The omniORB version 4.3 Users' Guide

Duncan Grisby
(*dgrisby@apasphere.com*)

Contents

1 Introduction **1**
- 1.1 Features . . . 1
  - 1.1.1 Multithreading . . . 2
  - 1.1.2 Portability . . . 2
  - 1.1.3 Missing features . . . 3
- 1.2 Setting up your environment . . . 3
- 1.3 Platform specific variables . . . 4

2 The Basics **6**
- 2.1 The Echo Object Example . . . 6
- 2.2 Specifying the Echo interface in IDL . . . 6
- 2.3 Generating the C++ stubs . . . 7
- 2.4 Object References and Servants . . . 8
- 2.5 A quick look at the C++ mapping . . . 8
  - 2.5.1 Mapping overview . . . 8
  - 2.5.2 Interface scope type . . . 8
  - 2.5.3 Object reference pointer type . . . 9
    - 2.5.3.1 Nil object reference . . . 9
    - 2.5.3.2 Object reference lifecycle . . . 10
    - 2.5.3.3 Object reference inheritance . . . 11
    - 2.5.3.4 Object reference equivalence . . . 11
  - 2.5.4 Servant Object Implementation . . . 12
- 2.6 Writing the servant implementation . . . 12
- 2.7 Writing the client . . . 14
- 2.8 Example 1 — Colocated Client and Servant . . . 15
  - 2.8.1 ORB initialisation . . . 16
  - 2.8.2 Obtaining the Root POA . . . 16
  - 2.8.3 Object initialisation . . . 17
  - 2.8.4 Activating the POA . . . 18
  - 2.8.5 Performing a call . . . 18
  - 2.8.6 ORB destruction . . . 18
- 2.9 Example 2 — Different Address Spaces . . . 18

2.9.1 Making a Stringified Object Reference . . . 18
2.9.2 Client: Using a Stringified Object Reference . . . 19
2.9.3 Catching System Exceptions . . . 20
2.9.4 Lifetime of a CORBA object . . . 20
2.10 Example 3 — Using the Naming Service . . . 21
2.10.1 Obtaining the Root Context Object Reference . . . 21
2.10.2 The Naming Service Interface . . . 22
2.11 Example 4 — Using tie implementation templates . . . 22
2.12 Source Listings . . . 23
2.12.1 eg1.cc . . . 23
2.12.2 eg2_impl.cc . . . 26
2.12.3 eg2_clt.cc . . . 28
2.12.4 eg3_impl.cc . . . 30
2.12.5 eg3_clt.cc . . . 34
2.12.6 eg3_tieimpl.cc . . . 36

3 C++ language mapping 40
3.1 omniORB 2 BOA compatibility . . . 40
3.2 omniORB 3.0 compatibility . . . 42
3.3 omniORB 4.0 compatibility . . . 42
3.4 omniORB 4.1 compatibility . . . 43
3.5 omniORB 4.2 compatibility . . . 43
3.6 Interoperability . . . 43
3.6.1 Exceptions in Anys . . . 43

4 omniORB configuration and API 45
4.1 Setting parameters . . . 45
4.1.1 Command line arguments . . . 45
4.1.2 ORB_init() parameter . . . 46
4.1.3 Environment variables . . . 46
4.1.4 Configuration file . . . 46
4.1.5 Windows registry . . . 46
4.2 Tracing options . . . 47
4.2.1 Tracing API . . . 48
4.3 Miscellaneous global options . . . 48
4.4 Client side options . . . 50
4.5 Server side options . . . 54
4.5.1 Main thread selection . . . 57
4.6 GIOP and interoperability options . . . 58
4.7 System Exception Handlers . . . 60
4.7.1 Minor codes . . . 60
4.7.2 CORBA::TRANSIENT handlers . . . 61

4.7.3 CORBA::TIMEOUT . . . 63
4.7.4 CORBA::COMM_FAILURE . . . 64
4.7.5 CORBA::SystemException . . . 64
4.7.6 Extended exception handlers . . . 65
4.8 Location forwarding . . . 65

5 The IDL compiler **67**
5.1 Common options . . . 67
5.1.1 Preprocessor interactions . . . 68
5.1.2 Forward-declared interfaces . . . 68
5.1.3 Comments . . . 68
5.2 C++ back-end options . . . 69
5.2.1 Optional code generation options . . . 69
5.2.1.1 Any and TypeCode . . . 69
5.2.1.2 Tie templates . . . 70
5.2.1.3 Asynchronous Method Invocation . . . 70
5.2.1.4 Example implementations . . . 70
5.2.2 Include file options . . . 70
5.2.3 Object reference operations . . . 71
5.2.4 Module splicing . . . 71
5.3 Examples . . . 72

6 Connection and Thread Management **73**
6.1 Background . . . 73
6.2 The model . . . 74
6.3 Client side behaviour . . . 74
6.3.1 Client side timeouts . . . 75
6.4 Server side behaviour . . . 77
6.4.1 Thread per connection mode . . . 77
6.4.2 Thread pool mode . . . 77
6.4.3 Policy transition . . . 78
6.5 Idle connection shutdown . . . 79
6.5.1 Interoperability Considerations . . . 79

7 Transports and Endpoints **81**
7.1 Port ranges . . . 82
7.2 IPv6 . . . 82
7.2.1 Link local addresses . . . 83
7.3 Endpoint publishing . . . 83
7.3.1 POA-specific endpoint publishing . . . 85
7.4 Connection selection and acceptance . . . 85
7.5 Client transport rules . . . 85

7.6 Server transport rules . . . 86
7.7 Bidirectional GIOP . . . 87
7.8 TLS / SSL transport . . . 88
7.8.1 Self-signed certificate authority . . . 88
7.9 HTTP, HTTPS and WebSocket transport . . . 89
7.9.1 HTTP limitations . . . 89
7.9.2 Endpoint and transport rule specification . . . 90
7.9.3 Client connections . . . 90
7.9.4 HTTP proxies . . . 91
7.9.5 HTTPS keys and certificates . . . 91
7.9.6 In-message encryption . . . 91
7.10 ZIOP . . . 92
7.10.1 Forcing ZIOP Policies . . . 93
7.11 Connection Management Extension . . . 93
7.11.1 Client-side parameters . . . 94
7.11.2 Server-side parameters . . . 95
7.11.3 Usage . . . 96

8 Interoperable Naming Service 97
8.1 Object URIs . . . 97
8.1.1 corbaloc . . . 97
8.1.2 Other transports . . . 98
8.1.3 HTTP URLs . . . 99
8.1.4 Resolve initial references . . . 99
8.1.5 corbaname . . . 99
8.2 Configuring resolve_initial_references . . . 100
8.2.1 ORBInitRef . . . 100
8.2.2 ORBDefaultInitRef . . . 100
8.3 omniNames . . . 100
8.3.1 NamingContextExt . . . 100
8.3.2 Use with corbaname . . . 102
8.4 omniMapper . . . 102
8.5 Creating objects with plain object keys . . . 102

9 Code set conversion 104
9.1 Native code sets . . . 104
9.2 Default code sets . . . 105
9.3 Code set library . . . 105
9.4 Implementing new code sets . . . 106

10 Interceptors 107
10.1 Interceptor registration . . . 108

10.2 Available interceptors 108
10.3 Server-side call interceptor 111

11 Type Any and TypeCode **112**
11.1 Example using type Any 112
11.1.1 Type Any in IDL 112
11.1.2 Inserting and Extracting Basic Types from an Any 113
11.1.3 Inserting and Extracting Constructed Types from an Any . . 114
11.2 Type Any in omniORB 115
11.2.1 Generating Insertion and Extraction Operators. 116
11.2.2 TypeCode comparison when extracting from an Any. . . . 116
11.2.3 Top-level aliases. 116
11.2.4 Removing aliases from TypeCodes. 117
11.2.5 Recursive TypeCodes. 117
11.2.6 Threads and type Any. 117
11.3 TypeCode in omniORB 118
11.3.1 TypeCodes in IDL. 118
11.3.2 orb.idl 118
11.3.3 Generating TypeCodes for constructed types. 118

12 Objects by value, etc. **119**
12.1 Features 119
12.2 Reference counting 119
12.3 Value sharing and local calls 120
12.4 Value box factories 120
12.5 Standard value boxes 121
12.6 Covariant returns 121
12.7 Values inside Anys 121
12.7.1 Values inside DynAnys 122
12.8 Local Interfaces 123
12.8.1 Simple local interfaces 123
12.8.2 Inheritance from unconstrained interfaces 123
12.8.3 Valuetypes supporting local interfaces 124

13 Asynchronous Method Invocation **126**
13.1 Implied IDL 126
13.2 Generating AMI stubs 127
13.3 AMI examples 127

14 Interface Type Checking **128**
14.1 Introduction 128
14.2 Interface Inheritance 129

15 Packaging stubs into DLLs **132**
15.1 Dynamic loading and unloading 132
15.2 Windows DLLs 132
15.2.1 Exporting symbols 132
15.2.2 Importing constant symbols 133

16 Resources **136**

Chapter 1

Introduction

omniORB is an Object Request Broker (ORB) that implements version 2.6 of the Common Object Request Broker Architecture (CORBA) [OMG01] specification. Where possible, backward compatibility has been maintained back to specification 2.0. It passed the Open Group CORBA compliant testsuite (for CORBA 2.1) and was one of the three ORBs to be granted the CORBA brand in June 1999.

This user guide tells you how to use omniORB to develop CORBA applications. It assumes a basic understanding of CORBA.

In this chapter, we give an overview of the main features of omniORB and what you need to do to set up your environment to run omniORB.

1.1 Features

omniORB is quite feature-rich, but it does not slavishly implement every last part of the CORBA specification. The goal is to provide the most generally useful parts of the specification in a clean and efficient manner. Highlights are:

- C++ and Python language bindings.
- Support for the complete Portable Object Adapter (POA) specification.
- Support for the Interoperable Naming Service (INS).
- Internet Inter-ORB Protocol (IIOP 1.2) is used as the native protocol.
- The omniORB runtime is fully multithreaded. It uses platform thread support encapsulated with a small class library, omnithread, to abstract away from differences in native thread APIs.
- TypeCode and type Any are supported.
- DynAny is supported.

- The Dynamic Invocation and Dynamic Skeleton interfaces are supported.
- Valuetype and abstract interfaces are supported.
- Asynchronous Method Invocation (AMI) supported, including both polling and callback models.
- Extensive control over connection management.
- Soft real-time features including call deadlines and timeouts.
- A COS Naming Service, omniNames.
- Many platforms are supported, including most Unix platforms and Windows.
- It has been successfully tested for interoperability via IIOP with other ORBs.

1.1.1 Multithreading

omniORB is fully multithreaded. To achieve low call overhead, unnecessary call multiplexing is eliminated. With the default policies, there is at most one call in-flight in each communication channel between two address spaces at any one time. To do this without limiting the level of concurrency, new channels connecting the two address spaces are created on demand and cached when there are concurrent calls in progress. Each channel is served by a dedicated thread. This arrangement provides maximal concurrency and eliminates any thread switching in either of the address spaces to process a call. Furthermore, to maximise the throughput in processing large call arguments, large data elements are sent as soon as they are processed while the other arguments are being marshalled. With GIOP 1.2, large messages are fragmented, so the marshaller can start transmission before it knows how large the entire message will be.

omniORB also supports a flexible thread pool policy, and supports sending multiple interleaved calls on a single connection. This policy leads to a small amount of additional call overhead, compared to the default thread per connection model, but allows omniORB to scale to extremely large numbers of concurrent clients.

1.1.2 Portability

omniORB runs on many flavours of Unix, Windows, several embedded operating systems, and relatively obscure systems such as OpenVMS and Fujitsu-Siemens

BS2000. It is designed to be easy to port to new platforms. The IDL to C++ mapping for all target platforms is the same.

omniORB uses real C++ exceptions and nested classes. It keeps to the CORBA specification's standard mapping as much as possible and does not use the alternative mappings for C++ dialects. The only small exception is the mapping of IDL modules, which can use either namespaces according to the standard, or nested classes for truly ancient C++ compilers without namespace support.

omniORB relies on native thread libraries to provide multithreading capability. A small class library (omnithread [Ric96]) is used to encapsulate the APIs of the native thread libraries. In application code, it is recommended but not mandatory to use this class library for thread management. It should be easy to port omnithread to any platform that either supports the POSIX thread standard or has a thread package that supports similar capabilities.

Partly for historical reasons, and partly to support users with archaic compilers, omniORB does not use the C++ standard library.

The omniORB IDL compiler, omniidl, requires Python version 3.5 or later or, for people stuck in the past, version 2.7.

1.1.3 Missing features

omniORB is not a complete implementation of the CORBA 2.6 core. The following is a list of the most significant missing features.

- For some very dynamic uses of CORBA, you may need an Interface Repository. omniORB does not have its own one, but it can act as a client to an IfR. The omniifr project (`https://github.com/omniorb/omniifr`) aims to create an IfR for omniORB.
- omniORB supports interceptors, but not the standard Portable Interceptor API.

1.2 Setting up your environment

To get omniORB running, you first need to install omniORB according to the instructions in the installation notes for your platform. See `README.FIRST.txt` at the top of the omniORB tree for instructions. Most Unix platforms can use the Autoconf `configure` script to automate the configuration process.

Once omniORB is installed in a suitable location, you must configure it according to your required setup. The configuration can be set with a configuration file, environment variables, command-line arguments or, on Windows, the Windows registry.

- On Unix platforms, the omniORB runtime looks for the environment variable `OMNIORB_CONFIG`. If this variable is defined, it contains the pathname of the omniORB configuration file. If the variable is not set, omniORB will use the compiled-in pathname to locate the file (by default `/etc/omniORB.cfg`).
- On Win32 / Win64 platforms, omniORB first checks the environment variable `OMNIORB_CONFIG` to obtain the pathname of the configuration file. If this is not set, it then attempts to obtain configuration data in the system registry. It searches for the data under the key `HKEY_LOCAL_MACHINE\SOFTWARE\omniORB`.

omniORB has a large number of parameters than can be configured. See chapter 4 for full details. The files `sample.cfg` and `sample.reg` contain an example configuration file and set of registry entries respectively.

To get all the omniORB examples running, the main thing you need to configure is the Naming service, omniNames. To do that, the configuration file or registry should contain an entry of the form

```
InitRef = NameService=corbaname::my.host.name
```

See section 8.1.5 for full details of corbaname URIs.

1.3 Platform specific variables

To compile omniORB programs correctly, several C++ preprocessor defines *must* be specified to identify the target platform. On Unix platforms where omniORB was configured with Autoconf, the `omniconfig.h` file sets these for you. On other platforms, and Unix platforms when Autoconf is not used, you must specify the following defines:

Platform	CPP defines			
Windows	`__x86__`	`__NT__`	`__OSVERSION__=4`	`__WIN32__`
Windows NT 3.5	`__x86__`	`__NT__`	`__OSVERSION__=3`	`__WIN32__`
Sun Solaris 2.5	`__sparc__`	`__sunos__`	`__OSVERSION__=5`	
HPUX 10.x	`__hppa__`	`__hpux__`	`__OSVERSION__=10`	
HPUX 11.x	`__hppa__`	`__hpux__`	`__OSVERSION__=11`	
IBM AIX 4.x	`__aix__`	`__powerpc__`	`__OSVERSION__=4`	
Digital Unix 3.2	`__alpha__`	`__osf1__`	`__OSVERSION__=3`	
Linux 2.x (x86)	`__x86__`	`__linux__`	`__OSVERSION__=2`	
Linux 2.x (powerpc)	`__powerpc__`	`__linux__`	`__OSVERSION__=2`	
OpenVMS 6.x (alpha)	`__alpha__`	`__vms`	`__OSVERSION__=6`	
OpenVMS 6.x (vax)	`__vax__`	`__vms`	`__OSVERSION__=6`	
SGI Irix 6.x	`__mips__`	`__irix__`	`__OSVERSION__=6`	
Reliant Unix 5.43	`__mips__`	`__SINIX__`	`__OSVERSION__=5`	
ATMos 4.0	`__arm__`	`__atmos__`	`__OSVERSION__=4`	
NextStep 3.x	`__m68k__`	`__nextstep__`	`__OSVERSION__=3`	
Unixware 7	`__x86__`	`__uw7__`	`__OSVERSION__=5`	

The preprocessor defines for new platform ports not listed above can be found in the corresponding platform configuration files. The preprocessor defines to identify a platform are in the make variable `IMPORT_CPPFLAGS`.

In a single source multi-target environment, you can put the preprocessor defines as the command-line arguments for the compiler. If you are building for a single platform, you can edit include/omniconfig.h to add the definitions.

Chapter 2

The Basics

In this chapter, we go through three examples to illustrate the practical steps to use omniORB. By going through the source code of each example, the essential concepts and APIs are introduced. If you have no previous experience with using CORBA, you should study this chapter in detail. There are pointers to other essential documents you should be familiar with.

If you have experience with using other ORBs, you should still go through this chapter because it provides important information about the features and APIs that are necessarily omniORB specific. With the Portable Object Adapter, there are very few omniORB specific details.

2.1 The Echo Object Example

Our example is an object which has only one method. The method simply echos the argument string. We have to:

1. define the object interface in IDL
2. use the IDL compiler to generate the stub code, which provides the object mapping as defined in the CORBA specification
3. provide the *servant* object implementation
4. write the client code.

These examples are in the `src/examples/echo` directory of the omniORB distribution; there are several other examples in `src/examples`.

2.2 Specifying the Echo interface in IDL

We define an object interface, called `Echo`, as follows:

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

If you are new to IDL, you can learn about its syntax in Chapter 3 of the CORBA 2.6 specification [OMG01]. For the moment, you only need to know that the interface consists of a single operation, `echoString()`, which takes a string as an input argument and returns a copy of the same string.

The interface is written in a file, called `echo.idl`. It is part of the CORBA standard that all IDL files must have the extension '`.idl`', although omniORB does not enforce this. In the omniORB distribution, this file is in `idl/echo.idl`.

For simplicity, the interface is defined in the global IDL namespace. You should normally avoid this practice for the sake of object reusability. If every CORBA developer defines their interfaces in the global IDL namespace, there is a danger of name clashes between two independently defined interfaces. Therefore, it is better to qualify your interfaces by defining them inside `module` names. Of course, this does not eliminate the chance of a name clash unless some form of naming convention is agreed globally. Nevertheless, a well-chosen module name can help a lot.

2.3 Generating the C++ stubs

From the IDL file, we use the IDL compiler to produce the C++ mapping of the interface. The IDL compiler for omniORB is called omniidl. Given the IDL file, omniidl produces two stub files: a C++ header file and a C++ source file. For example, from the file `echo.idl`, the following files are produced:

- `echo.hh`
- `echoSK.cc`

omniidl must be invoked with the `-bcxx` argument to tell it to generate C++ stubs. The following command line generates the stubs for `echo.idl`:

```
omniidl -bcxx echo.idl
```

Note that the names `echo.hh` and `echoSK.cc` are not defined in the C++ mapping standard. Other CORBA implementations may use different file names. To aid migration omniidl from other implementations, omniidl has options to override the default output file names. See section 5.2 for details.

If you are using our make environment, you don't need to invoke omniidl explicitly. In the example file `dir.mk`, we have the following line:

```
CORBA_INTERFACES = echo
```

That is all we need to instruct the build system to generate the stubs. You won't find the stubs in your working directory because all stubs are written into the `stub` directory at the top level of your build tree.

The full arguments to omniidl are detailed in chapter 5.

2.4 Object References and Servants

We contact a CORBA object through an *object reference*. The actual implementation of a CORBA object is termed a *servant*.

Object references and servants are quite separate entities, and it is important not to confuse the two. Client code deals purely with object references, so there can be no confusion; object implementation code must deal with both object references and servants. omniORB uses distinct C++ types for object references and servants, so the C++ compiler will complain if you use a servant when an object reference is expected, or vice-versa.

2.5 A quick look at the C++ mapping

The C++ stubs conform to the standard mapping defined in the CORBA specification [OMG03]. Sadly, since it pre-dates the C++ standard library, the C++ language mapping is quite hard to use, especially because it has complex memory management rules.

The best way to understand the mapping is to read either the specification or, better, a book about using CORBA from C++. Reading the code generated by omniidl is hard-going, and it is difficult to distinguish the parts you need to know from the implementation details.

2.5.1 Mapping overview

For interface `Echo`, omniidl generates four things of note:

- class `Echo`, containing static functions and type definitions
- `Echo_ptr`, an *object reference* type with pointer semantics
- `Echo_var`, a memory management helper for `Echo_ptr`
- class `POA_Echo`, the server-side *skeleton* class

2.5.2 Interface scope type

A C++ class `Echo` is defined to hold a number of static functions and type definitions. It looks like this:

```
class Echo {
public:
  typedef Echo_ptr _ptr_type;
  typedef Echo_var _var_type;

  static _ptr_type _duplicate(_ptr_type);
  static _ptr_type _narrow(CORBA::Object_ptr);
  static _ptr_type _nil();
};
```

The `_ptr_type` and `_var_type` typedefs are there to facilitate template programming. The static functions are described below.

2.5.3 Object reference pointer type

For interface `Echo`, the mapping defines the object reference type `Echo_ptr` which has pointer semantics. The `_ptr` type provides access to the interface's operations. The concrete type of an object reference is opaque, i.e. you must not make any assumptions about how an object reference is implemented. You can imagine it looks something like this:

```
class private_class : public some_base_class {
  char* echoString(const char* mesg);
};

typedef something Echo_ptr;
```

To use an object reference, you use the arrow operator '`->`' to invoke its operations, but you must not use it as a C++ pointer in any other respect. It is non-compliant to convert it to `void*`, perform arithmetic or relational operations including testing for equality using `operator==`.

In some CORBA implementations, `Echo_ptr` is a typedef to `Echo*`. In omniORB, it is not—the object reference type is distinct from class `Echo`.

2.5.3.1 Nil object reference

Object references can be *nil*. To obtain a nil object reference for interface `Echo`, call `Echo::_nil()`. To test if an object reference is nil, use `CORBA::_is_nil()`:

```
CORBA::Boolean true_result = CORBA::is_nil(Echo::_nil());
```

`Echo::_nil()` is the only compliant way to obtain a nil Echo reference, and `CORBA::is_nil()` is the only compliant way to check if an object reference is nil. You should not use the equality `operator==`. Many C++ ORBs use the null pointer to represent a nil object reference, but *omniORB does not*.

2.5.3.2 Object reference lifecycle

Object references are reference counted. That is, the opaque C++ objects on the client side that implement `Echo_ptr` are reference counted, so they are deleted when the count goes to zero. The lifetime of an object reference has no bearing at all on the lifetime of the CORBA object to which it is a reference—when an object reference is deleted, it has *no effect* on the object in the server.

Reference counting for `Echo` object references is performed with `Echo::_duplicate()` and `CORBA::release()`.

The `_duplicate()` function returns a new object reference of the Echo interface. The new object reference can be used interchangeably with the old object reference to perform an operation on the same object.

To indicate that an object reference will no longer be accessed, you must call the `CORBA::release()` operation. Its signature is as follows:

```
namespace CORBA {
  void release(CORBA::Object_ptr obj);
  ... // other methods
};
```

Once you have called `CORBA::release()` on an object reference, you may no longer use that reference. This is because the associated resources may have been deallocated. Remember that we are referring to the resources associated with the object reference and *not the servant object*. Servant objects are not affected by the lifetimes of object references. In particular, servants are not deleted when all references to them have been released—CORBA does not perform distributed garbage collection.

Nil object references are *not* reference counted, so there is no need to call `_duplicate()` and `release()` with them, although it does no harm.

Since object references must be released explicitly, their usage is prone to error and can lead to memory leaks or invalid memory accesses. The mapping defines the *object reference variable* type `Echo_var` to make life somewhat easier.

The `Echo_var` is more convenient to use because it automatically releases its object reference when it goes out of scope or when assigned a new object reference. For many operations, mixing data of type `Echo_var` and `Echo_ptr` is possible without any explicit operations or casting. For instance, the `echoString()` operation can be called using the arrow (‘`->`’) on a `Echo_var`, as one can do with a `Echo_ptr`.

The usage of `Echo_var` is illustrated below:

```
Echo_var a;
Echo_ptr p = ... // somehow obtain an object reference

a = p;           // a assumes ownership of p, must not use p any more
```

```
Echo_var b = a;  // implicit _duplicate

p = ...          // somehow obtain another object reference

a = Echo::_duplicate(p);     // release old object reference
                             // a now holds a copy of p.
```

The mappings of many other IDL data types include `_var` types with similar semantics.

2.5.3.3 Object reference inheritance

All CORBA objects inherit from the generic object `CORBA::Object`. `CORBA::Object_ptr` is the object reference type for base `CORBA::Object`. Object references can be implicitly *widened* to base interface types, so this is valid:

```
Echo_ptr echo_ref = // get reference from somewhere
CORBA::Object_ptr base_ref = echo_ref; // widen
```

An object reference such as `Echo_ptr` can be used in places where a `CORBA::Object_ptr` is expected. Conversely, the `Echo::_narrow()` function takes an argument of type `CORBA::Object_ptr` and returns a new object reference of the `Echo` interface. If the actual (runtime) type of the argument object reference can be narrowed to `Echo_ptr`, `_narrow()` will return a valid object reference. Otherwise it will return a nil object reference. Note that `_narrow()` performs an implicit duplication of the object reference, so the result must be released. Note also that `_narrow()` may involve a remote call to check the type of the object, so it may throw CORBA system exceptions such as `TRANSIENT` or `OBJECT_NOT_EXIST`.

