that is responsible for the server.
`exec`	The path of server executable (the `.exe` extension must be included on Windows platforms). If this attribute is not set, then the IMR will not activate the server.

`args`	The arguments to be supplied when starting the server executable. Note that "`-ORBserver_name` *server-name*" is automatically appended to the arguments before the server process is started.
`rundir`	The directory that the server process will be started from. If this attribute is not set, then the server process will be started from the root directory. For Windows platforms, the full path must be specified in the `exec` attribute even if this attribute is set.
`mode`	The activation mode. The possible values are: `shared`, only one server process is created which is used by all clients, and `persistent`, the server process is started when the IMR starts and is used by all clients.
`activate-poas`	If this attribute is set to `true` (default), then all persistent POAs will be registered automatically. If set to `false`, then persistent POAs are not registered automatically.
`update-timeout`	The amount of time (in milliseconds) to wait for server status updates.
`failure-timeout`	The amount of time (in seconds) to wait for the server to start.
`max-spawns`	The maximum number of tries to start the server.

The IMR also maintains various state information for each server:

- The internal ID of the server.
- The status of the server process. The valid values are `forked`, `starting`, `running`, `stopping`, and `stopped`.
- Whether or not the server was started manually.
- The number of times that the server process has been spawned.

Server processes inherit environment settings from the environment in which the OAD was started. Hence, path, library path, and class path environment variables can be used by the server application. This is especially useful in the case of shared library and class path settings. (Note that the class path may also be set in the `args` attribute.)

On Windows platforms, the `exec` attribute may refer to an executable or batch file. On UNIX platforms, the `exec` attribute may refer to an executable or a shell script with

```
#! interpreter
```

as its first line. However, if a batch file or shell script is used, then it should accept the `-ORBserver_name` option since it is automatically appended to the `args` attribute by the IMR.

In the case of Java servers, a batch file or shell script should be created to start the server. An alternative is to set the `exec` attribute to the Java interpreter and to use the `args` attribute to specify the class implementing the server.

POAs

The IMR allows implicit registration of POAs when the server is started. This can be enabled or disabled for each server using the `activate_poas` server attribute. If implicit registration is enabled, then the user does not have to register any of the POAs; instead, the server transparently notifies the IMR whenever a call to `create_POA` is made by the application code.

If the user disables implicit registration, then the user must register all persistent POAs (i.e., POAs with the `PERSISTENT` life span policy). POAs are registered using the name of its server and the name of the POA. Note that any transient POAs (POAs with the `TRANSIENT` life span policy) created by the server are not registered with the IMR.

The IMR also maintains the status for each POA, which indicates the state of its POA Manager. The valid values are `inactive`, `active`, `holding`, and `discarding`.

7.1.3 IMR Security

It is *very important* that *only* the IMR's public port (also referred to as its forward port) be accessible outside of the network firewall. Otherwise, anyone can mimic the IMR and cause an OAD to run any command they decide.

For additional security, the information managed by the IMR may only be modified when the IMR is running in *administrative* mode. That is:

- OAD registration and removal,
- server registration and removal,
- modification of server attributes, and
- POA registration and removal

are only possible when the IMR is running in administrative mode. An attempt to modify the information managed by the IMR when it is not running in administration mode will result in a `CORBA::NO_PERMISSION` exception.

7.2 Synopsis

7.2.1 Usage

The IMR and OAD are currently implemented using ORBACUS for C++, but ORBACUS for Java servers can also be launched by the IMR. Both the IMR and OAD are contained in the IMR server, which may be started in one of three modes:

`master`	Start only the IMR.
`slave`	Start only the OAD.
`dual`	Start both the IMR and OAD.

Command-line usage is as follows:

```
imr
   [-h,--help] [-v,--version] [-m,--master] [-s,--slave]
   [-a,--administrative] [-d,--database] [-A,--admin-port]
   [-F,--forward-port] [-S,--slave-port] [-L, --locator-port]
```

Options

`-h` `--help`	Display the command-line options supported by the server.
`-v` `--version`	Display the version of the server.
`-m` `--master`	Run the server in `master` mode.[a]
`-s` `--slave`	Run the server in `slave` mode.[a]
`-a` `--administrative`	Run the IMR in administrative mode. The IMR will run in non-administrative mode by default.
`-d DIRECTORY` `--database DIRECTORY`	Specifies the directory in which the IMR maintains its database files. If not specified, the current working directory is used.
`-A PORT` `--admin-port PORT`	Specifies the IMR's administrative port. This is the port that the OADs and IMR-enabled servers use to communicate with the IMR. For security reasons, access to this port can be restricted. If not specified, port 9999 is used.
`-F PORT` `--forward-port PORT`	Specifies the IMR's public port, which is used by clients for server requests. If not specified, the port 9998 is used.
`-S PORT` `--slave-port PORT`	Specifies the port used by the OAD. Note that all of the OADs in a domain must use the same port. If not specified, the port 9997 is used.
`-L PORT` `--locator-port PORT`	Specifies the port used by the Initial Service Locator (see “The Initial Service Locator” on page 111). If not specified, the port 2809 is used.

a. Note that only one of the `-m` or `-s` options may be specified. Also, if neither the `-m` or `-s` option is specified, then the server is started in `dual` mode.

7.2.2 Windows NT Native Service

The imr server is also available as a native Windows NT service.

```
ntimrservice
   [-h,--help] [-i,--install] [-s,--start-install]
```

```
   [-u,--uninstall] [-d,--debug]
```

`-h` `--help`	Display the command-line options supported by the service.
`-i` `--install`	Install the service. The service must be started manually.
`-s` `--start-install`	Install and start the service.
`-u` `--uninstall`	Uninstall the service.
`-d` `--debug`	Run the service in debug mode.

In order to use the IMR server as a native Windows NT service, first add the desired configuration properties to the `HKEY_LOCAL_MACHINE` NT registry key (see "Using the Windows NT Registry" on page 60 for more details). For example, add the `ooc.imr.admin_port`, `ooc.imr.forward_port`, and `ooc.imr.slave_port` properties so that the IMR and OAD will use non-default ports.

Next the service should be installed with:

```
ntimrservice -i
```

This adds the `ORBacus Implementation Repository` entry to the `Services` dialog in the Control Panel. To start the service, select the `ORBacus Implementation Repository` entry, and press `Start`. If the service is to be started automatically when the machine is booted, select the `ORBacus Implementation Repository` entry, then click `Startup`. Next select `Startup Type - Automatic`, and press `OK`. Alternatively, the service could have been installed using the `-s` option, which configures the service for automatic start-up:

```
ntimrservice -s
```

If you want to remove the service, run:

```
ntimrservice -u
```

Note: If the executable for the service is moved, it must be uninstalled and re-installed.

Any trace information provided by the service is be placed in the Windows NT Event Viewer with the title `IMRService`. To enable tracing information, add the desired trace configuration property (i.e., one of the `ooc.imr.trace` properties or one of the

`ooc.orb.trace` properties) to the `HKEY_LOCAL_MACHINE` NT registry key with a `REG_SZ` value of at least 1.

7.2.3 Configuration Properties

In addition to the standard configuration properties described in Chapter 4, the IMR also supports the following properties:

ooc.imr.mode

Value: `master`, `slave`, `dual`

Specifies the mode in which the imr server will be started.

ooc.imr.administrative

Value: `true`, `false`

If set to `true`, then run the IMR in administrative mode. For details refer to the `-a` command-line option.

ooc.imr.dbdir

Value: *directory*

Equivalent to the `-d` command-line option.

ooc.imr.admin_port

Value: *port*

Equivalent to the `-A` command-line option.

ooc.imr.forward_port

Value: *port*

Equivalent to the `-F` command-line option.

ooc.imr.slave_port

Value: *port*

Equivalent to the `-S` command-line option.

ooc.imr.locator_port

Value: *port*

Equivalent to the `-L` command-line option.

ooc.imr.trace.peer_status

Value: *level >= 0*

Defines the output level for IMR diagnostic messages related to communications with the OADs. The default level is 0, which produces no output.

ooc.imr.trace.process_control

Value: *level >= 0*

Defines the output level for IMR diagnostic messages related to the forking and death of server processes. The default level is 0, which produces no output.

ooc.imr.trace.server_status

Value: *level >= 0*

Defines the output level for IMR diagnostic messages related to the status of servers and POAs. The default level is 0, which produces no output.

7.3 Connecting to the Service

Servers that use the IMR must be configured with the IMR initial reference. The object key of the IMR is `IMR`, hence, a URL-style object reference of the IMR service running on host `imrhost` at port `10000` would be:

```
corbaloc::imrhost:10000/IMR
```

Using this object reference, a server can configure the IMR initial reference with the property:

```
ooc.orb.service.IMR=corbaloc::imrhost:10000/IMR
```

An alternative to using the above property is to use the `-ORBInitRef` command-line option. Refer to Chapter 6 for more information on URLs and configuring initial services.

7.4 Utilities

7.4.1 Implementation Repository Administration

The `imradmin` utility provides complete control over the IMR, OADs and servers in a domain. Its command interface is shown below:

`-h, --help`	Display this information.
`--add-oad [`*host*`]`	Register an OAD for the specified host.
`--add-server` *server-name* `[`*exec* `[`*host*`]]`	Register a server under the OAD specified by *host* with the given *exec* attribute.
`--add-poa` *server-name poa-name*	Register a POA for the specified server.
`--remove-oad [`*host*`]`	Unregister an OAD.
`--remove-server` *server-name*	Unregister a server.
`--remove-poa` *server-name poa-name*	Unregister a POA.
`--get-oad-status [`*host*`]`	Get the status of an OAD.
`--get-server-info` *server-name*	Get the attributes and state information for a server.
`--get-poa-status` *server-name poa-name*	Get the status of a POA.
`--list-oads`	List all OADs.
`--list-servers`	List all servers.
`--list-poas` *server-name*	List all POAs.
`--tree`	Display all OADs, servers and POAs in a tree like format.
`--tree-oad [`*host*`]`	Display an OAD and its associated servers and POAs in a tree like format.
`--tree-server` *server-name*	Display a server and its associated POAs in a tree like format.
`--set-server` *server-name* `{exec|host|args|rundir|mode|activate_poas|update_timeout|failure_timeout|max_spawns}` *value*	Set an attribute of a server (e.g., `--set-server srv max_spawns 2` sets the `max_spawns` attribute for the server `srv` to `2`).
`--start-server` *server-name*	Start a server.

`--stop-server` *server-name*	Stop a server.
`--reset-server` *server-name*	Reset a server.

Note that the `imradmin` utility also needs to be configured with the IMR initial reference (see "Connecting to the Service" on page 121).

The *`host`* argument is optional. If *`host`* is not specified the local host name is used. The *`server-name`* argument refers to the name of the server. The format of the *`poa-name`* argument is `poa1/poa2/poa3`, where `poa1` is a child of the Root POA, `poa2` is a child of `poa1`, and `poa3` is a child of `poa2`. Refer to "Information Managed by the IMR" on page 114 for further details.

In very rare circumstances, it's possible for the IMR and OAD to become confused as to the state of a server. In this case it might be necessary to manually reset the state of the server using the `--reset-server` command. It is also necessary to use this command if the server continually crashes on startup and has reached the maximum number of retries specified by its `max_spawns` attribute. This prevents the OAD from continually starting the same broken server.

7.4.2 Making References

The `mkref` utility creates IMR-based object references for use by clients. Since the Object ID is required to create a reference, this utility can only be used to create references for objects created by POAs using the `USER_ID` object identification policy. Its usage is shown below.

`mkref [-H` *`imr-host`*`]` *`server-name object-id poa1/poa2/.../poan`*

imr-host	The host that the imr server is running on. If the host is not specified, then `localhost` is used.
server-name	The name of the server as registered in the IMR.
object-id	The Object ID used by the object.
poa1/*poa2*/.../*poan*	The POA which creates the object, where poa1 is a child of the Root POA, *poa2* is a child of *poa1*, and so on.

The `mkref` utility uses the `ooc.imr.forward_port` property (see "Configuration Properties" on page 120). If this property is not set then `mkref` will use 9998.

7.4.3 Upgrading the IMR Database

The `imrdbupgrade` utility is used to upgrade an earlier version of the IMR database. Command-line usage is as follows:

`imrdbupgrade` *`database-directory`*

The *`database-directory`* parameter is used to specify the IMR database directory.

7.5 Getting Started with the Implementation Repository

To use the IMR, several steps must be taken. These steps are presented below and are explained by way of example. In this example we assume that ORBACUS has been installed in the directory `/usr/local/ORBacus` and the executables `imr`, `imradmin` and `mkref` all exist in a directory that is in the search path.

1. Determine the physical architecture.

   In this example, we have a network with three hosts: `master`, `slave1` and `slave2`. The host `master` is used to run only the IMR. The hosts `slave1` and `slave2` are used to run individual CORBA servers.

2. Create a configuration file for the IMR and OADs.

   First, create a configuration file for the IMR containing the following:

   ```
   # imr.conf
   ooc.imr.admin_port=10000
   ooc.imr.forward_port=10001
   ooc.imr.slave_port=10002
   ooc.imr.mode=master
   ooc.imr.dbdir=/usr/local/ORBacus/db
   ```

   This file is placed in the `/usr/local/ORBacus/etc` directory on host `master`. This configuration file can also be used by the `mkref` utility.

   Second, create a configuration file for the OADs containing the following:

   ```
   # oad.conf
   ooc.orb.service.IMR=corbaloc::master:10000/IMR
   ooc.imr.slave_port=10002
   ooc.imr.mode=slave
   ooc.imr.dbdir=/usr/local/ORBacus/db
   ```

   This files is placed in the `/usr/local/ORBacus/etc` directory on hosts `slave1` and `slave2`.

3. Start the IMR in administrative mode.

   On host `master`, run:

```
imr -ORBconfig /usr/local/ORBacus/etc/imr.conf --administrative
```

4. Start the OADs.

   On host `slave1`, run:

   ```
   imr -ORBconfig /usr/local/ORBacus/etc/oad.conf
   ```

   On host `slave2`, run:

   ```
   imr -ORBconfig /usr/local/ORBacus/etc/oad.conf
   ```

   Each OAD automatically registers itself with the IMR. Note that an OAD can also be registered manually using the `imradmin` utility. For example, to register the OAD on host `slave1`, run:

   ```
   imradmin -ORBInitRef IMR=corbaloc::master:10000/IMR \
       --add-oad slave1
   ```

5. Add each server to the IMR.

   In our example, we will run one server on each OAD. The server names are `Server1` and `Server2` and are located in `/usr/local/bin` on their respective hosts.

   First, we register the servers using the `imradmin` utility:

   ```
   imradmin -ORBInitRef IMR=corbaloc::master:10000/IMR \
       --add-server Server1 "/usr/local/bin/server1" slave1
   imradmin -ORBInitRef IMR=corbaloc::master:10000/IMR \
       --add-server Server2 "/usr/local/bin/server2" slave2
   ```

   Next, we set the server arguments:

   ```
   imradmin -ORBInitRef IMR=corbaloc::master:10000/IMR \
       --set-server Server1 args \
       "-ORBInitRef IMR=corbaloc::master:10000/IMR"
   imradmin -ORBInitRef IMR=corbaloc::master:10000/IMR \
       --set-server Server2 args \
       "-ORBInitRef IMR=corbaloc::master:10000/IMR"
   ```

   A C++ server can automatically register itself with the IMR using the `-ORBregister` command-line option. For example, to registered `Server1`, run the following on `slave1`:

   ```
   /usr/local/bin/server1 -ORBregister Server1 \
       -ORBInitRef IMR=corbaloc::master:10000/IMR
   ```

   If the server requires command-line options, then these options must be added using the `imradmin` utility.

6. Add each POA to the IMR.

In this example, the servers are registered without setting the `activate_poas` attribute, so the attribute defaults to `true`. Hence, all persistent POAs will be registered automatically. If this were not the case, the POAs would have to be registered manually.

7. Configure your servers to use the IMR.

   There are three ways to configure a server to use the IMR:

   a) Use the `-ORBregister` command-line option (only available for C++ servers). This option is used for server registration and can only be used when starting the server for the first time.

   b) Use the `-ORBserver_name` command-line option.

   c) Use the `ooc.orb.server_name` configuration property. This configuration property is equivalent to the `-ORBserver_name` command-line option and may be set in a configuration file or programmatically prior to initializing the ORB in a server.

   In this example, the IMR is responsible for starting the servers. Hence, when the server is started, the `-ORBserver_name` option is automatically added to the argument list.

8. Create object references for use by the clients.

   A server can always be used to create references for its objects. However, if an object is created by a POA that uses the `USER_ID` object identification policy, then the `mkref` utility can also be used to create a reference for the object. Using the `mkref` utility is discussed below.

   Assume each server has a single primary object. `Server1` uses `Object1` for its Object ID and `Server2` uses `Object2`. Also, each server creates a persistent POA called `Main` to hold its objects. To create object references for these objects, run the following on `master`:

   ```
   mkref -ORBconfig /usr/local/ORBacus/etc/imr.conf \
       Server1 Object1 Main > Object1.ref
   mkref -ORBconfig /usr/local/ORBacus/etc/imr.conf \
       Server2 Object2 Main > Object2.ref
   ```

   The `imr.conf` configuration file contains the properties needed by the `mkref` utility.

After all OADs, servers and POAs are registered, it is recommended to restart the IMR in non-administrative mode. This will prevent any accidental (or unauthorized) modifications.

7.6 Programming Example

In this section, we will show how to modify the C++ version of the "Hello World" server (see Chapter 2) to use a persistent object reference. This will allow the server to use the IMR for indirect binding. Modifications to the Java version of the server are similar. The code for both the C++ and Java persistent "Hello World" servers may be found in the `demo/hello_imr` directories of the ORBACUS for C++ and Java distributions.

The "Hello World" server presented in Chapter 2 uses the Root POA to activate its Hello servant. Since the Root POA uses the `TRANSIENT` life span policy, the object reference it creates will not be persistent. Hence, the "Hello World" server must be modified so that the Hello servant is activated using a child POA with the `PERSISTENT` life span policy. The new child POA will also use the `USER_ID` object identification policy so that the `mkref` utility may be used. Further, the Hello servant is no longer activated under the Root POA, so it becomes necessary for it to override the `_default_POA` method. The modified servant's class declaration is shown below:

```
1  // C++
2
3  #include <Hello_skel.h>
4
5  class Hello_impl : public POA_Hello,
6                     public PortableServer::RefCountServantBase
7  {
8      PortableServer::POA_var poa_;
9
10 public:
11
12     Hello_impl(PortableServer::POA_ptr);
13
14     virtual void say_hello() throw(CORBA::SystemException);
15
16     virtual PortableServer::POA_ptr _default_POA();
17 };
```

8 Private member to store the servant's default POA.

12 A constructor must be defined to allow the assignment of the servant's default POA.

16 Declaration of the `_default_POA` method.

The remainder of the class declaration is unchanged. The definition of the constructor and `_default_POA` method follow:

```
// C++

Hello_impl::Hello_impl(PortableServer::POA_ptr poa)
    : poa_(PortableServer::POA::_duplicate(poa)
{
}

PortableServer::POA_ptr Hello_impl::_default_POA()
{
    return PortableServer::POA::_duplicate(poa_);
}
```

The modified portion of the server program is shown below:

```
1  // C++
2
3  int
4  run(CORBA::ORB_ptr orb, int argc)
5  {
6      CORBA::Object_var poaObj =
7          orb -> resolve_initial_references("RootPOA");
8      PortableServer::POA_var rootPoa =
9          PortableServer::POA::_narrow(poaObj);
10
11     PortableServer::POAManager_var manager =
12         rootPoa -> the_POAManager();
13
14     CORBA::PolicyList pl(2);
15     pl.length(2);
16     pl[0] = rootPOA -> create_lifespan_policy(
17         PortableServer::PERSISTENT);
18     pl[1] = rootPOA -> create_id_assignment_policy(
19         PortableServer::USER_ID);
20
21     PortableServer::POA_var helloPOA =
22         rootPOA -> create_POA("hello", manager, pl);
23
24     Hello_impl* helloImpl = new Hello_impl(helloPOA);
25     PortableServer::ServantBase_var servant = helloImpl;
26     PortableServer::ObjectId_var oid =
27         PortableServer::string_to_ObjectId("hello");
28     helloPOA -> activate_object_with_id(oid, servant);
29     Hello_var hello = helloImpl -> _this();
30
```

```
31      CORBA::String_var s = orb -> object_to_string(hello);
32      ofstream out("Hello.ref");
33      out << s << endl;
34      out.close();
35
36      manager -> activate();
37      orb -> run();
38
39      return 0;
40  }
```

14-22 Create a new POA using `PERSISTENT` life span policy and the `USER_ID` object identification policy.

24-25 Create the Hello servant.

26-27 Using the string `"hello"`, create an object id.

28 Activate the servant with the new POA.

The remainder of the code is unchanged.

CHAPTER 8

The Implementation Repository Console

The ORBACUS Implementation Repository (IMR) includes a graphical client for administering the service called the ORBACUS IMR Console. The ORBACUS IMR Console provides complete control over the IMR, OADs and servers in a domain.

8.1 Synopsis

8.1.1 Usage

```
com.ooc.IMRConsole.Main
   [--look CLASS] [--windows] [--motif] [--mac] [-h,--help]
```

`--look CLASS`	Use the specified Look & Feel class.
`--windows`	Use the Windows Look & Feel (if available).
`--motif`	Use the Motif Look & Feel (if available).
`--mac`	Use the Macintosh Look & Feel (if available).
`-h` `--help`	Display the command-line options supported by the program.

8.1.2 CLASSPATH Requirements

The ORBACUS IMR Console requires the classes in `OB.jar`, `OBIMR.jar`, `OBUtil.jar` and the Java Foundation Classes (JFC). Note, JFC is part of version 1.2 (or greater) of JDK.

8.1.3 Implementation Repository Service Lookup

In order to locate an IMR Service, the application uses the initial IMR Service, as provided to the ORB with options such as `-ORBservice` or `-ORBconfig`. If the service is not found, an error is displayed and the IMR Console exits.

8.2 The Menus

The menus provide access to all of the features of the application. In addition, the most common actions are also available in the toolbar, as well as in a popup menu that is displayed when pressing the right mouse button over an item in the binding table or context tree.

8.2.1 The File Menu

The **File** menu contains the **Exit** menu item, which is used to exit the ORBACUS IMR Console.

8.2.2 The Edit Menu

The operations in the **Edit** menu provide the means for manipulating OADs, servers and POAs.

Create	Create a new OAD, server, or POA.
Modify	Modify the selected object.
Delete	Delete the selected object.
Cut	Move the selected server to the clipboard.
Paste	Insert the server contained in the clipboard under the selected OAD.
Start	Start the selected server.
Stop	Stop the selected server.
Reset	Reset the state of the selected server.

The **Create** menu item creates a child object under the selected object. OADs are created under the "IMR Domain" root object, servers are created under OADs, and POAs are created under servers.

The **Modify** menu item applies to all objects. However, servers are currently the only objects that have attributes that can be modified.

To delete an object, the **Delete** menu item is used. This operation recursively deletes all children under the selected item.

The **Cut** and **Paste** menu items only apply to servers and are used to move servers to different hosts. Note that OAD for the desired host must be selected when using **Paste**.

In very rare circumstances, it's possible for the IMR and OAD to become confused as to the state of a server. In this case it might be necessary to manually reset the state of the server using the **Reset** menu item. It also necessary to use this item if the server continually crashes on startup and has reached the maximum number of retries specified by its `max_spawns` attribute. This prevents the OAD from continually starting the same broken server.

8.2.3 The View Menu

The **View** menu contains the **Refresh** menu item. The **Refresh** menu item is used to update the console when the contents of the IMR have been changed from outside the console. Note that clicking or expanding an item will refresh the item.

8.3 *The Toolbar and the Popup Menu*

In addition to the operations offered by the menu bar, some frequently needed functions are available by icons located in the toolbar. The toolbar contains all of the items of the **Edit** menu and the **Refresh** item of the **View** menu. The toolbar is shown below in Figure 8.1.

Figure 8.1: A closer look at the toolbar

When selecting an OAD, server or POA with the right mouse button, a popup menu with a choice of operations will be displayed as shown in Figure 8.2. This popup menu provides

Figure 8.2: The popup menu

the same operations as the toolbar.

CHAPTER 9

ORBacus Names

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 their associated object references.

ORBACUS Names is compliant with [10]. This chapter does not provide a complete description of the service. It only provides an overview, suitable to get you started. For more information, please refer to the specification.

9.1 Synopsis

9.1.1 Usage

ORBACUS includes functionally equivalent implementations of the Naming Service in C++ and Java.

C++

```
nameserv
   [-h,--help] [-v,--version] [-i,--ior] [-n,--no-updates]
   [-s,--start] [-d,--database FILE] [-t,--timeout MINS]
   [-c, --callback-timeout SECS]
```

Java

```
com.ooc.CosNaming.Server
   [-h,--help] [-v,--version] [-i,--ior] [-n,--no-updates]
   [-s,--start] [-d,--database FILE] [-t,--timeout MINS]
   [-c, --callback-timeout SECS]
```

Options

`-h` `--help`	Display the command-line options supported by the server.
`-v` `--version`	Display the version of the server.
`-i` `--ior`	Prints the stringified IOR of the server to standard output.
`-n` `--no-updates`	Disables automatic updates, i.e., callbacks that notify interested clients of changes to the naming service.
`-s` `--start`	Use this option only when starting a persistent server using a new database.
`-d FILE` `--database FILE`	Enables persistence for the server. All of the bindings created 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.
`-t MINS` `--timeout MINS`	Specifies the timeout in minutes after which a persistent server automatically compacts its database. The default timeout is five minutes.
`-c SECS` `--callback-timeout SECS`	Specifies the timeout in seconds to be used for the ORBACUS timeout policy (`OB::TimeoutPolicy`). The default timeout is five seconds. See Chapter 15 for more information.

9.1.2 Windows NT Native Service

The C++ version of ORBACUS Names is also available as a native Windows NT service.

```
ntnameservice
   [-h,--help] [-i,--install] [-s,--start-install]
```

```
   [-u,--uninstall] [-d,--debug]
```

`-h` `--help`	Display the command-line options supported by the server.
`-i` `--install`	Install the service. The service must be started manually.
`-s` `--start-install`	Install the service. The service will be started automatically.
`-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 `ooc.naming.port` configuration property to the `HKEY_LOCAL_MACHINE` NT registry key (see "Using the Windows NT Registry" on page 60 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`. Alternatively, the service could have been installed using the `-s` option, which configures the service for automatic start-up:

```
ntnameservice -s
```

If you want to remove the service, run:

```
ntnameservice -u
```

1. Please note that services do not have access to network drives, so the path to the database must be on a local hard drive.

Note: If the executable for the Naming 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 `NamingService`. To enable tracing information, add the desired trace configuration property (i.e., the `ooc.naming.trace_level` property or one of the `ooc.orb.trace` properties) to the `HKEY_LOCAL_MACHINE` NT registry key with a `REG_SZ` value of at least 1.

9.1.3 Configuration Properties

In addition to the standard configuration properties described in Chapter 4, ORBACUS Names also supports the following properties:

`ooc.naming.callback_timeout=SECS`	Equivalent to the `-c` command-line option.
`ooc.naming.database=FILE`	Equivalent to the `-d` command-line option.
`ooc.naming.no_updates`	Equivalent to the `-n` command-line option.
`ooc.naming.port=PORT`	Specifies the port number on which the service should listen for new connections. Note that this property is only considered if the `ooc.oa.port` property is not set.
`ooc.naming.timeout=MINS`	Equivalent to the `-t` command-line option.
`ooc.naming.trace_level=LEVEL`	Defines the output level for diagnostic messages printed by ORBACUS Names. The default level is 0, which produces no output. A level of 1 or higher produces messages related to database operations, a level of 2 or higher produces messages related to adding and removing listeners, and a level of 3 or higher produces messages related to binding operations.

9.1.4 Persistence

ORBACUS Names can optionally be used in a persistent mode, in which all data managed by the service is saved in a file. If you do not run the service in its persistent mode, all of the data 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 Chapter 4 for more information).

9.1.5 CLASSPATH Requirements

ORBACUS Names for Java requires the classes in `OB.jar` and `OBNaming.jar`.

9.2 Connecting to the Service

The object key of the Naming Service is `NameService`, which identifies an object of type `CosNaming::OBNamingContext`. The `OBNamingContext` interface is derived from the standard interface `CosNaming::NamingContextExt` and provides additional ORBACUS-specific functionality. For a description of the `OBNamingContext` interface, please refer to the documented IDL file `naming/idl/OBNaming.idl`.

The object key can be used when composing URL-style object references. For example, the following URL identifies the naming service running on host `nshost` at port `10000`:

```
corbaloc::nshost:10000/NameService
```

Refer to Chapter 6 for more information on URLs and configuring initial services.

9.3 Using the Naming Service with the IMR

The Naming Service may be used with the Implementation Repository (IMR). However, if used with the IMR, it is important to note that the `corbaloc` URL-style object reference described in the previous section cannot be used. If the IMR is used, then the object reference for the Naming Service must be created using one of the following methods (where `NamingServer` refers to the server name configured with the IMR):

- start the Naming Service with the options:

  ```
  --ior -ORBserver_name NamingServer
  ```

  causing the Naming Service to print its reference to standard output.
- use the `mkref` utility:

  ```
  mkref NamingServer NameService RootContextPOA
  ```

When using the Naming Service with the IMR, the service must be started with the option `-ORBserver_name NamingServer`, where `NamingServer` refers to the server name configured with the IMR. When the IMR is configured to start the Naming Service, this option is automatically added to the service's arguments. However, when the Naming Service is started manually, the option must be present. For further information on configuring a service with the IMR, refer to "Getting Started with the Implementation Repository" on page 124.

9.4 Naming Service Concepts

9.4.1 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, AlreadyBound);

NamingContext new_context();

NamingContext bind_new_context(in Name n)
   raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
```

`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. New naming context objects are created with `new_context` or `bind_new_context`. `bind_context` and `bind_new_context` throw an `AlreadyBound` exception if the name is already in use in the target context.

To create a new binding without being concerned if the specified binding already exists, use the following operations:

```
// IDL
void rebind(in Name n, in Object obj)
   raises(NotFound, CannotProceed, InvalidName);

void rebind_context(in Name n, in NamingContext nc)
   raises(NotFound, CannotProceed, InvalidName);
```

Use the `unbind` operation to delete a particular binding:

```
// IDL
void unbind(in Name n)
   raises(NotFound, CannotProceed, InvalidName);
```

9.4.2 Name Resolution

Besides registering objects, an equally important task of the Naming Service is name resolution. A name is passed to the `resolve` or `resolve_str` operation and an object reference is returned if the name exists.

```
// IDL
Object resolve(in Name n)
   raises(NotFound, CannotProceed, InvalidName);
Object resolve_str(in StringName n)
   raises(NotFound, CannotProceed, InvalidName);
```

The `resolve` and `resolve_str` operations are 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.

9.5 Programming 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.

9.5.1 Initialization

The first code fragment deals with initializing the ORB.

```
1 // Java
2 java.util.Properties props = System.getProperties();
3 props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
4 props.put("org.omg.CORBA.ORBSingletonClass",
5           "com.ooc.CORBA.ORBSingleton");
6
7 org.omg.CORBA.ORB orb = null;
```

```
 8 try
 9 {
10    orb = ORB.init(args, props);
11
12    org.omg.CORBA.Object poaObj = null;
13    try
14    {
15       poaObj = orb.resolve_initial_references("RootPOA");
16    }
17    catch(org.omg.CORBA.ORBPackage.InvalidName ex)
18    {
19       throw new RuntimeException();
20    }
21    POA rootPOA = POAHelper.narrow(poaObj);
22    POAManager manager = rootPOA.the_POAManager();
23
24    org.omg.CORBA.Object obj = null;
25    try
26    {
27       obj = orb.resolve_initial_references("NameService");
28    }
29    catch(org.omg.CORBA.ORBPackage.InvalidName ex)
30    {
31       throw new RuntimeException();
32    }
33
34    if(obj == null)
35    {
36       throw new RuntimeException();
37    }
38
39    NamingContextExt nc = null;
40    try
41    {
42       nc = NamingContextExtHelper.narrow(obj);
43    }
44    catch(org.omg.CORBA.BAD_PARAM ex)
45    {
46       throw new RuntimeException();
47    }
```

10-22 Usually the application is initialized in the `main` method. For more information on ORB initialization see Chapter 4.

24-32 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.

34-47 If calling `resolve_initial_references` was successful, the object reference is checked and narrowed in order to verify that it supports the interface `CosNaming::NamingContextExt`. If the `narrow` operation raises `CORBA::BAD_PARAM`, the object does not support the interface. This is considered to be an error because we explicitly asked for a Naming Service instance.

9.5.2 Binding

In the next step some sample bindings are created and bound to the Naming Service.

```
1  // Java
2      Named_impl implA = new Named_impl();
3      Named_impl implA1 = new Named_impl();
4      Named_impl implA2 = new Named_impl();
5      Named_impl implA3 = new Named_impl();
6      Named_impl implR = new Named_impl();
7      Named_impl implC = new Named_impl();
8      Named a = implA._this(orb);
9      Named a1 = implA1._this(orb);
10     Named a2 = implA2._this(orb);
11     Named a3 = implA3._this(orb);
12     Named b = implB._this(orb);
13     Named c = implC._this(orb);
14
15     try
16     {
17         NameComponent[] nc1Name = new NameComponent[1];
18         nc1Name[0] = new NameComponent();
19         nc1Name[0].id = "nc1";
20         nc1Name[0].kind = "";
21         NamingContext nc1 = nc.bind_new_context(nc1Name);
22
23         NameComponent[] nc2Name = new NameComponent[2];
24         nc2Name[0] = new NameComponent();
25         nc2Name[0].id = "nc1";
26         nc2Name[0].kind = "";
27         nc2Name[1] = new NameComponent();
28         nc2Name[1].id = "nc2";
29         nc2Name[1].kind = "";
30         NamingContext nc2 = nc.bind_new_context(nc2Name);
```

```
31
32          NameComponent[] aName = new NameComponent[1];
33          aName[0] = new NameComponent();
34          aName[0].id = "a";
35          aName[0].kind = "";
36          nc.bind(aName, a);
37
38          NameComponent[] a1Name = new NameComponent[1];
39          a1Name[0] = new NameComponent();
40          a1Name[0].id = "a1";
41          a1Name[0].kind = "";
42          nc.bind(a1Name, a1);
43
44          NameComponent[] a2Name = new NameComponent[1];
45          a2Name[0] = new NameComponent();
46          a2Name[0].id = "a2";
47          a2Name[0].kind = "";
48          nc.bind(a2Name, a2);
49
50          NameComponent[] a3Name = new NameComponent[1];
51          a3Name[0] = new NameComponent();
52          a3Name[0].id = "a3";
53          a3Name[0].kind = "";
54          nc.bind(a3Name, a3);
55
56          NameComponent[] bName = new NameComponent[2];
57          bName[0] = new NameComponent();
58          bName[0].id = "nc1";
59          bName[0].kind = "";
60          bName[1] = new NameComponent();
61          bName[1].id = "b";
62          bName[1].kind = "";
63          nc.bind(bName, b);
64
65          NameComponent[] cName = new NameComponent[3];
66          cName[0] = new NameComponent();
67          cName[0].id = "nc1";
68          cName[0].kind = "";
69          cName[1] = new NameComponent();
70          cName[1].id = "nc2";
71          cName[1].kind = "";
72          cName[2] = new NameComponent();
73          cName[2].id = "c";
74          cName[2].kind = "";
```

```
75          nc.bind(cName, c);
76      }
```

2-13 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.

17-75 Create and bind some new contexts and bind the sample objects to these contexts. Each binding name consists of several `NameComponent`s 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 operations expect a unique binding 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`.

9.5.3 Exceptions

This code fragment deals with exceptions that may be thrown by the Naming Service operations.

```
1  // Java
2      catch(NotFound ex)
3      {
4          System.err.print("Got a 'NotFound' exception (");
5          switch(ex.why.value())
6          {
7              case NotFoundReason._missing_node:
8              System.err.print("missing node");
9              break;
10
11             case NotFoundReason._not_context:
12             System.err.print("not context");
13             break;
14
15             case NotFoundReason._not_object:
16             System.err.print("not object");
17             break;
18         }
19
20         System.err.println(")");
21         ex.printStackTrace();
22         throw new SystemException();
23     }
```

```
24      catch(CannotProceed ex)
25      {
26          System.err.println("Got a ‘CannotProceed’ exception");
27          ex.printStackTrace();
28          throw new SystemException();
29      }
30      catch(InvalidName ex)
31      {
32          System.err.println("Got an ‘InvalidName’ exception");
33          ex.printStackTrace();
34          throw new SystemException();
35      }
36      catch(AlreadyBound ex)
37      {
38          System.err.println("Got an ‘AlreadyBound’ exception");
39          ex.printStackTrace();
40          throw new SystemException();
41      }
```

2-41 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.

9.5.4 The Event Loop

Next we start listening for requests.

```
1  // Java
2      try
3      {
4          manager.activate();
5      }
6      catch(org.omg.PortableServer.POAManagerPackage.AdapterInactive
ex)
7      {
8          throw new RuntimeException();
9      }
10     orb.run();
```

2-10 Everything is ready now, so we can listen for requests by calling `actiavate` on the POA Manager and `run` on the ORB.

9.5.5 Releasing Resources

Some cleanup work should be done before exiting the program. Every binding must be properly unbound and the ORB must be destroyed.

```
1  // Java
2      nc.unbind(cName);
3      nc.unbind(bName);
4      nc.unbind(aName);
5      nc.unbind(a1Name);
6      nc.unbind(a2Name);
7      nc.unbind(a3Name);
8      nc.unbind(nc2Name);
9      nc.unbind(nc1Name);
10 }
11 catch(RuntimeException ex)
12 {
13     status = 1;
14 }
15
16 if (orb != null)
17 {
18     try
19     {
20         orb.destroy();
21     }
22     catch(const RuntimeException ex)
23     {
24         status = 1;
25     }
26 }
27
28 System.exit(status);
```

2-9 All bindings are unbound.

16-26 `destroy` is called on the ORB. This releases the resources used by the ORB.

The complete example can be found in the folder `naming/demo` included with the ORBACUS distribution.

CHAPTER 10

ORBacus Names Console

ORBACUS Names includes a graphical client for administering the service called the ORBACUS Names Console. The application can manage any CORBA-compliant Naming Service, but additional features are provided when used with ORBACUS Names.

10.1 Synopsis

10.1.1 Usage