2.5.3.4 Object reference equivalence

As described above, the equality `operator==` should not be used on object references. To test if two object references are equivalent, the member function `_is_equivalent()` of the generic object `CORBA::Object` can be used. Here is an example of its usage:

```
Echo_ptr a;
...            // initialise a to a valid object reference
Echo_ptr b = a;
CORBA::Boolean true_result = a->_is_equivalent(a);
// Note: the above call is guaranteed to be true
```

`_is_equivalent()` does *not* contact the object to check for equivalence—it uses purely local knowledge, meaning that it is possible to construct situations in which two object references refer to the same object, but `_is_equivalent()`

does not consider them equivalent. If you need a strong sense of object identity, you must implement it with explicit IDL operations.

2.5.4 Servant Object Implementation

For each object interface, a *skeleton* class is generated. In our example, the POA specification says that the skeleton class for interface `Echo` is named `POA_Echo`. A servant implementation can be written by creating an implementation class that derives from the skeleton class.

The skeleton class `POA_Echo` is defined in `echo.hh`. The relevant section of the code is reproduced below.

```
class POA_Echo :
  public virtual PortableServer::ServantBase
{
public:
  Echo_ptr _this();

  virtual char * echoString(const char* mesg) = 0;
};
```

The code fragment shows the only member functions that can be used in the object implementation code. Other member functions are generated for internal use only. As with the code generated for object references, other POA-based ORBs will generate code which looks different, but is functionally equivalent to this.

`echoString()`
: It is through this abstract function that an implementation class provides the implementation of the `echoString()` operation. Notice that its signature is the same as the `echoString()` function that can be invoked via the `Echo_ptr` object reference. This will be the case most of the time, but object reference operations for certain parameter types use special helper classes to facilitate correct memory management.

`_this()`
: The `_this()` function returns an object reference for the target object, provided the POA policies permit it. The returned value must be deallocated via `CORBA::release()`. See section 2.8 for an example of how this function is used.

2.6 Writing the servant implementation

You define a class to provide the servant implementation. There is little constraint on how you design your implementation class except that it has to inherit

from the skeleton class[1] and to implement all the abstract functions defined in the skeleton class. Each of these abstract functions corresponds to an operation of the interface. They are the hooks for the ORB to perform upcalls to your implementation. Here is a simple implementation of the Echo object.

```
class Echo_i : public POA_Echo
{
public:
  inline Echo_i() {}
  virtual ~Echo_i() {}
  virtual char* echoString(const char* mesg);
};

char* Echo_i::echoString(const char* mesg)
{
  return CORBA::string_dup(mesg);
}
```

There are four points to note here:

Storage Responsibilities

String, which is used both as an in argument and the return value of `echoString()`, is a variable sized data type. Other examples of variable sized data types include sequences, type 'any', etc. For these data types, you must be clear about whose responsibility it is to allocate and release the associated storage. As a rule of thumb, the client (or the caller to the implementation functions) owns the storage of all *in* arguments, the object implementation (or the callee) must copy the data if it wants to retain a copy. For *out* arguments and return values, the object implementation allocates the storage and passes the ownership to the client. The client must release the storage when the variables will no longer be used. For details, see the C++ mapping specification.

Multi-threading

As omniORB is fully multithreaded, multiple threads may perform the same upcall to your implementation concurrently. It is up to your implementation to synchronise the threads' accesses to shared data. In our simple example, we have no shared data to protect so no thread synchronisation is necessary.

Alternatively, you can create a POA which has the `SINGLE_THREAD_MODEL` Thread Policy. This guarantees that all calls to that POA are processed sequentially.

[1] Rather than deriving from the skeleton class, an alternative is to use a *tie* template, described in section 2.11.

Reference Counting

All servant objects are reference counted. The base `PortableServer::ServantBase` class from which all servant skeleton classes derive defines member functions named `_add_ref()` and `_remove_ref()`[2]. The reference counting means that an `Echo_i` instance will be deleted when no more references to it are held by application code or the POA itself. Note that this is totally separate from the reference counting which is associated with object references—a servant object is *never* deleted due to a CORBA object reference being released.

Instantiation

Servants are usually instantiated on the heap, i.e. using the `new` operator. However, they can also be created on the stack as automatic variables. If you do that, it is vital to make sure that the servant has been deactivated, and thus released by the POA, before the variable goes out of scope and is destroyed.

2.7 Writing the client

Here is an example of how an `Echo_ptr` object reference is used.

```
1  void
2  hello(CORBA::Object_ptr obj)
3  {
4    Echo_var e = Echo::_narrow(obj);
5
6    if (CORBA::is_nil(e)) {
7      cerr << "cannot invoke on a nil object reference."
8           << endl;
9      return;
10   }
11
12   CORBA::String_var src = (const char*) "Hello!";
13   CORBA::String_var dest;
14
15   dest = e->echoString(src);
16
17   cout << "I said, \"" << src << "\"."
18        << " The Object said,\"" << dest <<"\"" << endl;
19 }
```

[2] In the previous 1.0 version of the C++ mapping, servant reference counting was optional, chosen by inheriting from a mixin class named `RefCountServantBase`. That has been deprecated in the 1.1 version of the C++ mapping, but the class is still available as an empty struct, so existing code that inherits from `RefCountServantBase` will continue to work.

The `hello()` function accepts a generic object reference. The object reference (`obj`) is narrowed to `Echo_ptr`. If the object reference returned by `Echo::_narrow()` is not nil, the operation `echoString()` is invoked. Finally, both the argument to and the return value of `echoString()` are printed to `cout`.

The example also illustrates how `T_var` types are used. As was explained in the previous section, `T_var` types take care of storage allocation and release automatically when variables are reassigned or when the variables go out of scope.

In line 4, the variable `e` takes over the storage responsibility of the object reference returned by `Echo::_narrow()`. The object reference is released by the destructor of `e`. It is called automatically when the function returns. Lines 6 and 15 show how a `Echo_var` variable is used. As explained earlier, the `Echo_var` type can be used interchangeably with the `Echo_ptr` type.

The argument and the return value of `echoString()` are stored in `CORBA::String_var` variables `src` and `dest` respectively. The strings managed by the variables are deallocated by the destructor of `CORBA::String_var`. It is called automatically when the variable goes out of scope (as the function returns). Line 15 shows how `CORBA::String_var` variables are used. They can be used in place of a string (for which the mapping is `char*`)[3]. As used in line 12, assigning a constant string (`const char*`) to a `CORBA::String_var` causes the string to be copied. On the other hand, assigning a `char*` to a `CORBA::String_var`, as used in line 15, causes the latter to assume the ownership of the string[4].

Under the C++ mapping, `T_var` types are provided for all the non-basic data types. One should use automatic variables whenever possible both to avoid memory leaks and to maximise performance. However, when one has to allocate data items on the heap, it is a good practice to use the `T_var` types to manage the heap storage.

2.8 Example 1 — Colocated Client and Servant

Having introduced the client and the object implementation, we can now describe how to link up the two via the ORB and POA. In this section, we describe an example in which both the client and the object implementation are in the same address space. In the next two sections, we shall describe the case where the two are in different address spaces.

The code for this example is reproduced below:

```
1 int
2 main(int argc, char **argv)
3 {
```

[3] A conversion operator of `CORBA::String_var` converts a `CORBA::String_var` to a `char*`.

[4] Please refer to the C++ mapping specification for details of the String_var mapping.

```
4    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv, "omniORB4");
5
6    CORBA::Object_var       obj = orb->resolve_initial_references("RootPOA");
7    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
8
9    PortableServer::Servant_var<Echo_i> myecho = new Echo_i();
10   PortableServer::ObjectId_var myechoid = poa->activate_object(myecho);
11
12   Echo_var myechoref = myecho->_this();
13
14   PortableServer::POAManager_var pman = poa->the_POAManager();
15   pman->activate();
16
17   hello(myechoref);
18
19   orb->destroy();
20   return 0;
21 }
```

The example illustrates several important interactions among the ORB, the POA, the servant, and the client. Here are the details:

2.8.1 ORB initialisation

Line 4

The ORB is initialised by calling the `CORBA::ORB_init()` function. The function uses the optional 3rd argument to determine which ORB should be returned. Unless you are using omniORB specific features, it is usually best to leave it out, and get the default ORB. To explicitly ask for omniORB 4.x, this argument must be 'omniORB4'[5].

`CORBA::ORB_init()` takes the list of command line arguments and processes any that start '`-ORB`'. It removes these arguments from the list, so application code does not have to deal with them.

If any error occurs during ORB initialisation, such as invalid ORB arguments, or an invalid configuration file, the `CORBA::INITIALIZE` system exception is raised.

2.8.2 Obtaining the Root POA

Lines 6–7

To activate our servant object and make it available to clients, we must register it with a POA. In this example, we use the *Root POA*, rather than cre-

[5] For backwards compatibility, the ORB identifiers 'omniORB2' and 'omniORB3' are also accepted.

ating any child POAs. The Root POA is found with `orb->resolve_initial_references()`, which returns a plain `CORBA::Object`. In line 7, we narrow the reference to the right type for a POA.

A POA's behaviour is governed by its *policies*. The Root POA has suitable policies for many simple servers, and closely matches the 'policies' used by omniORB 2's BOA. See Chapter 11 of the CORBA 2.6 specification[OMG01] for details of all the POA policies which are available.

2.8.3 Object initialisation

Line 9

An instance of the Echo servant is initialised using the `new` operator. The `PortableServer::Servant_var<>` template is analogous to the `T_var` types generated by the IDL compiler. It releases our reference to the servant when it goes out of scope.

Line 10

The servant object is activated in the Root POA using `poa->activate_object()`, which returns an object identifier (of type `PortableServer::ObjectId*`). The object id must be passed back to various POA operations. The caller is responsible for freeing the object id, so it is assigned to a `_var` type.

Line 12

The object reference is obtained from the servant object by calling its `_this()` method. Like all object references, the return value of `_this()` must be released by `CORBA::release()` when it is no longer needed. In this case, we assign it to a `_var` type, so the release is implicit at the end of the function.

One of the important characteristics of an object reference is that it is completely location transparent. A client can invoke on the object using its object reference without any need to know whether the servant object is colocated in the same address space or is in a different address space.

In the case of colocated client and servant, omniORB is able to short-circuit the client calls so they do not involve IIOP. The calls still go through the POA, however, so the various POA policies affect local calls in the same way as remote ones. This optimisation is applicable not only to object references returned by `_this()`, but to any object references that are passed around within the same address space or received from other address spaces via remote calls.

2.8.4 Activating the POA

Lines 14–15

POAs are initially in the *holding* state, meaning that incoming requests are blocked. Lines 15 and 16 acquire a reference to the POA's POA manager, and use it to put the POA into the *active* state. Incoming requests are now served. **Failing to activate the POA is one of the most common programming mistakes. If your program appears deadlocked, make sure you activated the POA!**

2.8.5 Performing a call

Line 17

At long last, we can call `hello()` with this object reference. The argument is widened implicitly to the generic object reference `CORBA::Object_ptr`.

2.8.6 ORB destruction

Line 19

Shutdown the ORB permanently. This call causes the ORB to release all its resources, e.g. internal threads, and also to deactivate any servant objects which are currently active. When it deactivates the `Echo_i` instance, the servant's reference count drops to zero, so the servant is deleted.

2.9 Example 2 — Different Address Spaces

In this example, the client and the object implementation reside in two different address spaces. The code of this example is almost the same as the previous example. The only difference is the extra work which needs to be done to pass the object reference from the object implementation to the client.

The simplest (and quite primitive) way to pass an object reference between two address spaces is to produce a *stringified* version of the object reference and to pass this string to the client as a command-line argument. The string is then converted by the client into a proper object reference. This method is used in this example. In the next example, we shall introduce a better way of passing the object reference using the CORBA Naming Service.

2.9.1 Making a Stringified Object Reference

The `main()` function of the server side is reproduced below. The full listing (`eg2_impl.cc`) can be found at the end of this chapter.

```
1  int main(int argc, char** argv)
2  {
3    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
4
5    CORBA::Object_var       obj = orb->resolve_initial_references("RootPOA");
6    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
7
8    PortableServer::Servant_var<Echo_i> myecho = new Echo_i();
9
10   PortableServer::ObjectId_var myechoid = poa->activate_object(myecho);
11
12   obj = myecho->_this();
13   CORBA::String_var sior(orb->object_to_string(obj));
14   cerr << sior << endl;
15
16   PortableServer::POAManager_var pman = poa->the_POAManager();
17   pman->activate();
18
19   orb->run();
20   orb->destroy();
21   return 0;
22 }
```

The stringified object reference is obtained by calling the ORB's `object_to_string()` function (line 13). This results in a string starting with the signature '`IOR:`' and followed by quite a lot of hexadecimal digits. All CORBA compliant ORBs are able to convert the string into its internal representation of a so-called Interoperable Object Reference (IOR). The IOR contains the location information and a key to uniquely identify the object implementation in its own address space. From the IOR, an object reference can be constructed.

2.9.2 Client: Using a Stringified Object Reference

The stringified object reference is passed to the client as a command-line argument. The client uses the ORB's `string_to_object()` function to convert the string into a generic object reference (`CORBA::Object_ptr`). The relevant section of the code is reproduced below. The full listing (`eg2_clt.cc`) can be found at the end of this chapter.

```
try {
  CORBA::Object_var obj = orb->string_to_object(argv[1]);
  hello(obj);
}
catch (CORBA::TRANSIENT&) {
  ... // code to handle transient exception...
}
```

2.9.3 Catching System Exceptions

When omniORB detects an error condition, it may raise a system exception. The CORBA specification defines a series of exceptions covering most of the error conditions that an ORB may encounter. The client may choose to catch these exceptions and recover from the error condition[6]. For instance, the code fragment, shown in section 2.9.2, catches the `TRANSIENT` system exception which indicates that the object could not be contacted at the time of the call, usually meaning the server is not running.

All system exceptions inherit from `CORBA::SystemException`. Unless you have a truly ancient C++ compiler, a single catch of `CORBA::SystemException` will catch all the different system exceptions.

2.9.4 Lifetime of a CORBA object

CORBA objects are either *transient* or *persistent*. The majority are transient, meaning that the lifetime of the CORBA object (as contacted through an object reference) is the same as the lifetime of its servant object. Persistent objects can live beyond the destruction of their servant object, the POA they were created in, and even their process. Persistent objects are, of course, only contactable when their associated server processes are running, and their servants are active or can be activated by their POA with a servant manager[7]. A reference to a persistent object can be published, and will remain valid even if the server process is restarted.

To support persistent objects, the servants must be activated in their POA with the same object identifier each time. Also, the server must be configured with the same *endpoint* details so it is contactable in the same way as previous invocations. See chapter 7 for details.

A POA's Lifespan Policy determines whether objects created within it are transient or persistent. The Root POA has the `TRANSIENT` policy.

An alternative to creating persistent objects is to register object references in a *naming service* and bind them to fixed path names. Clients can bind to the object implementations at run time by asking the naming service to resolve the path names to the object references. CORBA defines a standard naming service, which is a component of the Common Object Services (COS) [OMG98], that can be used for this purpose. The next section describes an example of how to use the COS Naming Service.

[6] If a system exception is not caught, the C++ runtime will call the `terminate()` function. This function is defaulted to abort the whole process and on some systems will cause a core file to be produced.

[7] The POA itself can be activated on demand with an adapter activator.

2.10 Example 3 — Using the Naming Service

In this example, the object implementation uses the Naming Service [OMG98] to pass on the object reference to the client. This method is often more practical than using stringified object references. The full listing of the object implementation (`eg3_impl.cc`) and the client (`eg3_clt.cc`) can be found at the end of this chapter.

The names used by the Naming service consist of a sequence of *name components*. Each name component has an *id* and a *kind* field, both of which are strings. All name components except the last one are bound to *naming contexts*. A naming context is analogous to a directory in a filing system: it can contain names of object references or other naming contexts. The last name component is bound to an object reference.

Sequences of name components can be represented as a flat string, using '.' to separate the id and kind fields, and '/' to separate name components from each other[8]. In our example, the Echo object reference is bound to the stringified name '`test.my_context/Echo.Object`'.

The kind field is intended to describe the name in a syntax-independent way. The naming service does not interpret, assign, or manage these values. However, both the name and the kind attribute must match for a name lookup to succeed. In this example, the kind values for `test` and `Echo` are chosen to be '`my_context`' and '`Object`' respectively. This is an arbitrary choice as there is no standardised set of kind values.

2.10.1 Obtaining the Root Context Object Reference

The initial contact with the Naming Service can be established via the *root* context. The object reference to the root context is provided by the ORB and can be obtained by calling `resolve_initial_references()`. The following code fragment shows how it is used:

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv);
CORBA::Object_var obj = orb->resolve_initial_references("NameService");

CosNaming::NamingContext_var rootContext;
rootContext = CosNaming::NamingContext::_narrow(obj);
```

Remember from section 1.2, omniORB constructs its internal list of initial references at initialisation time using the information provided in the configuration file `omniORB.cfg`, or given on the command line. If this file is not present,

[8] There are escaping rules to cope with id and kind fields which contain '.' and '/' characters. See chapter 8 of this manual, and chapter 3 of the CORBA services specification, as updated for the Interoperable Naming Service [OMG00].

the internal list will be empty and `resolve_initial_references()` will raise a `CORBA::ORB::InvalidName` exception.

2.10.2 The Naming Service Interface

It is beyond the scope of this chapter to describe in detail the Naming Service interface. You should consult the CORBA services specification [OMG98] (chapter 3). The code listed in `eg3_impl.cc` and `eg3_clt.cc` are good examples of how the service can be used.

2.11 Example 4 — Using tie implementation templates

omniORB supports *tie* implementation templates as an alternative way of providing servant classes. If you use the `-Wbtp` option to omniidl, it generates an extra template class for each interface. This template class can be used to tie a C++ class to the skeleton class of the interface.

The source code in `eg3_tieimpl.cc` at the end of this chapter illustrates how the template class can be used. The code is almost identical to `eg3_impl.cc` with only a few changes.

Firstly, the servant class `Echo_i` does not inherit from any skeleton classes. This is the main benefit of using the template class because there are applications in which it is difficult to require every servant class to derive from CORBA classes.

Secondly, the instantiation of a CORBA object now involves creating an instance of the implementation class *and* an instance of the template. Here is the relevant code fragment:

```
class Echo_i { ... };

Echo_i *myimpl = new Echo_i();
POA_Echo_tie<Echo_i> myecho(myimpl);

PortableServer::ObjectId_var myechoid = poa->activate_object(&myecho);
```

For interface `Echo`, the name of its tie implementation template is `POA_Echo_tie`. The template parameter is the servant class that contains an implementation of each of the operations defined in the interface. As used above, the tie template takes ownership of the `Echo_i` instance, and deletes it when the tie object goes out of scope. The tie constructor has an optional boolean argument (defaulted to `true`) which indicates whether or not it should delete the servant object. For full details of using tie templates, see the CORBA C++ mapping specification.

2.12 Source Listings

2.12.1 eg1.cc

```
// eg1.cc - This is the source code of example 1 used in Chapter 2
//          "The Basics" of the omniORB user guide.
//
//          In this example, both the object implementation and the
//          client are in the same process.
//
// Usage: eg1
//

#include <echo.hh>

#include <iostream>
using namespace std;

// This is the object implementation.

class Echo_i : public POA_Echo
{
public:
  inline Echo_i() {}
  virtual ~Echo_i() {}
  virtual char* echoString(const char* mesg);
};


char* Echo_i::echoString(const char* mesg)
{
  // Memory management rules say we must return a newly allocated
  // string.
  return CORBA::string_dup(mesg);
}


//////////////////////////////////////////////////////////////////////

// This function acts as a client to the object.

static void hello(Echo_ptr e)
{
  if( CORBA::is_nil(e) ) {
    cerr << "hello: The object reference is nil!" << endl;
    return;
  }
```

```
  CORBA::String_var src = (const char*) "Hello!";
  // String literals are (char*) rather than (const char*) on some
  // old compilers.  Thus it is essential to cast to (const char*)
  // here to ensure that the string is copied, so that the
  // CORBA::String_var does not attempt to 'delete' the string
  // literal.

  CORBA::String_var dest = e->echoString(src);

  cout << "I said, \"" << (char*)src << "\"." << endl
       << "The Echo object replied, \"" << (char*)dest <<"\"." << endl;
}

//////////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
  try {
    // Initialise the ORB.
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Obtain a reference to the root POA.
    CORBA::Object_var       obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

    // We allocate the servant (implementation object) on the heap.
    // The servant is reference counted. We start out holding a
    // reference, and when the object is activated, the POA holds
    // another reference. The PortableServer::Servant_var<> template
    // automatically releases our reference when it goes out of scope.
    PortableServer::Servant_var<Echo_i> myecho = new Echo_i();

    // Activate the object. This tells the POA that this object is
    // ready to accept requests.
    PortableServer::ObjectId_var myechoid = poa->activate_object(myecho);

    // Obtain a reference to the object.
    Echo_var myechoref = myecho->_this();

    // Obtain a POAManager, and tell the POA to start accepting
    // requests on its objects.
    PortableServer::POAManager_var pman = poa->the_POAManager();
    pman->activate();

    // Do the client-side call.
    hello(myechoref);
```

```
    // Clean up all the resources.
    orb->destroy();
  }
  catch (CORBA::SystemException& ex) {
    cerr << "Caught CORBA::" << ex._name() << endl;
  }
  catch (CORBA::Exception& ex) {
    cerr << "Caught CORBA::Exception: " << ex._name() << endl;
  }
  return 0;
}
```

2.12.2 eg2_impl.cc

```
// eg2_impl.cc - This is the source code of example 2 used in Chapter 2
//               "The Basics" of the omniORB user guide.
//
//               This is the object implementation.
//
// Usage: eg2_impl
//
//        On startup, the object reference is printed to cout as a
//        stringified IOR. This string should be used as the argument to
//        eg2_clt.
//

#include <echo.hh>

#include <iostream>
using namespace std;


class Echo_i : public POA_Echo
{
public:
  inline Echo_i() {}
  virtual ~Echo_i() {}
  virtual char* echoString(const char* mesg);
};


char* Echo_i::echoString(const char* mesg)
{
  cout << "Upcall: " << mesg << endl;
  return CORBA::string_dup(mesg);
}

//////////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
  try {
    CORBA::ORB_var          orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var       obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

    PortableServer::Servant_var<Echo_i> myecho = new Echo_i();

    PortableServer::ObjectId_var myechoid = poa->activate_object(myecho);
```

```
    // Obtain a reference to the object, and print it out as a
    // stringified IOR.
    obj = myecho->_this();
    CORBA::String_var sior(orb->object_to_string(obj));
    cout << sior << endl;

    PortableServer::POAManager_var pman = poa->the_POAManager();
    pman->activate();

    // Block until the ORB is shut down.
    orb->run();
  }
  catch (CORBA::SystemException& ex) {
    cerr << "Caught CORBA::" << ex._name() << endl;
  }
  catch (CORBA::Exception& ex) {
    cerr << "Caught CORBA::Exception: " << ex._name() << endl;
  }
  return 0;
}
```

2.12.3 eg2_clt.cc

```
// eg2_clt.cc - This is the source code of example 2 used in Chapter 2
//              "The Basics" of the omniORB user guide.
//
//              This is the client. The object reference is given as a
//              stringified IOR on the command line.
//
// Usage: eg2_clt <object reference>
//

#include <echo.hh>

#include <iostream>
using namespace std;


static void hello(Echo_ptr e)
{
  CORBA::String_var src = (const char*) "Hello!";

  CORBA::String_var dest = e->echoString(src);

  cout << "I said, \"" << (char*)src << "\"." << endl
       << "The Echo object replied, \"" << (char*)dest <<"\"." << endl;
}

//////////////////////////////////////////////////////////////////////

int main(int argc, char** argv)
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    if (argc != 2) {
      cerr << "usage:  eg2_clt <object reference>" << endl;
      return 1;
    }

    CORBA::Object_var obj = orb->string_to_object(argv[1]);

    Echo_var echoref = Echo::_narrow(obj);

    if (CORBA::is_nil(echoref)) {
      cerr << "Can't narrow reference to type Echo (or it was nil)." << endl;
      return 1;
    }
```

```
      for (CORBA::ULong count=0; count<10; count++)
        hello(echoref);

      orb->destroy();
    }
    catch (CORBA::TRANSIENT&) {
      cerr << "Caught system exception TRANSIENT -- unable to contact the "
           << "server." << endl;
    }
    catch (CORBA::SystemException& ex) {
      cerr << "Caught a CORBA::" << ex._name() << endl;
    }
    catch (CORBA::Exception& ex) {
      cerr << "Caught CORBA::Exception: " << ex._name() << endl;
    }
    return 0;
  }
```

2.12.4 eg3_impl.cc

```
// eg3_impl.cc - This is the source code of example 3 used in Chapter 2
//               "The Basics" of the omniORB user guide.
//
//               This is the object implementation.
//
// Usage: eg3_impl
//
//        On startup, the object reference is registered with the
//        COS naming service. The client uses the naming service to
//        locate this object.
//
//        The name which the object is bound to is as follows:
//              root  [context]
//               |
//              test  [context] kind [my_context]
//               |
//              Echo  [object]  kind [Object]
//