```
com.ooc.CosNamingConsole.Main
   [-f,--file FILE] [-i,--ior] [-n,--no-updates] [--look CLASS]
   [--windows] [--motif] [--mac] [-h,--help] [-v, --version]
```

`-f FILE` `--file FILE`	Read the Naming Service IOR from `FILE`.
`-i` `--ior`	Print the stringified IOR of the Naming Service to standard output.
`-n` `--no-updates`	Disables automatic updates, i.e., callbacks that notify interested clients of changes to the naming service.
`--look CLASS`	Use the specified Look & Feel class.
`--windows`	Use the Windows Look & Feel (if available).

`--motif`	Use the Motif Look & Feel (if available).
`--mac`	Use the Macintosh Look & Feel (if available).
`-h` `--help`	Display the command-line options supported by the program.

10.1.2 CLASSPATH Requirements

The ORBACUS Names Console requires the classes in `OB.jar`, `OBNaming.jar`, `OBUtil.jar` and the Java Foundation Classes (JFC). Note, JFC is part of version 1.2 (or greater) of JDK.

10.1.3 Naming Service Lookup

In order to locate a Naming Service, the application takes the following steps on start-up:

- First it checks whether a Naming Service reference was given with the -f option.
- If this is not the case, then the initial Naming Service is used, as provided to the ORB with options like -ORBservice or -ORBconfig.

If both of the above steps fail, an error window is displayed and the Names console exits.

10.2 The Menus

The menus provide access to all of the features of the application. In addition, the most common actions are also available in the toolbar, as well as in a popup menu that is displayed when pressing the right mouse button over an item in the binding table or context tree.

10.2.1 The File Menu

This menu contains operations that create bindings and define the current root context.

New Window	Opens an additional control window.
Switch Root Context	Selects a new root naming context.
Load Context	Recursively loads a naming context from a file.
Save Context As	Recursively saves the selected naming context to a file.
Save IOR to File	Saves the stringified IOR of the currently selected item to a file.
Close Window	Closes the current window.
Exit	Quits the ORBACUS Names Console.

After starting the application, the current root context is the naming context corresponding to the IOR specified on the command line or the initial Naming Service, as provided to the ORB with options like -ORBservice or -ORBconfigby. You can make another naming context the root context using **Switch Root Context**. The new root context's IOR is specified in the **Enter IOR** dialog window, as shown in Figure 10.1. The IOR can be entered

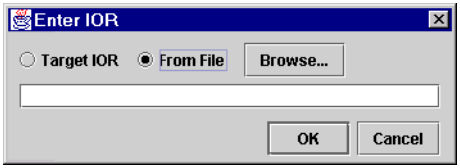


Figure 10.1: Entering an IOR

directly or can be read from a file. If an IOR is entered manually you usually either use the URL-style notation as described in Chapter 6, or you copy a stringified object reference into the dialog box using "Cut & Paste". After selecting **Browse** a file containing an IOR can be selected.

Sometimes it is not desirable to completely replace the currently visible root context by another root context. For example, you may need to copy bindings from one context to another. If this is the case, simply open an additional window for the new root context using **New Window**. You can then switch the root context in only one window without affecting the information displayed in the other one. Using two windows, you can easily transfer bindings from one context to another using "Cut & Paste".

Complete naming contexts can be loaded from a special file with naming context information. Such a file, which was previously created with **Save Context As**, is loaded with **Load Context**. The bindings saved to this file are added to the current naming context.

When saving a naming context, the console checks each context for accessibility. If a context cannot be accessed, i.e., if its contents cannot be saved, a message is displayed in the error window. You also get an error message if the console detects a recursion. The bindings contained in the naming context leading to the recursion is not saved.

Use **Save IOR to File** in order to create a file that contains the stringified IOR of the currently selected binding or context.

With **Close Window** the current window is closed. Closing the last window causes the application to terminate. **Exit** can be used to terminate the application regardless of how many windows are open.

10.2.2 The Edit Menu

The operations in this menu provide the means for creating and deleting objects and for changing the Naming Service structure.

New Context	Creates a new naming context.
New Binding	Creates a new binding for an object.
Delete	Deletes the selected items.
Link	Creates a new binding for an existing naming context.
Unlink	Unbinds the selected items.
Cut	Moves the selected items to the clipboard.
Copy	Copies the selected items to the clipboard.
Paste	Inserts the clipboard contents.
Change ID	Edits the ID field of the selected item.
Change Kind	Edits the Kind field of the selected item.
Change IOR	Edits the IOR of the selected item.
Select all	Selects all items in the object table.
Invert Selection	Inverts the current selection.

New contexts and bindings are created with the operations **New Context** and **New Binding**, respectively. If one of these functions is selected, a new context or object binding with a unique name is added to the current context. For new object bindings an IOR can be specified.

Use **Delete** to remove the selected items from a naming context. Deleting Naming Service entries removes all selected bindings from their parent context. The objects belonging to these bindings are not affected. Destroying Naming Service information only affects the actual Naming Service data, not the objects themselves.

Use **Link** to create a new binding for an existing naming context, where the naming context is specified by an IOR. The operation **Unlink** unbinds the selected items. For objects, **Unlink** is equivalent to **Delete**, but for contexts, **Unlink** differs in that the context is not destroyed. Since a context is not destroyed using Unlink, it should only be used when there are multiple bindings to a context in order to avoid orphaned contexts.

The console supports a clipboard that you can use to move bindings between different contexts. Data is transferred to the clipboard using the **Cut** or **Copy** commands. **Cut** moves the currently selected items to the clipboard and deletes the original entries, whereas **Copy**

simply creates a copy in the clipboard but keeps the source entry unchanged. When new data is transferred to the clipboard, the old clipboard contents are replaced. Using **Paste**, you can add the clipboard data into a naming context. The clipboard contents are not changed by this operation, i.e., you can **Paste** the same items several times. Note that if naming contexts are transferred to the clipboard, their contents are not evaluated before they are pasted. It is during the **Paste** operation that the bindings of a context are duplicated. This means that if new bindings are added to a context after a **Cut** or **Copy** operation, these bindings will be present after pasting this context.

An item registered with the Naming Service has three modifiable attributes: its ID, its Kind and its IOR. The ID and Kind attributes can be edited by simply double-clicking the **ID** or **Kind** field in the table. You can also change binding attributes with the corresponding menu operations **Change ID**, **Change Kind** and **Change IOR**. Entering a new IOR for an existing name effectively replaces an object registered with the Naming Service by another object with the same name.

Use **Select all** to select all of the entries in the binding table. The current table selection can be inverted using **Invert Selection**.

10.2.3 The View Menu

The operations in this menu control the appearance of the console window as well as the presentation of the Naming Service data.

Toolbar	Toggles the toolbar visibility.
Status Bar	Toggles the statusbar visibility.
Error Window	Toggles the error message window visibility.
Simple List	Displays minimum object information.
Details	Displays additional object information.
Sort	Sets sorting mode for object list.
Refresh	Updates the complete window contents

A toolbar that gives access to frequently needed operations is normally present below the menu. If you don't have a need for this toolbar or if you just want to save space on the screen, you can switch it off with the **Toolbar** toggle button. The same applies to the status bar where information about the currently selected item is displayed. The status bar displays an object's repository ID, the host where this object is located and the port it is bound to. If an item with a nil object reference is selected or if multiple items are selected, the status bar is empty.

If an error occurs while editing bindings, the console automatically displays a new window with information about what went wrong. Usually this information consists of exception data. The visibility of this window can be explicitly controlled with the **Error Window** toggle button.

If the console is connected to ORBACUS Names, as described in Chapter 9, the console can display timestamp information for each binding by making use of proprietary features of ORBACUS Names. This information is shown in the binding table if the **Details** display mode instead of the **Simple List** mode is active.

Usually the console sorts the items in the binding table in ascending alphabetical order, with naming contexts being listed at the top. You can change this order with the options available in the **Sort** menu. Bindings can be sorted by their ID or Kind fields. If the extended attributes are displayed, items can also be sorted by date and time. You can reverse the sort order by selecting the current sorting mode a second time in the **View** menu or by clicking on the table header cells. In this case, the display switches from ascending to descending order and vice versa.

If the contents of a naming context have been changed by a third party and you want to update the information displayed in the console window, selecting **Refresh** updates the display. If the console is connected to ORBACUS Names, a refresh is done automatically each time a change occurs.

10.2.4 The Tools Menu

The operations available in this menu are meant as tools for your everyday work.

Ping	Checks the accessibility of the selected items.
Clean up	Unbinds inaccessible objects from the current context.

Sometimes it is useful to check if an object bound to a name still exists or if the object reference associated with it has become invalid, for example, because of a server crash. To perform such a check, select all the objects you want to check and start the **Ping** operation. The console tries to contact each of the selected objects and displays the time it took to get a connection to them in a separate window. This is very similar to the Windows or Unix `ping` command for an IP address or a host name. If there is a time-out while trying to contact an object, this information is displayed in the Ping Window and the console continues with the next object.

If you want objects that cannot be contacted, for example because of a server breakdown, to be unbound from the current context, **Clean up** does the job. **Clean up** non-recursively tries to connect to the selected objects. If there is a communication failure or the

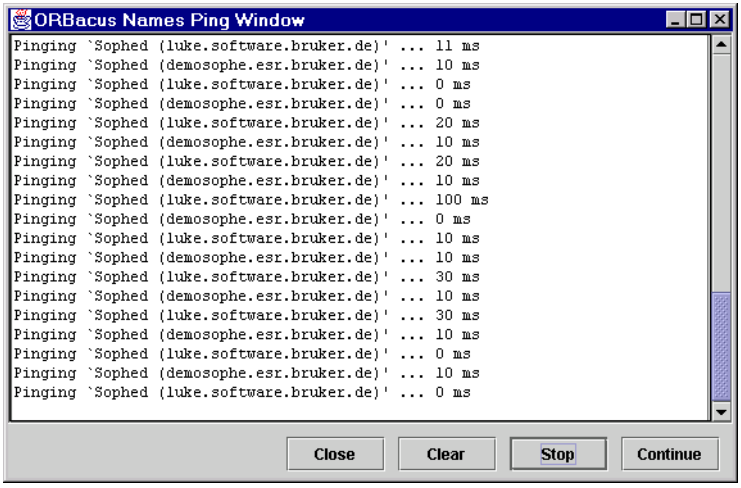


Figure 10.2: The Ping Window

`_non_existent()` operation returns true for a particular object, the corresponding binding is automatically removed. **Clean up** should be used with care.

10.3 The Toolbar

In addition to the operations offered by the menu bar, some frequently needed functions are available by icons located in the toolbar, as shown in Figure 10.3.

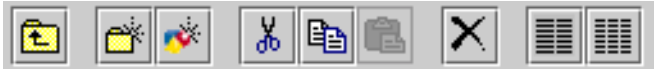


Figure 10.3: A closer look at the toolbar

The icon on the toolbar's left is the **Upwards** icon which changes the naming context to the parent of the context currently being displayed. The next five icons correspond to the **New Context**, **New Binding**, **Cut**, **Copy**, **Paste** and **Delete** items as described in "The Edit Menu" on page 152.

The **Simple List** and **Details** items from the **View** menu are the next two icons in the toolbar. They determine whether the binding table displays only the ID and Kind fields, or, if ORBACUS Names is available, also the date and time the binding was last modified.

The last item in the menubar corresponds to the **Refresh** operation from the **View** menu.

10.4 The Popup Menu

When selecting an item in the binding table or a tree node with the right mouse button, a popup menu with a choice of operations is displayed as shown in Figure 10.4. This is

Figure 10.4: A popup menu offers important operations

another convenient alternative for executing frequently used operations.

CHAPTER 11

ORBacus Properties

The CORBA Property Service[1] permits you to 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 CORBA `Any` data type.

ORBACUS Properties is compliant with [10]. This chapter does not provide a complete description of the service. It only provides an overview, suitable to get you started. For more information, please refer to the specification.

11.1 Synopsis

11.1.1 Usage

ORBACUS includes functionally equivalent implementations of the Property Service in C++ and Java.

C++

```
propserv
   [-h,--help] [-v,--version] [-i,--ior]
```

1. Note that the Property Service has nothing to do with the properties used for configuration purposes. Configuration properties are described in "ORB Properties" on page 49.

Java

```
com.ooc.CosPropertyService.Server
   [-h,--help] [-v,--version] [-i,--ior]
```

Options

`-h` `--help`	Display the command-line options supported by the server.
`-v` `--version`	Display the version of the server.
`-i` `--ior`	Prints the stringified IOR of the server to standard output.

11.1.2 Configuration Properties

In addition to the standard configuration properties described in Chapter 4, ORBACUS Properties also supports the following properties:

`ooc.property.port=PORT`	Specifies the port number on which the service should listen for new connections. Note that this property is only considered if the `ooc.oa.port` property is not set.

11.1.3 CLASSPATH Requirements

ORBACUS Properties for Java requires the classes in `OB.jar` and `OBProperty.jar`.

11.2 Connecting to the Service

The object key of the Property Service is `PropertyService`, which identifies an object of type `CosPropertyService::PropertySetDefFactory`.

The object key can be used when composing URL-style object references. For example, the following URL identifies the Property Service running on host `prophost` at port `10000`:

```
corbaloc::prophost:10000/PropertyService
```

Refer to Chapter 6 for more information on URLs and configuring initial services.

11.3 Using the Property Service with the IMR

The Property Service may be used with the Implementation Repository (IMR). However, if used with the IMR, it is important to note that the corbaloc URL-style object reference described in the previous section cannot be used. If the IMR is used, then the object reference for the Property Service must be created using one of the following methods (where `PropertyServer` refers to the server name configured with the IMR):

- start the Property Service with the options:

  ```
  --ior -ORBserver_name PropertyServer
  ```

  causing the Property Service to print its reference to standard output.
- use the `mkref` utility:

  ```
  mkref PropertyServer PropertyService PropertyServicePOA
  ```

When using the Property Service with the IMR, the service must be started with the option `-ORBserver_name PropertyServer`, where `PropertyServer` refers to the server name configured with the IMR. When the IMR is configured to start the Property Service, this option is automatically added to the service's arguments. However, when the Property Service is started manually, the option must be present. For further information on configuring a service with the IMR, refer to "Getting Started with the Implementation Repository" on page 124.

11.4 Property Service Concepts

11.4.1 Creating Properties

A property handled by the CORBA Property Service consists of two components: the property's name and its value. The name is 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;
};
```

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.

11.4.2 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 that 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`.

11.4.3 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 Property Service [9] specification for details.

11.5 Programming 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 {
16    // The object reference is invalid
17 }
18
19 PropertySetDefFactory factory = null;
20 try
21 {
22    factory = PropertySetDefFactoryHelper.narrow(obj);
23 }
24 catch(org.omg.CORBA.BAD_PARAM ex)
25 {
26    // This object does not implement the Property Service
```

```
27 }
28
29 PropertySetDef set = factory.create_propertysetdef();
30
31 Any anyLong = orb.create_any();
32 Any AnyString = orb.create_any();
33 Any anyShort = orb.create_any();
34 anyLong.insert_long(12345L);
35 anyString.insert_string("Foo");
36 anyShort.insert_short((short)0);
37
38 try
39 {
40    set.define_property("LongProperty", anyLong);
41    set.define_property("StringProperty", anyString);
42    set.define_property("ShortProperty", anyShort);
43 }
44 catch(ReadOnlyProperty ex)
45 {
46    // An error occurred
47 }
48 catch(ConflictingProperty ex)
49 {
50    // An error occurred
51 }
52 catch(UnsupportedProperty ex)
53 {
54    // An error occurred
55 }
56 catch(UnsupportedTypeCode ex)
57 {
58    // An error occurred
59 }
60 catch(InvalidPropertyName ex)
61 {
62    // An error occurred
63 }
64
65 PropertiesHolder ph = new PropertiesHolder();
66 PropertiesIteratorHolder ih = new PropertiesIteratorHolder();
67 set.get_all_properties(0, ph, ih);
68
69 PropertyHolder h = new PropertyHolder();
70 while(ih.value.next_one(h))
71 {
```

```
72      // The next property is now stored in h.value
73  }
74
75  ih.value.destroy();
```

5-27 Get a Property Service reference and check for errors.

29 The `PropertySetDefFactory` object is used to create a `PropertySetDef` instance. Note that `PropertySetDef` is a subclass of `PropertySet`.

31-36 Each property consists of a name and a value in the form of a CORBA `Any`.

38-63 Three properties are defined. The first has the name “LongProperty” and stores a `long` value. The second one is called “StringProperty” and stores a `string`. The remaining property represents a `short` value. If for some reason a property cannot be created, an exception is thrown.

65-73 Now we try to get a list of all the properties that were previously defined. With `get_all_properties` the `PropertySetDef` returns its properties. As we have set the `how_many` parameter to 0, we have to use the `PropertiesIterator` for each item. An application would normally provide a positive integer for `how_many`.

75 The iterator has fulfilled its duty and can now be destroyed.

CHAPTER 12

ORBacus Time

The CORBA Time Service provides the means to obtain the current time along with an error estimate. The service also provides operations related to time and intervals.

This chapter does not provide a complete description of the Time Service. It only provides an overview, suitable for getting started. For detailed information, please refer to the Time Service specification [9].

12.1 Compliance Statement

ORBACUS Time is compliant with [9] and conforms to the Basic Time Service. The following presents the necessary documentation for conformance.

12.1.1 Criteria to Be Followed for Secure Time

The Time Service specification states:

The following types of operations must be protected against unauthorized invocation. They must also be mandatorily audited:

- *Operations that set or reset the current time*
- *Operations that designate a time source as authoritative*
- *Operations that modify the accuracy of the time service or the uncertainty interval of generated timestamps.*

The UNIX and Windows NT/2000 operating systems do not provide mandatory auditing for the system time, which is the source of time for ORBACUS Time. Hence, ORBACUS Time cannot provide secure time. As required, the `TimeUnavailable` exception is raised when the `secure_universal_time` operation of the `TimeService` interface is invoked.

12.1.2 Proxies and Time Uncertainty

The specification states:

In a CORBA system, the use of proxy objects can render time values unreliable by introducing unpredictable latency between the time the time server object generates a timestamp and the time the caller's time server proxy receives the timestamp and returns it to the caller.

ORBACUS Time prevents this problem from occurring by requiring a Time Service implementation in every address space that will need to make Time Service calls.

12.2 Synopsis

12.2.1 Usage

As stated in 12.1.2, ORBACUS Time can only be used as a collocated server, so there is no server executable. In C++, the Time Service is initialized with `OB::TimeServiceInit()`, which is declared in `OB/TimeService.h`. For example:

```
// C++
int main(int argc, char* argv[])
{
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    OB::TimeServiceInit(orb, argc, argv);
    // ...
}
```

In Java, the Time Service is initialized in a similar manner:

```
// Java
public static void main(String args[])
{
    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
       "com.ooc.CORBA.ORBSingleton");
```

```
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
    com.ooc.CosTime.TimeService.init(orb, args);
    // ...
}
```

To obtain a reference to the Time Service after initialization, the client must invoke `resolve_initial_references("TimeService")` on the ORB.

In addition to the standard command-line arguments described in Chapter 4, clients that use ORBACUS Time accept the following command-line option:

`-TIMEinaccuracy VALUE`	Specifies the inaccuracy of time in 100 nanosecond units. The default is 0.

12.2.2 Configuration Properties

In addition to the standard configuration properties described in Chapter 4, ORBACUS Time also supports the following property:

`ooc.time.inaccuracy=VALUE`	Equivalent to the `-TIMEinaccuracy` command-line option.

12.2.3 CLASSPATH Requirements

ORBACUS Time for Java requires the classes in `OB.jar` and `OBTime.jar`.

12.3 Time Service Concepts

The section presents an overview of the Time Service specification. For details refer to [9]. ORBACUS specific extensions are also presented.

12.3.1 Representation of Time

The Time Service uses the Universal Time Coordinated (UTC) representation from the *X/Open DCE Time Service*.

Absolute UTC time is defined as follows:

Time units	100 nanoseconds (10^{-7} seconds)

Base time	15 Oct 1582 00:00:00
Approximate range	AD 30,000

Absolute UTC time in the Time Service always refers to time in Greenwich Time (GMT) Zone.

Note that the base time used in the Time Service differs from the base time of the commonly used POSIX time representation. The base time of the POSIX time representation is 1 January 1970 00:00:00.

Relative UTC time is defined as follows:

Time units	100 nanoseconds (10^{-7} seconds)
Approximate range	+/- 30,000

12.3.2 Basic Types

Data structures used by the Time Service are declared in the `TimeBase` module. An overview of the various structure follow:

```
1  // IDL
2  module TimeBase
3  {
4     unsigned long long TimeT;
5     unsigned long long InaccuracyT;
6     short TdfT;
7     struct UtcT
8     {
9        TimeT time;
10       unsigned long inacclo;
11       unsigned short inacchi;
12       TdfT tdf;
13    };
14    struct IntervalT
15    {
16       TimeT lower_bound;
17       TimeT upper_bound;
18    };
19 };
```

4 An absolute or relative time.

5 The value of inaccuracy in an absolute or relative time in units of 100 nanoseconds. The inaccuracy defines an error envelope around a time value that has a lower bound of `max(0, time - inaccuracy)` and an upper bound of `min(maximum time, time + inaccuracy)`. Only the first 48 bits of the 64 available bits are used to hold inaccuracy. Using a value requiring more than 48 bits as an inaccuracy argument of a Time Service operation will raise a `CORBA::BAD_PARAM` exception.

6 The time displacement factor is in the form of minutes of displacement from the Greenwich Meridian. Displacements East of the meridian are positive, and displacements West are negative. Hence, adding the time displacement factor to an absolute time will give the time in the local time zone.

7-13 Represents an absolute or relative time along with its associated inaccuracy and time displacement factor. Note that `inacclo` and `inacchi` together make the 48 bit inaccuracy value.

14-18 Represents a time interval.

12.3.3 Enumerations

Enumerations used by the Time Service are declared in the `CosTime` module. An overview of the various enumerations follow:

```
1  // IDL
2  module CosTime
3  {
4      enum ComparisonType
5      {
6          IntervalC,
7          MidC
8      };
9      enum TimeComparison
10     {
11         TCEqualTo,
12         TCLessThan,
13         TCGreaterThan,
14         TCIndeterminate
15     };
16     enum OverlapType
17     {
18         OTContainer,
19         OTContained,
20         OTOverlap,
```

```
21          OTNoOverlap
22      };
23  };
```

4-8 This enumeration defines two types of time comparisons. `IntervalC` comparisons take into account the error envelope. `MidC` comparisons ignore the error envelope and performs the comparison based solely on the time values.

9-14 This enumeration defines the possible results of a time comparison. `MidC` comparisons can never result in `TCIndeterminate`. `IntervalC` comparisons result in `TCIndeterminate` if the error envelops overlap.

16-21 This enumeration defines the type of overlap between two intervals.

- `OTContainer` implies that interval A wholly contains interval B. The overlap interval is equal to interval B.
- `OTContained` implies that interval B wholly contains interval A or intervals A and B are equal. The overlap interval is equal to interval A. (Note: the specification does not define the result comparing equivalent intervals.)
- `OTOverlap` indicates that neither interval wholly contains the other but there is overlap. The overlap interval is the intersection of the two intervals.
- `OTNoOverlap` indicates that the two intervals do not intersect. The overlap interval is the gap between the two intervals.

12.3.4 Exceptions

In addition to the standard CORBA exceptions, the Time Service may raise the `TimeUnavailable` exception that is declared in the `CosTime` module. ORBACUS Time will always raise the `TimeUnavailable` exception when the `secure_universal_time` operation of the `TimeService` interface is invoked. This is because ORBACUS Time does not provide secure time.

12.3.5 The Universal Time Object

The Universal Time Object (UTO) represents an absolute or relative time along with its associated inaccuracy and time displacement factor. The UTO also provides various operations related to time. The UTO is declared in the `CosTime` module as follows:

```
1  // IDL
2  module CosTime
3  {
4      interface TIO; // forward declaration
```