#include <echo.hh>

#include <iostream>
using namespace std;

static CORBA::Boolean bindObjectToName(CORBA::ORB_ptr, CORBA::Object_ptr);


class Echo_i : public POA_Echo
{
public:
  inline Echo_i() {}
  virtual ~Echo_i() {}
  virtual char* echoString(const char* mesg);
};


char* Echo_i::echoString(const char* mesg)
{
  return CORBA::string_dup(mesg);
}

//////////////////////////////////////////////////////////////////////

int
main(int argc, char **argv)
```

```
{
  try {
    CORBA::ORB_var          orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var       obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

    PortableServer::Servant_var<Echo_i> myecho = new Echo_i();

    PortableServer::ObjectId_var myechoid = poa->activate_object(myecho);

    // Obtain a reference to the object, and register it in
    // the naming service.
    obj = myecho->_this();

    CORBA::String_var sior(orb->object_to_string(obj));
    cout << sior << endl;

    if (!bindObjectToName(orb, obj))
      return 1;

    PortableServer::POAManager_var pman = poa->the_POAManager();
    pman->activate();

    orb->run();
  }
  catch (CORBA::SystemException& ex) {
    cerr << "Caught CORBA::" << ex._name() << endl;
  }
  catch (CORBA::Exception& ex) {
    cerr << "Caught CORBA::Exception: " << ex._name() << endl;
  }
  return 0;
}

//////////////////////////////////////////////////////////////////////

static CORBA::Boolean
bindObjectToName(CORBA::ORB_ptr orb, CORBA::Object_ptr objref)
{
  CosNaming::NamingContext_var rootContext;

  try {
    // Obtain a reference to the root context of the Name service:
    CORBA::Object_var obj = orb->resolve_initial_references("NameService");

    // Narrow the reference returned.
    rootContext = CosNaming::NamingContext::_narrow(obj);
```

```
    if (CORBA::is_nil(rootContext)) {
      cerr << "Failed to narrow the root naming context." << endl;
      return 0;
    }
  }
  catch (CORBA::NO_RESOURCES&) {
    cerr << "Caught NO_RESOURCES exception. You must configure omniORB "
         << "with the location" << endl
         << "of the naming service." << endl;
    return 0;
  }
  catch (CORBA::ORB::InvalidName&) {
    // This should not happen!
    cerr << "Service required is invalid [does not exist]." << endl;
    return 0;
  }

  try {
    // Bind a context called "test" to the root context:

    CosNaming::Name contextName;
    contextName.length(1);
    contextName[0].id   = (const char*) "test";       // string copied
    contextName[0].kind = (const char*) "my_context"; // string copied

    CosNaming::NamingContext_var testContext;
    try {
      // Bind the context to root.
      testContext = rootContext->bind_new_context(contextName);
    }
    catch(CosNaming::NamingContext::AlreadyBound& ex) {
      // If the context already exists, this exception will be raised.
      // In this case, just resolve the name and assign testContext
      // to the object returned:
      CORBA::Object_var obj = rootContext->resolve(contextName);
      testContext = CosNaming::NamingContext::_narrow(obj);
      if (CORBA::is_nil(testContext)) {
        cerr << "Failed to narrow naming context." << endl;
        return 0;
      }
    }

    // Bind objref with name Echo to the testContext:
    CosNaming::Name objectName;
    objectName.length(1);
    objectName[0].id   = (const char*) "Echo";   // string copied
    objectName[0].kind = (const char*) "Object"; // string copied
```

```
      try {
        testContext->bind(objectName, objref);
      }
      catch(CosNaming::NamingContext::AlreadyBound& ex) {
        testContext->rebind(objectName, objref);
      }
      // Note: Using rebind() will overwrite any Object previously bound
      //       to /test/Echo with obj.
      //       Alternatively, bind() can be used, which will raise a
      //       CosNaming::NamingContext::AlreadyBound exception if the name
      //       supplied is already bound to an object.
    }
    catch (CORBA::TRANSIENT& ex) {
      cerr << "Caught system exception TRANSIENT -- unable to contact the "
           << "naming service." << endl
           << "Make sure the naming server is running and that omniORB is "
           << "configured correctly." << endl;

      return 0;
    }
    catch (CORBA::SystemException& ex) {
      cerr << "Caught a CORBA::" << ex._name()
           << " while using the naming service." << endl;
      return 0;
    }
    return 1;
  }
```

2.12.5 eg3_clt.cc

```
// eg3_clt.cc - This is the source code of example 3 used in Chapter 2
//              "The Basics" of the omniORB user guide.
//
//              This is the client. It uses the COS naming service
//              to obtain the object reference.
//
// Usage: eg3_clt
//
//
//        On startup, the client lookup the object reference from the
//        COS naming service.
//
//        The name which the object is bound to is as follows:
//              root  [context]
//               |
//              text  [context] kind [my_context]
//               |
//              Echo  [object]  kind [Object]
//

#include <echo.hh>

#include <iostream>
using namespace std;

static void hello(Echo_ptr e)
{
  if (CORBA::is_nil(e)) {
    cerr << "hello: The object reference is nil!\n" << endl;
    return;
  }

  CORBA::String_var src = (const char*) "Hello!";

  CORBA::String_var dest = e->echoString(src);

  cerr << "I said, \"" << (char*)src << "\"." << endl
       << "The Echo object replied, \"" << (char*)dest <<"\"." << endl;
}

//////////////////////////////////////////////////////////////////////

int
main (int argc, char **argv)
{
```

```
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // We use a corbaname URI to resolve the name
    const char*       uri = "corbaname:rir:#test.my_context/Echo.Object";
    CORBA::Object_var obj = orb->string_to_object(uri);

    Echo_var echoref = Echo::_narrow(obj);

    for (CORBA::ULong count=0; count < 10; count++)
      hello(echoref);

    orb->destroy();
    return 0;
  }
  catch (CORBA::TRANSIENT&) {
    cerr << "Caught system exception TRANSIENT -- unable to contact the "
         << "server." << endl;
  }
  catch (CORBA::NO_RESOURCES&) {
    cerr << "Caught NO_RESOURCES exception." << endl
         << "You must configure omniORB with the location of the naming service."
         << endl;
  }
  catch (CORBA::BAD_PARAM&) {
    cerr << "Caught BAD_PARAM exception." << endl
         << "The object is not registered in the naming service."
         << endl;
  }
  catch (CORBA::SystemException& ex) {
    cerr << "Caught a CORBA::" << ex._name() << endl;
  }
  catch (CORBA::Exception& ex) {
    cerr << "Caught CORBA::Exception: " << ex._name() << endl;
  }
  return 1;
}
```

2.12.6 eg3_tieimpl.cc

```
// eg3_tieimpl.cc - This example is similar to eg3_impl.cc except that
//                  the tie implementation skeleton is used.
//
//               This is the object implementation.
//
// Usage: eg3_tieimpl
//
//        On startup, the object reference is registered with the
//        COS naming service. The client uses the naming service to
//        locate this object.
//
//        The name which the object is bound to is as follows:
//              root  [context]
//               |
//              test  [context] kind [my_context]
//               |
//              Echo  [object]  kind [Object]
//

#include <echo.hh>

#include <iostream>
using namespace std;

static CORBA::Boolean bindObjectToName(CORBA::ORB_ptr, CORBA::Object_ptr);


// This is the object implementation.  Notice that it does not inherit
// from any skeleton class, and notice that the echoString() member
// function does not have to be virtual.

class Echo_i {
public:
  inline Echo_i() {}
  inline ~Echo_i() {}
  char* echoString(const char* mesg);
};


char* Echo_i::echoString(const char* mesg)
{
  return CORBA::string_dup(mesg);
}

//////////////////////////////////////////////////////////////////////
```

```
int main(int argc, char** argv)
{
  try {
    CORBA::ORB_var          orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var       obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

    // Note that the <myecho> tie object is constructed on the stack
    // here. It will delete its implementation (myimpl) when it it
    // itself destroyed (when it goes out of scope).  It is essential
    // however to ensure that such servants are not deleted whilst
    // still activated.
    //
    // Tie objects can of course be allocated on the heap using new,
    // in which case they are deleted when their reference count
    // becomes zero, as with any other servant object.
    Echo_i* myimpl = new Echo_i();
    POA_Echo_tie<Echo_i> myecho(myimpl);

    PortableServer::ObjectId_var myechoid = poa->activate_object(&myecho);

    // Obtain a reference to the object, and register it in
    // the naming service.
    obj = myecho._this();
    if (!bindObjectToName(orb, obj))
      return 1;

    PortableServer::POAManager_var pman = poa->the_POAManager();
    pman->activate();

    orb->run();
  }
  catch (CORBA::SystemException& ex) {
    cerr << "Caught CORBA::" << ex._name() << endl;
  }
  catch (CORBA::Exception& ex) {
    cerr << "Caught CORBA::Exception: " << ex._name() << endl;
  }
  return 0;
}

//////////////////////////////////////////////////////////////////////

static CORBA::Boolean
bindObjectToName(CORBA::ORB_ptr orb, CORBA::Object_ptr objref)
{
```

```
CosNaming::NamingContext_var rootContext;

try {
  // Obtain a reference to the root context of the Name service:
  CORBA::Object_var obj = orb->resolve_initial_references("NameService");

  // Narrow the reference returned.
  rootContext = CosNaming::NamingContext::_narrow(obj);
  if (CORBA::is_nil(rootContext)) {
    cerr << "Failed to narrow the root naming context." << endl;
    return 0;
  }
}
catch (CORBA::NO_RESOURCES&) {
  cerr << "Caught NO_RESOURCES exception. You must configure omniORB "
       << "with the location" << endl
       << "of the naming service." << endl;
  return 0;
}
catch (CORBA::ORB::InvalidName&) {
  // This should not happen!
  cerr << "Service required is invalid [does not exist]." << endl;
  return 0;
}

try {
  // Bind a context called "test" to the root context:

  CosNaming::Name contextName;
  contextName.length(1);
  contextName[0].id   = (const char*) "test";       // string copied
  contextName[0].kind = (const char*) "my_context"; // string copied

  CosNaming::NamingContext_var testContext;
  try {
    // Bind the context to root.
    testContext = rootContext->bind_new_context(contextName);
  }
  catch(CosNaming::NamingContext::AlreadyBound& ex) {
    // If the context already exists, this exception will be raised.
    // In this case, just resolve the name and assign testContext
    // to the object returned:
    CORBA::Object_var obj = rootContext->resolve(contextName);
    testContext = CosNaming::NamingContext::_narrow(obj);
    if (CORBA::is_nil(testContext)) {
      cerr << "Failed to narrow naming context." << endl;
      return 0;
```

```
      }
    }

    // Bind objref with name Echo to the testContext:
    CosNaming::Name objectName;
    objectName.length(1);
    objectName[0].id   = (const char*) "Echo";   // string copied
    objectName[0].kind = (const char*) "Object"; // string copied

    try {
      testContext->bind(objectName, objref);
    }
    catch(CosNaming::NamingContext::AlreadyBound& ex) {
      testContext->rebind(objectName, objref);
    }
    // Note: Using rebind() will overwrite any Object previously bound
    //       to /test/Echo with obj.
    //       Alternatively, bind() can be used, which will raise a
    //       CosNaming::NamingContext::AlreadyBound exception if the name
    //       supplied is already bound to an object.
  }
  catch (CORBA::TRANSIENT& ex) {
    cerr << "Caught system exception TRANSIENT -- unable to contact the "
         << "naming service." << endl
         << "Make sure the naming server is running and that omniORB is "
         << "configured correctly." << endl;

    return 0;
  }
  catch (CORBA::SystemException& ex) {
    cerr << "Caught a CORBA::" << ex._name()
         << " while using the naming service." << endl;
    return 0;
  }
  return 1;
}
```

Chapter 3

C++ language mapping

Now that you are familiar with the basics, it is important to familiarise yourself with the standard IDL to C++ language mapping. The mapping is described in detail in [OMG03]. If you have not done so, you should obtain a copy of the document and use that as the programming guide to omniORB.

The specification is not an easy read. The alternative is to use one of the books on CORBA programming. For instance, Henning and Vinoski's 'Advanced CORBA Programming with C++' [HV99] includes many example code fragments to illustrate how to use the C++ mapping.

3.1 omniORB 2 BOA compatibility

Before the Portable Object Adapter (POA) specification, many of the details of how servant objects should be implemented and registered with the system were unspecified, so server-side code was not portable between ORBs. The POA specification rectifies that. For compatibility, omniORB 4 still supports the old omniORB 2.x BOA mapping, but you should always use the POA mapping for new code. BOA code and POA code can coexist within a single program.

If you use the `-WbBOA` option to omniidl, it will generate skeleton code with (nearly) the same interface as the old omniORB 2 BOA mapping, as well as code to be used with the POA. Note that since the major problem with the BOA specification was that server code was not portable between ORBs, it is unlikely that omniORB's BOA compatibility will help you much if you are moving from a different BOA-based ORB.

The BOA compatibility permits the majority of BOA code to compile without difficulty. However, there are a number of constructs which relied on omniORB 2 implementation details which no longer work.

- omniORB 2 did not use distinct types for object references and servants, and often accepted a pointer to a servant when the CORBA specification

says it should only accept an object reference. Such code will not compile under omniORB 4.

- The reverse is true for `BOA::obj_is_ready()`. It now only works when passed a pointer to a servant object, not an object reference. The more commonly used mechanism of calling `_obj_is_ready(boa)` on the servant object still works as expected.

- It used to be the case that the skeleton class for interface `I` (`_sk_I`) was derived from class `I`. This meant that the names of any types declared in the interface were available in the scope of the skeleton class. This is no longer true. If you have an interface:

```
interface I {
  struct S {
    long a,b;
  };
  S op();
};
```

then where before the implementation code might have been:

```
class I_impl : public virtual _sk_I {
  S op();  // _sk_I is derived from I
};
I::S I_impl::op() {
  S ret;
  // ...
}
```

it is now necessary to fully qualify all uses of `S`:

```
class I_impl : public virtual _sk_I {
  I::S op(); // _sk_I is not derived from I
};
I::S I_impl::op() {
  I::S ret;
  // ...
}
```

- The proprietary omniORB 2 LifeCycle extensions are no longer supported. All of the facilities it offered can be implemented with the POA interfaces, and the `omniORB::LOCATION_FORWARD` exception (see section 4.8). Code which used the old interfaces will have to be rewritten.

3.2 omniORB 3.0 compatibility

omniORB 4 is almost completely source-code compatible with omniORB 3.0. There are two main cases where code may have to change. The first is code that uses the omniORB API, some aspects of which have changed. The omniORB configuration file also has a new format. See the next chapter for details of the new API and configuration file.

The second case of code that may have to change is code using the Dynamic Any interfaces. The standard changed quite significantly between CORBA 2.2 and CORBA 2.3; omniORB 3.0 supported the old CORBA 2.2 interfaces; omniORB 4 uses the new mapping. The changes are largely syntax changes, rather than semantic differences.

3.3 omniORB 4.0 compatibility

omniORB 4.3 is source-code compatible with omniORB 4.0, with four exceptions:

1. As required by the 1.1 version of the CORBA C++ mapping specification, the `RefCountServantBase` class has been deprecated, and the reference counting functionality moved into `ServantBase`. For backwards compatibility, `RefCountServantBase` still exists, but is now defined as an empty struct. Most code will continue to work unchanged, but code that explicitly calls `RefCountServantBase::_add_ref()` or `_remove_ref()` will no longer compile.

2. omniORB 4.0 had an option for Any extraction semantics that was compatible with omniORB 2.7, where ownership of extracted values was not maintained by the Any. That option is no longer available.

3. The members of the `clientSendRequest` interceptor have been changed, replacing all the separate variables with a single member of type `GIOP_C`. All the values previously available can be accessed through the `GIOP_C` instance.

4. The C++ mapping contains Any insertion operators for sequence types that are passed by pointer, which cause the Any to take ownership of the inserted sequence. In omniORB 4.0 and earlier, the sequence was immediately marshalled into the Any's internal buffer, and the sequence was deleted. Since omniORB 4.1, the sequence pointer is stored by the Any, and the sequence is deleted later when the Any is destroyed.

   For most uses, this change is not visible to application code. However, if a sequence is constructed using an application-supplied buffer with the release flag set to `false` (meaning that the application continues to own

the buffer), it is now important that the buffer is not deleted or modified while the Any exists, since the Any continues to refer to the buffer contents. This change means that code that worked with omniORB 4.0 may now fail with 4.1, with the Any seeing modified data or the process crashing due to accessing deleted data. To avoid this situation, use the alternative Any insertion operator using a const reference, which copies the sequence.

3.4 omniORB 4.1 compatibility

omniORB 4.3 is source-code compatible with omniORB 4.1 with two exceptions:

1. When omniORB 4.1 and earlier detected a timeout condition, they would throw the `CORBA::TRANSIENT` system exception. omniORB 4.2 and later support the `CORBA::TIMEOUT` system exception that was introduced with the CORBA Messaging specification. Application code that caught `CORBA::TRANSIENT` to handle timeouts should be changed to catch `CORBA::TIMEOUT` instead. Alternatively, to avoid code changes, omniORB can be configured to throw `CORBA::TRANSIENT` for timeouts, by setting the `throwTransientOnTimeOut` parameter to `1`. See section 4.4.

2. `sslContext` has moved to the `omni` namespace.

3.5 omniORB 4.2 compatibility

omniORB 4.3 is source-code compatible with omniORB 4.2, with one exception:

1. `sslContext` has moved to the `omni` namespace.

3.6 Interoperability

In general, all versions of omniORB interoperate with each other, and with other CORBA implementations. There are a number of configuration options that can be set to work around interoperability with other CORBA implementations, described in section 4.6.

3.6.1 Exceptions in Anys

Normally, exceptions—both system exceptions and user-defined exceptions—are thrown by server operations and caught by clients. When that occurs, the GIOP protocol is completely clear that the exception is marshalled as the exception's repository id, followed by the exception's data members.

It is not permitted to specify an Exception as a normal operation parameter or within a constructed IDL type, but it *is* permitted to insert an Exception into an Any, and transmit that in an operation parameter. In that case, when the Any is marshalled, first the TypeCode is sent, and then the data value. It is somewhat ambiguous in the GIOP specification whether the value part of an Exception sent this way should contain the repository id, or not — the id is contained in the TypeCode, so sending it as part of the value too is redundant.

Versions of omniORB for C++ prior to 4.3 did not send the Exception repository id in the value part of an Any. All versions of omniORBpy *do* send it, as do some other CORBA implementations. For interoperability, omniORB 4.3 has changed to send the repository id. This means that it is compatible with omniORBpy and other ORBs, but that it is *incompatible* with previous versions of omniORB for C++. The `exceptionIdInAny` configuration parameter can be set to `false` to revert to the prior behaviour if interoperability with earlier omniORB versions is required.

Chapter 4

omniORB configuration and API

omniORB has a wide range of parameters that can be configured. They can be set in the configuration file / Windows registry, as environment variables, on the command line, or within a proprietary extra argument to `CORBA::ORB_init()`. A few parameters can be configured at run time. This chapter lists all the configuration parameters, and how they are used.

4.1 Setting parameters

When `CORBA::ORB_init()` is called, the value for each configuration parameter is searched for in the following order:

1. Command line arguments
2. `ORB_init()` options
3. Environment variables
4. Configuration file / Windows registry
5. Built-in defaults

4.1.1 Command line arguments

Command line arguments take the form '`-ORB`*parameter*', and usually expect another argument. An example is '`-ORBtraceLevel 10`'.

4.1.2 ORB_init() parameter

`ORB_init()`'s extra argument accepts an array of two-dimensional string arrays, like this:

```
const char* options[][2] = { { "traceLevel", "1" }, { 0, 0 } };
orb = CORBA::ORB_init(argc,argv,"omniORB4",options);
```

4.1.3 Environment variables

Environment variables consist of the parameter name prefixed with 'ORB'. Using bash, for example

```
export ORBtraceLevel=10
```

4.1.4 Configuration file

The best way to understand the format of the configuration file is to look at the `sample.cfg` file in the omniORB distribution. Each parameter is set on a single line like

```
traceLevel = 10
```

Some parameters can have more than one value, in which case the parameter name may be specified more than once, or you can leave it out:

```
InitRef = NameService=corbaname::host1.example.com
        = InterfaceRepository=corbaloc::host2.example.com:1234/IfR
```

Command line arguments and environment variables prefix parameter names with '-ORB' and 'ORB' respectively, but the configuration file and the extra argument to `ORB_init()` do not use a prefix.

4.1.5 Windows registry

On Windows, configuration parameters can be stored in the registry, under the key `HKEY_LOCAL_MACHINE\SOFTWARE\omniORB`.

The file `sample.reg` shows the settings that can be made. It can be edited and then imported into regedit.

4.2 Tracing options

The following options control debugging trace output.

`traceLevel` *default* = `1`

omniORB can output tracing and diagnostic messages to the standard error stream. The following levels are defined:

level 0	critical errors only
level 1	informational messages only
level 2	configuration information and warnings
level 5	notifications when server threads are created and communication endpoints are shutdown
level 10	execution and exception traces
level 25	trace each send or receive of a GIOP message
level 30	dump up to 128 bytes of each GIOP message
level 40	dump complete contents of each GIOP message

The trace level is cumulative, so at level 40, all trace messages are output.

`traceExceptions` *default* = `0`

If the `traceExceptions` parameter is set `true`, all system exceptions are logged as they are thrown, along with details about where the exception is thrown from. This parameter is enabled by default if the traceLevel is set to 10 or more.

`traceInvocations` *default* = `0`

If the `traceInvocations` parameter is set `true`, all local and remote invocations are logged, in addition to any logging that may have been selected with `traceLevel`.

`traceInvocationReturns` *default* = `0`

If the `traceInvocationReturns` parameter is set `true`, a log message is output as an operation invocation returns. In conjunction with `traceInvocations` and `traceTime` (described below), this provides a simple way of timing CORBA calls within your application.

`traceThreadId` *default* = `1`

If `traceThreadId` is set `true`, all trace messages are prefixed with the id of the thread outputting the message. This can be handy for making sense of multi-

threaded code, but it adds overhead to the logging so it can be disabled.

`traceTime` *default* = `1`

If `traceTime` is set `true`, all trace messages are prefixed with the time. This is useful, but on some platforms it adds a very large overhead, so it can be turned off.

`traceFile` *default* =

omniORB's tracing is normally sent to stderr. If `traceFile` it set, the specified file name is used for trace messages.

4.2.1 Tracing API

The tracing parameters can be modified at runtime by assigning to the following variables

```
namespace omniORB {
  CORBA::ULong   traceLevel;
  CORBA::Boolean traceExceptions;
  CORBA::Boolean traceInvocations;
  CORBA::Boolean traceInvocationReturns;
  CORBA::Boolean traceThreadId;
  CORBA::Boolean traceTime;
};
```

Log messages can be sent somewhere other than stderr by registering a logging function which is called with the text of each log message:

```
namespace omniORB {
  typedef void (*logFunction)(const char*);
  void setLogFunction(logFunction f);
};
```

The log function must not make any CORBA calls, since that could lead to infinite recursion as outputting a log message caused other log messages to be generated, and so on.

4.3 Miscellaneous global options

These options control miscellaneous features that affect the whole ORB runtime.

`dumpConfiguration` *default* = `0`

If set `true`, the ORB dumps the values of all configuration parameters at start-up.

`scanGranularity` *default* = `5`

As explained in chapter 6, omniORB regularly scans incoming and outgoing connections, so it can close unused ones. This value is the granularity in seconds at which the ORB performs its scans. A value of zero turns off the scanning altogether.

`nativeCharCodeSet` *default* = `ISO-8859-1`

The native code set the application is using for `char` and `string`. See chapter 9.

`nativeWCharCodeSet` *default* = `UTF-16`

The native code set the application is using for `wchar` and `wstring`. See chapter 9.

`defaultCharCodeSet` *default* = *none*

The default code set used for `char` and `string` if the server does not specify it in its IORs. See chapter 9.

`defaultWCharCodeSet` *default* = *none*

The default code set used for `wchar` and `wstring` if the server does not specify it in its IORs. See chapter 9.

`copyValuesInLocalCalls` *default* = `1`

Determines whether valuetype parameters in local calls are copied or not. See chapter 12.

`abortOnInternalError` *default* = `0`

If this is set `true`, internal fatal errors will abort immediately, rather than throwing the `omniORB::fatalException` exception. This can be helpful for tracking down bugs, since it leaves the call stack intact.

`abortOnNativeException` *default* = `0`

On Windows, 'native' exceptions such as segmentation faults and divide by zero appear as C++ exceptions that can be caught with `catch (...)`. Setting this parameter to `true` causes such exceptions to abort the process instead.

`maxSocketSend`

`maxSocketRecv`

On some platforms, calls to send() and recv() have a limit on the buffer size that can be used. These parameters set the limits in bytes that omniORB uses when sending / receiving bulk data.

The default values are platform specific. It is unlikely that you will need to change the values from the defaults.

The minimum valid limit is 1KB, 1024 bytes.

`socketSendBuffer` *default* = `-1`

In old Windows versions, there is a kernel buffer used during send operations. A bug in Windows means that if a send uses the entire kernel buffer, a select() on the socket blocks until all the data has been acknowledged by the receiver, resulting in dreadful performance. This parameter modifies the socket send buffer from its default (8192 bytes on Windows) to the value specified. If this parameter is set to -1, the socket send buffer is left at the system default.

`validateUTF8` *default* = `0`

When transmitting a string that is supposed to be UTF-8, omniORB usually passes it directly, assuming that it is valid. With this parameter set `true`, omniORB checks that all UTF-8 strings are valid, and throws DATA_CONVERSION if not.

4.4 Client side options

These options control aspects of client-side behaviour.

`InitRef` *default = none*

Specify objects available from `ORB::resolve_initial_references()`. The arguments take the form <*key*>=<*uri*>, where *key* is the name given to `resolve_initial_references()` and *uri* is a valid CORBA object reference URI, as detailed in chapter 8.

`DefaultInitRef` *default = none*

Specify the default URI prefix for `resolve_initial_references()`. See chapter 8.

`clientTransportRule` *default* = `* unix,tcp,ssl`

Used to specify the way the client contacts a server, depending on the server's address. See section 7.5 for details.

`clientCallTimeOutPeriod` *default* = `0`

Call timeout in milliseconds for the client side. If a call takes longer than the specified number of milliseconds, the ORB closes the connection to the server and raises a `TRANSIENT` exception. A value of zero means no timeout; calls can block for ever. See section 6.3.1 for more information about timeouts.

Note: omniORB 3 had timeouts specified in seconds; omniORB 4.0 and later use milliseconds for timeouts.

`clientConnectTimeOutPeriod` *default* = `0`

The timeout that is used in the case that a new network connection is established to the server. A value of zero means that the normal call timeout is used. See section 6.3.1 for more information about timeouts.

`supportPerThreadTimeOut` *default* = `0`

If this parameter is set `true`, timeouts can be set on a per thread basis, as well as globally and per object. Checking per-thread storage has a noticeable performance impact, so it is turned off by default.

`resetTimeOutOnRetries` *default* = `0`

If `true`, the call timeout is reset when an exception handler causes a call to be retried. If `false`, the timeout is not reset, and therefore applies to the call as a whole, rather than to each individual call attempt.

`throwTransientOnTimeOut` *default* = `0`

omniORB 4.2 and later support the `CORBA::TIMEOUT` exception that is part of the CORBA Messaging specification. By default, that is the exception thrown when timeouts occur. Previous omniORB releases did not have the `CORBA::TIMEOUT` exception, and instead used `CORBA::TRANSIENT`. If this parameter is set `true`, omniORB follows the old behaviour of throwing `CORBA::TRANSIENT` when a timeout occurs.

`outConScanPeriod` *default* = `120`

Idle timeout in seconds for outgoing (i.e. client initiated) connections. If a connection has been idle for this amount of time, the ORB closes it. See section 6.5.

`maxGIOPConnectionPerServer` *default* = `5`

The maximum number of concurrent connections the ORB will open to a *single* server. If multiple threads on the client call the same server, the ORB opens additional connections to the server, up to the maximum specified by this parameter. If the maximum is reached, threads are blocked until a connection becomes free for them to use.

`oneCallPerConnection` *default* = `1`

When this parameter is set to `true` (the default), the ORB will only send a single call on a connection at a time. If multiple client threads invoke on the same server, multiple connections are opened, up to the limit specified by `maxGIOP ConnectionPerServer`. With this parameter set to `false`, the ORB will allow concurrent calls on a single connection. This saves connection resources, but requires slightly more management work for both client and server. Some server-side ORBs (including omniORB versions before 4.0) serialise all incoming calls on a single connection.

`maxInterleavedCallsPerConnection` *default* = `5`

The maximum number of calls that can be interleaved on a connection. If more concurrent calls are made, they are queued.

`offerBiDirectionalGIOP` *default* = `0`

If set `true`, the client will indicate to servers that it is willing to accept callbacks on client-initiated connections using bidirectional GIOP, provided the relevant POA policies are set. See section 7.7.

`diiThrowsSysExceptions` *default* = `0`

If this is `true`, DII functions throw system exceptions; if it is `false`, system exceptions that occur are passed through the `Environment` object.

`verifyObjectExistsAndType` *default* = `1`

By default, omniORB uses the GIOP `LOCATE_REQUEST` message to verify the existence of an object prior to the first invocation. In the case that the full type of the object is not known, it instead calls the `_is_a()` operation to check the object's type. Some ORBs have bugs that mean one or other of these operations fail. Setting this parameter `false` prevents omniORB from making these calls.

`giopTargetAddressMode` *default* = `0`

GIOP 1.2 supports three addressing modes for contacting objects. This parameter selects the mode that omniORB uses. A value of 0 means `GIOP::KeyAddr`; 1 means `GIOP::ProfileAddr`; 2 means `GIOP::ReferenceAddr`.

`immediateAddressSwitch` *default* = `0`

If `true`, the client will immediately switch to use a new address to contact an object after a failure. If `false` (the default), the current address will be retried in certain circumstances.

`resolveNamesForTransportRules` *default* = `1`

If `true`, names in IORs will be resolved when evaluating client transport rules, and remembered from then on; if `false`, names will not be resolved until connect time. Client transport rules based on IP address will therefore not match, but some platforms can use external knowledge to pick the best address to use if given a name to connect to.

`retainAddressOrder` *default* = `1`

For IORs with multiple addresses, determines how the address to connect to is chosen. When first establishing a connection, the addresses are ordered according to the client transport rules (after resolving names if `resolveNamesForTransportRules` is `true`), and the addresses are tried in priority order until one connects successfully. For as long as there is at least one connection open to the address, new connections continue to use the same address.

After a failure, or after all open connections have been scavenged and closed, this parameter determines the address used to reconnect on the next call. If this parameter is `true` (the default), the address order and chosen address within the order is remembered; if `false`, a new connection attempt causes re-evaluation of the order (in case name resolutions change), and the highest priority address is tried first.

`bootstrapAgentHostname` *default* = *none*

If set, this parameter indicates the hostname to use for look-ups using the obsolete Sun bootstrap agent. This mechanism is superseded by the interoperable naming service.

`bootstrapAgentPort` *default* = `900`

The port number for the obsolete Sun bootstrap agent.

`principal` *default = none*

GIOP 1.0 and 1.1 have a request header field named 'principal', which contains a sequence of octets. It was never defined what it should mean, and its use is now deprecated; GIOP 1.2 has no such field. Some systems (e.g. Gnome) use the principal field as a primitive authentication scheme. This parameter sets the data omniORB uses in the principal field. The default is an empty sequence.

4.5 Server side options

These parameters affect server-side operations.

`endPoint` *default =* `giop:tcp::`
`endPointPublish`
`endPointNoPublish`

These options determine the end-points the ORB should listen on, and the details that should be published in IORs. See chapter 7 for details.

`serverTransportRule` *default =* `* unix,tcp,ssl`

Configure the rules about whether a server should accept an incoming connection from a client. See section 7.6 for details.

`serverCallTimeOutPeriod` *default =* `0`

This timeout is used to catch the situation that the server starts receiving a request, but the end of the request never comes. If a calls takes longer than the specified number of milliseconds to arrive, the ORB shuts the connection. A value of zero means never timeout.

`inConScanPeriod` *default =* `180`

Idle timeout in seconds for incoming connections. If a connection has been idle for this amount of time, the ORB closes it. See section 6.5.

`threadPerConnectionPolicy` *default =* `1`

If `true` (the default), the ORB dedicates one server thread to each incoming connection. Setting it `false` means the server should use a thread pool.

`maxServerThreadPerConnection` *default =* `100`

If the client multiplexes several concurrent requests on a single connection, omniORB uses extra threads to service them. This parameter specifies the maximum number of threads that are allowed to service a single connection at any one time.

`maxServerThreadPoolSize` *default* = `100`

The maximum number of threads the server will allocate to do various tasks, including dispatching calls in the thread pool mode. This number does not include threads dispatched under the thread per connection server mode.

`threadPerConnectionUpperLimit` *default* = `10000`

If the `threadPerConnectionPolicy` parameter is `true`, the ORB can automatically transition to thread pool mode if too many connections arrive. This parameter sets the number of connections at which thread pooling is started. The default of 10000 is designed to mean that it never happens.

`threadPerConnectionLowerLimit` *default* = `9000`

If thread pooling was started because the number of connections hit the upper limit, this parameter determines when thread per connection should start again.

`threadPoolWatchConnection` *default* = `1`

After dispatching an upcall in thread pool mode, the thread that has just performed the call can watch the connection for a short time before returning to the pool. This leads to less thread switching for a series of calls from a single client, but is less fair if there are concurrent clients. The connection is watched if the number of threads concurrently handling the connection is less than or equal to the value of this parameter. i.e. if the parameter is zero, the connection is never watched; if it is 1, the last thread managing a connection watches it; if 2, the connection is still watched if there is one other thread still in an upcall for the connection, and so on. See section 6.4.2.

`connectionWatchPeriod` *default* = `50000`

For each endpoint, the ORB allocates a thread to watch for new connections and to monitor existing connections for calls that should be handed by the thread pool. The thread blocks in select() or similar for a period, after which it rescans the lists of connections it should watch. This parameter is specified in microseconds.

`connectionWatchImmediate` *default* = `0`

When a thread handles an incoming call, it unmarshals the arguments then marks the connection as watchable by the connection watching thread, in case the client sends a concurrent call on the same connection. If this parameter is set to the default `false`, the connection is not actually watched until the next connection watch period (determined by the `connectionWatchPeriod` parameter). If this parameter is set `true`, the connection watching thread is immediately signalled to watch the connection. That leads to faster interactive response to clients that multiplex calls, but adds significant overhead along the call chain.

Note that this setting has no effect on Windows, since it has no mechanism for signalling the connection watching thread.

`acceptBiDirectionalGIOP` *default* = `0`

Determines whether a server will ever accept clients' offers of bidirectional GIOP connections. See section 7.7.

`unixTransportDirectory` *default* = `/tmp/omni-%u`

(Unix platforms only). Selects the location used to store Unix domain sockets. The '%u' is expanded to the user name.

`unixTransportPermission` *default* = `0777`

(Unix platforms only). Determines the octal permission bits for Unix domain sockets. By default, all users can connect to a server, just as with TCP.

`supportCurrent` *default* = `1`

omniORB supports the `PortableServer::Current` interface to provide thread context information to servants. Supporting current has a small but noticeable run-time overhead due to accessing thread specific storage, so this option allows it to be turned off.

`objectTableSize` *default* = `0`

Hash table size of the Active Object Map. If this is zero, the ORB uses a dynamically resized open hash table. This is normally the best option, but it leads to less predictable performance since any operation which adds or removes a table entry may trigger a resize. If set to a non-zero value, the hash table has the specified number of entries, and is never resized. Note that the hash table is open, so this does not limit the number of active objects, just how efficiently they can be located.

`poaHoldRequestTimeout` *default* = `0`

If a POA is put in the `HOLDING` state, calls to it will be timed out after the specified number of milliseconds, by raising a `CORBA::TIMEOUT` exception. Zero means no timeout.

`poaUniquePersistentSystemIds` *default* = `1`

The POA specification requires that object ids in POAs with the PERSISTENT and SYSTEM_ID policies are unique between instantiations of the POA. Older versions of omniORB did not comply with that, and reused object ids. With this value `true`, the POA has the correct behaviour; with `false`, the POA uses the old scheme for compatibility.

`idleThreadTimeout` *default* = `10`

When a thread created by omniORB becomes idle, it is kept alive for a while, in case a new thread is required. Once a thread has been idle for the number of seconds specified in this parameter, it exits.

`supportBootstrapAgent` *default* = `0`

If set `true`, servers support the Sun bootstrap agent protocol.

4.5.1 Main thread selection

There is one server-side parameter that must be set with an API function, rather than a normal configuration parameter:

```
namespace omniORB {
  void setMainThread();
};
```

POAs with the `MAIN_THREAD` policy dispatch calls on the ‘main’ thread. By default, omniORB assumes that the thread that initialised the omnithread library is the ‘main’ thread. To choose a different thread, call this function from the desired ‘main’ thread. The calling thread must have an `omni_thread` associated with it (i.e. it must have been created by omnithread, or `omni_thread::create_dummy()` must have been called). If it does not, the function throws `CORBA::INITIALIZE`.

Note that calls are only actually dispatched to the ‘main’ thread if `ORB::run()` or `ORB::perform_work()` is called from that thread.

4.6 GIOP and interoperability options

These options control omniORB's use of GIOP, and cover some areas where omniORB can work around buggy behaviour by other ORBs.

`maxGIOPVersion` *default* = `1.2`

Choose the maximum GIOP version the ORB should support. Valid values are 1.0, 1.1 and 1.2.

`giopMaxMsgSize` *default* = `2097152`

The largest message, in bytes, that the ORB will send or receive, to avoid resource starvation. If the limit is exceeded, a `MARSHAL` exception is thrown. The size must be >= 8192.

`giopBufferSize` *default* = `8192`

When transmitting data, it is marshalled into a buffer, and flushed across the network when the buffer is full. If there is a large amount of data to transmit, using a relatively small buffer allows some of the data to be in transit across the network while later marshalling activity happens in parallel. However, in some circumstances it may be beneficial to use larger buffers, and so make it more likely that the complete request/response body fits in a single buffer.

`giopDirectReceiveCutOff` *default* = `1024`

Some values, such as sequences of base integer types and strings that do not require codeset conversion, have an in-memory representation that is identical to the GIOP marshalled form. When receiving such data, if the marshalled value is larger than the direct receive cut-off, omniORB directly receives from the network into the application data structure, rather than first copying it to the GIOP buffer.

`giopDirectSendCutOff` *default* = `16384`

Some values, such as sequences of base integer types and strings that do not require codeset conversion, have an in-memory representation that is identical to the GIOP marshalled form. When sending such data, if the marshalled value is larger than the direct send cut-off, omniORB directly sends it from the application data structure, rather than first copying it to the GIOP buffer.

`giopMinChunkBeforeDirectSend` *default* = `1024`

When directly sending GIOP data (due to `giopDirectSendCutOff`), omniORB must first send any data that it has already buffered. If the buffer only contains a small amount of data, that would cause transmission of an inefficiently small network packet. If the buffer before a direct send contains less data than the minimum chunk size, enough data is copied from the application data structure into the buffer to reach the minimum size, before the rest of the data is sent directly.

`strictIIOP` *default* = `1`

If `true`, be strict about interpretation of the IIOP specification; if `false`, permit some buggy behaviour to pass.

`lcdMode` *default* = `0`

If `true`, select 'Lowest Common Denominator' mode. This disables various IIOP and GIOP features that are known to cause problems with some ORBs.

`tcAliasExpand` *default* = `0`

This flag is used to indicate whether TypeCodes associated with Anys should have aliases removed. This functionality is included because some ORBs will not recognise an Any containing a TypeCode with aliases to be the same as the actual type contained in the Any. There is a performance penalty when inserting into an Any if `tcAliasExpand` is set to 1.

`useTypeCodeIndirections` *default* = `1`

TypeCode Indirections reduce the size of marshalled TypeCodes, and are essential for recursive types, but some old ORBs do not support them. Setting this flag to `false` prevents the use of indirections (and, therefore, prevents the use of recursive TypeCodes).

`acceptMisalignedTcIndirections` *default* = `0`

If `true`, try to fix a mis-aligned indirection in a typecode. This is used to work around a bug in some old versions of Visibroker's Java ORB.

`exceptionIdInAny` *default* = `1`

If `true` (the default), when an Exception is inserted in an Any, it is transmitted as the TypeCode, followed by the Exception repository id, followed by the Exception data members. This is the format used by other CORBA implementations, including omniORBpy. Versions of omniORB for C++ prior to 4.3 did not

send the repository id after the TypeCode. If you require interoperability with earlier omniORB versions that send or receive exceptions inside Anys, set this parameter to `false`.

4.7 System Exception Handlers

By default, all system exceptions that are raised during an operation invocation, with the exception of some cases of `CORBA::TRANSIENT`, are propagated to the application code. Some applications may prefer to trap these exceptions within the proxy objects so that the application logic does not have to deal with the error condition. For example, when a `CORBA::COMM_FAILURE` is received, an application may just want to retry the invocation until it finally succeeds. This approach is useful for objects that are persistent and have idempotent operations.

omniORB provides a set of functions to install exception handlers. Once they are installed, proxy objects will call these handlers when the associated system exceptions are raised by the ORB runtime. Handlers can be installed for `CORBA::TRANSIENT`, `CORBA::TIMEOUT`, `CORBA::COMM_FAILURE` and `CORBA::SystemException`. This last handler covers all system exceptions other than the three specific ones covered by the first three handlers. An exception handler can be installed for individual proxy objects, or it can be installed for all proxy objects in the address space.

4.7.1 Minor codes

omniORB makes extensive use of exception minor codes to indicate the specific circumstances surrounding a system exception. The file `include/omniORB4/minorCode.h` contains definitions of all the minor codes used in omniORB, covering codes allocated in the CORBA specification, and ones specific to omniORB. The minor code constants appear in namespace `omni` unless you have a truly ancient C++ compiler with no namespace support, in which case they are in the global scope.

Applications can use minor codes to adjust their behaviour according to the condition, e.g.

```
try {
  ...
}
catch (CORBA::TRANSIENT& ex) {
  if (ex.minor() == omni::TRANSIENT_ConnectFailed) {
    // retry with a different object reference...
  }
  else {
```

```
      // print an error message...
    }
  }
```

4.7.2 CORBA::TRANSIENT handlers

TRANSIENT exceptions can occur in many circumstances. One circumstance is as follows:

1. The client invokes on an object reference.
2. The object replies with a LOCATION_FORWARD message.
3. The client caches the new location and retries to the new location.
4. Time passes...
5. The client tries to invoke on the object again, using the cached, forwarded location.
6. The attempt to contact the object fails.
7. The ORB runtime resets the location cache and throws a TRANSIENT exception with minor code TRANSIENT_FailedOnForwarded.

In this situation, the default TRANSIENT exception handler retries the call, using the object's original location. If the retry results in another LOCATION_FORWARD, to the same or a different location, and *that* forwarded location fails immediately, the TRANSIENT exception will occur again, and the pattern will repeat. With repeated exceptions, the handler starts adding delays before retries, with exponential back-off.

In all other circumstances, the default TRANSIENT handler just passes the exception on to the caller.

Applications can override the default behaviour by installing their own exception handler. The API to do so is summarised below:

```
namespace omniORB {

  typedef CORBA::Boolean
  (*transientExceptionHandler_t)(void* cookie,
                                 CORBA::ULong n_retries,
                                 const CORBA::TRANSIENT& ex);

  void
  installTransientExceptionHandler(void* cookie,
                                   transientExceptionHandler_t fn);
```

```
  void
  installTransientExceptionHandler(CORBA::Object_ptr obj,
                                   void* cookie,
                                   transientExceptionHandler_t fn);
}
```

The overloaded `installTransientExceptionHandler()` function is used to install the exception handlers for `CORBA::TRANSIENT`. Two forms are available: the first form installs an exception handler for all object references except for those which have an exception handler installed by the second form, which takes an additional argument to identify the target object reference. The argument `cookie` is an opaque pointer which will be passed on by the ORB when it calls the exception handler.

An exception handler will be called by proxy objects with three arguments. The `cookie` is the opaque pointer registered by `installTransientExceptionHandler()`. The argument `n_retries` is the number of times the proxy has called this handler for the same invocation. The argument `ex` is the value of the exception caught. The exception handler is expected to do whatever is appropriate and return a boolean value. If the return value is `true`, the proxy object retries the operation. If the return value is `false`, the original exception is propagated into the application code. In the case of a `TRANSIENT` exception due to a failed location forward, the exception propagated to the application is the *original* exception that caused the `TRANSIENT` (e.g. a `COMM_FAILURE` or `OBJECT_NOT_EXIST`), rather than the `TRANSIENT` exception[1].

The following sample code installs a simple exception handler for all objects and for a specific object:

```
CORBA::Boolean my_transient_handler1(void* cookie,
                                     CORBA::ULong retries,
                                     const CORBA::TRANSIENT& ex)
{
   cerr << "transient handler 1 called." << endl;
   return true;           // retry immediately.
}

CORBA::Boolean my_transient_handler2(void* cookie,
                                     CORBA::ULong retries,
                                     const CORBA::TRANSIENT& ex)
{
   cerr << "transient handler 2 called." << endl;
   return false;          // do not retry.
}
```

[1] This is different from omniORB 4.0 and earlier, where it was the `TRANSIENT` exception that was propagated to the application.

```
static Echo_ptr myobj;

void installhandlers()
{
   omniORB::installTransientExceptionHandler(0, my_transient_handler1);
   // All proxy objects will call my_transient_handler1 from now on.

   omniORB::installTransientExceptionHandler(myobj, 0, my_transient_handler2);
   // The proxy object of myobj will call my_transient_handler2 from now on.
}
```

4.7.3 CORBA::TIMEOUT

When a call timeout occurs, by default the ORB throws `CORBA::TIMEOUT`. The default behaviour of the proxy objects is to propagate this exception to the application. Applications can override the default behaviour by installing their own exception handlers. The API to do so is summarised below:

```
typedef CORBA::Boolean
(*timeoutExceptionHandler_t)(void* cookie,
                             CORBA::ULong n_retries,
                             const CORBA::TIMEOUT& ex);

void
installTimeoutExceptionHandler(void* cookie,
                               timeoutExceptionHandler_t fn);

void
installTimeoutExceptionHandler(CORBA::Object_ptr obj,
                               void* cookie,
                               timeoutExceptionHandler_t fn);
```

The functions are equivalent to their counterparts for `CORBA::TRANSIENT`.

omniORB version 4.1 and earlier did not have the `CORBA::TIMEOUT` exception, and threw `CORBA::TRANSIENT` instead. If the `throwTransientOnTimeOut` configuration parameter is set to `1`, omniORB 4.3 reverts to this behaviour, and calls the transient exception handler instead of the timeout exception handler.

The timeout exception handler is used when a CORBA call times out. It is *not* called when an AMI poller operation throws `CORBA::TIMEOUT`. In that situation, the exception is always propagated to the caller.

4.7.4 CORBA::COMM_FAILURE

If the ORB has successfully contacted a server at some point, and access to it subsequently fails (and the condition for TRANSIENT described above does not occur), the ORB raises a CORBA::COMM_FAILURE exception.

The default behaviour of the proxy objects is to propagate this exception to the application. Applications can override the default behaviour by installing their own exception handlers. The API to do so is summarised below:

```
typedef CORBA::Boolean
(*commFailureExceptionHandler_t)(void* cookie,
                                 CORBA::ULong n_retries,
                                 const CORBA::COMM_FAILURE& ex);

void
installCommFailureExceptionHandler(void* cookie,
                                   commFailureExceptionHandler_t fn);

void
installCommFailureExceptionHandler(CORBA::Object_ptr obj,
                                   void* cookie,
                                   commFailureExceptionHandler_t fn);
```

The functions are equivalent to their counterparts for CORBA::TRANSIENT.

4.7.5 CORBA::SystemException

If a system exceptions other than TRANSIENT, TIMEOUT or COMM_FAILURE occurs, the default behaviour of the proxy objects is to propagate this exception to the application. Applications can override the default behaviour by installing their own exception handlers. The API to do so is summarised below:

```
typedef CORBA::Boolean
(*systemExceptionHandler_t)(void* cookie,
                            CORBA::ULong n_retries,
                            const CORBA::SystemException& ex);

void
installSystemExceptionHandler(void* cookie,
                              systemExceptionHandler_t fn);

void
installSystemExceptionHandler(CORBA::Object_ptr obj,
                              void* cookie,
                              systemExceptionHandler_t fn);
```

The functions are equivalent to their counterparts for CORBA::TRANSIENT.

4.7.6 Extended exception handlers

New in omniORB 4.2, each of the exception handlers described above also has an 'extended' form in which the exception handler takes two additional parameters, the object reference being invoked upon, and a string containing the name of the operation invoked. e.g.:

```
namespace omniORB {

  typedef CORBA::Boolean
  (*transientExceptionHandlerExt_t)(void*                   cookie,
                                    CORBA::ULong            n_retries,
                                    const CORBA::TRANSIENT& ex,
                                    CORBA::Object_ptr       obj,
                                    const char*             op);

  void
  installTransientExceptionHandlerExt(void* cookie,
                                      transientExceptionHandlerExt_t fn);

  void
  installTransientExceptionHandlerExt(CORBA::Object_ptr obj,
                                      void* cookie,
                                      transientExceptionHandlerExt_t fn);
}
```

Note that the operation parameter can sometimes be null. By default, omniORB sends a `LocateRequest` message prior to the first operation invocation on an object reference. That `LocateRequest` is subject to the same exception handling mechanism as a normal operation invocation, but it is represented with a null operation name. Exception handler code that uses the operation name must correctly handle a null operation name pointer.

4.8 Location forwarding

Any CORBA operation invocation can return a `LOCATION_FORWARD` message to the caller, indicating that it should retry the invocation on a new object reference. The standard allows ServantManagers to trigger `LOCATION_FORWARD`s by raising the `PortableServer::ForwardRequest` exception, but it does not provide a similar mechanism for normal servants. omniORB provides the `omniORB::LOCATION_FORWARD` exception for this purpose. It can be thrown by any operation implementation.