```
5     interface UTO
6     {
7         readonly attribute TimeBase::TimeT time;
8         readonly attribute TimeBase::InaccuracyT inaccuracy;
9         readonly attribute TimeBase::TdfT tdf;
10        readonly attribute TimeBase::UtcT utc_time;
11        UTO absolute_time();
12        TimeComparison compare_time(
13            in ComparisonType comparison_type, in UTO uto);
14        TIO time_to_interval(in UTO uto);
15        TIO interval();
16    };
17 };
```

7-9 Attributes of the UTO.

10 UtcT structure containing the same attributes as the UTO.

11 Return a UTO with the absolute time corresponding to the relative time of the UTO. Will raise a `CORBA::DATA_CONVERSION` exception in the case that the operation would result in a overflow.

12-13 Compare the time of the UTO parameter with the time of the UTO (the time of the UTO parameter is the second parameter in the comparison). See “Enumerations” on page 169 for details. Will raise a `CORBA::BAD_PARAM` exception if the UTO parameter is nil or if its inaccuracy attribute is greater then the maximum allowable inaccuracy.

14 Return a TIO representing the time interval between the time of the UTO and the time of the UTO parameter. Will raise a `CORBA::BAD_PARAM` exception if the UTO parameter is nil or if its inaccuracy attribute is greater then the maximum allowable inaccuracy.

15 Return a TIO representing the error envelope of the UTO.

12.3.6 The Time Interval Object

The Time Interval Object (TIO) represents a time interval and provides various operations related to time intervals. The TIO is declared in the `CosTime` module as follows:

```
1  // IDL
2  module CosTime
3  {
4      interface TIO
5      {
6          readonly attribute TimeBase::IntervalT time_interval;
7          OverlapType spans(in UTO time, out TIO overlap);
```

```
8         OverlapType overlaps(in TIO interval, out TIO overlap);
9         UTO time();
10     };
11 };
```

6 The time interval represented by the TIO.

7 Compare the time interval of the TIO (interval A) with the associated error envelope of the UTO parameter (interval B). See “Enumerations” on page 169 for details. Will raise a `CORBA::BAD_PARAM` exception if the UTO parameter is nil or if its inaccuracy attribute is greater then the maximum allowable inaccuracy.

8 Compare the time interval of the TIO (interval A) with the time interval of the TIO parameter (interval B). See “Enumerations” on page 169 for details. Will raise a `CORBA::BAD_PARAM` exception if the TIO parameter is nil or if its lower bound is greater than its upper bound.

9 Return a UTO with an associated error envelop equal to the time interval of the TIO and a time equal to the midpoint of the time interval. Will raises a `CORBA::DATA_CONVERSION` exception if the resulting inaccuracy is greater than the maximum allowable inaccuracy.

12.3.7 The TimeService Object

The TimeService Object provides operations for getting the current time and creating arbitrary UTOs and TIOs. The TimeService is declared in the `CosTime` module as follows:

```
1  // IDL
2  module CosTime
3  {
4     interface TimeService
5     {
6        UTO universal_time()
7           raises(TimeUnavailable);
8        UTO secure_universal_time()
9           raises(TimeUnavailable);
10       UTO new_universal_time(in TimeBase::TimeT time,
11          in TimeBase::InaccuracyT inaccuracy,
12          in TimeBase::TdfT tdf);
13       UTO uto_from_utc(in TimeBase::UtcT utc);
14       TIO new_interval(in TimeBase::TimeT lower,
15          in TimeBase::TimeT upper);
```

```
16      };
17  };
```

6-7 Return a UTO with the current time.

8-9 Since ORBACUS Time is not secure, this operation will always raise a `TimeUnavailable` exception.

10-13 Both `new_universal_time` and `uto_from_utc` return a UTO with the specified attributes.

14-15 Return a TIO with the specified lower and upper bounds.

12.4 Time Service Extensions

ORBACUS provides additional operation for working with the structures defined in the `TimeBase` module. In C++, these operations are provided by the `TimeHelper` class and global functions and are part of the `OB` module. Declarations are as follows:

```
1  // C++
2  module OB
3  {
4
5  class TimeHelper
6  {
7  public:
8      static const CORBA::ULongLong MaxTimeT =
9          OB_ULONGLONG(0xffffffffffffffff);
10     static const CORBA::ULongLong MaxInaccuracyT =
11         OB_ULONGLONG(0xfffffffffff);
12
13     static TimeBase::UtcT utcNow(TimeBase::InaccuracyT = 0);
14     static TimeBase::UtcT utcMin();
15     static TimeBase::UtcT utcMax();
16     static timeval toTimeval(const TimeBase::UtcT&);
17     static timeval toTimeval(TimeBase::TimeT);
18     static TimeBase::IntervalT toIntervalT(TimeBase::TimeT,
19         TimeBase::InaccuracyT);
20     static TimeBase::UtcT toUtcT(TimeBase::TimeT,
21         TimeBase::InaccuracyT, TimeBase::TdfT = 0);
22     static TimeBase::UtcT toUtcT(const TimeBase::IntervalT&);
23     static char* toString(const TimeBase::UtcT&);
24     static char* toTimeString(const TimeBase::UtcT&);
```

```
25     static char* toString(TimeBase::TimeT);
26 };
27
28 bool operator<(const TimeBase::UtcT&, const TimeBase::UtcT&);
29 bool operator<=(const TimeBase::UtcT&, const TimeBase::UtcT&);
30 bool operator>(const TimeBase::UtcT&, const TimeBase::UtcT&);
31 bool operator>=(const TimeBase::UtcT&, const TimeBase::UtcT&);
32 bool operator==(const TimeBase::UtcT&, const TimeBase::UtcT&);
33 bool operator!=(const TimeBase::UtcT&, const TimeBase::UtcT&);
34 TimeBase::UtcT operator+(const TimeBase::UtcT&, TimeBase::TimeT);
35 TimeBase::UtcT operator+(TimeBase::TimeT, const TimeBase::UtcT&);
36 TimeBase::UtcT operator-(const TimeBase::UtcT&, TimeBase::TimeT);
37
38 }; // end of module OB
```

8-9 The maximum value of the `TimeT` type.

10-11 The maximum value of the `InaccuracyT` type.

13 Return the current UTC time.

14 Return the minimum UTC time.

15 Return the maximum UTC time.

16-17 Return the equivalent POSIX time.

18-19 Create an interval from a time value and its inaccuracy (similar to `UTC::time_to_interval`).

20-21 Create a UTC time from its different components.

22 Create a UTC time from an interval (similar to `TIO::time`).

23-25 Return the string representation of the UTC time.

- `toString(const TimeBase::UtcT&)` returns the time and date,
- `toTimeString(const TimeBase::UtcT&)` returns only the time, and
- `toString(TimeBase::TimeT)` returns a string of the form `seconds:milliseconds`.

Use `CORBA::string_free()` to free the return value.

28-36 Various relational and arithmetic operators.

Similar operations are available in Java. Declarations are as follows:

```
1  // Java
2  package com.ooc.CosTime
3
4  import org.omg.CORBA.*;
5  import org.omg.TimeBase.*;
6
7  public class TimeHelper
8  {
9      public final static long MaxTimeT = 0xffffffffffffffffL;
10     public final static long MaxInaccuracyT = 0xfffffffffffL;
11
12     public static UtcT utcNow(long inaccuracy);
13     public static UtcT utcMin();
14     public static UtcT utcMax();
15     public static long toJavaMillis(UtcT utc);
16     public static long toJavaMillis(long time);
17     public static IntervalT toIntervalT(long time, long inaccuracy);
18     public static UtcT toUtcT(long time, long inaccuracy, short tdf);
19     public static UtcT toUtcT(long time, long inaccuracy);
20     public static UtcT toUtcT(IntervalT inter);
21     public static String toString(UtcT utc);
22     public static String toTimeString(UtcT utc);
23     public static String toString(long time);
24
25     public static boolean lessThan(UtcT a, UtcT b);
26     public static boolean lessThanEqual(UtcT a, UtcT b);
27     public static boolean greaterThan(UtcT a, UtcT b);
28     public static boolean greaterThanEqual(UtcT a, UtcT b);
29     public static boolean equal(UtcT a, UtcT b);
30     public static boolean notEqual(UtcT a, UtcT b);
31     public static UtcT add(UtcT a, long t);
32     public static UtcT add(long t, UtcT a);
33     public static UtcT subtract(UtcT a, long t);
34 }
```

9 The maximum value of the `TimeT` type.

10 The maximum value of the `InaccuracyT` type.

12 Return the current UTC time.

13 Return the minimum UTC time.

14 Return the maximum UTC time.

15 Return the equivalent POSIX time in milliseconds.

16 Return the equivalent POSIX time in milliseconds.

17 Create an interval from a time value and its inaccuracy (similar to `UTC::time_to_interval`).

18 Create a UTC time from its different components.

19 Create a UTC time from an interval (similar to `TIO::time`).

21-23 Return the string representation of the UTC time. See description of C++ versions above.

25-33 Various relational and arithmetic operators.

12.5 Programming Example

This section presents a simple program that uses ORBACUS Time to implement a stop watch. The program is split into three functions `main()`, `run()` and `stopwatch()`. `main()` only creates the ORB, initializes the Time Service, and calls `run()`:

```
1  // C++
2  #include <OB/CORBA.h>
3  #include <OB/TimeService.h>
4  #include <OB/TimeHelper.h>
5
6  #include <iostream>
7
8  using namespace std;
9
10 int run(CORBA::ORB_ptr);
11 int stopwatch(CosTime::TimeService_ptr);
12
13 int main(int argc, char* argv[])
14 {
15    int status = EXIT_SUCCESS;
16    CORBA::ORB_var orb;
17
18    try
19    {
20       orb = CORBA::ORB_init(argc, argv);
21       OB::TimeServiceInit(orb, argc, argv);
22       status = run(orb);
23    }
24    catch(const CORBA::Exception& ex)
25    {
26       cerr << ex << endl;
```

```
27          status = EXIT_FAILURE;
28      }
29
30      if(!CORBA::is_nil(orb))
31      {
32          try
33          {
34              orb -> destroy();
35          }
36          catch(const CORBA::Exception& ex)
37          {
38              cerr << ex << endl;
39              status = EXIT_FAILURE;
40          }
41      }
42
43      return status;
44  }
```

2-6 Several header files are included. `OB/CORBA.h` provides standard CORBA definitions, `OB/TimeService.h` provides ORBACUS Time definitions, and `OB/TimeHelper.h` provides definitions for ORBACUS Time extensions.

10-11 Forward declarations for the `run()` and `stopwatch()` functions.

20 Initialize the ORB.

21 Initialize the Time Service.

22 Call the `run()` helper function.

30-41 If the ORB was successfully created, it is destroyed.

The run method resolves the Time Service initial reference and calls `stopwatch()`:

```
1   // C++
2   int run(CORBA::ORB_ptr orb)
3   {
4       CORBA::Object_var obj;
5
6       try
7       {
8           obj = orb -> resolve_initial_references("TimeService");
9       }
10      catch(const CORBA::ORB::InvalidName&)
11      {
```

```
12        cerr << "Can't resolve 'TimeService'" << endl;
13        return EXIT_FAILURE;
14    }
15
16    if(CORBA::is_nil(obj))
17    {
18        cerr << "'TimeService' is a nil object reference" << endl;
19        return EXIT_FAILURE;
20    }
21
22    CosTime::TimeService_var ts =
23        CosTime::TimeService::_narrow(obj);
24
25    if(CORBA::is_nil(ts))
26    {
27        cerr << "'TimeService' is not a TimeService "
28            << "object reference" << endl;
29        return EXIT_FAILURE;
30    }
31
32    return stopwatch(ts);
33 }
```

8 Using the ORB reference, `resolve_initial_reference` is invoked to obtain a reference to the Time Service.

23-23 The reference is then narrowed to a reference of type `CosTime::TimeService`.

`stopwatch()` uses the Time Service to implement a stop watch:

```
1  // C++
2  int stopwatch(CosTime::TimeService_ptr ts)
3  {
4      CosTime::UTO_var start;
5      CosTime::UTO_var stop;
6
7      try
8      {
9          cout << "Press Enter to start " << flush;
10         cin.get();
11         start = ts -> universal_time();
12         cout << "Press Enter to stop " << flush;
13         cin.get();
14         stop = ts -> universal_time();
15     }
```

```
16      catch(const CosTime::TimeUnavailable&)
17      {
18          cout << "Time not available" << endl;
19          return EXIT_FAILURE;
20      }
21
22      CORBA::String_var str =
23          OB::TimeHelper::toTimeString(start -> utc_time());
24      cout << "Start time:   " << str << endl;
25
26      str = OB::TimeHelper::toTimeString(stop -> utc_time());
27      cout << "Stop time:    " << str << endl;
28
29      TimeBase::TimeT elapsed = stop -> time() - start -> time();
30      cout << "Elapsed time: " << (unsigned long)(elapsed / 10000)
31          << " ms" << endl;
32
33      return EXIT_SUCCESS;
34  }
```

7-20 Get the start and stop times.

22-27 Output the start and stop times.

29-31 Compute elapsed time and output result.

CHAPTER 13

ORBacus Events

Some applications need 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.

ORBACUS Events is compliant with [9]. This chapter does not provide a complete description of the service. It only provides an overview, suitable to get you started. For more information, please refer to the specification.

13.1 Synopsis

13.1.1 Usage

ORBACUS includes functionally equivalent implementations of the Event Service in C++ and Java.

C++

```
eventserv
   [-h,--help] [-v,--version] [-i,--ior] [-t,--typed-service]
   [-u,--untyped-service]
```

Java

```
com.ooc.CosEvent.Server
   [-h,--help] [-v,--version] [-i,--ior] [-t,--typed-service]
   [-u,--untyped-service]
```

Options

`-h` `--help`	Display the command-line options supported by the server.
`-v` `--version`	Display the version of the server.
`-i` `--ior`	Print the stringified IOR of the server to standard output.
`-t` `--typed-service`	Run a typed event service.
`-u` `--untyped-service`	Run an untyped event service. This is the default behavior.

13.1.2 Windows NT Native Service

The C++ version of ORBACUS Events is also available as a native Windows NT service.

```
nteventservice
   [-h,--help] [-i,--install] [-s,--start-install]
   [-u,--uninstall] [-d,--debug]
```

`-h` `--help`	Display the command-line options supported by the server.
`-i` `--install`	Install the service. The service must be started manually.
`-s` `--start-install`	Install and start 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 `ooc.event.port` property to the `HKEY_LOCAL_MACHINE` NT registry key (see "Using the Windows NT Registry" on page 60 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`. Alternatively, the service could have been installed using the `-s` option, which configures the service for automatic start-up:

```
nteventservice -s
```

If you want to remove the service, run:

```
nteventservice -u
```

Note: If the executable for the Event Service is moved, it must be uninstalled and re-installed.

Any trace information provided by the service is be placed in the Windows NT Event Viewer with the title `EventService`. To enable tracing information, add the desired trace configuration property (i.e., one of the `ooc.event.trace` properties or one of the `ooc.orb.trace` properties) to the `HKEY_LOCAL_MACHINE` NT registry key with a `REG_SZ` value of at least 1.

13.1.3 Configuration Properties

In addition to the standard configuration properties described in Chapter 4, ORBACUS Events also supports the following properties:

`ooc.event.inactivity_timeout=sec`	Proxies that are inactive for the specified number of seconds will be reaped. The default value is four hours.
`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`.
`ooc.event.port=port`	Specifies the port number on which the service should listen for new connections. Note that this property is only considered if the `ooc.oa.port` property is not set.
`ooc.event.pull_interval=msec`	This specifies the number of milliseconds between successive calls to pull on `PullSupplier`. Default value is 0.
`ooc.event.reap_frequency=sec`	This specifies the frequency (in seconds) in which inactive proxies will be reaped. The default value is thirty minutes. Setting this property to 0 disables the reaping of proxies.
`ooc.event.retry_timeout=msec`	Specifies the initial amount of time in milliseconds that the service waits 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.trace.events=LEVEL`	Defines the output level for event diagnostic messages printed by ORBACUS Events. The default level is 0, which produces no output. A level of 1 or higher produces event processing information and a level of 2 or higher produces event creation and destruction information.
`ooc.event.trace.lifecycle=LEVEL`	Defines the output level for lifecycle diagnostic messages printed by ORBACUS Events. The default level is 0, which produces no output. A level of 1 or higher produces lifecycle information (e.g. creation and destruction of Suppliers and Consumers).
`ooc.event.typed_service`	Equivalent to the `-t` command-line option.

13.1.4 Diagnostics

ORBACUS Events generates diagnostic messages if the `ooc.orb.trace_level` property is set to `2`.

13.1.5 CLASSPATH Requirements

ORBACUS Events for Java requires the classes in `OB.jar` and `OBEvent.jar`.

13.2 Connecting to the Service

The object key of the Event Service depends on whether it is running as a "typed" or "untyped" service. The object keys and corresponding interface types are shown in Table 13.1.

	Object Key	Interface Type
Event Service	`DefaultEventChannel`	`CosEventChannelAdmin::EventChannel`
Typed Event Service	`DefaultTypedEventChannel`	`CosTypedEventChannelAdmin::TypedEventChannel`

Table 13.1: Primary Object Keys and Interface Types

The object key can be used when composing URL-style object references. For example, the following URL identifies the untyped event service running on host `evhost` at port `10000`:

```
corbaloc::evhost:10000/DefaultEventChannel
```

Refer to Chapter 6 for more information on URLs and configuring initial services.

ORBACUS Events also provides proprietary "factory" interfaces which allow construction and administration of multiple event channels in a single service. The object keys and corresponding interface types of the factories are shown in Table 13.2.

	Object Key	Interface Type
Event Channel Factory	`DefaultEventChannelFactory`	`OBEventChannelFactory::EventChannelFactory`
Typed Event Channel Factory	`DefaultTypedEventChannelFactory`	`OBTypedEventChannelFactory::TypedEventChannelFactory`

Table 13.2: Factory Object Keys and Interface Types

For a description of the factory interfaces, please refer to the documented IDL files `event/idl/OBEventChannelFactory.idl` and `event/idl/OBTypedEventChannelFactory.idl`.

13.3 Using the Event Service with the IMR

The Event Service may be used with the Implementation Repository (IMR). However, if used with the IMR, it is important to note that the `corbaloc` URL-style object reference described in the previous section cannot be used. If the IMR is used, then the object reference for the "untyped" Event Service must be created using one of the following methods (where `EventServer` refers to the server name configured with the IMR):

- start the Event Service with the options:

  ```
  -ORBserver_name EventServer --ior
  ```

  causing the Event Service to print its reference to standard output.
- use the `mkref` utility:

  ```
  mkref EventServer DefaultEventChannel EventServicePOA
  ```

For the "typed" Event Service, the object reference must be created using one of the following methods:

- start the Event Service with the options:

  ```
  -ORBserver_name EventServer --typed-service --ior
  ```

  causing the Event Service to print its reference to standard output.
- use the `mkref` utility:

  ```
  mkref EventServer DefaultTypedEventChannel EventServicePOA
  ```

Object references for the ORBACUS proprietary "factory" objects can be created using the following commands:

```
mkref EventServer DefaultEventChannelFactory EventServicePOA
mkref EventServer DefaultTypedEventChannelFactory EventServicePOA
```

When using the Event Service with the IMR, the service must be started with the option `-ORBserver_name EventServer`, where `EventServer` refers to the server name configured with the IMR. When the IMR is configured to start the Event Service, this option is automatically added to the service's arguments. However, when the Event Service is started manually, the option must be present. For further information on configuring a service with the IMR, refer to "Getting Started with the Implementation Repository" on page 124.

13.4 Event Service Concepts

13.4.1 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 [9].

13.4.2 Event Suppliers and Consumers

Applications participating in generating and accepting events are called *suppliers* and *consumers*, respectively. Suppliers and consumers each come in two different versions, namely, *push suppliers* and *pull suppliers*, and *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. Some consumers must be immediately informed when new events become available on an event channel. Such 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();
};
```

Push consumers are passive, that is, are servers. Conversely, pull consumers are active, that is, are clients. Pull consumers poll an event channel for new events. As events may arrive at a greater rate than they are polled for by a pull consumer or accepted and processed by a push consumer, some events might get lost. A buffering policy implemented by the event channel determines whether events are buffered and what happens in case of an event queue overflow.

Like consumers, suppliers can also use push or pull behavior. Push suppliers are the more common type, in which the supplier directly forwards data to the event channel and thus plays the client 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();
};
```

13.4.3 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 queue, i.e., a certain number of events are stored and, in case of a buffer overflow, the oldest events are discarded.

13.4.4 Event Channel Factories

The standard CORBA Event Service provides no support for managing the lifecycle of event channels; as a result, applications requiring multiple channels are often forced to run a separate instance of the Event Service for each channel. To remedy this situation, ORBACUS Events provides optional, proprietary interfaces for event channel administration.

The `OBEventChannelFactory::EventChannelFactory` interface describes the factory for untyped event channels:

```
// IDL
module OBEventChannelFactory
{
typedef string ChannelId;
typedef sequence<ChannelId> ChannelIdSeq;

exception ChannelAlreadyExists {};
```

```
exception ChannelNotAvailable {};

interface EventChannelFactory
{
    CosEventChannelAdmin::EventChannel
    create_channel(in ChannelId id)
         raises(ChannelAlreadyExists);

    CosEventChannelAdmin::EventChannel
    get_channel_by_id(in ChannelId id)
         raises(ChannelNotAvailable);

    ChannelIdSeq get_channels();

    void shutdown();
};
};
```

The `OBTypedEventChannelFactory::TypedEventChannelFactory` interface describes the factory for typed event channels:

```
// IDL
module OBTypedEventChannelFactory
{
interface TypedEventChannelFactory
{
    CosTypedEventChannelAdmin::TypedEventChannel
    create_channel(in OBEventChannelFactory::ChannelId id)
         raises(OBEventChannelFactory::ChannelAlreadyExists);

    CosTypedEventChannelAdmin::TypedEventChannel
    get_channel_by_id(in OBEventChannelFactory::ChannelId id)
         raises(OBEventChannelFactory::ChannelNotAvailable);

    OBEventChannelFactory::ChannelIdSeq get_channels();

    void shutdown();
};
};
```

At start-up, the untyped Event Service creates a single channel having the identifier `DefaultEventChannel`, and the typed Event Service creates a single channel having the identifier `DefaultTypedEventChannel`. A channel's identifier also serves as its object key; therefore, a channel can be located using a `corbaloc:` URL (see "corbaloc: URLs"

on page 105). For example, a channel with the identifier `TelemetryData` can be located on the host `myhost` at port `2098` using the following URL:

```
corbaloc::myhost:2098/TelemetryData
```

To obtain the object reference of a channel factory, use a `corbaloc:` URL with the object key as shown in Table 13.1 on page 185. For example, assuming the untyped Event Service is running on host `myhost` at port `2098`, here is how a C++ application can obtain the object reference of the channel factory and create a channel with the identifier `TelemetryData`:

```
// C++
CORBA::Object_var obj = orb -> string_to_object(
   "corbaloc::myhost:2098/DefaultEventChannelFactory");
OBEventChannelFactory::EventChannelFactory_var factory =
   OBEventChannelFactory::EventChannelFactory::_narrow(obj);
CosEventChannelAdmin::EventChannel_var channel =
   factory -> create_channel("TelemetryData");
```

Here is the same example in Java:

```
// Java
org.omg.CORBA.Object obj = orb.string_to_object(
   "corbaloc::myhost:2098/DefaultEventChannelFactory");
com.ooc.OBEventChannelFactory.EventChannelFactory factory =
   com.ooc.OBEventChannelFactory.EventChannelFactoryHelper.
   narrow(obj);
org.omg.CosEventChannelAdmin.EventChannel channel =
   factory.create_channel("TelemetryData");
```

13.5 Programming 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 provides 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:

```
1 // Java
2 while(consumer_ != null)
```

```
3  {
4      java.util.Date date = new java.util.Date();
5      String s = "PushSupplier says: " + date.toString();
6
7      Any any = orb_.create_any();
8      any.insert_string(s);
9
10     try
11     {
12        consumer_.push(any);
13     }
14     catch(Disconnected ex)
15     {
16        // Supplier was disconnected from event channel
17     }
18
19     try
20     {
21        Thread.sleep(1000);
22     }
23     catch(InterruptedException ex)
24     {
25     }
26 }
```

4-8 The current date and time is inserted into the `Any`.

10-17 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.

19-25 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:

```
1  // Java
2  public Any pull()
3  {
4      java.util.Date date = new java.util.Date();
5      String s = "PullSupplier says: " + date.toString();
6
```

```
 7      Any any = orb.create_any();
 8      any.insert_string(s);
 9
10      return any;
11  }
12
13  public Any
14  try_pull(BooleanHolder has_event)
15  {
16      has_event.value = true;
17
18      return pull();
19  }
```

4-8 Date and time are inserted into the `Any`.

13-19 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  public void push(Any any)
 3  {
 4      try
 5      {
 6          String s = any.extract_string();
 7          System.out.println(s);
 8      }
 9      catch(MARSHAL ex)
10      {
11         // Ignore unknown event data
12      }
13  }
```

2-13 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  while(supplier_ != null)
3  {
4      Any any = null;
5
6      try
7      {
8          any = supplier_.pull();
9      }
10     catch(Disconnected ex)
11     {
12         // Supplier was diconnected from event channel
13     }
14
15     try
16     {
17         String s = any.extract_string();
18         System.out.println(s);
19     }
20     catch(MARSHAL ex)
21     {
22         // Ignore unknown event data
23     }
24 }
```

4 A CORBA `Any` is prepared for later use.

6-13 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.

15-23 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.

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.

CHAPTER 14

The Interface Repository

A CORBA Interface Repository (IFR) is essential for applications using the dynamic features of CORBA, such as the Dynamic Invocation Interface and DynAny. The IFR holds IDL type definitions and can be queried and traversed by applications.

The ORBACUS Interface Repository is compliant with [4]. This chapter does not provide a complete description of the IFR. For more information, please refer to the specification.

14.1 Synopsis

14.1.1 Usage

The ORBACUS Interface Repository is currently only provided with ORBACUS for C++.

```
irserv
   [-h,--help] [-v,--version] [-e NAME,--cpp NAME] [-d,--debug]
   [-i,--ior] [-DNAME] [-DNAME=DEF] [-UNAME] [-IDIR] [-E]
   [--case-sensitive] [FILE ...]
```

`-h` `--help`	Display the command-line options supported by the server.
`-v` `--version`	Display the version of the server.

`-e NAME` `--cpp NAME`	Use `NAME` as the preprocessor program.
`-d` `--debug`	Print diagnostic messages. This option is for ORBACUS internal debugging purposes only.
`-i` `--ior`	Print the stringified IOR of the server to standard output.
`-DNAME` `-DNAME=DEF`	Defines `NAME` as `DEF`, or `1` if `DEF` is not provided. This option is passed directly to the preprocessor.
`-UNAME`	Removes any definition for `NAME`. This option is passed directly to the preprocessor.
`-IDIR`	Adds `DIR` to the include file search path. This option is passed directly to the preprocessor.
`-E`	Run only the preprocessor.
`--case-sensitive`	The semantics of OMG IDL forbid identifiers in the same scope to differ only in case. This option relaxes these semantics, but is only provided for backward compatibility with non-compliant IDL.
`FILE ...`	IDL files to be loaded into the repository.

14.1.2 Windows NT Native Service

```
ntirservice
   [-h,--help] [-i,--install] [-s,--start-install]
   [-u,--uninstall] [-d,--debug]
```

`-h` `--help`	Display the command-line options supported by the server.
`-i` `--install`	Install the service. The service must be started manually.
`-s` `--start-install`	Install the service and start it.
`-u` `--uninstall`	Uninstall the service.
`-d` `--debug`	Run the service in debug mode.

In order to use the IFR as a native Windows NT service, it is first necessary to add the `ooc.ifr.port` configuration property to the `HKEY_LOCAL_MACHINE` NT registry key (see "Using the Windows NT Registry" on page 60 for more details).

Next the service should be installed with:

```
ntirservice -i
```

This adds the `ORBacus Interface Repository Service` entry to the `Services` dialog in the Control Panel. To start the naming service, select the `ORBacus Interface Repository Service` entry, and press `Start`. If the service is to be started automatically when the machine is booted, select the `ORBacus Interface Repository Service` entry, then click `Startup`. Next select `Startup Type - Automatic`, and press `OK`. Alternatively, the service could have been installed using the `-s` option, which configures the service for automatic start-up:

```
ntirservice -s
```

If you want to remove the service, run:

```
ntirservice -u
```

Note: If the executable for the Interface Repository is moved, it must be uninstalled and re-installed.

Any trace information provided by the service is placed in the Windows NT Event Viewer with the title `IRService`. To enable tracing information, add the desired trace configuration property (i.e., one of the `ooc.orb.trace` properties) to the `HKEY_LOCAL_MACHINE` NT registry key with a `REG_SZ` value of at least 1.

14.1.3 Configuration Properties

In addition to the standard configuration properties described in Chapter 4, the ORBACUS Interface Repository also supports the following properties:

`ooc.ifr.options=OPTS`	Allows command-line options to be passed to the Windows NT Native service at start-up. Note that absolute pathnames should be used when specifying include directives, IDL files, etc.
`ooc.ifr.port=PORT`	Specifies the port number on which the service should listen for new connections. Note that this property is only considered if the `ooc.oa.port` property is not set.

14.2 Connecting to the Interface Repository

The object key of the IFR is `DefaultRepository`, which identifies an object of type `CORBA::Repository`.

The object key can be used when composing URL-style object references. For example, the following URL identifies the IFR running on host `ifrhost` at port `10000`:

```
corbaloc::ifrhost:10000/DefaultRepository
```

Refer to Chapter 6 for more information on URLs and configuring initial services.

14.3 Configuration Issues

Although applications can interact with the IFR as with any other CORBA server, it does have special status within the ORB. Specifically, use of the standard operation `Object::get_interface()` requires the presence of an IFR:

```
// PIDL
interface Object
{
   ...
   InterfaceDef get_interface();
   ...
};
```

The exact semantics of `get_interface` can be a source of confusion. In ORBACUS, as with most other ORBs, the `get_interface` operation is a *remote* operation. That is, when a client invokes `get_interface` on an object reference, the request is sent to the server. The server knows the interface type of the object reference and interacts with the IFR to locate the appropriate `CORBA::InterfaceDef` object to return to the client. *Therefore, the server must be configured for the IFR. It is not necessary to configure the client for the IFR if the client's only interaction with the IFR is via* `get_interface`.

14.4 Interface Repository Utilities

14.4.1 irfeed

IDL files can be loaded into the IFR at runtime using `irfeed`. See the description of the `irserv` command for more information on the command-line options.

```
irfeed [-h,--help] [-v,--version] [-e NAME,--cpp NAME] [-d,--debug]
       [-DNAME] [-DNAME=DEF] [-UNAME] [-IDIR] [-E] FILE ...
```

14.4.2 irdel

Type definitions can be removed from the IFR using `irdel`. See the description of the `irserv` command for more information on the command-line options.

```
irdel [-h,--help] [-v,--version] name ...
```

The `name` argument represents the scoped name of the type to be removed. 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".

14.5 Programming Example

Below is a simple example in Java that demonstrates how to obtain an `InterfaceDef` object and display its contents:

```
1  // Java
2  import org.omg.CORBA.*;
3  ...
4
5  org.omg.CORBA.ORB = ... // initialize the ORB
6  org.omg.CORBA.Object obj = ... // get object reference somehow
7
8  org.omg.CORBA.Object defObj = obj._get_interface_def();
9  if(defObj == null)
10 {
11    System.err.println("No Interface Repository available");
12    System.exit(1);
13 }
14
15 InterfaceDef def = InterfaceDefHelper.narrow(defObj);
16 org.omg.CORBA.InterfaceDefPackage.FullInterfaceDescription desc =
17    def.describe_interface();
18
19 int i;
20
21 System.out.println("name = " + desc.name);
22 System.out.println("id = " + desc.id);
23 System.out.println("defined_in = " + desc.defined_in);
24 System.out.println("version = " + desc.version);
25 System.out.println("operations:");
26 for(i = 0 ; i < desc.operations.length ; i++)
27 {
28    System.out.println(i + ": " + desc.operations[i].name);
```

```
29 }
30 System.out.println("attributes:");
31 for(i = 0 ; i < desc.attributes.length ; i++)
32 {
33    System.out.println(i + ": " + desc.attributes[i].name);
34 }
35 System.out.println("base_interfaces:");
36 for(i = 0 ; i < desc.base_interfaces.length ; i++)
37 {
38    System.out.println(i + ": " + desc.base_interfaces[i]);
39 }
```

5-8 After initializing the ORB and obtaining an object reference, we invoke `_get_interface_def`[1] on the object.

9-13 If no interface definition could be found, `_get_interface_def` returns nil.

15 Narrow the object reference to `InterfaceDef`. We now have a reference to an object in the IFR that describes the most-derived type of our object reference.

16-17 Request a complete description of the interface.

19-39 Print information about the interface, including the names of its operations and attributes.

A complete example of how to use the IFR can be found in the `ob/demo/repository` subdirectory.

1. Recent versions of the IDL-to-Java mapping introduced the `_get_interface_def` operation, which returns `org.omg.CORBA.Object` instead of `org.omg.CORBA.InterfaceDef`. Portable Java applications should use `_get_interface_def`. In C++, the operation is `_get_interface`.

CHAPTER 15

Using Policies

15.1 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`.

There are two basic types of policies: those used to configure the ORB and those used to create a new POA. Furthermore, the configuration of ORB policy objects is accomplished at two 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.
- **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.

For more information on Policies, the `PolicyManager` interface and the `CORBA::Object` policy operations see [8] and [4].

15.2 Supported Policies

The following is a brief description of the ORBACUS-specific policies that are currently supported. For a detailed description, please refer to Appendix B. For standard policies, please refer to [4].

OB::ACMTimeoutPolicy

This policy determines whether the ORB performs "active connection management" (ACM) on the connection associated with an object reference. The policy specifies a time after which idle connections are shutdown. A value of 0 means no timeout. The default for this policy is the value of the `ooc.orb.client_timeout` property (see "ooc.orb.client_timeout" on page 50).

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::ConnectTimeoutPolicy

If an object has this policy and a connection cannot be established after `value` milliseconds, a `CORBA::NO_RESPONSE` exception is raised.

OB::InterceptorPolicy

This policy determines whether interceptors will be called. This policy can be set on an ORB or object reference to control client-side interceptors, and can be set on a POA to control server-side interceptors.

OB::LocationTransparencyPolicy

This policy determines how strictly the ORB will enforce location transparency. The default behavior is strict enforcement, but an application may wish to sacrifice strict CORBA compliance to improve performance for local invocations.

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::RequestTimeoutPolicy

If an object has this policy and no response is available for a request after `value` milliseconds, a `CORBA::NO_RESPONSE` exception is raised.

OB::RetryPolicy

This policy is used to specify whether requests should be retried after communication failures.

OB::TimeoutPolicy

If an object has this policy and a connection cannot be established or no response is available for a request after `value` milliseconds, a `CORBA::NO_RESPONSE` exception is raised. If an object has `OB::ConnectTimeoutPolicy` or `OB::RequestTimeoutPolicy` set, those policies have precedence.

15.3 Programming Examples

15.3.1 Connection Reuse Policy

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.

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(0);
4  CORBA::PolicyList policies;
5  policies.length(1);
6  policies[0] = orb -> create_policy(OB::CONNECTION_REUSE_POLICY_ID,
7     boolAny);
8  CORBA::Object_var pmObj =
9      orb -> resolve_initial_references("ORBPolicyManager");
10 CORBA::PolicyManager_var pm = CORBA::PolicyManager::_narrow(pmObj);
11 pm -> add_policy_overrides(policies);
```

2-3 Create an any and insert the value 0 (false).

4-5 Create a sequence containing one policy object.

6-7 Ask the ORB to create a connection reuse policy. Pass the any that contains the value for this policy.

8-10 Obtain the ORB level policy manager object.

11 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_POLICY_ID.value,
7        boolAny);
8  org.omg.CORBA.PolicyManager pm =
9     org.omg.CORBA.PolicyManagerHelper.narrow(
10       orb.resolve_initial_references("ORBPolicyManager"));
11 pm.add_policy_overrides(policies);
```

1-11 This is equivalent to the C++ version.

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(0);
4  CORBA::PolicyList policies(1);
5  policies.length(1);
6  policies[0] = orb -> create_policy(OB::CONNECTION_REUSE_POLICY_ID,
7      boolAny);
8  CORBA::Object_var newObj =
9      obj -> _set_policy_overrides(policies, CORBA::ADD_OVERRIDE);
```

2-6 This is the same as in the example for the ORB level.

Set the policy 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_POLICY_ID.value,
7          boolAny);
8  org.omg.CORBA.Object newObj =
9      obj._set_policy_override(policies,
10         org.omg.CORBA.SetOverrideType.ADD_OVERRIDE);
```

1-9 This is equivalent to the C++ version.

15.3.2 Timeout Policy

This example shows how to configure timeouts at the object level. As usual, the C++ version is presented first, followed by the Java version:

```
1  // C++
2  CORBA::Any ULongAny;
3  ULongAny <<= (CORBA::ULong)1000;
4  CORBA::PolicyList policies(1);
5  policies.length(1);
6  policies[0] = orb -> create_policy(OB::TIMEOUT_POLICY_ID, ULongAny);
```

```
7  CORBA::Object_var newObj =
8      obj -> _set_policy_overrides(policies, CORBA::ADD_OVERRIDE);
```

2-6 We want to set the timeout to a value of 1000 milliseconds.

7-8 Set the policy on the object by using the `_set_policy_overrides` method. This method returns a new object that has the set of policies applied.

And now the same example in Java:

```
1  // Java
2  org.omg.CORBA.Any ULongAny = orb.create_any();
3  ULongAny.insert_ulong(1000);
4  org.omg.CORBA.Policy[] policies = new org.omg.CORBA.Policy[1];
5  policies[0] =
6      orb.create_policy(com.ooc.OB.TIMEOUT_POLICY_ID.value,
7                        ULongAny);
8  org.omg.CORBA.Object newObj =
9      obj._set_policy_override(policies,
10         org.omg.CORBA.SetOverrideType.ADD_OVERRIDE);
```

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

Note that you can also set the timeout policy at the ORB level.

CHAPTER 16

Concurrency Models

16.1 Introduction

16.1.1 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.

16.1.2 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.

16.1.3 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 *Reactive*, *Threaded*, *Thread-per-Client*, *Thread-per-Request* and *Thread Pool*.

16.2 Single-Threaded Concurrency Models

16.2.1 Blocking Clients

The blocking concurrency model is the simplest one. It only applies to the client side and means that the ORB blocks while sending requests.

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.

Because of its simplicity, the blocking concurrency model is the fastest model available. There is no overhead, neither for calls to operations like `select`[2] (because the ORB is allowed to block on a single connection), nor for any thread creation or context switches.

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.

16.2.2 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 is shown in Figure 16.1.

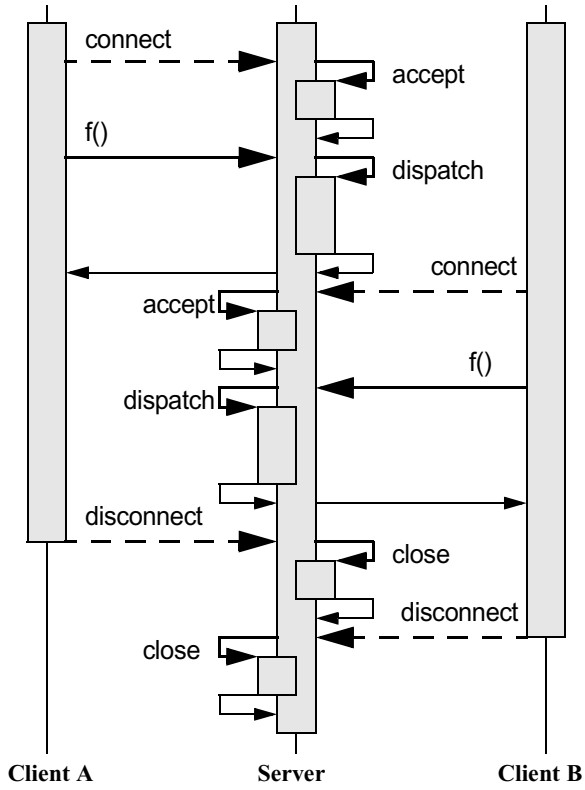


Figure 16.1: Reactive Server

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 application 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 `f` 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 [1].

For blocking clients this scenario is shown in Figure 16.2. As you can see, the callback `g`

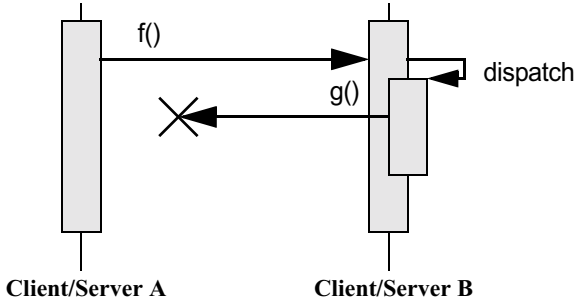


Figure 16.2: Blocking Client

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 is used, A can dispatch incoming requests while waiting for B's reply for `f`. This is shown in Figure 16.3.

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]

The maximum nesting level for the reactive concurrency model is usually much higher than for threaded concurrency models. The reason is that the maximum nesting level for

1. For more information on `send_deferred`, `get_response` and `poll_response`, see the chapter "The Dynamic Invocation Interface" in [4].

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 "The Reactor" on page 215 for more information.

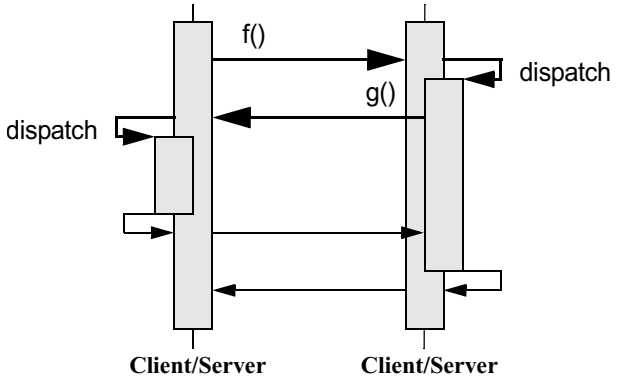


Figure 16.3: Reactive Client/Server

threaded models 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.

16.3 Multi-Threaded Concurrency Models

16.3.1 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. 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 16.4. (For simplicity, the "dispatch" arrows and the corresponding return arrows are omitted in this and all following 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`

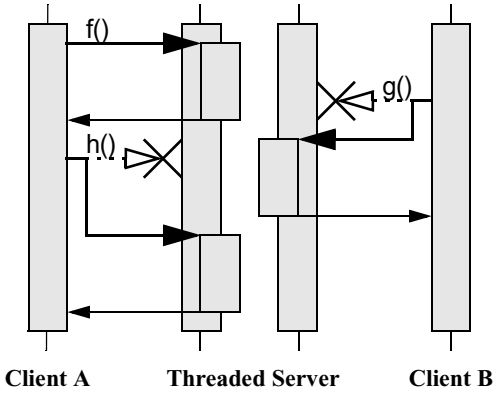


Figure 16.4: Threaded Server

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 reactive concurrency model, at least on a single-processor computer.

16.3.2 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 16.5. A's call to `f` and B's call to `g` are carried out simultaneously, 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.

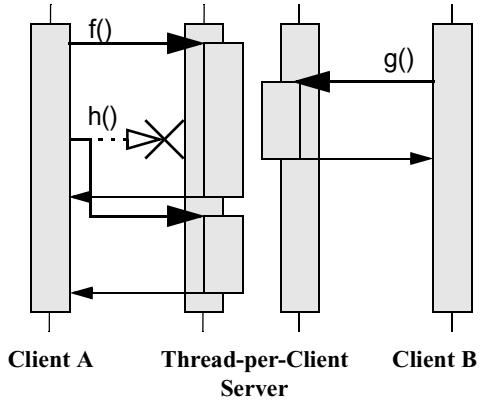


Figure 16.5: Thread-per-Client Server

16.3.3 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 16.6. (For simplicity there are no separate

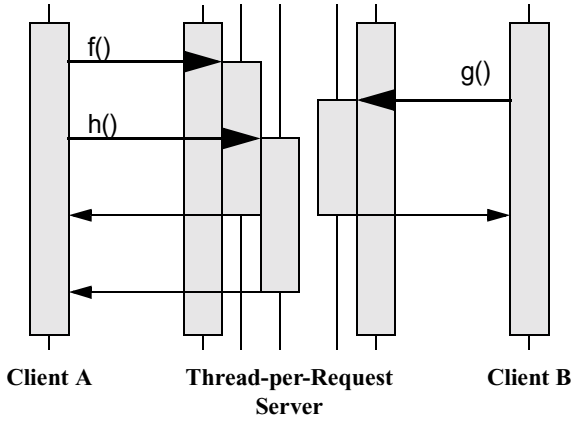


Figure 16.6: Thread-per-Request Server

arrows for dispatch and thread creation in the diagram.) With the thread-per-request

model, requests are never delayed. When they arrive, 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.

16.3.4 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 16.7 shows an

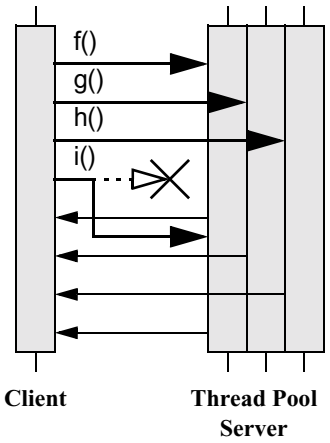


Figure 16.7: Thread Pool Server

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.

16.4 Selecting Concurrency Models

Concurrency models can be selected either by properties or command-line parameters (see Chapter 4). The default concurrency models are shown in Table 16.1.

	Client	Server
Java	Blocking	Threaded
C++	Blocking	Reactive

Table 16.1: Default Concurrency Models

16.5 The Reactor

16.5.1 What is a Reactor?

In "reactive" mode (see "Reactive Clients and Servers" on page 209), ORBACUS uses a so-called "Reactor" for event dispatching [14]. 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 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++.

16.5.2 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 95/98/NT/2000. 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.

The X11 Reactor

An application that wants to use the X11 Reactor can obtain a special X11 Reactor using `OB::GetX11Reactor()`, which it must pass to `OBCORBA::ORB_init()`:

```
1  // C++
2  #include <X11/Intrinsic.h>
3
4  #include <OB/CORBA.h>
5  #include <OB/Logger.h>
6  #include <OB/Properties.h>
7  #include <OB/X11.h>
8
9  int main(int argc, char* argv[])
10 {
11    XtAppContext appContext;
12    Widget topLevel = XtAppInitialize(&appContext, "MyApplication",
13                                      0, 0, &argc, argv, 0, 0, 0);
14
15    OB::Reactor_var reactor = OB::GetX11Reactor(appContext);
16
17    CORBA::ORB_var = OBCORBA::ORB_init(argc, argv,
18       OB::Properties::_nil(), OB::Logger::_nil(), reactor);
19
20    ... // POA initialization not shown
21
22    orb -> run();
23
24    ... // Cleanup not shown
25 }
```

1-7 Include header files.

11-13 Initialize the X11 application.

15 Use the X11 application context to obtain a X11 Reactor.

17 Initialize the ORB using the ORBACUS-specific `OBCORBA::ORB_init()`.

22 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.

The Windows Reactor

Using a Windows Reactor is very similar to using a X11 Reactor:

```
1  // C++
2  #include <Windows.h>
3
4  #include <OB/CORBA.h>
5  #include <OB/Logger.h>
6  #include <OB/Properties.h>
7  #include <OB/OBWindows.h>
8
9  int main(int argc, char* argv[])
10 {
11     HINSTANCE hInstance = GetModuleHandle(0);
12
13     OB::Reactor_var reactor = OB::GetWindowsReactor(hInstance);
14
15     CORBA::ORB_var = OBCORBA::ORB_init(argc, argv,
16        OB::Properties::_nil(), OB::Logger::_nil(), reactor);
17
18     ... // POA initialization not shown
19
20     orb -> run();
21
22     ... // Cleanup not shown
23 }
```

2-7 Include header files.

13 Use the Windows application instance to get a Windows Reactor.

15-16 Initialize the ORB using the ORBACUS-specific `OBCORBA::ORB_init()`.

20 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.

CHAPTER 17

The Open Communications Interface

17.1 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.

17.2 Interface Summary

17.2.1 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.

17.2.2 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.

17.2.3 Acceptor and Connector

Acceptors and Connectors are Factories [2] 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.

17.2.4 Acceptor and Connector Factories

Acceptor and Connector Factories are used by clients to create Acceptors and Connectors. Acceptors are created infrequently, usually only when POA Managers are created. Connectors, however, need to be created by clients whenever a new connection to a server has to be established.

The only component of the OCI that is configurable by applications is the Acceptor. When creating a new Acceptor, an Acceptor Factory takes a sequence of protocol-specific parameters which are used to configure the Acceptor. Each plug-in implementation should document these configuration parameters. The configuration parameters for the plug-ins included with ORBACUS are described later in this chapter.

17.2.5 The Registries

The ORB provides Acceptor and Connector Factory Registries. These registries allow the plugging-in of new protocols. Transport, Connector, Connector Factory, Acceptor 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 Factory must be registered with the ORB's Acceptor Factory Registry.

17.2.6 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`.

17.2.7 Class Diagram

Figure 17.1 shows the classes and interfaces of the OCI (except for the Buffer and Info

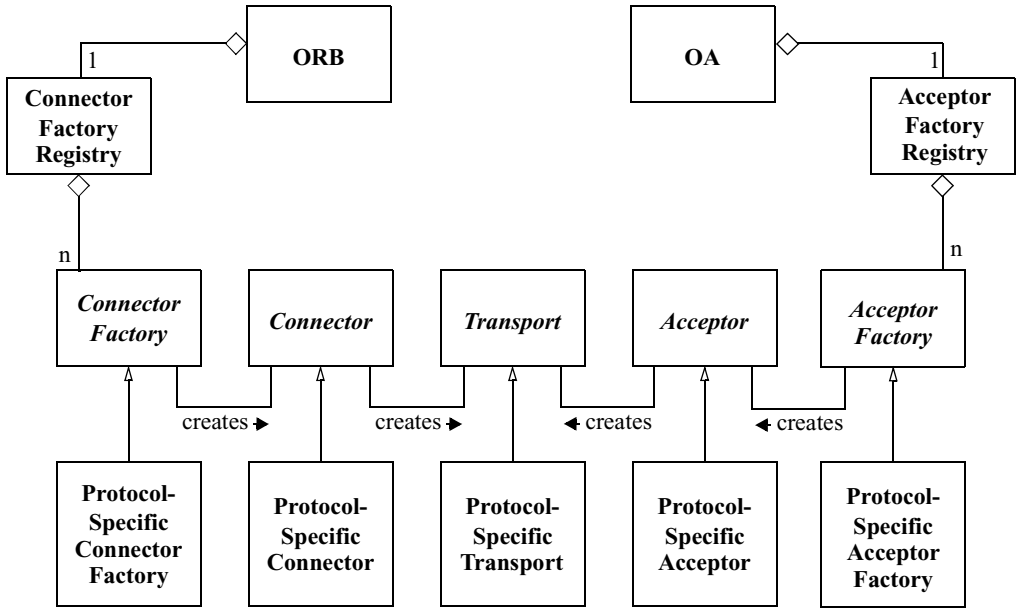


Figure 17.1: OCI Class Diagram

interfaces). ORBACUS provides abstract base classes for the interfaces Connector Factory, Connector, Transport, Acceptor Factory 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 Factory Registry. Instances of Connector Factory Registry and Acceptor Factory Registry can be obtained via the ORB operation `resolve_initial_references`, using the identifiers "OCIConFactoryRegistry" and "OCIAccFactoryRegistry", respectively. Concrete implementations of Connector Factory

must be registered with the Connector Factory Registry, and concrete implementations of Acceptor Factory must be registered with the Acceptor Factory Registry.

17.3 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 E.

17.4 OCI for the Application Programmer

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

17.4.1 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:

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

4-10 Converts `short` port numbers to `int`.

12-22 Converts `byte[]` IP addresses to `int[]`.

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

17.4.2 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  OCI::AcceptorSeq_var acceptors = poaManager -> get_acceptors();
3
4  for(CORBA::ULong i = 0 ; i < acceptors -> length() ; i++)
5  {
6      OCI::AcceptorInfo_var info = acceptors[i] -> get_info();
7      OCI::IIOP::AcceptorInfo_var iiopInfo =
8          OCI::IIOP::AcceptorInfo::_narrow(info);
9
10     if(!CORBA::is_nil(iiopInfo))
11     {
12         CORBA::StringSeq_var hosts = iiopInfo -> hosts();
13         CORBA::UShort port = iiopInfo -> port();
14
15         cout << "host: " << host[0] << endl;
16         cout << "port: " << port << endl;
17     }
18 }
```

2 The list of registered acceptors is requested from the POA Manager.

4 The `for` loop iterates over all acceptors.

6-8 The info object for the acceptor is requested and narrowed to an IIOP acceptor info object.

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

12-16 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  com.ooc.OCI.Acceptor[] acceptors = poaManager.get_acceptors();
3
4  for(int i = 0 ; i < acceptors.length ; i++)
5  {
6     com.ooc.OCI.AcceptorInfo info = acceptors[i].get_info();
7     com.ooc.OCI.IIOP.AcceptorInfo iiopInfo =
8        com.ooc.OCI.IIOP.AcceptorInfoHelper.narrow(info);
9
10    if(iiopInfo != null)
11    {
12       String[] hosts = iiopInfo.hosts();
13       short port = Converter.port(iiopInfo.port());
14
15       System.out.println("host: " + host[0]);
16       System.out.println("port: " + port);
17    }
18 }
```

2-12 This is equivalent to the C++ version.

13 The converter class is used to get a port number in `int` format.

15-16 Like in the C++ version, the hostname and port number are printed on standard output.

17.4.3 Determining a Client's IP Address

To determine 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  CORBA::Object_var baseCurrent =
3     orb -> resolve_initial_references("OCICurrent");
4  OCI::Current_var current = OCI::Current::_narrow(baseCurrent);
5
6  OCI::TransportInfo_var info = current -> get_oci_transport_info();
7  OCI::IIOP::TransportInfo_var iiopInfo =
8     OCI::IIOP::TransportInfo::_narrow(info);
```

```
 9
10 if(!CORBA::is_nil(iiopInfo))
11 {
12     OCI::IIOP::InetAddr remoteAddr = iiopInfo -> remote_addr();
13     CORBA::UShort remotePort = iiopInfo -> remote_port();
14
15     cout << "Call from: "
16          << remoteAddr[0] << '.' << remoteAddr[1] << '.'
17          << remoteAddr[2] << '.' << remoteAddr[3]
18          << ":" << remotePort << endl;
19 }
```

2-4 The OCI current object is requested and `narrow`'d to the correct `OCI::Current` type.

6-8 The info object for the transport is requested and `narrow`'d to an IIOP transport info object.

10 The remainder of the example code is only executed if this was really an IIOP transport info object.

12-18 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 // Java
 2 org.omg.CORBA.Object baseCurrent =
 3     orb.resolve_initial_references("OCICurrent");
 4 com.ooc.OCI.Current current =
 5     com.ooc.OCI.CurrentHelper.narrow(baseCurrent);
 6
 7 com.ooc.OCI.TransportInfo info = current.get_oci_transport_info();
 8 com.ooc.OCI.IIOP.TransportInfo iiopInfo =
 9     com.ooc.OCI.IIOP.TransportInfoHelper.narrow(baseInfo);
10
11 if(iiopInfo != null)
12 {
13     int[] remoteAddr = Converter.addr(iiopInfo.remote_addr());
14     int remotePort = Converter.port(iiopInfo.remote_port());
15
16    System.out.println("Call from: " +
17                       remoteAddr[0] + "." +
18                       remoteAddr[1] + "." +
19                       remoteAddr[2] + "." +
```

```
20                      remoteAddr[3] + ":" + remotePort);
21  }
```

2-11 This code is equivalent to the C++ version.

13-14 Again, the port number must be converted from `short` to `int`.

16-20 This is also equivalent to the C++ version.

17.4.4 Determining a Server's IP Address

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

```
1  // C++
2  CORBA::Object_var obj = ... // Get an object reference somehow
3
4  OCI::ConnectorInfo_var info = obj -> get_oci_connector_info();
5  OCI::IIOP::ConnectorInfo_var iiopInfo =
6     OCI::IIOP::ConnectorInfo::_narrow(info);
7
8  if(!CORBA::is_nil(iiopInfo))
9  {
10    OCI::IIOP::InetAddr_var remoteAddr = iiopInfo -> remoteAddr();
11    CORBA::UShort remotePort = iiopInfo -> remote_port();
12
13    cout << "Will connect to: "
14         << remoteAddr[0] << '.' << remoteAddr[2] << '.'
15         << remoteAddr[2] << '.' << remoteAddr[3]
16         << ":" << remotePort << endl;
17 }
```

4-6 Get the OCI connector info and narrow to an IIOP connector info

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

10-16 The address and port are obtained and displayed on standard output.

The Java version looks as follows:

```
1  // Java
2  org.omg.CORBA.Object obj = ... // Get an object reference somehow
3
4  org.omg.CORBA.portable.ObjectImpl objImpl =
5     (org.omg.CORBA.portable.ObjectImpl)obj;
6  com.ooc.CORBA.Delegate objDelegate =
```

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

4-7 We need to retrieve the ORBACUS-specific `Delegate` object so that we can get the connector info.

9-12 Get the OCI connector info and narrow to an IIOP connector info.

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

16-23 The address and port are obtained and displayed on standard output.

17.5 The IIOP OCI Plug-in

The IIOP plug-in implements the Internet Inter-ORB Protocol as described in [4]. By default, the ORB automatically installs the client and server (i.e., Connector Factory and Acceptor Factory) components of the IIOP plug-in, and IIOP is the default protocol used by the ORB.

17.5.1 IIOP Acceptor Configuration

The configuration parameters for the IIOP Acceptor are described in Table 17.2.

Table 17.2: IIOP Acceptor Configuration Parameters

Name	Type	Description
backlog	unsigned short	The maximum length of the listen backlog queue. Note that the operating system may have a smaller limit which will override this value. If not specified, a default value of 50 is used in Java, and 5 in C++.
bind	string	The hostname or dotted decimal address of the network interface on which to bind the socket. If not specified, the socket will be bound to all available interfaces.
host	CORBA::StringSeq	A sequence of one or more hostnames and/or dotted decimal addresses representing the addresses that should be advertised in IORs. Using IIOP 1.0, multiple addresses are represented as multiple tagged profiles. Using IIOP 1.1 or later, multiple addresses can be represented as either multiple tagged profiles, or as a single tagged profile with a tagged component for each additional address. The `multi_profile` parameter determines how multiple addresses are represented in IIOP 1.1 or later. If this parameter is not specified, the canonical hostname is used.
multi_profile	boolean	If `true`, multiple addresses in the `host` parameter are represented as multiple tagged profiles in an IOR. If `false`, multiple addresses are represented as a single tagged profile (using the first address in the `host` sequence as the primary address), with all additional addresses represented as alternate addresses in tagged components. If IIOP 1.0 is in use, multiple addresses are always represented as multiple tagged profiles, because IIOP 1.0 does not support tagged components. If not specified, the default value is `false`.
numeric	boolean	If `true`, and if `host` is not specified, then the canonical dotted decimal address is advertised in IORs. If not specified, the default value is `false`.
port	unsigned short	Specifies the port on which the acceptor should listen for new connections. If not specified, the default value is 0, which causes the operating system to select an unused port.

Typical configurations include the following:

- No parameters: The Acceptor will use the canonical hostname and a port selected by the operating system.
- Port only: Specifying a port is typically required to create “persistent” object references.
- Host and port: The canonical hostname is incorrect or cannot be determined reliably, so a hostname is specified.
- Multiple hosts and port: The host is known by several names. The client should attempt at least one connection to each address until finding one that succeeds or until all addresses have been exhausted.
- Bind: A host has multiple network interfaces, but the acceptor should only be bound to one of them.

17.6 The Bi-directional OCI Plug-in

The ORBACUS Bi-directional plug-in offers a solution for distributed systems where security restrictions interfere with a client's ability to receive callbacks.

This capability is especially useful in two common situations:

- Firewalls prevent the server from establishing a separate connection back to the client
- Browser restrictions prevent an applet from accepting connections

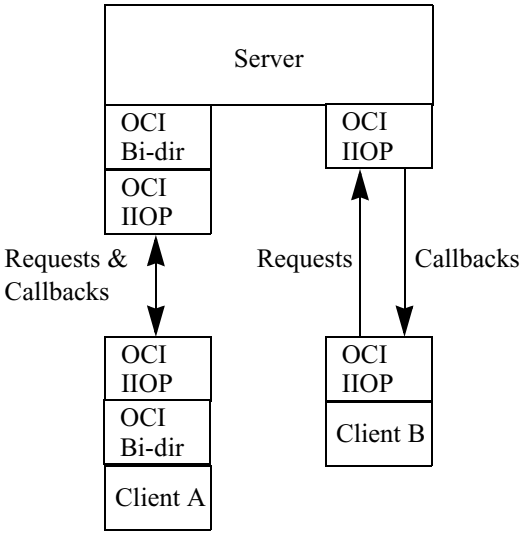
Note: This plug-in does not implement the Bi-directional IIOP standard defined by CORBA 2.3. This plug-in uses a proprietary protocol that is not interoperable with other ORBs.

17.6.1 How does it work?

The Bi-directional plug-in uses a layered design that theoretically enables any connection-oriented OCI plug-in to support bi-directional functionality. Initially however, only bi-directional IIOP is supported.

In Figure 17.1, a server is shown that is capable of receiving both bi-directional IIOP connections and regular IIOP connections.

Figure 17.1: Connection Requirements



Any callback requests from the Server to Client A will travel down the existing connection already established by the client. On the other hand, any callback requests from the Server to Client B require a new IIOP connection to be established from the server to the client.

17.6.2 Peers

The Bi-directional plug-in requires each peer in a bi-directional connection to have a unique identifier, called the “peer ID”. Currently, this identifier is just a simple ISO-LATIN1 string. In IIOP terms, a unique endpoint is derived from the hostname/port combination. However, since the Bi-directional OCI plug-in has no knowledge of the underlying protocol, a separate identification scheme is currently required, and must be provided by the application. It is therefore the application's responsibility to ensure that each server and client has a unique peer ID.

In IIOP, object references can be made “persistent” (i.e., valid across process restarts) by ensuring that the process is restarted on the same host and port, and that the object keys in the object references will continue to be valid. The same is true of peer IDs. If you want a bi-directional IIOP object reference to remain valid across process restarts, you must use

the same peer ID, host, port and object key. Conversely, if an object reference is transient, then the peer ID can vary along with the host, port and object key.

17.6.3 POA Managers

When using the Bi-directional plug-in, a POA Manager must be created specifically for accepting bi-directional connections. If the application only wishes to accept bi-directional connections, then only one POA Manager is required. If the application wishes to accept both bi-directional and regular IIOP connections, then at least two POA Managers are required.

There are really two ways a POA Manager can accept an incoming bi-directional IIOP connection:

- By listening on a port for new TCP/IP connections, just like with regular IIOP
- By listening for new callback connections on existing outgoing connections

A server will typically want to enable both options, but a security-restricted client will only want to enable the second option, since listening on a port is often forbidden (or pointless, if a firewall prevents incoming TCP/IP connections).

The implication of enabling only the second option is that a server wishing to connect to a client will only be able to do so if there is an existing bi-directional connection from the client to the server. If not, the server will receive a TRANSIENT exception.

17.6.4 Initialization and Configuration

The initialization steps can be summarized as follows:

- Initialize the ORB
- Create a POA Manager with one or more Bi-directional acceptors
- Create POAs, activate servants, etc.
- Activate POAManager

The `createPOAManager` method initializes the Bi-directional OCI plug-in for IIOP and returns a new POA Manager:

```
// Java
package com.ooc.BiDir;
public class IIOP
{
   public static org.omg.PortableServer.POAManager
   createPOAManager(
```

```
        String name,
        org.omg.CORBA.ORB orb,
        String[] args,
        java.util.Properties properties);
}

// C++
#include <OB/BiDirIIOPInit.h>

namespace OBBiDir
{
namespace IIOP
{
PortableServer::POAManager_ptr createPOAManager(
    const char* name,
    CORBA::ORB_ptr orb,
    int& argc, char** argv,
    OB::Properties_ptr properties):
};
};
```

The `createPOAManager` method supports the following properties:

`ooc.bidir.peer=id`	Specifies the peer identifier for this ORB. This property is required.
`ooc.bidir.mode=client|server|both`	Specifies the bi-directional mode. If the value is `client`, then only callback connections will be accepted. If the value is `server`, then bi-directional connections will only be accepted on a port. The default value is `both`, in which both kinds of connections are accepted.

The following command-line option is also supported:

`-BIDIRpeer ID`	Equivalent to `ooc.bidir.peer`.

If the command-line option and the `ooc.bidir.peer` property are both defined, the command-line option has precedence.

Because the Bi-directional plug-in can be layered upon the IIOP plug-in, the IIOP configuration properties are still applicable. See Chapter 4 for more information.

Several common scenarios are described in detail below.

Notes:

- In general, the plug-in should be initialized prior to resolving the RootPOA.
- The POA Manager returned by `createPOAManager` must be passed to `create_POA` for all POAs you want associated with that POA Manager.

Scenario: Server accepts only bi-directional IIOP connections

Solution: Configure the Root POA Manager for bi-directional IIOP. As long as no other POA Manager is created, the server will accept only bi-directional connections.

Scenario: Server accepts both regular and bi-directional IIOP connections

Solution: Use the Root POA Manager for regular IIOP connections, and create a new POA Manager for bi-directional connections.

Remember that each POA Manager must be activated in order to dispatch requests to its POAs.

Scenario: Client only accepts connections on existing outgoing bi-directional connections

Solution: Use the Root POA Manager in "client mode", i.e., do not listen on a port. This is accomplished by setting the `ooc.bidir.mode` property to `client`.

For details on how to create POA Managers, see "Creating POA Managers" on page 65. A complete example of using the Bi-directional plug-in is provided in the subdirectory `bidir/demo/hello`.

17.6.5 Bi-directional Acceptor Configuration

The configuration parameters for the Bi-directional Acceptor are described in Table 17.1.

Table 17.1: Bi-directional Acceptor Configuration Parameters

Name	Type	Description
callback	boolean	Specifies whether the factory should create a “callback acceptor”, i.e., one that can only receive callback connections. Note that only one callback acceptor can be created per ORB. If not specified, the default value is `false`.
params	OCI::ParamSeq	Provides a sequence of parameters for use in creating a protocol-specific acceptor. For example, when creating a bi-directional IIOP acceptor, these parameters are used by the Bi-directional plug-in to create an IIOP acceptor. This parameter can only be specified when the `callback` parameter is `false`.

CHAPTER 18

Exceptions and Error Messages

18.1 CORBA System Exceptions

The CORBA specification defines the standard system exceptions shown in Table 18.1. In

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 18.1: Standard CORBA System Exceptions

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
TRANSACTION_REQUIRED	Active transaction context required
TRANSACTION_ROLLEDBACK	Transaction has rolled back or is marked to be rolled back
INVALID_TRANSACTION	Invalid transaction context
INV_POLICY	Invalid Policy
CODESET_INCOMPATIBLE	Incompatible client and server native code sets
REBIND	Thrown on a OBJECT_FORWARD or LOCATION_FORWARD status, depending on the RebindPolicy
TIMEOUT	Time-to-live period was exceeded
TRANSACTION_UNAVAILABLE	Transaction service context could not be processed

Table 18.1: Standard CORBA System Exceptions

TRANSACTION_MODE	Mismatch between TransactionPolicy and current transaction mode
BAD_QOS	Object cannot support the required QOS

Table 18.1: Standard CORBA System Exceptions

the following subsections the minor exception codes are presented. Minor codes that are ORBACUS-specific are presented as *MinorCodeName**, that is, are tagged with the superscript '*'.

18.1.1 INITIALIZE Minor Exception Code

`MinorORBDestroyed`	ORB already destroyed

18.1.2 UNKNOWN Minor Exception Code

`MinorUnknownUserException`	Unknown user exception

18.1.3 BAD_PARAM Minor Exception Code

`MinorValueFactoryError`	Failure to register, unregister or lookup value factory
`MinorRepositoryIdExists`	Repository ID already exists in Interface Repository
`MinorNameExists`	Name already used in Interface Repository
`MinorInvalidContainer`	Target is not a valid container
`MinorNameClashInInheritedContext`	Name clash in inherited context
`MinorBadAbstractInterfaceType`	Incorrect type for abstract interface
`MinorBadSchemeName`	Bad scheme name
`MinorBadAddress`	Bad address
`MinorBadSchemeSpecificPart`	Bad scheme specific part
`MinorOther`	Other
`MinorInvalidAbstractInterfaceInheritance`	Invalid abstract interface inheritance
`MinorInvalidValueInheritance`	Invalid valuetype inheritance
`MinorInvalidServiceContextId`	Invalid service context ID
`MinorObjectIsNull`	Object parameter to `object_to_ior()` is null
`MinorInvalidComponentId`	Invalid component ID

`MinorInvalidProfileId`	Invalid profile ID
`MinorDuplicateDeclarator`*	Duplicate declarator
`MinorInvalidValueModifier`*	Invalid valuetype modifier
`MinorDuplicateValueInit`*	Duplicate valuetype initializer
`MinorAbstractValueInit`*	Abstract valuetype cannot have initializer
`MinorDuplicateBaseType`*	Base type appears more than once
`MinorSingleThreadedOnly`*	ORB does not support multiple threads
`MinorNameRedefinitionInImmediateScope`*	Invalid name redefinition in an immediate scope
`MinorInvalidValueBoxType`*	Invalid type for valuebox

18.1.4 NO_MEMORY Minor Exception Code

`MinorAllocationFailure`*	Memory allocation failure

18.1.5 IMP_LIMIT Minor Exception Code

`MinorMessageSizeLimit`*	Maximum message size exceeded
`MinorThreadLimit`*	Can't create new thread

18.1.6 COMM_FAILURE Minor Exception Code

`MinorRecv`*	`recv()` failed
`MinorSend`*	`send()` failed
`MinorRecvZero`*	`recv()` returned zero
`MinorSendZero`*	`send()` returned zero
`MinorSocket`*	`socket()` failed
`MinorSetsockopt`*	`setsockopt()` failed
`MinorGetsockopt`*	`getsockopt()` failed
`MinorBind`*	`bind()` failed
`MinorListen`*	`listen()` failed
`MinorConnect`*	`connect()` failed
`MinorAccept`*	`accept()` failed
`MinorSelect`*	`select()` failed
`MinorGethostname`*	`gethostname()` failed
`MinorGethostbyname`*	`gethostbyname()` failed
`MinorWSAStartup`*	`WSAStartup()` failed
`MinorWSACleanup`*	`WSACleanup()` failed
`MinorNoGIOP`*	Not a GIOP message

`MinorUnknownMessage`*	Unknown GIOP message
`MinorWrongMessage`*	Wrong GIOP message
`MinorMessageError`*	Got a message error message
`MinorFragment`*	Invalid fragment message
`MinorUnknownReqId`*	Unknown request ID
`MinorVersion`*	Incompatible GIOP version
`MinorPipe`*	Creation of pipe failed

18.1.7 MARSHAL Minor Exception Code

`MinorNoValueFactory`	Unable to locate value factory
`MinorReadOverflow`*	Input stream buffer overflow
`MinorReadBooleanOverflow`*	Overflow while reading boolean
`MinorReadCharOverflow`*	Overflow while reading char
`MinorReadWCharOverflow`*	Overflow while reading wchar
`MinorReadOctetOverflow`*	Overflow while reading octet
`MinorReadShortOverflow`*	Overflow while reading short
`MinorReadUShortOverflow`*	Overflow while reading ushort
`MinorReadLongOverflow`*	Overflow while reading long
`MinorReadULongOverflow`*	Overflow while reading ulong
`MinorReadLongLongOverflow`*	Overflow while reading longlong
`MinorReadULongLongOverflow`*	Overflow while reading ulonglong
`MinorReadFloatOverflow`*	Overflow while reading float
`MinorReadDoubleOverflow`*	Overflow while reading double
`MinorReadLongDoubleOverflow`*	Overflow while reading longdouble
`MinorReadStringOverflow`*	Overflow while reading string
`MinorReadStringZeroLength`*	Encountered zero-length string
`MinorReadStringNullChar`*	Encountered null char in string
`MinorReadStringNoTerminator`*	Terminating null char missing in string
`MinorReadWStringOverflow`*	Overflow while reading wstring
`MinorReadWStringZeroLength`*	Encountered zero-length wstring
`MinorReadWStringNullWChar`*	Encountered null char in wstring
`MinorReadWStringNoTerminator`*	Terminating null char missing in wstring
`MinorReadFixedOverflow`*	Overflow while reading fixed

`MinorReadFixedInvalid`*	Invalid encoding for fixed value
`MinorReadBooleanArrayOverflow`*	Overflow while reading boolean array
`MinorReadCharArrayOverflow`*	Overflow while reading char array
`MinorReadWCharArrayOverflow`*	Overflow while reading wchar array
`MinorReadOctetArrayOverflow`*	Overflow while reading octet array
`MinorReadShortArrayOverflow`*	Overflow while reading short array
`MinorReadUShortArrayOverflow`*	Overflow while reading ushort array
`MinorReadLongArrayOverflow`*	Overflow while reading long array
`MinorReadULongArrayOverflow`*	Overflow while reading ulong array
`MinorReadLongLongArrayOverflow`*	Overflow while reading longlong array
`MinorReadULongLongArrayOverflow`*	Overflow while reading ulonglong array
`MinorReadFloatArrayOverflow`*	Overflow while reading float array
`MinorReadDoubleArrayOverflow`*	Overflow while reading double array
`MinorReadLongDoubleArrayOverflow`*	Overflow while reading longdouble array
`MinorReadInvTypeCodeIndirection`*	Invalid type code indirection
`MinorWriteObjectLocal`*	Attempt to marshal a locality-constrained object
`MinorLongDoubleNotSupported`*	Long double is not supported

18.1.8 NO_IMPLEMENT Minor Exception Code

`MinorMissingLocalValueImplementation`	Missing local value implementation
`MinorIncompatibleValueImplementationVersion`	Incompatible value implementation version

18.1.9 NO_RESOURCES Minor Exception Code

`MinorInvalidBinding`	Portable Interceptor operation not supported in binding

18.1.10 BAD_INV_ORDER Minor Exception Code

`MinorDependencyPreventsDestruction`	Dependency exists in Interface Repository prevents destruction of object
`MinorIndestructibleObject`	Attempt to destroy indestructible object in Interface Repository
`MinorDestroyWouldBlock`	Operation would deadlock
`MinorShutdownCalled`	ORB has shutdown
`MinorInvalidPICall`	Invalid Portable Interceptor call
`MinorServiceContextExists`	A service context already exists with the given ID
`MinorPolicyFactoryExists`	A factory already exists for the given PolicyType
`MinorBadConcModel`*	Invalid concurrency model
`MinorORBRunning`*	`ORB::run()` already called
`MinorRequestAlreadySent`*	Request has already been sent
`MinorRequestNotSent`*	Request has not yet been sent
`MinorResponseAlreadyReceived`*	Response has already been received

18.1.11 TRANSIENT Minor Exception Code

`MinorRequestCancelled`	Request has been cancelled
`MinorConnectFailed`*	Request has been cancelled
`MinorCloseConnection`*	Got a 'close connection' message
`MinorActiveConnectionManagement`*	Active connection management closed connection
`MinorForcedShutdown`*	Forced connection shutdown because of timeout

18.1.12 INTF_REPOS Minor Exception Code

`MinorNoIntfRepos`*	Interface Repository is not available
`MinorLookupAmbiguous`*	Search name for `lookup()` is ambiguous
`MinorIllegalRecursion`*	Illegal Recursion
`MinorNoEntry`*	IFR is not populated with a required definition.

18.1.13 OBJECT_NOT_EXIST Minor Exception Code

`MinorUnregisteredValue`	Attempt to pass unactivated (unregistered) value as an object reference

18.1.14 INV_POLICY Minor Exception Code

`MinorNoPolicyFactory`	No PolicyFactory for the PolicyType has been registered

18.2 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 a deprecated feature that is not implemented anymore

This is not an application error. This error message appears if an application attempts to use a feature that is no longer implemented in ORBACUS. In this case the only thing that can be done is to avoid using this particular feature.

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 greater 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 `operator[]` or `remove` exceeds the sequence length.

Buffer size not equal to sequence bound

This message indicates that the application attempted to call `allocbuf` on a bounded sequence with an argument not equal to the sequence bound.

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.

Null value passed as parameter

This message indicates that an application attempted to pass a null value across an IDL interface.

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.

CORBA::Any::operator<<=(Exception*) cannot be used with --no-type-codes

This message indicates that `CORBA::Any::operator<<=(Exception*)` was invoked for an exception for which no TypeCode is available. That is, the IDL defining the exception was compiled with the `--no-typecodes` option.

An operation on an unembedded recursive TypeCode was invoked

If this message appears, an operation was invoked on a recursive TypeCode that has not yet been embedded.

An already embedded TypeCode was reused

This message indicates that an application attempted to embed a recursive TypeCode that was already embedded.

LongDouble type is not supported on this platform

This message appears when an application uses the `CORBA::LongDouble` type on a platform which does not support this type.

APPENDIX A

Boot Manager Reference

A.1 Interface OB::BootManager

interface **BootManager**

Interface to manage bootstrapping of objects.

Exceptions

NotFound

```
exception NotFound
{
};
```

This exception indicates that a binding has not been found.

AlreadyExists

```
exception AlreadyExists
{
};
```

This exception indicates that a binding already exists.

Operations

add_binding

```
void add_binding(in PortableServer::ObjectId oid,
                 in Object obj)
   raises(AlreadyExists);
```

Add a new binding to the internal table.

Parameters:

`oid` - The object id to bind.
`obj` - The object reference.

Raises:

`AlreadyExists` - Thrown if binding already exists.

remove_binding

```
void remove_binding(in PortableServer::ObjectId oid)
   raises(NotFound);
```

Remove a binding from the internal table.

Parameters:

`oid` - The object id to remove.

Raises:

`NotFound` - Thrown if no binding found.

set_locator

```
void set_locator(in BootLocator locator);
```

Set the BootLocator. The BootLocator is called when a binding for an object id does not exist in the internal table.

Parameters:

`locator` - The BootLocator reference.

See Also:

BootLocator

A.2 Interface OB::BootLocator

interface **BootLocator**

Interface used by BootManager to assist in locating objects.

See Also:
BootManager

Operations

locate

```
void locate(in PortableServer::ObjectId oid,
            out Object obj,
            out boolean add)
   raises(BootManager::NotFound);
```

Locate the object coresponding to the given object id.

Parameters:
`oid` - The object id.
`obj` - The object reference to associate with the id.
`add` - Whether the binding should be added to the internal table.

Raises:
`NotFound` - Raised if no binding found.

APPENDIX B

ORBacus Policy Reference

B.1 Module OB

Constants

ACM_TIMEOUT_POLICY_ID

```
const CORBA::PolicyType ACM_TIMEOUT_POLICY_ID = 1330577418;
```

This policy type identifies the active connection management (ACM) timeout policy.

CONNECTION_REUSE_POLICY_ID

```
const CORBA::PolicyType CONNECTION_REUSE_POLICY_ID = 1330577411;
```

This policy type identifies the connection reuse policy.

CONNECT_TIMEOUT_POLICY_ID

```
const CORBA::PolicyType CONNECT_TIMEOUT_POLICY_ID = 1330577416;
```

This policy type identifies the connect timeout policy.

INTERCEPTOR_POLICY_ID

```
const CORBA::PolicyType INTERCEPTOR_POLICY_ID = 1330577415;
```

This policy type identifies the interceptor policy.

LOCATION_TRANSPARENCY_POLICY_ID
```
const CORBA::PolicyType LOCATION_TRANSPARENCY_POLICY_ID = 1330577414;
```

This policy type identifies the location transparency policy.

LOCATION_TRANSPARENCY_RELAXED
```
const short LOCATION_TRANSPARENCY_RELAXED = 1;
```

The `LOCATION_TRANSPARENCY_RELAXED` LocationTransparencyPolicy value.

LOCATION_TRANSPARENCY_STRICT
```
const short LOCATION_TRANSPARENCY_STRICT = 0;
```

The `LOCATION_TRANSPARENCY_STRICT` LocationTransparencyPolicy value.

PROTOCOL_POLICY_ID
```
const CORBA::PolicyType PROTOCOL_POLICY_ID = 1330577410;
```

This policy type identifies the protocol policy.

REQUEST_TIMEOUT_POLICY_ID
```
const CORBA::PolicyType REQUEST_TIMEOUT_POLICY_ID = 1330577417;
```

This policy type identifies the request timeout policy.

RETRY_ALWAYS
```
const short RETRY_ALWAYS = 2;
```

The `RETRY_ALWAYS` RetryPolicy value.

RETRY_NEVER
```
const short RETRY_NEVER = 0;
```

The `RETRY_NEVER` RetryPolicy value.

RETRY_POLICY_ID
```
const CORBA::PolicyType RETRY_POLICY_ID = 1330577412;
```

This policy type identifies the retry policy.

RETRY_STRICT

```
const short RETRY_STRICT = 1;
```

The `RETRY_STRICT` RetryPolicy value.

TIMEOUT_POLICY_ID

```
const CORBA::PolicyType TIMEOUT_POLICY_ID = 1330577413;
```

This policy type identifies the timeout policy.

B.2 Interface OB::ACMTimeoutPolicy

interface **ACMTimeoutPolicy**
inherits from CORBA::Policy

The active connection management (ACM) timeout policy. This policy can be used to specify a time after which idle connections are shut down by the client.

Attributes

value

```
readonly attribute unsigned long value;
```

If an object has an `ACMTimeoutPolicy` set, the connection associated with this object reference will be shut down after it has been idle for `value` seconds. The default value is the value of the property `ooc.orb.client_timeout`. A value of `0` means no timeout.

B.3 Interface OB::ConnectTimeoutPolicy

interface **ConnectTimeoutPolicy**
inherits from CORBA::Policy

The connect timeout policy. This policy can be used to specify a maximum time limit for connection establishment.

See Also:
TimeoutPolicy

Attributes

value

```
readonly attribute unsigned long value;
```

If an object has a `ConnectTimeoutPolicy` set and a connection cannot be established after `value` milliseconds, a `CORBA::NO_RESPONSE` exception is raised. The default value is `-1`, which means no timeout.

B.4 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 object references will not be permitted to use connections made on behalf of this object. If set to `TRUE`, then connections are shared. The default value is `TRUE`.

B.5 Interface OB::InterceptorPolicy

interface **InterceptorPolicy**
inherits from CORBA::Policy

The interceptor policy. This policy can be used to control whether interceptors are called on method invocations on both the client and the server side.

Attributes

value

```
readonly attribute boolean value;
```

If an object has an `InterceptorPolicy` set and `value` is `FALSE` then any installed interceptors are not called. Otherwise, interceptors are called for each method invocation. The default value is `TRUE`.

B.6 Interface OB::LocationTransparencyPolicy

interface **LocationTransparencyPolicy**
inherits from CORBA::Policy

The location transparency policy. This policy is used to control how strict the ORB is in enforcing location transparency. This is useful for performance reasons.

Attributes

value

```
readonly attribute short value;
```

`LOCATION_TRANSPARENCY_STRICT` ensures strict location transparency is followed. `LOCATION_TRANSPARENCY_RELAXED` relaxes the location transparency guarantees for performance reasons. Specifically for collocated method invocations, the dispatch concurrency model will be ignored, and policy overrides are not removed. The default value is `LOCATION_TRANSPARENCY_RELAXED`.

B.7 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 OCI::ProtocolId value;
```

If a `ProtocolPolicy` is set, then the protocol with the given identifier will be used, if possible. If it is not possible to use this protocol, a `CORBA::NO_RESOURCES` exception will be raised. By default, the ORB chooses the protocol to be used.

B.8 Interface OB::RequestTimeoutPolicy

interface **RequestTimeoutPolicy**
inherits from CORBA::Policy

The request timeout policy. This policy can be used to specify a maximum time limit for requests.

See Also:
TimeoutPolicy

Attributes

value

```
readonly attribute unsigned long value;
```

If an object has a `RequestTimeoutPolicy` set and no response to a request is available after `value` milliseconds, a `CORBA::NO_RESPONSE` exception is raised. The default value is `-1`, which means no timeout.

B.9 Interface OB::RetryPolicy

interface **RetryPolicy**
inherits from CORBA::Policy

The retry policy. This policy is used to specify whether requests are retried after communication failures (i.e., `CORBA::TRANSIENT` and `CORBA::COMM_FAILURE` exceptions).

Attributes

value

```
readonly attribute short value;
```

`RETRY_NEVER` indicates that requests should never be retried, and the exception is re-thrown to the application. `RETRY_STRICT` will retry once if the exception completion status is `COMPLETED_NO`, in order to guarantee at-most-once semantics. `RETRY_ALWAYS` will retry once, regardless of the exception completion status. The default value is `RETRY_STRICT`. **Note:** Many TCP/IP stacks do not provide a reliable indication of communication failure when sending smaller requests, therefore the failure may not be detected until the ORB attempts to read the reply. In this case, the ORB must assume that the remote end has received the request, in order to guarantee at-most-once semantics for the request. The implication is that when using the default setting of `RETRY_STRICT`, most communication failures will not cause a retry. This behavior can be relaxed using `RETRY_ALWAYS`.

B.10 Interface OB::TimeoutPolicy

interface **TimeoutPolicy**
inherits from CORBA::Policy

The timeout policy. This policy can be used to specify the default timeout for connection establishment and requests. If an object also has `ConnectTimeoutPolicy` or `RequestTimeoutPolicy` set, those values have precedence.

See Also:
ConnectTimeoutPolicy, RequestTimeoutPolicy

Attributes

value

```
readonly attribute unsigned long value;
```

If an object has a `TimeoutPolicy` set and a connection cannot be established or no response to a request is available after `value` milliseconds, a `CORBA::NO_RESPONSE` exception is raised. The default value is `-1`, which means no timeout.

APPENDIX C

Reactor Reference

C.1 Interface OB::Reactor

interface **Reactor**

A generic Reactor interface.

Operations

register_handler

```
void register_handler(in EventHandler handler,
                      in Mask handler_mask,
                      in TypeMask type_mask,
                      in Handle h);
```

Register an event handler with the Reactor, or change the registration of an already registered event handler.

Parameters:

`handler` - The event handler to register.
`mask` - The type of events the event handler is interested in.
`type_mask` - The category the event handler belongs to.
`h` - The event handler's handle.

unregister_handler

```
void unregister_handler(in EventHandler handler);
```

Remove an event handler from the Reactor.

Parameters:

`handler` - The event handler to remove.

dispatch

```
boolean dispatch(in TypeMask type_mask);
```

Dispatch events.

Parameters:

`type_mask` - If not zero, this operation will return once all registered event handlers that match the type mask have unregistered.

Returns:

`TRUE` if all event handlers that match the type mask have unregistered, or `FALSE` if event dispatching has been interrupted.

interrupt_dispatch

```
void interrupt_dispatch();
```

Interrupt event dispatching. After calling this operation, `interrupt()` will return with `FALSE`.

dispatch_one_event

```
boolean dispatch_one_event(in long timeout);
```

Dispatch at least one event.

Parameters:

`timeout` - The timeout in milliseconds. A negative value means no timeout, i.e., the operation will not return before at least one event has been dispatched. A zero timeout means that the operation will return immediately if there is no event to dispatch.

Returns:

`TRUE` if at least one event has been dispatched, or `FALSE` otherwise.

event_ready

```
boolean event_ready();
```

Check whether an event is available.

Returns:
TRUE if an event is ready, or FALSE otherwise.

APPENDIX D

Logger Reference

D.1 Interface OB::Logger

interface **Logger**

The ORBacus message logger interface.

Operations

info

```
void info(in string msg);
```

Log an informational message.

Parameters:

`msg` - The message.

error

```
void error(in string msg);
```

Log an error message.

Parameters:

`msg` - The error message.

warning

```
void warning(in string msg);
```

Log a warning message.

Parameters:

`msg` - The warning message.

trace

```
void trace(in string category,
           in string msg);
```

Log a trace message.

Parameters:

`category` - The trace category.

`msg` - The trace message.

APPENDIX E

Open Communications Interface Reference

E.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 CORBA::OctetSeq ObjectKey;
```

Alias for an object key, which is a sequence of octets.

TaggedComponentSeq

```
typedef IOP::TaggedComponentSeq TaggedComponentSeq;
```

Alias for a sequence of tagged components.

Handle

```
typedef long Handle;
```

Alias for a system-specific handle type.

ProtocolId

```
typedef unsigned long ProtocolId;
```

Alias for a protocol id.

ProfileInfoSeq

```
typedef sequence<ProfileInfo> ProfileInfoSeq;
```

Alias for a sequence of basic information about profiles.

ParamSeq

```
typedef sequence<Param> ParamSeq;
```

Alias for a sequence of parameters.

CloseCBSeq

```
typedef sequence<CloseCB> CloseCBSeq;
```

Alias for a sequence of close callback objects.

ConnectorSeq
```
typedef sequence<Connector> ConnectorSeq;
```

Alias for a sequence of Connectors.

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.

AccFactorySeq
```
typedef sequence<AccFactory> AccFactorySeq;
```

Alias for a sequence of AccFactory objects.

ConFactorySeq
```
typedef sequence<ConFactory> ConFactorySeq;
```

Alias for a sequence of Connector factories.

Structs

ProfileInfo
```
struct ProfileInfo
{
   ObjectKey key;
   octet major;
   octet minor;
   ProfileId id;
   unsigned long index;
```

```
    TaggedComponentSeq components;
};
```

Basic information about an IOR profile. Profiles for specific protocols contain additional data. (For example, an IIOP profile also contains a hostname and a port number.)

Members:

`key` - The object key.
`major` - The major version number of the ORB's protocol. (For example, the major GIOP version, if the underlying ORB uses GIOP.)
`minor` - The minor version number of the ORB's protocol. (For example, the minor GIOP version, if the underlying ORB uses GIOP.)
`id` - The id of the profile that contains this information.
`index` - The position index of this profile in an IOR.
`components` - A sequence of tagged components.

Param

```
struct Param
{
    string name;
    any value;
};
```

A parameter represented as a name/value pair.

Members:

`name` - The parameter name.
`value` - The parameter value.

Exceptions

InvalidParam

```
exception InvalidParam
{
    Param p;
    string reason;
};
```

A parameter is invalid. Either the name is unrecognized, the value has the wrong type, or the value is invalid.

Members:

`p` - The offending parameter.

`reason` - The reason why this parameter is invalid.

FactoryAlreadyExists

```
exception FactoryAlreadyExists
{
   ProtocolId id;
};
```

A factory with the given protocol id already exists.

Members:

`id` - The protocol id.

NoSuchFactory

```
exception NoSuchFactory
{
   ProtocolId id;
};
```

No factory with the given protocol id could be found.

Members:

`id` - The protocol id.

E.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.

E.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

id
```
readonly attribute ProtocolId id;
```

The protocol id.

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.

Operations

close
```
void close();
```

Closes the Transport. After calling `close`, no operations on this Transport object and its associated TransportInfo object may be called. To ensure that no messages get lost when `close` is called, `shutdown` should be called first. Then dummy data should be read from the Transport,

using one of the `receive` operations, until either an exception is raised, or until connection closure is detected. After that its save to call `close`, i.e., no messages can get lost.

Raises:
`COMM_FAILURE` - In case of an error.

shutdown
```
void shutdown();
```

Shutdown the Transport. Upon a successful shutdown, threads blocking in the `receive` operations will return or throw an exception. After calling `shutdown`, no operations on associated TransportInfo object may be called. To fully close the Transport, `close` must be called.

Raises:
`COMM_FAILURE` - In case of an error.

receive
```
void receive(in Buffer buf,
             in boolean block);
```

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.

Raises:
`COMM_FAILURE` - In case of an error.

receive_detect
```
boolean receive_detect(in Buffer buf,
                       in boolean block);
```

Similar to `receive`, but it signals a connection loss by returning `FALSE` instead of raising `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.

Raises:
`COMM_FAILURE` - In case of an error.

receive_timeout

```
void receive_timeout(in Buffer buf,
                     in unsigned long 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)`.

Raises:
`COMM_FAILURE` - In case of an error.

send

```
void send(in Buffer buf,
          in boolean block);
```

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.

Raises:
`COMM_FAILURE` - In case of an error.

send_detect

```
boolean send_detect(in Buffer buf,
                    in boolean block);
```

Similar to `send`, but it signals a connection loss by returning `FALSE` instead of raising `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.

Raises:

`COMM_FAILURE` - In case of an error.

send_timeout

```
void send_timeout(in Buffer buf,
                  in unsigned long 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)`.

Raises:

`COMM_FAILURE` - In case of an error.

get_info

```
TransportInfo get_info();
```

Returns the information object associated with the Transport.

Returns:

The Transport information object.

E.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

id
```
readonly attribute ProtocolId id;
```

The protocol id.

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 TransportInfo 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

describe
```
string describe();
```

Returns a human readable description of the transport.

Returns:
The description.

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.

E.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.

E.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

id
```
readonly attribute ProtocolId id;
```

The protocol id.

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.

Raises:
`TRANSIENT` - If the server cannot be contacted.
`COMM_FAILURE` - In case of other errors.

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.

Raises:
`TRANSIENT` - If the server cannot be contacted.
`COMM_FAILURE` - In case of other errors.

get_usable_profiles

```
ProfileInfoSeq get_usable_profiles(in IOR ref,
                                   in CORBA::PolicyList policies);
```

From the given IOR and list of policies, get basic information about all profiles for which this Connector can be used.

Parameters:
`ref` - The IOR from which the profiles are taken.
`policies` - The policies that must be satisfied.

Returns:
The sequence of basic information about profiles. If this sequence is empty, there is no profile in the IOR that matches this Connector and the list of policies.

equal

```
boolean equal(in Connector con);
```

Find out whether this Connector is equal to another Connector. Two Connectors are considered equal if they are interchangeable.

Parameters:
`con` - The connector to compare with.

Returns:
`TRUE` if the Connectors are equal, `FALSE` otherwise.

get_info

```
ConnectorInfo get_info();
```

Returns the information object associated with the Connector.

Returns:

The Connector information object.

E.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

id
```
readonly attribute ProtocolId id;
```

The protocol id.

tag
```
readonly attribute ProfileId tag;
```

The profile id tag.

Operations

describe
```
string describe();
```

Returns a human readable description of the transport.

Returns:
The description.

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.

E.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.

E.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
AccFactory
Transport

Attributes

id
```
readonly attribute ProtocolId id;
```

The protocol id.

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 Acceptor. `accept` or `listen` may not be called after `close` has been called.

Raises:
`COMM_FAILURE` - In case of an error.

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.

Raises:

`COMM_FAILURE` - In case of an error.

accept

```
Transport accept(in boolean block);
```

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.

Parameters:

`block` - If set to `TRUE`, the operation blocks until a new connection has been accepted. If set to `FALSE`, the operation returns a nil object reference if there is no new connection ready to be accepted.

Returns:

The new Transport object.

Raises:

`COMM_FAILURE` - In case of an error.

connect_self

```
Transport connect_self();
```

Connect to this acceptor. This operation can be used to unblock threads that are blocking in `accept`.

Returns:

The new Transport object.

Raises:

`TRANSIENT` - If the server cannot be contacted.
`COMM_FAILURE` - In case of other errors.

add_profiles

```
void add_profiles(in ProfileInfo profile_info,
```

```
                    inout IOR ref);
```

Add new profiles that match this Acceptor to an IOR.

Parameters:
`profile_info` - The basic profile information to use for the new profiles.
`ref` - The IOR.

get_local_profiles
```
ProfileInfoSeq get_local_profiles(in IOR ref);
```

From the given IOR, get basic information about all profiles for which are local to this Acceptor.

Parameters:
`ref` - The IOR from which the profiles are taken.

Returns:
The sequence of basic information about profiles. If this sequence is empty, there is no profile in the IOR that is local to the Acceptor.

get_info
```
AcceptorInfo get_info();
```

Returns the information object associated with the Acceptor.

Returns:
The Acceptor information object.

E.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

id
```
readonly attribute ProtocolId id;
```

The protocol id.

tag
```
readonly attribute ProfileId tag;
```

The profile id tag.

Operations

describe
```
string describe();
```

Returns a human readable description of the transport.

Returns:
The description.

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.

E.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.

E.12 Interface OCI::AccFactory

interface **AccFactory**

An interface for an AccFactory object, which is used by CORBA servers to create Acceptors.

See Also:
Acceptor
AccFactoryRegistry

Attributes

id
```
readonly attribute ProtocolId id;
```

The protocol id.

tag
```
readonly attribute ProfileId tag;
```

The profile id tag.

Operations

create_acceptor
```
Acceptor create_acceptor(in ParamSeq params)
   raises(InvalidParam);
```

Create an Acceptor using the given configuration parameters. Refer to the plug-in documentation for a description of the configuration parameters supported for a particular protocol.

Parameters:
`params` - The configuration parameters.

Returns:
The new Acceptor.

Raises:
`InvalidParam` - If any of the parameters are invalid.

get_info

```
AccFactoryInfo get_info();
```

Returns the information object associated with the Acceptor factory.

Returns:

The Acceptor

E.13 Interface OCI::AccFactoryInfo

interface **AccFactoryInfo**

Information on an OCI AccFactory object.

See Also:
AccFactory

Attributes

id
```
readonly attribute ProtocolId id;
```

The protocol id.

tag
```
readonly attribute ProfileId tag;
```

The profile id tag.

Operations

describe
```
string describe();
```

Returns a human readable description of the transport.

Returns:
The description.

E.14 Interface OCI::AccFactoryRegistry

interface **AccFactoryRegistry**

A registry for Acceptor factories.

See Also:
Acceptor
AccFactory

Operations

add_factory

```
void add_factory(in AccFactory factory)
   raises(FactoryAlreadyExists);
```

Adds an Acceptor factory to the registry.

Parameters:
`factory` - The Acceptor factory to add.

Raises:
`FactoryAlreadyExists` - If a factory already exists with the same protocol id as the given factory.

get_factory

```
AccFactory get_factory(in ProtocolId id)
   raises(NoSuchFactory);
```

Returns the factory with the given protocol id.

Parameters:
`id` - The protocol id.

Returns:
The Acceptor factory.

Raises:
`NoSuchFactory` - If no factory was found with a matching protocol id.

get_factories

```
AccFactorySeq get_factories();
```

Returns all registered factories.

Returns:
The Acceptor factories.

E.15 Interface OCI::ConFactory

interface **ConFactory**

A factory for Connector objects.

See Also:
Connector
ConFactoryRegistry

Attributes

id
```
readonly attribute ProtocolId id;
```

The protocol id.

tag
```
readonly attribute ProfileId tag;
```

The profile id tag.

Operations

create_connectors
```
ConnectorSeq create_connectors(in IOR ref,
                               in CORBA::PolicyList policies);
```

Returns a sequence of Connectors for a given IOR and a list of policies. The sequence includes one or more Connectors for each IOR profile that matches this Connector factory and satisfies the list of policies.

Parameters:
`ref` - The IOR for which Connectors are returned.
`policies` - The policies that must be satisfied.

Returns:
The sequence of Connectors.

equivalent
```
boolean equivalent(in IOR ior1,
```

```
                        in IOR ior2);
```

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

```
unsigned long hash(in IOR ref,
                   in unsigned long maximum);
```

Calculates a hash value for an IOR.

Parameters:

`ref` - 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 Connector factory information object.

E.16 Interface OCI::ConFactoryInfo

interface **ConFactoryInfo**

Information on an OCI ConFactory object.

See Also:
ConFactory

Attributes

id
```
readonly attribute ProtocolId id;
```

The protocol id.

tag
```
readonly attribute ProfileId tag;
```

The profile id tag.

Operations

describe
```
string describe();
```

Returns a human readable description of the transport.

Returns:
The description.

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.

E.17 Interface OCI::ConFactoryRegistry

interface **ConFactoryRegistry**

A registry for Connector factories.

See Also:
Connector
ConFactory

Operations

add_factory

```
void add_factory(in ConFactory factory)
   raises(FactoryAlreadyExists);
```

Adds a Connector factory to the registry.

Parameters:
`factory` - The Connector factory to add.

Raises:
`FactoryAlreadyExists` - If a factory already exists with the same protocol id as the given factory.

get_factory

```
ConFactory get_factory(in ProtocolId id)
   raises(NoSuchFactory);
```

Returns the factory with the given protocol id.

Parameters:
`id` - The protocol id.

Returns:
The Connector factory.

Raises:
`NoSuchFactory` - If no factory was found with a matching protocol id.

get_factories

```
ConFactorySeq get_factories();
```

Returns all registered factories.

Returns:
The Connector factories.

E.18 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.

E.19 Module OCI::IIOP

This module contains interfaces to support 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.

Constants

TAG_IIOP

```
const ProtocolId TAG_IIOP = 1330577409;
```

The protocol id for the ORBACUS IIOP plug-in.

E.20 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.

E.21 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.

E.22 Interface OCI::IIOP::AcceptorInfo

interface **AcceptorInfo**
inherits from OCI::AcceptorInfo

Information on an IIOP OCI Acceptor object.

See Also:
Acceptor
AcceptorInfo

Attributes

hosts
```
readonly attribute CORBA::StringSeq hosts;
```

Hostnames 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.

E.23 Interface OCI::IIOP::AccFactoryInfo

interface **AccFactoryInfo**
inherits from OCI::AccFactoryInfo

Information on an IIOP OCI Acceptor Factory object.

See Also:
AccFactory

E.24 Interface OCI::IIOP::ConFactoryInfo

interface **ConFactoryInfo**
inherits from OCI::ConFactoryInfo

Information on an IIOP OCI Connector Factory object.

See Also:
ConFactory
ConFactoryInfo

References

[1] Buschman, F., et al. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. New York: Wiley.

[2] Gamma, E., et al. 1994. *Design Patterns*. Reading, MA: Addison-Wesley

[3] Henning, M., and S. Vinoski. 1999. *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.

[4] Object Management Group. 1999. *The Common Object Request Broker: Architecture and Specification*. Revision 2.3.1. ftp://www.omg.org/pub/docs/formal/99-10-07.pdf. Framingham, MA: Object Management Group.

[5] Object Management Group. 1999. *C++ Language Mapping*. ftp://www.omg.org/pub/docs/formal/99-07-45.pdf. Framingham, MA: Object Management Group.

[6] Object Management Group. 1999. *IDL/Java Language Mapping*. ftp://www.omg.org/pub/docs/formal/99-07-53.pdf. Framingham, MA: Object Management Group.

[7] Object Management Group. 1999. *Portable Interceptors*. ftp://ftp.omg.org/pub/docs/orbos/99-12-02.pdf. Framingham, MA: Object Management Group.

[8] Object Management Group. 1998. *CORBA Messaging*. ftp://ftp.omg.org/pub/docs/orbos/98-05-06.pdf. Framingham, MA: Object Management Group.

[9] Object Management Group. 1998. *CORBAservices: Common Object Services Specification*. ftp://www.omg.org/pub/docs/formal/98-12-09.pdf. Framingham, MA: Object Management Group.

[10] Object Management Group. 1999. *Naming Service Specification*. ftp://ftp.omg.org/pub/docs/ptc/99-12-03.pdf. Framingham, MA: Object Management Group.

[11] IONA Technologies, Inc. 2001. *JTHREADS/C++*. http://www.orbacus.com/jtc/. Waltham, MA: IONA Technologies, Inc.

[12] IONA Technologies, Inc. 2001. *JTHREADS/C++ User's Manual*. Waltham, MA: IONA Technologies, Inc.

[13] IONA Technologies, Inc. 2001. *ORBACUS*. http://www.orbacus.com/ob/. Waltham, MA: IONA Technologies, Inc.

[14] Schmidt, D. C. 1995. "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching." In *Pattern Languages of Program Design*, ed. James O. Coplien and Douglas C. Schmidt. Reading, MA: Addison-Wesley.

P