```
namespace omniORB {
  class LOCATION_FORWARD {
  public:
```

```
      LOCATION_FORWARD(CORBA::Object_ptr objref);
    };
  };
```

The exception object consumes the object reference it is passed.

Chapter 5

The IDL compiler

omniORB's IDL compiler is called omniidl. It consists of a generic front-end parser written in C++, and a number of back-ends written in Python. omniidl is very strict about IDL validity, so you may find that it reports errors in IDL which compiles fine with other IDL compilers.

The general form of an omniidl command line is:

```
omniidl [options] -b<back-end> [back-end options] <file>
```

5.1 Common options

The following options are common to all back-ends:

`-b`*back-end*	Run the specified back-end. For the C++ ORB, use `-bcxx`.
`-D`*name*[`=`*value*]	Define *name* for the preprocessor.
`-U`*name*	Undefine *name* for the preprocessor.
`-I`*dir*	Include *dir* in the preprocessor search path.
`-E`	Only run the preprocessor, sending its output to stdout.
`-Y`*cmd*	Use *cmd* as the preprocessor, rather than the normal C preprocessor.
`-N`	Do not run the preprocessor.
`-T`	Use a temporary file, not a pipe, for preprocessor output.
`-Wp`*arg*[`,`*arg*`...`]	Send arguments to the preprocessor.
`-Wb`*arg*[`,`*arg*`...`]	Send arguments to the back-end.
`-nf`	Do not warn about unresolved forward declarations.
`-k`	Keep comments after declarations, to be used by some back-ends.
`-K`	Keep comments before declarations, to be used by some back-ends.
`-C`*dir*	Change directory to *dir* before writing output files.
`-d`	Dump the parsed IDL then exit, without running a back-end.
`-p`*dir*	Use *dir* as a path to find omniidl back-ends.
`-V`	Print version information then exit.

`-u`	Print usage information.
`-v`	Verbose: trace compilation stages.

Most of these options are self explanatory, but some are not so obvious.

5.1.1 Preprocessor interactions

IDL is processed by the C preprocessor before omniidl parses it. omniidl always uses the GNU C preprocessor (which it builds with the name omnicpp). The `-D`, `-U`, and `-I` options are just sent to the preprocessor. Note that the current directory is not on the include search path by default—use '`-I.`' for that. The `-Y` option can be used to specify a different preprocessor to omnicpp. Beware that line directives inserted by other preprocessors are likely to confuse omniidl.

5.1.2 Forward-declared interfaces

If you have an IDL file like:

```
interface I;
interface J {
  attribute I the_I;
};
```

then omniidl will normally issue a warning:

```
test.idl:1: Warning: Forward declared interface 'I' was never
fully defined
```

It is illegal to declare such IDL in isolation, but it *is* valid to define interface `I` in a separate file. If you have a lot of IDL with this sort of construct, you will drown under the warning messages. Use the `-nf` option to suppress them.

5.1.3 Comments

By default, omniidl discards comments in the input IDL. However, with the `-k` and `-K` options, it preserves the comments for use by the back-ends. The C++ back-end ignores this information, but it is relatively easy to write new back-ends which *do* make use of comments.

The two different options relate to how comments are attached to declarations within the IDL. Given IDL like:

```
interface I {
  void op1();
  // A comment
  void op2();
};
```

the `-k` flag will attach the comment to `op1()`; the `-K` flag will attach it to `op2()`.

5.2 C++ back-end options

When you specify the C++ back-end (with `-bcxx`), the following `-Wb` options are available. Note that the `-Wb` options must be specified *after* the `-bcxx` option, so omniidl knows which back-end to give the arguments to.

`-Wbh=`*suffix*	Use *suffix* for generated header files. Default '`.hh`'.
`-Wbs=`*suffix*	Use *suffix* for generated stub files. Default '`SK.cc`.'
`-Wbd=`*suffix*	Use *suffix* for generated dynamic files. Default '`DynSK.cc`.'
`-Wba`	Generate stubs for TypeCode and Any.
`-Wbtp`	Generate 'tie' implementation skeletons.
`-Wbtf`	Generate flattened 'tie' implementation skeletons.
`-Wbami`	Generate AMI types and operations.
`-Wbexample`	Generate example implementation code.
`-Wbinline`	Output stubs for `#included` IDL files in line with the main file.
`-Wbuse-quotes`	Use quotes in '`#include`' directives (e.g. `"foo"` rather than `<foo>`.)
`-Wbkeep-inc-path`	Preserve IDL '`#include`' paths in generated '`#include`' directives.
`-Wbvirtual-objref`	Use virtual functions for object reference operations.
`-Wbimpl-mapping`	Use the 'implementation' mapping for object reference methods.
`-Wbsplice-modules`	Splice together multiply-opened modules into one.
`-WbBOA`	Generate BOA compatible skeletons.
`-Wbold`	Generate old CORBA 2.1 signatures for skeletons.
`-Wbold-prefix`	Map C++ reserved words with prefix '_' rather than '`_cxx_`'.
`-WbF`	Generate code fragments (only for use during omniORB build).

5.2.1 Optional code generation options

By default, omniidl generates the minimum code required to provide all the IDL-defined types and interfaces, which is sufficient for the majority of applications. Additional code can also be generated, for various purposes:

5.2.1.1 Any and TypeCode

To generate TypeCodes and Any insertion operators, give the `-Wba` option. See chapter 11 for details.

By default, omniidl separates the normal stub and skeleton file (the `SK.cc` file) from these 'dynamic' stubs (the `DynSK.cc` file), so applications that do not need support for Any and TypeCode for a particular IDL file do not waste space with unnecessary definitions. It is possible to output both the normal stubs and

the dynamic stubs to a single file, by simply specifying the same extension for both files. This command places both the normal stubs and the dynamic stubs in `aSK.cc`:

```
omniidl -bcxx -Wba -Wbd=SK.cc a.idl
```

5.2.1.2 Tie templates

As described in section 2.11, tie templates can be used to provide servant implementations, instead of using inheritance from the normal skeleton classes. To generate tie templates, give the `-Wbtp` option to omniidl.

When using a pre-namespace C++ compiler, IDL modules are mapped to C++ classes, which causes a problem with tie templates. The C++ mapping says that for the interface `M::I`, the C++ tie template class should be named `POA_M::I_tie`. However, since template classes cannot be declared inside other classes, this naming scheme cannot be used if `POA_M` is a class.

The C++ mapping has an alternative option of 'flattened' tie class names, in which the template class is declared at global scope with the name `POA_M_I_tie`. i.e. all occurrences of '`::`' are replaced by '`_`'. Generate the flattened ties using the `-Wbtf` command line argument.

5.2.1.3 Asynchronous Method Invocation

Generate asynchronous invocation operations and the various types required by AMI by specifying `-Wbami`. See chapter 13 for details.

5.2.1.4 Example implementations

If you use the `-Wbexample` flag, omniidl will generate an example implementation file as well as the stubs and skeletons. For IDL file `foo.idl`, the example code is written to `foo_i.cc`. The example file contains class and method declarations for the operations of all interfaces in the IDL file, along with a `main()` function which creates an instance of each object. You still have to fill in the operation implementations, of course.

5.2.2 Include file options

IDL files regularly `#include` other files. By default, if file `a.idl` says `#include <b/c.idl>` then the generated header `a.hh` has an include of the form `#include <c.idl>`, and `aSK.cc` and `aDynSK.cc` contain only code corresponding to the declarations in `a.idl`.

If the `-Wbinline` option is provided, all the `#included` declarations are generated in `a.hh`, `aSK.cc` and `aDynSK.cc`, meaning the application code should only use that single set of files.

If `-Wbuse-quotes` is specified, then the directive in `a.hh` uses quotes rather than angle brackets: `#include "c.hh"`.

Normally any path details contained in the IDL `#include` directive are removed, leaving just the base name. If `-Wbkeep-inc-path` is specified, the directive in `a.hh` is `#include <b/c.hh>`.

5.2.3 Object reference operations

Some of the C++ mapping's parameter passing rules are problematic in terms of memory management. For example, if an IDL operation has a parameter of type `inout string`, the standard mapping has a C++ parameter of type `char*&`. If application code passes a `String_var` for the parameter, some C++ compilers choose the wrong conversion operator and cause a violation of the memory management rules[1].

To avoid this, omniORB uses some helper classes as the parameter types in object reference operations, meaning that the correct memory management rules are always followed. Normally, that is invisible to application code, but occasionally it becomes problematic. One example is that if a `local interface` is derived from a normal unconstrained interface, the C++ mapping of the local interface derives from the object reference class, and so the base object reference class must use the standard mapping rather than omniORB's usual enhanced mapping. To choose the standard 'implementation mapping', give the `-Wbimpl-mapping` option to omniidl.

Similarly, omniidl usually uses non-virtual methods in its object reference classes, since there is no usual need to override them. The local interface situation also requires method overrides, so omniidl must be instructed to generate object references as virtual. Use `-Wbvirtual-objref` to achieve this.

More details about the local interface mapping can be found in section 12.8.

5.2.4 Module splicing

On ancient C++ compilers without namespace support, IDL modules map to C++ classes, and so cannot be reopened. For some IDL, it is possible to 'splice' reopened modules on to the first occurrence of the module, so all module definitions are in a single class. It is possible in this sort of situation:

```
module M1 {
  interface I {};
```

[1] For this reason, the `_var` types define an `inout()` method that ensures use of the correct conversion and thus avoids this kind of trouble.

```
};
module M2 {
  interface J {
    attribute M1::I ok;
  };
};
module M1 {
  interface K {
    attribute I still_ok;
  };
};
```

but not if there are cross-module dependencies:

```
module M1 {
  interface I {};
};
module M2 {
  interface J {
    attribute M1::I ok;
  };
};
module M1 {
  interface K {
    attribute M2::J oh_dear;
  };
};
```

In both of these cases, the `-Wbsplice-modules` option causes omniidl to put all of the definitions for module `M1` into a single C++ class. For the first case, this will work fine. For the second case, class `M1::K` will contain a reference to `M2::J`, which has not yet been defined; the C++ compiler will complain.

5.3 Examples

Generate the C++ headers and stubs for a file `a.idl`:

```
omniidl -bcxx a.idl
```

Generate with Any support:

```
omniidl -bcxx -Wba a.idl
```

As above, but also generate Python stubs (assuming omniORBpy is installed):

```
omniidl -bcxx -Wba -bpython a.idl
```

Just check the IDL files for validity, generating no output:

```
omniidl a.idl b.idl
```

Chapter 6

Connection and Thread Management

This chapter describes how omniORB manages threads and network connections.

6.1 Background

In CORBA, the ORB is the 'middleware' that allows a client to invoke an operation on an object without regard to its implementation or location. In order to invoke an operation on an object, a client needs to 'bind' to the object by acquiring its object reference. Such a reference may be obtained as the result of an operation on another object (such as a naming service or factory object) or by conversion from a stringified representation. If the object is in a different address space, the binding process involves the ORB building a proxy object in the client's address space. The ORB arranges for invocations on the proxy object to be transparently mapped to equivalent invocations on the implementation object.

For the sake of interoperability, CORBA mandates that all ORBs should support IIOP as the means to communicate remote invocations over a TCP/IP connection. IIOP is usually[1] asymmetric with respect to the roles of the parties at the two ends of a connection. At one end is the client which can only initiate remote invocations. At the other end is the server which can only receive remote invocations.

Notice that in CORBA, as in most distributed systems, remote bindings are established implicitly without application intervention. This provides the illusion that all objects are local, a property known as 'location transparency'. CORBA does not specify when such bindings should be established or how they should

[1] GIOP 1.2 supports 'bidirectional GIOP', which permits the rôles to be reversed.

be multiplexed over the underlying network connections. Instead, ORBs are free to implement implicit binding by a variety of means.

The rest of this chapter describes how omniORB manages network connections and the programming interface to fine tune the management policy.

6.2 The model

omniORB is designed from the ground up to be fully multi-threaded. The objective is to maximise the degree of concurrency and at the same time eliminate any unnecessary thread overhead. Another objective is to minimise the interference by the activities of other threads on the progress of a remote invocation. In other words, thread 'cross-talk' should be minimised within the ORB. To achieve these objectives, the degree of multiplexing at every level is kept to a minimum by default.

Minimising multiplexing works well when the system is relatively lightly loaded. However, when the ORB is under heavy load, it can sometimes be beneficial to conserve operating system resources such as threads and network connections by multiplexing at the ORB level. omniORB has various options that control its multiplexing behaviour.

6.3 Client side behaviour

On the client side of a connection, the thread that invokes on a proxy object drives the GIOP protocol directly and blocks on the connection to receive the reply. The first time the client makes a call to a particular address space, the ORB opens a suitable connection to the remote address space (based on the client transport rule as described in section 7.5). After the reply has been received, the ORB caches the open network connection, ready for use by another call.

If two (or more) threads in a multi-threaded client attempt to contact the same address space simultaneously, there are two different ways to proceed. The default way is to open another network connection to the server. This means that neither the client or server ORB has to perform any multiplexing on the network connections—multiplexing is performed by the operating system, which has to deal with multiplexing anyway. The second possibility is for the client to multiplex the concurrent requests on a single network connection. This conserves operating system resources (network connections), but means that both the client and server have to deal with multiplexing issues themselves.

In the default one call per connection mode, there is a limit to the number of concurrent connections that are opened, set with the `maxGIOPConnectionPerServer` parameter. To tell the ORB that it may multiplex calls on a single con-

nection, set the `oneCallPerConnection` parameter to zero. If the `oneCallPerConnection` parameter is set to the default value of one, and there are more concurrent calls than specified by `maxGIOPConnectionPerServer`, calls block waiting for connections to become free.

Note that some server-side ORBs, including omniORB versions before version 4.0, are unable to deal with concurrent calls multiplexed on a single connection, so they serialise the calls. It is usually best to keep to the default mode of opening multiple connections.

6.3.1 Client side timeouts

omniORB can associate a timeout with a call, meaning that if the call takes too long a `CORBA::TIMEOUT` exception[2] is thrown. Timeouts can be set for the whole process, for a specific thread, or for a specific object reference.

Timeouts are set using this API:

```
namespace omniORB {
  void setClientCallTimeout(CORBA::ULong millisecs);
  void setClientCallTimeout(CORBA::Object_ptr obj, CORBA::ULong millisecs);
  void setClientThreadCallTimeout(CORBA::ULong millisecs);
  void setClientThreadCallDeadline(const omni_time_t& tt);
  void setClientConnectTimeout(CORBA::ULong millisecs);
};
```

`setClientCallTimeout()` sets either the global timeout or the timeout for a specific object reference.

`setClientThreadCallTimeout()` sets the timeout for the calling thread. Alternatively, `setClientThreadCallDeadline()` sets an absolute deadline for calls on thread. The calling thread must have an `omni_thread` associated with it.

Accessing per-thread state is a relatively expensive operation, so per thread timeouts are disabled by default. The `supportPerThreadTimeOut` parameter must be set `true` to enable them.

Setting any timeout value to zero disables it.

To choose the timeout value to use for a call, the ORB first looks to see if there is a timeout for the object reference, then to the calling thread, and finally to the global timeout.

When a client has no existing connection to communicate with a server, it must open a new connection before performing the call. `setClientConnectTimeout()` sets an overriding timeout for cases where a new connection must be established. The effect of the connect timeout depends upon whether the connect timeout is greater or less than the timeout that would otherwise be used.

[2] Or `CORBA::TRANSIENT` if the backwards-compatibility `throwTransientOnTimeOut` parameter is set to 1.

As an example, imagine that the usual call timeout is 10 seconds:

Connect timeout > usual timeout

If the connect timeout is set to 20 seconds, then a call that establishes a new connection will be permitted 20 seconds before it times out. Subsequent calls using the same connection have the normal 10 second timeout. If establishing the connection takes 8 seconds, then the call itself takes 5 seconds, the call succeeds despite having taken 13 seconds in total, longer than the usual timeout.

This kind of configuration is good when connections are slow to be established.

If an object reference has multiple possible endpoints available, and connecting to the first endpoint times out, only that one endpoint will have been tried before an exception is raised. However, once the timeout has occurred, the object reference will switch to use the next endpoint. If the application attempts to make another call, it will use the next endpoint.

Connect timeout < usual timeout

If the connect timeout is set to 2 seconds, the actual network-level connect is only permitted to take 2 seconds. As long as the connection is established in less than 2 seconds, the call can proceed. The 10 second call timeout still applies to the time taken for the whole call (including the connection establishment). So, if establishing the connection takes 1.5 seconds, and the call itself takes 9.5 seconds, the call will time out because although it met the connection timeout, it exceeded the 10 second total call timeout. On the other hand, if establishing the connection takes 3 seconds, the call will fail after only 2 seconds, since only 2 seconds are permitted for the connect.

If an object reference has multiple possible endpoints available, the client will attempt to connect to them in turn, until one succeeds. The connect timeout applies to each connection attempt. So with a connect timeout of 2 seconds, the client will spend up to 2 seconds attempting to connect to the first address and then, if that fails, up to 2 seconds trying the second address, and so on. The 10 second timeout still applies to the call as a whole, so if the total time taken on timed-out connection attempts exceeds 10 seconds, the call will time out.

This kind of configuration is useful where calls may take a long time to complete (so call timeouts are long), but a fast indication of connection failure is required.

6.4 Server side behaviour

The server side has two primary modes of operation: thread per connection and thread pooling. It is able to dynamically transition between the two modes, and it supports a hybrid scheme that behaves mostly like thread pooling, but has the same fast turn-around for sequences of calls as thread per connection.

6.4.1 Thread per connection mode

In thread per connection mode (the default, and the only option in omniORB versions before 4.0), each connection has a single thread dedicated to it. The thread blocks waiting for a request. When it receives one, it unmarshals the arguments, makes the up-call to the application code, marshals the reply, and goes back to watching the connection. There is thus no thread switching along the call chain, meaning the call is very efficient.

As explained above, a client can choose to multiplex multiple concurrent calls on a single connection, so once the server has received the request, and just before it makes the call into application code, it marks the connection as 'selectable', meaning that another thread should watch it to see if any other requests arrive. If they do, extra threads are dispatched to handle the concurrent calls. GIOP 1.2 actually allows the argument data for multiple calls to be interleaved on a connection, so the unmarshalling code has to handle that too. As soon as any multiplexing occurs on the connection, the aim of removing thread switching cannot be met, and there is inevitable inefficiency due to thread switching.

The `maxServerThreadPerConnection` parameter can be set to limit the number of threads that can be allocated to a single connection containing concurrent calls. Setting the parameter to 1 mimics the behaviour of omniORB versions before 4.0, that did not support calls multiplexed on one connection.

6.4.2 Thread pool mode

In thread pool mode, selected by setting the `threadPerConnectionPolicy` parameter to zero, a single thread watches all incoming connections. When a call arrives on one of them, a thread is chosen from a pool of threads, and set to work unmarshalling the arguments and performing the up-call. There is therefore at least one thread switch for each call.

The thread pool is not pre-initialised. Instead, threads are started on demand, and idle threads are stopped after a period of inactivity. The maximum number of threads that can be started in the pool is set with the `maxServer ThreadPoolSize` parameter. The default is 100.

A common pattern in CORBA applications is for a client to make several calls to a single object in quick succession. To handle this situation most efficiently, the default behaviour is to not return a thread to the pool immediately after a call is finished. Instead, it is set to watch the connection it has just served for a short while, mimicking the behaviour in thread per connection mode. If a new call comes in during the watching period, the call is dispatched without any thread switching, just as in thread per connection mode. Of course, if the server is supporting a very large number of connections (more than the size of the thread pool), this policy can delay a call coming from another connection. If the `threadPoolWatchConnection` parameter is set to zero, connection watching is disabled and threads return to the pool immediately after finishing a single request.

In the face of multiplexed calls on a single connection, multiple threads from the pool can be dispatched for one connection, just as in thread per connection mode. With `threadPoolWatchConnection` set to the default value of 1, only the last thread servicing a connection will watch it when it finishes a request. Setting the parameter to a larger number allows the last *n* connections to watch the connection.

6.4.3 Policy transition

If the server is dealing with a relatively small number of connections, it is most efficient to use thread per connection mode. If the number of connections becomes too large, however, operating system limits on the number of threads may cause a significant slowdown, or even prevent the acceptance of new connections altogether.

To give the most efficient response in all circumstances, omniORB allows a server to start in thread per connection mode, and transition to thread pooling if many connections arrive. This is controlled with the `threadPerConnectionUpperLimit` and `threadPerConnectionLowerLimit` parameters. The upper limit must always be larger than the lower limit. The upper limit chooses the number of connections at which time the ORB transitions to thread pool mode; the lower limit selects the point at which the transition back to thread per connection is made.

For example, setting the upper limit to 50 and the lower limit to 30 would mean that the first 49 connections would receive dedicated threads. The 50th to arrive would trigger thread pooling. All future connections to arrive would make use of threads from the pool. Note that the existing dedicated threads continue to service their connections until the connections are closed. If the number of connections falls below 30, thread per connection is reactivated and new connections receive their own dedicated threads (up to the limit of 50 again). Once again, existing connections in thread pool mode stay in that mode until

they are closed.

6.5 Idle connection shutdown

It is wasteful to leave a connection open when it has been left unused for a considerable time. Too many idle connections could block out new connections when the system runs out of spare communication channels. For example, most platforms have a limit on the number of file handles a process can open. Many platforms have a very small default limit like 64. The value can often be increased to a maximum of a thousand or more by changing the 'ulimit' in the shell.

Every so often, a thread scans all open connections to see which are idle. The scanning period (in seconds) is set with the `scanGranularity` parameter. The default is 5 seconds.

Outgoing connections (initiated by clients) and incoming connections (initiated by servers) have separate idle timeouts. The timeouts are set with the `outConScanPeriod` and `inConScanPeriod` parameters respectively. The values are in seconds, and must be a multiple of the scan granularity.

Beware that setting `outConScanPeriod` or `inConScanPeriod` to be equal to (or less than) `scanGranularity` means that connections are considered candidates for closure immediately after they are opened. That can mean that the connections are closed before any calls have been sent through them. If oneway calls are used, such connection closure can result in silent loss of calls.

6.5.1 Interoperability Considerations

The IIOP specification allows both the client and the server to shutdown a connection unilaterally. When one end is about to shutdown a connection, it should send a `CloseConnection` message to the other end. It should also make sure that the message will reach the other end before it proceeds to shutdown the connection.

The client should distinguish between an orderly and an abnormal connection shutdown. When a client receives a `CloseConnection` message before the connection is closed, the condition is an orderly shutdown. If the message is not received, the condition is an abnormal shutdown. In an abnormal shutdown, the ORB should raise a `COMM_FAILURE` exception whereas in an orderly shutdown, the ORB should *not* raise an exception and should try to re-establish a new connection transparently.

omniORB implements these semantics completely. However, it is known that some ORBs are not (yet) able to distinguish between an orderly and an abnormal shutdown. Usually this is manifested as the client in these ORBs seeing a

COMM_FAILURE occasionally when connected to an omniORB server. The workaround is either to catch the exception in the application code and retry, or to turn off the idle connection shutdown inside the omniORB server.

Chapter 7

Transports and Endpoints

omniORB can support multiple network transports. All platforms (usually) have a TCP transport available. Unix platforms support a Unix domain socket transport. Platforms with the OpenSSL library available can support a TLS/SSL and HTTP(s) transport.

Servers must be configured in two ways with regard to transports: the transports and interfaces on which they listen, and the details that are published in IORs for clients to see. Usually the published details will be the same as the listening details, but there are times when it is useful to publish different information.

Details are selected with the `endPoint` family of parameters. The simplest is plain `endPoint`, which chooses a transport and interface details, and publishes the information in IORs. Endpoint parameters are in the form of URIs, with a scheme name of '`giop:`', followed by the transport name. Different transports have different parameters following the transport.

TCP endpoints have the format:

`giop:tcp:`<*host*>`:`<*port*>

The host must be a valid host name or IP address for the server machine. It determines the network interface on which the server listens. The port selects the TCP port to listen on, which must be unoccupied. Either the host or port, or both can be left empty. If the host is empty, the ORB publishes the IP address of the first non-loopback network interface it can find (or the loopback if that is the only interface), but listens on *all* network interfaces. If the port is empty, the operating system chooses an *ephemeral* port.

Multiple TCP endpoints can be selected, either to specify multiple network interfaces on which to listen, or (less usefully) to select multiple TCP ports on which to listen.

If no `endPoint` parameters are set, the ORB assumes a single parameter of `giop:tcp::`, meaning IORs contain the address of the first non-loopback net-

work interface, the ORB listens on all interfaces, and the OS chooses a port number.

TLS/SSL endpoints have the same format as TCP ones, except 'tcp' is replaced with 'ssl'. Unix domain socket endpoints have the format:

```
giop:unix:<filename>
```

where the filename is the name of the socket within the filesystem. If the filename is left blank, the ORB chooses a name based on the process id and a timestamp.

To listen on an endpoint without publishing it in IORs, specify it with the endPointNoPublish configuration parameter. See below for more details about endpoint publishing.

7.1 Port ranges

Sometimes it is useful to restrict a server to listen on one of a range of ports, rather than pinning it to one particular port or allowing the OS to choose an ephemeral port. omniORB 4.2 introduced the ability to specify a range of ports using a hyphen. e.g. to listen on a port between 5000 and 5010 inclusive:

```
giop:tcp::5000-5010
```

omniORB randomly chooses a port in the range. If it finds that the chosen port is already occupied, it keeps trying different ports until it finds a free one. If all the ports in the range are occupied, it throws CORBA::INITIALIZE.

7.2 IPv6

On platforms where it is available, omniORB supports IPv6. On most Unix platforms, IPv6 sockets accept both IPv6 and IPv4 connections, so omniORB's default giop:tcp:: endpoint accepts both IPv4 and IPv6 connections. On Windows versions before Windows Vista, each socket type only accepts incoming connections of the same type, so an IPv6 socket cannot be used with IPv4 clients. For this reason, the default giop:tcp:: endpoint only listens for IPv4 connections. Since endpoints with a specific host name or address only listen on a single network interface, they are inherently limited to just one protocol family.

To explicitly ask for just IPv4 or just IPv6, an endpoint with the wildcard address for the protocol family should be used. For IPv4, the wildcard address is '0.0.0.0', and for IPv6 it is '::'. So, to listen for IPv4 connections on all IPv4 network interfaces, use an endpoint of:

```
giop:tcp:0.0.0.0:
```

All IPv6 addresses contain colons, so the address portion in URIs must be contained within `[]` characters. Therefore, to listen just for IPv6 connections on all IPv6 interfaces, use the somewhat cryptic:

```
giop:tcp:[::]:
```

To listen for both IPv4 and IPv6 connections on Windows versions prior to Vista, both endpoints must be explicitly provided.

7.2.1 Link local addresses

In IPv6, all network interfaces are assigned a *link local* address, starting with the digits `fe80`. The link local address is only valid on the same 'link' as the interface, meaning directly connected to the interface, or possibly on the same subnet, depending on how the network is switched. To connect to a server's link local address, a client has to know which of its network interfaces is on the same link as the server. Since there is no way for omniORB to know which local interface a remote link local address may be connected to, and in extreme circumstances may even end up contacting the wrong server if it picks the wrong interface, link local addresses are not considered valid. Servers do not publish link local addresses in their IORs.

7.3 Endpoint publishing

For clients to be able to connect to a server, the server publishes endpoint information in its IORs (Interoperable Object References). Normally, omniORB publishes the first available address for each of the endpoints it is listening on.

The endpoint information to publish is determined by the `endPointPublish` configuration parameter. It contains a comma-separated list of publish rules. The rules are applied in turn to each of the configured endpoints; if a rule matches an endpoint, it causes one or more endpoints to be published.

The following core rules are supported:

`addr`	the first natural address of the endpoint
`ipv4`	the first IPv4 address of a TCP or SSL endpoint
`ipv6`	the first IPv6 address of a TCP or SSL endpoint
`name`	the first address that can be resolved to a name
`hostname`	the result of the gethostname() system call
`fqdn`	the fully-qualified domain name

The core rules can be combined using the vertical bar operator to try several rules in turn until one succeeds. e.g:

`name|ipv6|ipv4`	the name of the endpoint if it has one; failing that, its first IPv6 address; failing that, its first IPv4 address.

Multiple rules can be combined using the comma operator to publish more than one endpoint. e.g.

`name,addr`	the name of the endpoint (if it has one), followed by its first address.

For endpoints with multiple addresses (e.g. TCP endpoints on multi-homed machines), the `all()` manipulator causes all addresses to be published. e.g.:

`all(addr)`	all addresses are published
`all(name)`	all addresses that resolve to names are published
`all(name|addr)`	all addresses are published by name if they have one, address otherwise.
`all(name,addr)`	all addresses are published by name (if they have one), and by address.
`all(name), all(addr)`	first the names of all addresses are published, followed by all the addresses.

A specific endpoint can be published by giving its endpoint URI, even if the server is not listening on that endpoint. e.g.:

```
giop:tcp:not.my.host:12345
giop:unix:/not/my/socket-file
```

If the host or port number for a TCP or SSL URI are missed out, they are filled in with the details from each listening TCP/SSL endpoint. This can be used to publish a different name for a TCP/SSL endpoint that is using an ephemeral port, for example.

omniORB 4.0 supported two options related to endpoint publishing that are superseded by the `endPointPublish` parameter, and so are now deprecated. Setting `endPointPublishAllIFs` to 1 is equivalent to setting `endPointPublish` to '`all(addr)`'. The `endPointNoListen` parameter is equivalent to adding endpoint URIs to the `endPointPublish` parameter.

7.3.1 POA-specific endpoint publishing

It can sometimes to useful to publish different sets of endpoints for different objects in the same server. To enable that, you can create a POA with the omniORB-specific EndpointPublishPolicy, giving a sequence of endpoint strings:

```
EndPointPublishPolicyValue eps;
eps.length(2);
eps[0] = "giop:tcp:hostname1:10000";
eps[1] = "giop:tcp:hostname2:20000";

policy = omniPolicy::create_endpoint_publish_policy(eps);
// ... use the policy in a call to create_POA()
```

The endpoints must be fully specified—unlike the global publish specifications, an empty port number is *not* filled with the chosen listening port.

Note that the publishing policy only affects the *published* endpoints—it does not change the global endpoints that the ORB is listening on to receive incoming connections.

7.4 Connection selection and acceptance

In the face of IORs containing details about multiple different endpoints, clients have to know how to choose the one to use to connect a server. Similarly, servers may wish to restrict which clients can connect to particular transports. This is achieved with *transport rules*.

7.5 Client transport rules

The `clientTransportRule` parameter is used to filter and prioritise the order in which transports specified in an IOR are tried. Each rule has the form:

<address mask> [action]+

The address mask can be one of

1. `localhost` — The address of this machine
2. *w.x.y.z/m1.m2.m3.m4* — An IPv4 address with bits selected by the mask, e.g. `172.16.0.0/255.240.0.0`
3. *w.x.y.z/prefixlen* — An IPv4 address with *prefixlen* significant bits, e.g. `172.16.2.0/24`
4. *a:b:c:d:e:f:g:h/prefixlen* — An IPv6 address with *prefixlen* significant bits, e.g. `3ffe:505:2:1::/64`
5. * — Wildcard that matches any address

The action is one or more of the following:

1. `none` Do not use this address
2. `tcp` Use a TCP transport
3. `ssl` Use an SSL transport
4. `unix` Use a Unix socket transport
5. `http` Use an HTTP transport
6. `bidir` Connections to this address can be used bidirectionally (see section 7.7)

The transport-selecting actions form a prioritised list, so an action of '`unix,ssl,tcp`' means to use a Unix transport if there is one, failing that an SSL transport, failing *that* a TCP transport. In the absence of any explicit rules, the client uses the implicit rule of '`* unix,ssl,tcp`'.

If more than one rule is specified, they are prioritised in the order they are specified. For example, the configuration file might contain:

```
clientTransportRule = 192.168.1.0/255.255.255.0  unix,tcp
clientTransportRule = 172.16.0.0/255.240.0.0     unix,tcp
                    =       *                    none
```

This would be useful if there is a fast network (192.168.1.0) which should be used in preference to another network (172.16.0.0), and connections to other networks are not permitted at all.

In general, the result of filtering the endpoint specifications in an IOR with the client transport rule will be a prioritised list of transports and networks. (If the transport rules do not prioritise one endpoint over another, the order the endpoints are listed in the IOR is used.) When trying to contact an object, the ORB tries its possible endpoints in turn, until it finds one with which it can contact the object. Only after it has unsuccessfully tried all permissible endpoints will it raise a `TRANSIENT` exception to indicate that the connect failed.

7.6 Server transport rules

Server transport rules have the same format as client transport rules. Rather than being used to select which of a set of ways to contact a machine, they are used to determine whether or not to accept connections from particular clients. In this example, we only allow connections from our intranet:

```
serverTransportRule = localhost                  unix,tcp,ssl
                    = 172.16.0.0/255.240.0.0     tcp,ssl
                    = *                          none
```

And in this one, we accept only SSL connections if the client is not on the intranet:

```
serverTransportRule = localhost                  unix,tcp,ssl
                    = 172.16.0.0/255.240.0.0     tcp,ssl
                    = *                          ssl,bidir
```

In the absence of any explicit rules, the server uses the implicit rule of '`* unix,ssl,tcp`', meaning any kind of connection (other than HTTP) is accepted from any client.

7.7 Bidirectional GIOP

omniORB supports bidirectional GIOP, which allows callbacks to be made using a connection opened by the original client, rather than the normal model where the server opens a new connection for the callback. This is important for negotiating firewalls, since they tend not to allow connections back on arbitrary ports.

There are several steps required for bidirectional GIOP to be enabled for a callback. Both the client and server must be configured correctly. On the client side, these conditions must be met:

- The `offerBiDirectionalGIOP` parameter must be set to `true`.
- The client transport rule for the target server must contain the `bidir` action.
- The POA containing the callback object (or objects) must have been created with a `BidirectionalPolicy` value of `BOTH`.

On the server side, these conditions must be met:

- The `acceptBiDirectionalGIOP` parameter must be set to `true`.
- The server transport rule for the requesting client must contain the `bidir` action.
- The POA hosting the object contacted by the client must have been created with a `BidirectionalPolicy` value of `BOTH`.

7.8 TLS / SSL transport

omniORB supports a TLS / SSL transport, using OpenSSL. It is only built if OpenSSL is available. On platforms using Autoconf, it is autodetected in many locations, or its location can be given with the `--with-openssl=` argument to `configure`. On other platforms, the `OPEN_SSL_ROOT` make variable must be set in the platform file.

To use the SSL transport, you must link your application with the `omnisslTP` library, and correctly set up certificates. See the `src/examples/ssl_echo` directory for an example. That directory contains a `README` file with more details.

7.8.1 Self-signed certificate authority

By default, omniORB configures OpenSSL to require both clients and servers to have certificates that are signed by a Certificate Authority (CA). It is possible to use a public CA to obtain keys that can be independently verified, but for many purposes, it is sufficient to use a private CA to sign all the keys in use in an application. The following is a brief description of how to become your own certificate authority and issue and sign certificates, using the OpenSSL command line tools.

Before starting, find the default `openssl.cnf` file that was installed with OpenSSL, copy it to a suitable location, and edit it as you feel appropriate. Now, build a certificate directory structure, authority key and certificate:

```
mkdir demoCA demoCA/private demoCA/newcerts

openssl req -config openssl.cnf -x509 -newkey rsa:2048 \
   -keyout demoCA/private/cakey.pem -out demoCA/cacert.pem -days 3650

echo 01 >demoCA/serial
touch demoCA/index.txt
```

Next, issue a key request and sign it:

```
openssl req -config openssl.cnf -new -keyout server_key.pem \
   -out server_req.pem -days 3650

openssl ca -config openssl.cnf -policy policy_anything \
   -out server_cert.pem -in server_req.pem
```

Amongst other things, you now have a server key file in `server_key.pem` and a certificate in `server_cert.pem`. To make a single file containing both the key

and the certificate, suitable for use in omniORB, concatenate the key and certificate files together. You can skip the human-readable(ish) text in the certificate file before the `-----BEGIN CERTIFICATE-----` marker.

If need be, create more certificates for servers and clients in the same way.

7.9 HTTP, HTTPS and WebSocket transport

omniORB includes an omniORB-specific HTTP transport. It works by encapsulating GIOP messages into HTTP messages, which can make it easier to transport through firewalls and proxies that understand HTTP but not GIOP. It supports the use of HTTP proxies, reverse proxies and load balancers. It can optionally establish WebSocket connections, which avoid many of the limitations of pure HTTP, but are not quite as widely accepted.

To use the HTTP transport, you must link with the `omnihttpTP` library.

7.9.1 HTTP limitations

Normal client-server CORBA calls match the semantics of HTTP — the client sends a request and blocks waiting for the server to respond, meaning that in simple uses, the communication is pure, valid HTTP, and can therefore safely pass through proxies. However, there are a number of situations in which GIOP semantics do not match HTTP. In these cases, the communication works if an omniORB client is connected directly to an omniORB server, but can fail if a proxy or other system is interpreting the HTTP messages along the communication path:

- If the `oneCallPerConnection` configuration parameter is set to the non-default value of false, a multi-threaded client can send multiple concurrent calls on a single connection, meaning multiple HTTP requests are made without waiting for each to receive a reply.
- If bidirectional GIOP is enabled, the server can send a message to the client that is not in response to a client request.
- Web proxies can choose to open new connections at any time, and therefore the client's idea of the lifetime of a connection does not necessarily match what the server sees. This means that codeset information, which is only transmitted on the first call on a connection, may be missing from the point of view of the server.
- Some web proxies and load balancers routinely close connections after a client has sent a request, without sending the request on to the server. The client therefore sees the connection fail, but cannot know that the

server did not receive the request. It therefore has to fail the call with a `COMM_FAILURE` exception with completion status `COMPLETED_MAYBE`. Application code must be prepared to handle the situation.

To minimise these issues with proxies, it is safest to keep `oneCallPerConnection` set to true, avoid bidirectional GIOP, and make use of the `defaultCharCodeSet` and `defaultWCharCodeSet` parameters to choose suitable codesets.

Alternatively, clients can 'upgrade' the HTTP connection to a WebSockets connection, in which cases messages are sent in a binary framing format that *does* permit the use of full GIOP semantics. Upgrading an HTTP connection to a WebSockets connection requires that all intermediary servers (proxies, load balancers, etc.) support the WebSocket protocol, so there are some environments in which plain HTTP communication succeeds but WebSockets fail.

7.9.2 Endpoint and transport rule specification

The HTTP transport uses endpoint URIs of the form `giop:http:`*url*, where the *url* is a full `http://`, `https://`, `ws://` or `wss://` URL. That means the URIs appear to duplicate the 'http':

```
giop:http:http://server.example.com:8000/call
giop:http:https://secure.example.com:443/invoke
giop:http:ws://websocket.example.com:8080/ws
giop:http:wss://websocket-secure.example.com:8443/ws
```

The path part of the URL can be set to any value. On the client side, the client makes an HTTP `POST` request to the URL, specifying the path; the server checks that the path matches what it expects. The ability to specify a path this way can be helpful when configuring path-based routing in load balancer configurations.

If no port is specified on the server endpoint, the server uses an ephemeral port; it does not default to 80 or 443.

Servers always support upgrade of connections to WebSocket. The use of a `ws://` or `wss://` URL determines the URL that is published in object references, and thus whether clients will attempt to use WebSockets or not.

The transport specification '`http`' is not in the default client or server transport rules (as described in section 7.5). To enable the use of HTTP, the client and server transport rules could for example contain '`* unix,tcp,ssl,http`'.

7.9.3 Client connections

As with all other transports, a client can connect to a server that uses the HTTP transport using an IOR in string form, or an object reference received

from another CORBA call. Clients can *also* connect by giving a suitable URL to `ORB::string_to_object()`.

In this case, the URL should be a simple plain URL, not a `corbaloc` URI. The URL path must match the patch in the server's endpoint specification. The CORBA object key should follow a # character:

```
obj = orb->string_to_object("https://server.host.com/call#obj_key");
```

For this to work, the server should publish objects with plain object keys, as described in section 8.5.

7.9.4 HTTP proxies

Clients can connect via a web proxy. The `httpProxy` configuration can be set to an http or https proxy URL. If the proxy requires basic authentication, the `httpProxyUsername` and `httpProxyPassword` parameters can be set.

When connecting through a proxy, the behaviour depends on whether the client is connecting to an HTTP or HTTPS server, and whether it is attempting to use WebSockets. When connecting with plain HTTP, the request is sent directly to the proxy, and the proxy forwards it at an HTTP level. In this situation, it is essential that only pure HTTP-compliant messages are sent.

When connecting to an HTTPS server or upgrading the connection to WebSockets, the client makes a `CONNECT` request to the proxy. If accepted by the proxy, that opens a transparent tunnel through to the server. In that situation, as long as nothing else intercepts and attempts to interpret the data, the omniORB client is in effect directly connected to the omniORB server, and there is more chance that transfers that are not strictly legal HTTP will succeed.

7.9.5 HTTPS keys and certificates

Support for HTTPS uses the same concepts as the SSL transport. For each configuration option with a name starting '`ssl`', there is an equivalent option starting '`https`'.

7.9.6 In-message encryption

In some environments, the use of HTTPS is not sufficient to guarantee end-to-end security between client and server. There are at least two situations in which that can arise:

- Some transparent web proxies intercept HTTPS connections and act as a decrypting man-in-the-middle. For that to work, the client and server must accept the proxy's certificates of course, but that may be the only way for the client to connect to the server.

- On the server side, it can be useful to have a load balancer that performs HTTPS off-load in front of the server. In that case, the client connects with HTTPS, but the server sees a plain HTTP connection.

To ensure message security in those cases, the HTTP transport supports in-message encryption, with pluggable encryption implementations. See `include/omniORB4/httpCrypto.h` for the interfaces. The `omnihttpCrypto` library provides an implementation that uses AES-256 for message encryption and RSA with pre-shared public keys for key transfer. To use it, on the server side, registering the public keys of two known clients:

```
httpCryptoManager_AES_RSA* manager = new httpCryptoManager_AES_RSA();
manager->init("server-ident", "server_private.pem", true);
manager->addClient("client-ident1", "client1_public.pem", true);
manager->addClient("client-ident2", "client2_public.pem", true);

httpContext::crypto_manager = manager;
```

and on the client:

```
httpCryptoManager_AES_RSA* manager = new httpCryptoManager_AES_RSA();
manager->init("client-ident1", "client1_private.pem", true);
manager->addServer("http://server.example.com:8000/call",
                   "server_public.pem", true);

httpContext::crypto_manager = manager;
```

7.10 ZIOP

omniORB has support for ZIOP, which compresses transmitted messages. To use it, link with the `omniZIOP4` library.

On Unix platforms, ZIOP support is automatically enabled if the configure script detects zlib. To enable it on Windows, set the `EnableZIOP` make variable in the platform configuration file.

omniORB has an almost complete implementation of the ZIOP specification, with the following extensions and differences:

1. To avoid a dependency on `CORBA::Any`, compression policies can be obtained with functions in the `omniZIOP` namespace, rather than with the standard `orb->create_policy()`. See `include/omniORB4/omniZIOP.h` for details. To use `Any` with the standard `orb->create_policy()`, link with the `omniZIOPDynamic4` library in addition to `omniZIOP4`.

2. Client-side policies are global, set with `omniZIOP::setGlobalPolicies()`. `CORBA::Object::_set_policy_overrides()` is not supported.

3. POAs can be given ZIOP policies as shown in the `src/examples/ziop/ziop_impl.cc`, but they can also use the global policies set with `omniZIOP::setGlobalPolicies()`. This is useful to apply ZIOP policies to the Root-POA or omniINSPOA.

In addition to the standard policies, whether or not to enable ZIOP is determined by client and server transport rules. For a client to use ZIOP, the matching client transport rule must include 'ziop'; similarly, for a server to use ZIOP, the matching server transport rule must include 'ziop'. e.g. to use the examples:

```
ziop_impl -ORBserverTransportRule "* unix,ssl,tcp,ziop"

ziop_clt -ORBclientTransportRule "* unix,ssl,tcp,ziop" IOR:...
```

This allows you to enable ZIOP for WAN links, but disable it for LAN communication, for example.

7.10.1 Forcing ZIOP Policies

The fact that a server supports ZIOP is encoded in its IORs. This means that if a client uses a `corbaloc` URI to reference an object, the object reference does not contain ZIOP details, and thus the communication cannot use ZIOP. If a client is absolutely certain that a server supports ZIOP, it can extend an object reference with ZIOP details using `omniZIOP::setServerPolicies()`. Using the new object reference, the client will be able to make ZIOP calls.

```
namespace omniZIOP {
  CORBA::Object_ptr
  setServerPolicies(CORBA::Object_ptr obj, const CORBA::PolicyList& policies);
};
```

Creating a ZIOP-enabling object reference in this way is dangerous! If the server does not actually support ZIOP, it will receive compressed messages that it cannot handle. A well-behaved server will throw a `CORBA::MARSHAL` exception in response, or perhaps just drop the invalid connection.

7.11 Connection Management Extension

The `omniConnectionMgmt` library provides an omniORB-specific extension for application-level connection management. Its purpose is to allow clients and servers to negotiate private GIOP connections, and to control how the connections are used in multi-threaded situations.

The `omniConnectionMgmt` library has two functions, defined in `include/omniORB4/omniConnectionMgmt.h`:

```
namespace omniConnectionMgmt {

  void init();

  CORBA::Object_ptr
  makeRestrictedReference(CORBA::Object_ptr obj,
                          CORBA::ULong      connection_id,
                          CORBA::ULong      max_connections,
                          CORBA::ULong      max_threads,
                          CORBA::Boolean    data_batch,
                          CORBA::Boolean    permit_interleaved,
                          CORBA::Boolean    server_hold_open);
};
```

The `init()` function must be called before `CORBA::ORB_init()` in every process that is to take part in the connection management.

The `makeRestrictedReference()` function is the single entry-point to the connection management functionality. It builds an annotated object reference that contains information for the connection management system. It returns a new reference, leaving the original object reference unchanged.

7.11.1 Client-side parameters

These parameters affect the client side of a connection:

`connection_id`

This number identifies the private connection set. All object references with the same `connection_id` will share the same set of GIOP connections. Object references with different connection ids are guaranteed to use different connections from each other, and from object references that have not been annotated with `makeRestrictedReference()`.

`max_connections`

This parameter overrides the omniORB `maxGIOPConnectionPerServer` configuration parameter for the given `connection_id`. It determines the maximum number of separate GIOP connections that will be opened to the object's server to service concurrent calls. It is common to set this value to 1, indicating that only one connection will be used for the given `connection_id`. Note that this parameter can only be used to reduce the default `maxGIOPConnectionPerServer` value, not increase it.

`data_batch`

omniORB usually configures its TCP connections to disable Nagle's algorithm, which batches small messages together into single IP packages, since that is best for the usual CORBA usage pattern of two-way requests. Setting this parameter to true overrides that, and enables Nagle's algorithm on TCP connections or equivalent functionality on other transports. This can increase throughput if a client is sending a large number of small oneway calls.

`permit_interleaved`

This parameter overrides the `oneCallPerConnection` configuration parameter that determines whether multi-threaded clients can interleave calls on a single connection, issuing a new request message while a previous request is still waiting for a reply. If `permit_interleaved` is true, clients can interleave messages; if it is false, they cannot.

7.11.2 Server-side parameters

These parameters affect the client side of a connection:

`max_threads`

This parameter overrides the global `maxServerThreadPerConnection` configuration parameter that determines the maximum number of concurrent threads the server will use to service requests coming from a connection. Note that this parameter is only relevant if either the client permits interleaved calls, or if oneway operations are used, since those are the only circumstances under which the server can receive a new request on a connection while already handling a request. As with the `max_connections` client-side parameter, this parameter can only reduce the default number of threads, not increase it.

`server_hold_open`

Normally, both clients and servers can decide to close a GIOP connection at any time. When using normal two-way calls, this is no problem since if a server closes a connection, the client is guaranteed to notice it when it waits for a reply, and can retry the call if necessary. With oneway calls, however, if a server closes a connection just as the client is sending a request, the client will not know whether the oneway call was received or not, and the call will potentially be lost. By setting the `server_hold_open` parameter to true, the server will not close the connection, relying on the client to do so. In that case, oneway calls will not be lost unless there is a network problem that breaks the GIOP connection.

7.11.3 Usage

The omniConnectionMgmt extension is very easy to use—simply call the `init()` method in all processes involved, then restrict references as required. The `makeRestrictedReference()` function adds profile information to the object reference's IOR, meaning that the parameters become part of the object reference and are transmitted along with it. In other words, a server can create a restricted reference and send it to a client, and the client will automatically make use of the restricted parameters when it invokes operations on the object reference. Alternatively, a client can restrict a normal reference it receives, in order to change its own behaviour.

Chapter 8

Interoperable Naming Service

omniORB supports the Interoperable Naming Service (INS). The following is a summary of its facilities.

8.1 Object URIs

As well as accepting IOR-format strings, `ORB::string_to_object()` also supports two Uniform Resource Identifier (URI) [BLFIM98] formats, which can be used to specify objects in a convenient human-readable form.

8.1.1 corbaloc

`corbaloc` URIs allow you to specify object references which can be contacted by IIOP, or found through `ORB::resolve_initial_references()`. To specify an IIOP object reference, you use a URI of the form:

`corbaloc:iiop:`*<host>*`:`*<port>*`/`*<object key>*

for example:

```
corbaloc:iiop:myhost.example.com:1234/MyObjectKey
```

which specifies an object with key 'MyObjectKey' within a process running on myhost.example.com listening on port 1234. Object keys containing non-ASCII characters can use the standard URI % escapes:

```
corbaloc:iiop:myhost.example.com:1234/My%efObjectKey
```

denotes an object key with the value 239 (hex ef) in the third octet.

The protocol name '`iiop`' can be abbreviated to the empty string, so the original URI can be written:

```
corbaloc::myhost.example.com:1234/MyObjectKey
```

The IANA has assigned port number 2809[1] for use by `corbaloc`, so if the server is listening on that port, you can leave the port number out. The following two URIs refer to the same object:

```
corbaloc::myhost.example.com:2809/MyObjectKey
corbaloc::myhost.example.com/MyObjectKey
```

You can specify an object which is available at more than one location by separating the locations with commas:

```
corbaloc::myhost.example.com,:localhost:1234/MyObjectKey
```

Note that you must restate the protocol for each address, hence the '`:`' before '`localhost`'. It could equally have been written '`iiop:localhost`'.

You can also specify an IIOP version number:

```
corbaloc::1.2@myhost.example.com/MyObjectKey
```

Specifying IIOP versions above 1.0 is slightly risky since higher versions make use of various information stored in IORs that is not present in a corbaloc URI. It is generally best to contact initial corbaloc objects with IIOP 1.0, and rely on higher versions for all other object references.

8.1.2 Other transports

The only transport specified in the CORBA standard is `iiop`, but omniORB also supports the following extensions:

`ssliop`
: Equivalent semantics to `iiop`, but the server is contacted using SSL / TLS. As with `iiop`, the address details are of the form *`host:port`*.

`omniunix`
: The omniORB Unix domain socket transport. The address details are of the form *`filename`*.

[1] Not 2089 as printed in [OMG00]!

8.1.3 HTTP URLs

If the omniORB-specific HTTP transport is enabled (see section 7.9), then HTTP URLs are also supported. The format is

scheme`://`<*host*>[`:`*port*]`/`<*path*>`#`<*object key*>

Where *scheme* is one of `http`, `https`, `ws` or `wss`. For example:

```
http://myhost.example.com/call#MyObjectKey
wss://myhost.example.com/websocket#MyObjectKey
```

8.1.4 Resolve initial references

A `corbaloc:` can also specify a call to `resolve_initial_references()`. This

```
orb->string_to_object("corbaloc:rir:/NameService");
```

is identical in behaviour to

```
orb->resolve_initial_references("NameService");
```

8.1.5 corbaname

`corbaname` URIs cause `string_to_object()` to look-up a name in a CORBA Naming service. They are an extension of the `corbaloc` syntax:

`corbaname:`<*corbaloc location*>`/`<*object key*>`#`<*stringified name*>

for example:

```
corbaname::myhost/NameService#project/example/echo.obj
corbaname:rir:/NameService#project/example/echo.obj
```

The object found with the `corbaloc`-style portion must be of type `CosNaming::NamingContext`, or something derived from it. If the object key (or `rir` name) is '`NameService`', it can be left out:

```
corbaname::myhost#project/example/echo.obj
corbaname:rir:#project/example/echo.obj
```

The stringified name portion can also be left out, in which case the URI denotes the `CosNaming::NamingContext` which would have been used for a look-up:

```
corbaname::myhost.example.com
corbaname:rir:
```

The first of these examples is the easiest way of specifying the location of a naming service.

8.2 Configuring resolve_initial_references

The INS specifies two standard command line arguments that provide a portable way of configuring `ORB::resolve_initial_references()`:

8.2.1 ORBInitRef

`-ORBInitRef` takes an argument of the form *<ObjectId>=<ObjectURI>*. So, for example, with command line arguments of:

```
-ORBInitRef NameService=corbaname::myhost.example.com
```

`resolve_initial_references("NameService")` will return a reference to the object with key 'NameService' available on myhost.example.com, port 2809. Since IOR-format strings are considered URIs, you can also say things like:

```
-ORBInitRef NameService=IOR:00ff...
```

8.2.2 ORBDefaultInitRef

`-ORBDefaultInitRef` provides a prefix string which is used to resolve otherwise unknown names. When `resolve_initial_references()` is unable to resolve a name which has been specifically configured (with `-ORBInitRef`), it constructs a string consisting of the default prefix, a '`/`' character, and the name requested. The string is then fed to `string_to_object()`. So, for example, with a command line of:

```
-ORBDefaultInitRef corbaloc::myhost.example.com
```

a call to `resolve_initial_references("MyService")` will return the object reference denoted by '`corbaloc::myhost.example.com/MyService`'.

Similarly, a `corbaname` prefix can be used to cause look-ups in the naming service. Note, however, that since a '`/`' character is always added to the prefix, it is impossible to specify a look-up in the root context of the naming service—you have to use a sub-context, like:

```
-ORBDefaultInitRef corbaname::myhost.example.com#services
```

8.3 omniNames

8.3.1 NamingContextExt

omniNames supports the extended `CosNaming::NamingContextExt` interface:

```
module CosNaming {
  interface NamingContextExt : NamingContext {
    typedef string StringName;
    typedef string Address;
    typedef string URLString;

    StringName  to_string(in Name n)        raises(InvalidName);
    Name        to_name  (in StringName sn) raises(InvalidName);

    exception InvalidAddress {};

    URLString   to_url(in Address addr, in StringName sn)
      raises(InvalidAddress, InvalidName);

    Object      resolve_str(in StringName n)
      raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
  };
};
```

`to_string()` and `to_name()` convert from `CosNaming::Name` sequences to flattened strings and vice-versa. Note that calling these operations involves remote calls to the naming service, so they are not particularly efficient. You can use the omniORB specific local `omniURI::nameToString()` and `omniURI::stringToName()` functions instead.

A `CosNaming::Name` is stringified by separating name components with '`/`' characters. The `kind` and `id` fields of each component are separated by '`.`' characters. If the `kind` field is empty, the representation has no trailing '`.`'; if the `id` is empty, the representation starts with a '`.`' character; if both `id` and `kind` are empty, the representation is just a '`.`'. The backslash '`\`' is used to escape the meaning of '`/`', '`.`' and '`\`' itself.

`to_url()` takes a `corbaloc` style address and key string (but without the `corbaloc:` part), and a stringified name, and returns a `corbaname` URI (incorrectly called a URL) string, having properly escaped any invalid characters. The specification does not make it clear whether or not the address string should also be escaped by the operation; omniORB does not escape it. For this reason, it is best to avoid calling `to_url()` if the address part contains escapable characters. To avoid remote calls, omniORB provides the equivalent local function `omniURI::addrAndNameToURI()`.

`resolve_str()` is equivalent to calling `to_name()` followed by the inherited `resolve()` operation. There are no string-based equivalents of the various bind operations.

8.3.2 Use with corbaname

To make it easy to use omniNames with `corbaname` URIs, it starts with the default port of 2809, and an object key of '`NameService`' for the root naming context.

8.4 omniMapper

omniMapper is a simple daemon which listens on port 2809 (or any other port), and redirects IIOP requests for configured object keys to associated persistent object references. It can be used to make a naming service (even an old non-INS aware version of omniNames or other ORB's naming service) appear on port 2809 with the object key '`NameService`'. The same goes for any other service you may wish to specify, such as an interface repository. omniMapper is started with a command line of:

```
omniMapper [-port <port>] [-config <config file>] [-v]
```

The `-port` option allows you to choose a port other than the default 2809 to listen on. The `-config` option specifies a location for the configuration file. The default name is `/etc/omniMapper.cfg`, or `C:\omniMapper.cfg` on Windows. omniMapper does not normally print anything; the `-v` option makes it verbose so it prints configuration information and a record of the redirections it makes, to standard output.

The configuration file is very simple. Each line contains a string to be used as an object key, some white space, and an IOR (or any valid URI) that it will redirect that object key to. Comments should be prefixed with a '#' character. For example:

```
# Example omniMapper.cfg
NameService         IOR:000f...
InterfaceRepository IOR:0100...
```

omniMapper can either be run on a single machine, in much the same way as omniNames, or it can be run on *every* machine, with a common configuration file. That way, each machine's omniORB configuration file could contain the line:

```
ORBDefaultInitRef corbaloc::localhost
```

8.5 Creating objects with plain object keys

In normal use, omniORB creates object keys containing various information including POA names and various non-ASCII characters. Since object keys are

supposed to be opaque, this is not usually a problem. The INS breaks this opacity and requires servers to create objects with human-friendly keys.

If you wish to make your objects available with human-friendly URIs, there are two options. The first is to use omniMapper as described above, in conjunction with a `PERSISTENT` POA. The second is to create objects with the required keys yourself. You do this using a POA with the special property that the object keys it creates contain only the object ids given to the POA, and no other data.

There is a pre-defined POA with the correct policies named '`omniINSPOA`', acquired from `resolve_initial_references()`. This POA has the `USER_ID` and `PERSISTENT` policies. It is a normal POA in all other respects, so you can activate/deactivate it, create children, and so on, in the usual way. Children of the omniINSPOA do not inherit its special properties of creating plain object keys.

If the omniINSPOA's policies are not suitable for your application, you can create a different POA, giving it the omniORB-specific 'plain object keys' policy:

```
policy = omniPolicy::create_plain_object_keys_policy(
                                  omniPolicy::PLAIN_OBJECT_KEYS_ENABLE);
```

The other policies must be compatible:

- `PERSISTENT` lifecycle policy
- `USER_ID` id assignment policy
- `USE_ACTIVE_OBJECT_MAP_ONLY` request processing policy

You can choose any thread policy and provide an omniORB endpoint publishing policy.

Chapter 9

Code set conversion

omniORB supports full code set negotiation, used to select and translate between different character code sets when transmitting chars, strings, wchars and wstrings. The support is mostly transparent to application code, but there are a number of options that can be selected. This chapter covers the options, and also gives some pointers about how to implement your own code sets, in case the ones that come with omniORB are not sufficient.

9.1 Native code sets

For the ORB to know how to handle strings and wstrings given to it by the application, it must know what code set they are represented with, so it can properly translate them if need be. The defaults are ISO 8859-1 (Latin 1) for char and string, and UTF-16 for wchar and wstring. Different code sets can be chosen at initialisation time with the `nativeCharCodeSet` and `nativeWCharCodeSet` parameters. The supported code sets are printed out at initialisation time if the ORB traceLevel is 15 or greater.

For many applications, the defaults are fine. The most common non-default choice is to set the native char code set to UTF-8, allowing the full Unicode range to be supported in strings.

Note that the default for wchar is always UTF-16, even on Unix platforms where wchar is a 32-bit type. Select the UCS-4 code set to select characters outside the basic multilingual plane without having to use UTF-16 surrogates[1].

[1] If you have no idea what this means, don't worry—you're better off not knowing unless you *really* have to.

9.2 Default code sets

The way code set conversion is meant to work in CORBA communication is that each client and server has a *native* code set that it uses for character data in application code, and supports a number of *transmission* code sets that it uses for communication. When a client connects to a server, the client picks one of the server's transmission code sets to use for the interaction. For that to work, the client plainly has to know the server's supported transmission code sets.

Code set information from servers is embedded in IORs. A client with an IOR from a server should therefore know what transmission code sets the server supports. This approach can fail for two reasons:

1. A `corbaloc` URI (see chapter 8) does not contain any code set information.
2. Some badly-behaved servers that do support code set conversion fail to put codeset information in their IORs.

Similarly, a badly-behaved client may fail to specify codeset information when it first uses a connection to a server, meaning that the server does not know the transmission codeset used by the client.

The CORBA standard says that if a server has not specified transmission code set information, clients must assume that they only support ISO-8859-1 for char and string, and do not support wchar and wstring at all. Similarly, a server that does not receive codeset information from a client must assume that the client is using ISO-8859-1 for char and string, and will not send wchar and wstring. The effect is that `DATA_CONVERSION` or `BAD_PARAM` exceptions are thrown.

To avoid these issues, omniORB allows you to configure *default* code sets that are assumed to be supported by the other end of the communication, if they are not otherwise known. Set `defaultCharCodeSet` for char and string data, and `defaultWCharCodeSet` for wchar and wstring data.

9.3 Code set library

To save space in the main ORB core library, most of the code set implementations are in a separate library named omniCodeSets4. To use the extra code sets, you must link your application with that library. On most platforms, if you are using dynamic linking, specifying the omniCodeSets4 library in the link command is sufficient to have it initialised, and for the code sets to be available. With static linking, or platforms with less intelligent dynamic linkers, you must force the linker to initialise the library. You do that by including the `omniORB4/optionalFeatures.h` header. By default, that header enables several optional features. Look at the file contents to see how to turn off particular features.

9.4 Implementing new code sets

It is quite easy to implement new code sets, if you need support for code sets (or marshalling formats) that do not come with the omniORB distribution. There are extensive comments in the headers and ORB code that explain how to implement a code set; this section just serves to point you in the right direction.

The main definitions for the code set support are in `include/omniORB4/codeSets.h`. That defines a set of base classes use to implement code sets, plus some derived classes that use look-up tables to convert simple 8-bit and 16-bit code sets to Unicode.

When sending or receiving string data, there are a total of four code sets in action: a native char code set, a transmission char code set, a native wchar code set, and a transmission wchar code set. The native code sets are as described above; the transmission code sets are the ones selected to communicate with a remote machine. They are responsible for understanding the GIOP marshalling formats, as well as the code sets themselves. Each of the four code sets has an object associated with it which contains methods for converting data.

There are two ways in which a string/wstring can be transmitted or received. If the transmission code set in action knows how to deal directly with the native code set (the trivial case being that they are the same code set, but more complex cases are possible too), the transmission code set object can directly marshal or unmarshal the data into or out of the application buffer. If the transmission code set does not know how to handle the native code set, it converts the string/wstring into UTF-16, and passes that to the native code set object (or vice-versa). All code set implementations must therefore know how to convert to and from UTF-16.

With this explanation, the classes in `codeSets.h` should be easy to understand. The next place to look is in the various existing code set implementations, which are files of the form `cs-*.cc` in the `src/lib/omniORB/orbcore` and `src/lib/omniORB/codesets`. Note how all the 8-bit code sets (the ISO 8859-* family) consist entirely of data and no code, since they are driven by look-up tables.

Chapter 10

Interceptors

omniORB supports interceptors that allow the application to insert processing in various points along the call chain, and in various other locations. It does not support the standard Portable Interceptors API.

The interceptor interfaces are defined in a single header, `include/omniORB4/omniInterceptors.h`. Each interception point consists of a singleton object with `add()` and `remove()` methods, and the definition of an 'interceptor info' class. For example:

```
class omniInterceptors {
  ...
  class clientSendRequest_T {
  public:

    class info_T {
    public:
      GIOP_C&                 giop_c;
      IOP::ServiceContextList service_contexts;

      info_T(GIOP_C& c) : giop_c(c), service_contexts(5) {}

      // Accessors for operation / connection details
      const char* operation();
      const char* myaddress();
      const char* peeraddress();
      const char* peeridentity();
      void*       peerdetails();

      //... private things
    };

    typedef CORBA::Boolean (*interceptFunc)(info_T& info);
```

```
      void add(interceptFunc);
      void remove(interceptFunc);
    };
    ...
  };
```

You can see that the interceptors themselves are functions that take the `info_T` object as their argument and return boolean. Interceptors are called in the order they are registered; normally, all interceptor functions return `true`, meaning that processing should continue with subsequent interceptors. If an interceptor returns `false`, later interceptors are not called. You should only do that if you really know what you are doing.

Notice that the `info_T` contains references to omniORB internal data types. The definitions of these types can be found in other header files within `include/omniORB4` and `include/omniORB4/internal`.

10.1 Interceptor registration

All the interceptor singletons are registered within another singleton object of class `omniInterceptors`. You retrieve a pointer to the object with the `omniORB::getInterceptors()` function, which must be called after the ORB has been initialised with `CORBA::ORB_init()`, but before the ORB is used. The code to register an interceptor looks, for example, like:

```
omniInterceptors* interceptors = omniORB::getInterceptors();
interceptors->clientSendRequest.add(myInterceptorFunc);
```

10.2 Available interceptors

The following interceptors are available:

encodeIOR
: Called when encoding an IOR to represent an object reference. This interception point allows the application to insert extra profile components into IORs. Note that you must understand and adhere to the rules about data stored in IORs, otherwise the IORs created may be invalid. omniORB itself uses this interceptor to insert various items, so you can see an example of its use in the `insertSupportedComponents()` function defined in `src/lib/omniORB/orbcore/ior.cc`.

decodeIOR
: Called when decoding an IOR. The application can use this to get out whatever information they put into IORs with encodeIOR. Again, see `extract`

`SupportedComponents()` in `src/lib/omniORB/orbcore/ior.cc` for an example.

clientOpenConnection
Called as a client opens a new connection to a server, after the connection is opened but before it is used to send a request. The interceptor function can set the `info_T`'s `reject` member to `true` to cause the client to immediately close the new connection and throw CORBA::TRANSIENT to the calling code. In that case, the interceptor function can also set the `why` member to provide a message that is logged.

clientSendRequest
Called just before a request header is sent over the network. The application can use it to insert service contexts in the header. See the `setCodeSetServiceContext()` function in `src/lib/omniORB/orbcore/cdrStream.cc` for an example of its use.

clientReceiveReply
Called as the client receives a reply, just after unmarshalling the reply header. Called for normal replies and exceptions.

serverAcceptConnection
Called when a server accepts a new incoming connection, but before it reads any data from it. The interceptor function can set the `info_T`'s `reject` member to `true` to cause the server to immediately close the new connection. In that case, the interceptor function can also set the `why` member to provide a message that is logged.

serverReceiveRequest
Called when the server receives a request, just after unmarshalling the request header. See the `getCodeSetServiceContext()` function in `src/lib/omniORB/orbcore/cdrStream.cc` for an example.

serverSendReply
Called just before the server marshals a reply header.

serverSendException
Called just before the server marshals an exception reply header.

createRope
Called when the ORB is about to create a 'rope' that encapsulates a bundle of connections ('strands') to a remote address space. It allows application code to override omniORB's normal connection management.

createIdentity
Called when the ORB is about to create an 'identity' object to represent a CORBA object. It allows application code to provide its own identity implementations. It is very unlikely that an application will need to do this.

createORBServer
Used internally by the ORB to register different kinds of server. At present, only a GIOP server is registered. It is very unlikely that application code will need to do this.

createThread
Called whenever the ORB creates a thread. The `info_T` class for this interceptor is

```
class info_T {
public:
  virtual void run() = 0;
  virtual omni_thread* self() = 0;
};
```

The interceptor is called in the context of the newly created thread. The function *must* call the `info_T`'s `run()` method, to pass control to the thread body. `run()` returns just before the thread exits. This arrangement allows the interceptor to initialise some per-thread state before the thread body runs, then release it just before the thread exits.

The `info_T`'s `self()` method returns a pointer to the `omni_thread` object for the thread, equivalent to calling `omni_thread::self()`.

assignUpcallThread
The ORB maintains a general thread pool, from which threads are drawn for various purposes. One purpose is for performing upcalls to application code, in response to incoming CORBA calls. The `assignUpcallThread` interceptor is called when a thread is assigned to perform upcalls. In the thread per connection model, the thread stays assigned to performing upcalls for the entire lifetime of the underlying network connection; in the thread pool model, threads are assigned for upcalls on a per call basis, so this interceptor is triggered for every incoming call[1]. As with the `createThread` interceptor, the interceptor function must call the `info_T`'s `run()` method to pass control to the upcall.

[1] Except that with the `threadPoolWatchConnection` parameter set `true`, a thread can perform multiple upcalls even when thread pool mode is active.

When a thread finishes its assignment of processing upcalls, it returns to the pool (even in thread per connection mode), so the same thread can be reassigned to perform more upcalls, or reused for a different purpose.

assignAMIThread

Asynchronous Method Invocation (AMI) uses threads to perform outgoing calls. The `assignAMIThread` interceptor is called when a thread is assigned to perform AMI calls. As with the other thread interceptors, the interceptor function must call the `info_T`'s `run()` method to pass control to the AMI call.

Unlike the other interceptors, the interceptor functions for `createThread`, `assignUpcallThread` and `assignAMIThread` have no return values. Interceptor chaining is performed by calls through the `info_T::run()` method, rather than by visiting interceptor functions in turn.

10.3 Server-side call interceptor

Calls can be intercepted on the server just before the upcall into application code. This interceptor is registered with omniORB's `callDescriptor` class, which is responsible for encapsulating the state of a call. Unlike the transport-related `serverReceiveRequest`, `serverSendReply` and `serverSendException` interceptors, the `callDescriptor` interceptor is invoked for *all* calls, even ones from colocated clients in the same address space.

The types used for the call interceptor are defined in `include/omniORB4/callDescriptor.h`. The interceptor takes the form of a bare function with two parameters. The first parameter is a pointer to the `callDescriptor`; the second is a pointer to `omniServant`, which is the base class of all servant classes. The interceptor function must call the `callDescriptor`'s `interceptedCall()` method to pass on the call.

This interception point allows access to various parts of omniORB's call machinery. The `callDescriptor` includes access to the operation name and, if cast to the concrete subclass defined by the IDL compiler, the call arguments and return values too.

Chapter 11

Type Any and TypeCode

The CORBA specification provides for a type that can hold the value of any OMG IDL type. This type is known as type `Any`. The OMG also specifies a pseudo-object, `TypeCode`, that can encode a description of any type specifiable in OMG IDL.

In this chapter, an example demonstrating the use of type Any is presented. The example code is in the `src/examples/anyExample` directory in the omniORB distribution. The example is followed by sections describing the behaviour of type Any and TypeCode in omniORB. For further information on type Any, refer to the C++ Mapping specification., and for more information on TypeCode, refer to the Interface Repository chapter in the CORBA core section of the CORBA specification.

11.1 Example using type Any

Before going through this example, you should make sure that you have read and understood the examples in chapter 2.

11.1.1 Type Any in IDL

Type Any allows one to delay the decision on the type used in an operation until run-time. To use type any in IDL, use the keyword `any`, as in the following example:

```
// IDL
interface anyExample {
  any testOp(in any mesg);
};
```

The operation `testOp()` in this example can now take any value expressible in OMG IDL as an argument, and can also return any type expressible in OMG IDL.

Type Any is mapped into C++ as the type `CORBA::Any`. When passed as an argument or as a result of an operation, the following rules apply:

In	InOut	Out	Return
`const CORBA::Any&`	`CORBA::Any&`	`CORBA::Any*&`	`CORBA::Any*`

So, the above IDL would map to the following C++:

```
// C++

class anyExample_i : public virtual POA_anyExample {
public:
  anyExample_i() { }
  virtual ~anyExample_i() { }
  virtual CORBA::Any* testOp(const CORBA::Any& a);
};
```

11.1.2 Inserting and Extracting Basic Types from an Any

The question now arises as to how values are inserted into and removed from an Any. This is achieved using two overloaded operators: <<= and >>=.

To insert a value into an Any, the <<= operator is used, as in this example:

```
// C++
CORBA::Any an_any;
CORBA::Long l = 100;
an_any <<= l;
```

Note that the overloaded <<= operator has a return type of `void`.

To extract a value, the >>= operator is used, as in this example (where the Any contains a long):

```
// C++
CORBA::Long l;
an_any >>= l;

cout << "This is a long: " << l << endl;
```

The overloaded >>= operator returns a `CORBA::Boolean`. If an attempt is made to extract a value from an Any when it contains a different type of value (e.g. an attempt to extract a long from an Any containing a double), the overloaded >>= operator will return `false`; otherwise it will return `true`. Thus, a common tactic to extract values from an Any is as follows:

```
// C++
CORBA::Long l;
CORBA::Double d;
const char* str;
```

```
if (an_any >>= l) {
  cout << "Long: " << l << endl;
}
else if (an_any >>= d) {
  cout << "Double: " << d << endl;
}
else if (an_any >>= str) {
  cout << "String: " << str << endl;
  // The storage of the extracted string is still owned by the any.
}
else {
  cout << "Unknown value." << endl;
}
```

11.1.3 Inserting and Extracting Constructed Types from an Any

It is also possible to insert and extract constructed types and object references from an Any. omniidl will generate insertion and extraction operators for the constructed type. Note that it is necessary to specify the `-Wba` command-line flag when running omniidl in order to generate these operators. The following example illustrates the use of constructed types with type Any:

```
// IDL
struct testStruct {
  long l;
  short s;
};

interface anyExample {
  any testOp(in any mesg);
};
```

Upon compiling the above IDL with `omniidl -bcxx -Wba`, the following overloaded operators are generated:

1. `void operator<<=(CORBA::Any&, const testStruct&)`
2. `void operator<<=(CORBA::Any&, testStruct*)`
3. `CORBA::Boolean operator>>=(const CORBA::Any&,`
   `const testStruct*&)`

Operators of this form are generated for all constructed types, and for interfaces.

The first operator, *(1)*, copies the constructed type, and inserts it into the Any. The second operator, *(2)*, inserts the constructed type into the Any, and

then manages it. Note that if the second operator is used, the Any consumes the constructed type, and the caller should not use the pointer to access the data after insertion. The following is an example of how to insert a value into an Any using operator *(1)*:

```
// C++
CORBA::Any an_any;

testStruct t;
t.l = 456;
t.s = 8;

an_any <<= t;
```

The third operator, *(3)*, is used to extract the constructed type from the Any, and can be used as follows:

```
const testStruct* tp;

if (an_any >>= tp) {
    cout << "testStruct: l: " << tp->l << endl;
    cout << "            s: " << tp->s << endl;
}
else {
    cout << "Unknown value contained in Any." << endl;
}
```

As with basic types, if an attempt is made to extract a type from an Any that does not contain a value of that type, the extraction operator returns `false`. If the Any does contain that type, the extraction operator returns `true`. If the extraction is successful, the caller's pointer will point to memory managed by the Any. The caller must not delete or otherwise change this storage, and should not use this storage after the contents of the Any are replaced (either by insertion or assignment), or after the Any has been destroyed. In particular, management of the pointer should not be assigned to a `_var` type.

If the extraction fails, the caller's pointer will be set to point to null.

Note that there are special rules for inserting and extracting arrays (using the `_forany` types), and for inserting and extracting bounded strings, booleans, chars, and octets. Please refer to the C++ Mapping specification for further information.

11.2 Type Any in omniORB

This section contains some notes on the use and behaviour of type Any in omniORB.

11.2.1 Generating Insertion and Extraction Operators.

To generate type Any insertion and extraction operators for constructed types and interfaces, the `-Wba` command line flag should be specified when running omniidl.

11.2.2 TypeCode comparison when extracting from an Any.

When an attempt is made to extract a type from an Any, the TypeCode of the type is checked for *equivalence* with the TypeCode of the type stored by the Any. The `equivalent()` test in the TypeCode interface is used for this purpose. For example:

```
// IDL 1
typedef double Double1;

struct Test1 {
  Double1 a;
};
```

```
// IDL 2
typedef double Double2;

struct Test1 {
  Double2 a;
};
```

If an attempt is made to extract the type `Test1` defined in IDL 1 from an Any containing the `Test1` defined in IDL 2, this will succeed (and vice-versa), as the two types differ only by an alias.

11.2.3 Top-level aliases.

When a type is inserted into an Any, the Any stores both the value of the type and the TypeCode for that type. However, in some cases, a top-level alias can be lost due to the details of the C++ mapping. For example, consider these IDL definitions:

```
// IDL 3
typedef sequence<double> seqDouble1;
typedef sequence<double> seqDouble2;
typedef seqDouble2       seqDouble3;
```

omniidl generates distinct types for `seqDouble1` and `seqDouble2`, and therefore each has its own set of C++ operators for Any insertion and extraction. That means inserting a `seqDouble1` into an Any sets the Any's TypeCode to include the alias 'seqDouble1', and inserting a `seqDouble2` sets the TypeCode to the alias 'seqDouble2'.

However, in the C++ mapping, `seqDouble3` is required to be just a C++ typedef to `seqDouble2`, so the C++ compiler uses the Any insertion operator for `seqDouble2`. Therefore, inserting a `seqDouble3` sets the Any's TypeCode to the `seqDouble2` alias. If this is not desirable, you can use the member function '`void type(TypeCode_ptr)`' of the Any interface to explicitly set the TypeCode to the correct one.

11.2.4 Removing aliases from TypeCodes.

Some ORBs (such as old versions of Orbix) will not accept TypeCodes containing `tk_alias` TypeCodes. When using type Any while interoperating with these ORBs, it is necessary to remove `tk_alias` TypeCodes from throughout the TypeCode representing a constructed type.

To remove all `tk_alias` TypeCodes from TypeCodes transmitted in Anys, supply the `-ORBtcAliasExpand 1` command-line flag when running an omniORB executable. There will be some (small) performance penalty when transmitting Any values.

Note that the `_tc_` TypeCodes generated for all constructed types will contain the complete TypeCode for the type (including any `tk_alias` TypeCodes), regardless of whether the `-ORBtcAliasExpand` flag is set to 1 or not. It is only when Anys are transmitted that the aliases are stripped.

11.2.5 Recursive TypeCodes.

omniORB supports recursive TypeCodes. This means that types such as the following can be inserted or extracted from an Any:

```
// IDL 4
struct Test4 {
  sequence<Test4> a;
};
```

11.2.6 Threads and type Any.

Inserting and extracting simultaneously from the same Any (in 2 threads) results in undefined behaviour.

In versions of omniORB before 4.0, extracting simultaneously from the same Any (in 2 or more different threads) also led to undefined behaviour. That is no longer the case—Any extraction is now thread safe.

11.3 TypeCode in omniORB

This section contains some notes on the use and behaviour of TypeCode in omniORB

11.3.1 TypeCodes in IDL.

When using TypeCodes in IDL, note that they are defined in the CORBA scope. Therefore, `CORBA::TypeCode` should be used. Example:

```
// IDL 5
struct Test5 {
  long length;
  CORBA::TypeCode desc;
};
```

11.3.2 orb.idl

The CORBA specification says that IDL using `CORBA::TypeCode` must include the file `orb.idl`. That is not required in omniORB, but a suitable `orb.idl` is available.

11.3.3 Generating TypeCodes for constructed types.

To generate a TypeCode for constructed types, specify the `-Wba` command-line flag when running omniidl. This will generate a `_tc_` TypeCode describing the type, at the same scope as the type. Example:

```
// IDL 6
struct Test6 {
  double a;
  sequence<long> b;
};
```

A TypeCode, `_tc_Test6`, will be generated to describe the struct `Test6`. The operations defined in the TypeCode interface can be used to query the TypeCode about the type it represents.

Chapter 12

Objects by value, abstract interfaces and local interfaces

omniORB supports objects by value, declared with the `valuetype` keyword in IDL, and both abstract and local interfaces. This chapter outlines some issues to do with using these types in omniORB. You are assumed to have read the relevant parts of the CORBA specification, specifically chapters 3, 4, 5 and 6 of the CORBA 2.6 specification, and sections 1.17, 1.18 and 1.35 of the C++ mapping specification, version 1.1.

12.1 Features

omniORB supports the complete objects by value specification, with the exception of custom valuetypes. All other valuetype features including value boxes, value sharing semantics, abstract valuetypes, and abstract interfaces are supported. Local interfaces are supported, with a number of caveats outlined in section 12.8.

12.2 Reference counting

Values are reference counted. This means that, as long as your application properly manages reference counts, values are usually automatically deleted when they are no longer required. However, one of the features of valuetypes is that they support the representation of cyclic graph structures. In that kind of situation, the reference counting garbage collection does not work, because references internal to the graph prevent the reference counts ever becoming zero.

To avoid memory leaks, application code must explicitly break any reference cycles in values it manipulates. This includes graphs of values received as parameters and return values from CORBA operations.

12.3 Value sharing and local calls

When valuetypes are passed as parameters in CORBA calls (i.e. calls on CORBA objects declared with `interface` in IDL), the structure of related values is maintained. Consider, for example, the following IDL definitions (which are from the example code in `src/examples/valuetype/simple`:

```
module ValueTest {
  valuetype One {
    public string s;
    public long   l;
  };

  interface Test {
    One op1(in One a, in One b);
  };
};
```

If the client to the `Test` object passes the same value in both parameters, just one value is transmitted, and the object implementation receives a copy of the single value, with references to it in both parameters.

In the case that the object is remote from the client, there is obviously a copying step involved. In the case that the object is in the same address space as the client, the same copying semantics must be maintained so that the object implementation can modify the values it receives without the client seeing the modifications. To support that, omniORB must copy the entire parameter list in one operation, in case there is sharing between different parameters. Such copying is a rather more time-consuming process than the parameter-by-parameter copy that takes place in calls not involving valuetypes.

To avoid the overhead of copying parameters in this way, applications can choose to relax the semantics of value copying in local calls, so values are not copied at all, but are passed by reference. In that case, the client to a call *will* see any modifications to the values it passes as parameters (and similarly, the object implementation will see any changes the client makes to returned values). To choose this option, set the `copyValuesInLocalCalls` configuration parameter to zero.

12.4 Value box factories

With normal valuetypes, omniidl generates factory classes (with names ending `_init`) as required by the C++ mapping specification. The application is responsible for registering the factories with the ORB.

Unfortunately, the C++ mapping makes no mention of factories for value boxes. In omniORB, factories for value boxes are automatically registered with

the ORB, and there are no application-visible factory classes generated for them. Some other CORBA implementations generate application visible factories, and the application *does* have to register the factories with the ORB.

12.5 Standard value boxes

The standard `CORBA::StringValue` and `CORBA::WStringValue` value boxes are available to application code. To make the definitions available in IDL, #include the standard `orb.idl`.

12.6 Covariant returns

As required by the C++ mapping, on C++ compilers that support covariant return types, omniidl generates code for the `_copy_value()` function that returns the most derived type of the value. On older compilers, `_copy_value()` returns `CORBA::ValueBase`.

If you write code that calls `_copy_value()`, and you need to support older compilers, you should assign the result to a variable of type `CORBA::ValueBase*` and downcast to the target type, rather than using the covariant return.

If you are overriding `_copy_value()`, you must correctly take account of the `OMNI_HAVE_COVARIANT_RETURNS` preprocessor definition.

12.7 Values inside Anys

Valuetypes inserted into Anys cause a number of interesting issues. Even when inside Anys, values are required to support complete sharing semantics. Take this IDL for example:

```
module ValueTest {
  valuetype One {
    public string s;
    public long   l;
  };

  interface AnyTest {
    void op1(in One v, in Any a);
  };
};
```

Now, suppose the client behaves as follows:

```
ValueTest::One* v = new One_impl("hello", 123);
CORBA::Any a;
a <<= v;
```

```
obj->op1(v, a);
```

then on the server side:

```
void AnyTest_impl::op1(ValueTest::One* v, CORBA::Any& a)
{
  ValueTest::One* v2;
  a >>= v2;
  assert(v2 == v);
}
```

This is all very well in this kind of simple situation, but problems can arise if truncatable valuetypes are used. Imagine this derived value:

```
module ValueTest {
  valuetype Two : truncatable One {
    public double d;
  };
};
```

Now, suppose that the client shown above sends an instance of valuetype `Two` in both parameters, and suppose that the server has not seen the definition of valuetype `Two`. In this situation, as the first parameter is unmarshalled, it will be truncated to valuetype `One`, as required. Now, when the Any is unmarshalled, it refers to the same value, which has been truncated. So, even though the TypeCode in the Any indicates that the value has type `Two`, the stored value actually has type `One`. If the receiver of the Any tries to pass it on, transmission will fail because the Any's value does not match its TypeCode.

In the opposite situation, where an Any parameter comes before a valuetype parameter, a different problem occurs. In that case, as the Any is unmarshalled, there is no type information available for valuetype `Two`, so the value inside the Any has an internal omniORB type used for unknown valuetypes. As the next parameter is unmarshalled, omniORB sees that the shared value is unknown, and is able to convert it to the target `One` valuetype with truncation. In this case, the Any and the plain valuetype both have the correct types and values, but the fact that both should have referred to the same value has been lost.

Because of these issues, it is best to avoid defining interfaces that mix valuetypes and Anys in a single operation, and certainly to avoid trying to share plain values with values inside Anys.

12.7.1 Values inside DynAnys

The sharing semantics of valuetypes can also cause difficulties for DynAny. The CORBA 2.6 specification does not mention how shared values inside DynAnys should be handled; the CORBA 3.x specification slightly clarifies the situation, but it is still unclear. To write portable code it is best to avoid manipulating DynAnys containing values that are shared.

In omniORB, when a value inside an Any is converted into a DynAny, the value's state is copied into the DynAny, and manipulated there. When converting back to an Any a new value is created. This means that any other references to the original value (whether themselves inside Anys of not) still relate to the original value, with unchanged state. However, this copying only occurs when a DynValue is actually created, so for example a structure with two value members referring to the same value can manipulated inside a DynAny without breaking the sharing, provided the value members are not accessed as DynAnys. Extracting the value members as ValueBase will reveal the sharing, for example.

12.8 Local Interfaces

Local interfaces are somewhat under-specified in the C++ mapping. This section outlines the way local interfaces are supported in omniORB, and details the limitations and issues.

12.8.1 Simple local interfaces

With simple IDL, there are no particular issues:

```
module Test {
  local interface Example {
    string hello(in string arg);
  };
};
```

The IDL compiler generates an abstract base class `Test::Example`. The application defines a class derived from it that implements the abstract `hello()` member function. Instances of that class can then be used where the IDL specifies interface `Example`.

Note that, by default, local interface implementations have no reference counting behaviour. If the local object should be deleted when the last reference is released, the application must implement the `_add_ref()` and `_remove_ref()` virtual member functions within the implementation class. Make sure that the implementations are thread safe.

12.8.2 Inheritance from unconstrained interfaces

Local interfaces can inherit from unconstrained (i.e. non-local) interfaces:

```
module Test {
  interface One {
    void problem(inout string arg);
  };
  local interface Two : One {
```

```
  };

  interface Receiver {
    void setOne(in One a);
  };
};
```

IDL like this leads to two issues to do with omniORB's C++ mapping implementation.

First, an instance of local interface `Two` should be suitable to pass as the argument to the `setOne()` method of a `Receiver` object (as long as the object is in the same address space as the caller). Therefore, the `Two` abstract base class has to inherit from the internal class omniORB uses to map object references of type `One`. For performance reasons, the class that implements `One` object references normally has non-virtual member functions. That means that the application-supplied `problem()` member function for the implementation of local interface `Two` will not override the base class's version. To overcome this, the IDL for the base unconstrained interface must be compiled with the `-Wbvirtual-objref` switch to omniidl. That makes the member functions of the mapping of `One` into virtual functions, so they can be overridden.

The second problem is that, in some cases, omniORB uses a different mapping for object reference member functions than the mapping used in servant classes. For example, in the `problem()` operation, it uses an internal type for the inout string argument that avoids memory issues if the application uses a `String_var` in the argument. This means that the abstract member function declared in the `Two` class (and implemented by the application) has a different signature to the member function in the base class. The application-supplied class will therefore not properly override the base class method. In all likelihood, the C++ compiler will also complain that the two member functions are ambiguous. The solution to this problem is to use the implementation mapping in the base object reference class, rather than the normal object reference mapping, using the `-Wbimpl-mapping` switch to omniidl. The consequence of this is that some uses of `_var` types for inout arguments that are normally acceptable in omniORB can now lead to memory management problems.

In summary, to use local interfaces derived from normal unconstrained interfaces, you should compile all your IDL with the omniidl flags:

```
-Wbvirtual-objref -Wbimpl-mapping
```

12.8.3 Valuetypes supporting local interfaces

According to the IDL specification, it should be possible to declare a valuetype that supports a local interface:

```
local interface I {
  void my_operation();
};
valuetype V supports I {
  public string s;
};
```

omniidl accepts the IDL, but unfortunately the resulting C++ code does not compile. The C++ mapping specification has a problem in that both the `CORBA::LocalObject` and `CORBA::ValueBase` classes have `_add_ref()` and `_remove_ref()` member functions defined. The classes generated for the valuetype inherit from both these base classes, and therefore have an ambiguity. Until the C++ mapping resolves this conflict, valuetypes supporting local interfaces cannot be used in omniORB.

Chapter 13

Asynchronous Method Invocation

omniORB 4.2 and later support Asynchronous Method Invocation, AMI, as defined in the CORBA Messaging specification. It supports both the polling and callback models of asynchronous calls. Note that omniORB does not support the other parts of the Messaging specification such as Quality of Service, Routing and Persistent requests.

While omniORB mainly targets the 2.6 version of the CORBA specification, the AMI support follows the CORBA Messaging specification as described in the CORBA 3.1 specification, chapter 17 [OMG08]. That version of the specification is largely the same as the one in CORBA 2.6. The only significant difference is that exception replies in the callback model use a simpler interface-independent mapping.

13.1 Implied IDL

AMI works by defining some additional *implied IDL* for each interface in the real IDL. The implied IDL contains type and operation definitions that enable asynchronous calls.

As a guide to the implied IDL, there is a special `ami` back-end to omniidl that outputs the implied IDL for the given input IDL. For example, given the Echo example IDL:

```
// echo.idl
interface Echo {
  string echoString(in string mesg);
};
```

You can output the implied IDL using

```
omniidl -bami echo.idl
```

That outputs the following to standard out:

```
// ReplyHandler for interface Echo
interface AMI_EchoHandler : Messaging::ReplyHandler {
  void echoString(in string ami_return_val);
  void echoString_excep(in ::Messaging::ExceptionHolder excep_holder);
};

// Poller valuetype for interface Echo
abstract valuetype AMI_EchoPoller : Messaging::Poller {
  void echoString(in unsigned long ami_timeout, out string ami_return_val);
};

// AMI implied operations for interface Echo
interface Echo {
  void sendc_echoString(in ::AMI_EchoHandler ami_handler, in string mesg);
  ::AMI_EchoPoller sendp_echoString(in string mesg);
};
```

Alternatively, you can use the `-Wbdump` option to output an interleaved version that shows the original IDL and the implied IDL together.

Note that the implied IDL output is for information only. You should not compile it, but rather instruct the omniidl C++ back-end to generate the corresponding C++ definitions.

13.2 Generating AMI stubs

To generate stub code including AMI types and operations, give the `-Wbami` command line option to omniidl's `cxx` back-end:

```
omniidl -bcxx -Wbami echo.idl
```

That generates the normal C++ stubs and skeletons, plus all the definitions in the implied IDL.

13.3 AMI examples

Example AMI clients for the `Echo` server can be found in `src/examples/ami`.

Chapter 14

Interface Type Checking

This chapter describes the mechanism used by omniORB to ensure type safety when object references are exchanged across the network. This mechanism is handled completely within the ORB. There is no programming interface visible at the application level. However, for the sake of diagnosing the problem when there is a type violation, it is useful to understand the underlying mechanism in order to interpret the error conditions reported by the ORB.

14.1 Introduction

In GIOP/IIOP, an object reference is encoded as an Interoperable Object Reference (IOR) when it is sent across a network connection. The IOR contains a Repository ID (RepoId) and one or more communication profiles. The communication profiles describe where and how the object can be contacted. The RepoId is a string which uniquely identifies the IDL interface of the object.

Unless the `ID` pragma is specified in the IDL, the ORB generates the RepoId string in the so-called OMG IDL Format[1]. For instance, the RepoId for the `Echo` interface used in the examples of chapter 2 is `IDL:Echo:1.0`.

When interface inheritance is used in the IDL, the ORB always sends the RepoId of the most derived interface. For example:

```
// IDL
interface A {
  ...
};
interface B : A {
  ...
};
interface C {
```

[1] For further details of the repository ID formats, see section 10.6 in the CORBA 2.6 specification.

```
    void op(in A arg);
};

// C++
C_ptr server;
B_ptr objB;
A_ptr objA = objB;
server->op(objA);  // Send B as A
```

In the example, the operation `C::op()` accepts an object reference of type `A`. The real type of the reference passed to `C::op()` is `B`, which inherits from `A`. In this case, the RepoId of `B`, and not that of `A`, is sent across the network.

The GIOP/IIOP specification allows an ORB to send a null string in the RepoId field of an IOR. It is up to the receiving end to work out the real type of the object. omniORB never sends out null strings as RepoIds, but it may receive null RepoIds from other ORBs. In that case, it will use the mechanism described below to ensure type safety.

14.2 Interface Inheritance

When the ORB receives an IOR of interface type B when it expects the type to be A, it must find out if B inherits from A. When the ORB has no local knowledge of the type B, it must work out the type of B dynamically.

The CORBA specification defines an Interface Repository (IR) from which IDL interfaces can be queried dynamically. In the above situation, the ORB could contact the IR to find out the type of B. However, this approach assumes that an IR is always available and contains the up-to-date information of all the interfaces used in the domain. This assumption may not be valid in many applications.

An alternative is to use the `_is_a()` operation to work out the actual type of an object. This approach is simpler and more robust than the previous one because no 3rd party is involved, so this is what omniORB does.

```
class Object{
    CORBA::Boolean _is_a(const char* type_id);
};
```

The `_is_a()` operation is part of the `CORBA::Object` interface and must be implemented by every object. The input argument is a RepoId. The function returns `true` if the object is really an instance of that type, including if that type is a base type of the most derived type of that object.

In the situation above, the ORB would invoke the `_is_a()` operation on the object and ask if the object is of type A *before* it processes any application invocation on the object.

Notice that the `_is_a()` call is *not* performed when the IOR is unmarshalled. It is performed just prior to the first application invocation on the object. This leads to some interesting failure modes if B reports that it is not an A. Consider the following example:

```
// IDL
interface A { ... };
interface B : A { ... };
interface D { ... };
interface C {
  A      op1();
  Object op2();
};
```

```
1  // C++
2  C_ptr objC = //... an object reference from somewhere;
3  A_ptr objA;
4  CORBA::Object_ptr objR;
5
6  objA =  objC->op1();
7  objA->_non_existent();
8
9  objR =  objC->op2();
10 objA =  A::_narrow(objR);
```

If the stubs of A,B,C,D are linked into the executable and:

Case 1 `C::op1()` and `C::op2()` return a B. Lines 6–10 complete successfully. The remote object is only contacted at line 7.

Case 2 `C::op1()` and `C::op2()` return a D. This condition only occurs if the runtime of the remote end is buggy. Even though the IDL definitions show that D is not derived from A, omniORB gives it the benefit of the doubt, in case it actually has a more derived interface that is derived from both A and D. At line 7, the object is contacted to ask if it is an A. The answer is no, so a `CORBA::INV_OBJREF` exception is raised. At line 10, the narrow operation will fail, and objA will be set to nil.

If only the stubs of A are linked into the executable and:

Case 1 `C::op1()` and `C::op2()` return a B. Lines 6–10 complete successfully. When lines 7 and 10 are executed, the object is contacted to ask if it is an A.

Case 2 `C::op1()` and `C::op2()` return a D. This condition only occurs if the runtime of the remote end is buggy. Line 6 completes and no exception is raised. At line 7, the object is contacted to ask if it is an A. If the answer

is no, a `CORBA::INV_OBJREF` exception is raised. At line 10, the narrow operation will fail, and objA will be set to nil.

Chapter 15

Packaging stubs into DLLs

omniORB's stubs can be packaged into shared libraries or DLLs. On Unix platforms this is mostly painless, but on Windows things are slightly more tricky.

15.1 Dynamic loading and unloading

As long as your platform supports running static initialisers and destructors as libraries are loaded and unloaded, you can package stubs into shared libraries / DLLs, and load them dynamically at runtime.

There is one minor problem with this, which is that normally nil object references are heap allocated, and only deallocated when the ORB is destroyed. That means that if you unload a stub library from which nil references have been obtained (just by creating an object reference _var for example), there is a risk of a segmentation fault when the ORB is destroyed. To avoid that problem, define the `OMNI_UNLOADABLE_STUBS` C pre-processor symbol while you are compiling the stub files. Unfortunately, with that define set, there is a risk that object reference _vars at global scope will segfault as they are unloaded. You must not create _vars at global scope if you are using `OMNI_UNLOADABLE_STUBS`.

15.2 Windows DLLs

On Unix platforms, the linker figures out how to link the symbols exported by a library in to the running program. On Windows, unfortunately, you have to tell the linker where symbols are coming from. This causes all manner of difficulties.

15.2.1 Exporting symbols

To (statically) link with a DLL file in Windows, you link with a LIB file which references the symbols exported from the DLL. To build the LIB and DLL files,

the correct symbols must be exported. One way to do that is to decorate the source code with magic tags that tell the compiler to export the symbols. The alternative is to provide a DEF file that lists all the symbols to be exported. omniORB uses a DEF file.

The question is, how do you create the DEF file? The answer is to use a Python script named `makedeffile.py` that lives in the `bin\scripts` directory in the omniORB distribution. `makedeffile.py` runs the dumpbin program that comes with Visual C++, and processes its output to extract the necessary symbols. Although it is designed for exporting the symbols from omniORB stub files, it can actually be used for arbitrary C++ code. To use it to create a DLL from a single source file, use the following steps:

1. Compile the source:

   ```
   cl -c -O2 -MD -GX -Fofoo.o -Tpfoo.cc
   ```

2. Build a static library (It probably won't work on its own due to the -MD switch to cl, but we just need it to get the symbols out):

   ```
   lib -out:foo_static.lib foo.o
   ```

3. Use the script to build a .def file:

   ```
   makedeffile.py foo_static.lib foo 1.0 foo.def
   ```

4. Build the .dll and .lib with the def file.

   ```
   link -out:foo.dll -dll -def:foo.def -implib:foo.lib foo.o
   ```

Of course, you can link together many separate C++ files, rather than just the one shown here.

15.2.2 Importing constant symbols

As if exporting the symbols from a DLL was not complicated enough, any constant values exported by a DLL have to be explicitly *imported* into the code using them. omniORB's stub files declare a number of such constants. This time, the constant declarations in the generated header files are decorated in a way that tells the compiler what to do. When the stub headers are #included, the correct pre-processor defines must be set. If things are not set correctly, the code all links without problems, but then mysteriously blows up at run time.

Depending on how complex your situation is, there are a range of solutions. Starting with the simplest, here are some scenarios you may find yourself in:

1. All stub code, and all code that uses it is wrapped up in a single DLL.

   Do nothing special.

2. All stub code is in a single DLL. Code using it is in another DLL, or not in a DLL at all.

   `#define USE_stub_in_nt_dll` before `#include` of the stub headers.

3. The stubs for each IDL file are in separate DLLs, one DLL per IDL file.

   In this case, if the IDL files `#include` each other, then when the stub files are compiled, import declarations are needed so that references between the separate DLLs work. To do this, first compile the IDL files with the `-Wbdll_stubs` flag:

   ```
   omniidl -bcxx -Wbdll_stubs example.idl
   ```

   Then define the `INCLUDED_stub_in_nt_dll` pre-processor symbol when compiling the stub files. As above, define `USE_stub_in_nt_dll` when including the stub headers into application code.

4. Stubs and application code are packaged into multiple DLLs, but DLLs contain the stubs for more than one IDL file.

   This situation is handled by 'annotating' the IDL files to indicate which DLLs they will be compiled into. The annotation takes the form of some `#ifdefs` to be inserted in the stub headers. For example,

   ```
   // one.idl

   #pragma hh #ifndef COMPILING_FIRST_DLL
   #pragma hh # ifndef USE_stub_in_nt_dll
   #pragma hh #   define USE_stub_in_nt_dll
   #pragma hh # endif
   #pragma hh #endif

   #include <two.idl>

   module ModuleOne {
     ...
   };


   // two.idl

   #pragma hh #ifndef COMPILING_SECOND_DLL
   #pragma hh # ifndef USE_stub_in_nt_dll
   #pragma hh #   define USE_stub_in_nt_dll
   #pragma hh # endif
   #pragma hh #endif

   #include <three.idl>
   ...
   ```

Here, `one.idl` is packaged into `first.dll` and `two.idl` is in `second.dll`. When compiling `first.dll`, the `COMPILING_FIRST_DLL` define is set, meaning definitions from `one.idl` (and any other files in that DLL) are not imported. Any other module that includes the stub header for `one.idl` does not define `COMPILING_FIRST_DLL`, and thus imports the necessary symbols from the DLL.

Rather than explicitly listing all the pre-processor code, it can be cleaner to use a C++ header file for each DLL. See the COS services IDL files in `idl/COS` for an example.

Chapter 16

Resources

There are a number of useful online resources related to omniORB:

- https://www.omniorb.net/ is the main omniORB web site.
- The omniORB mailing list is the first port of call for questions that are not answered in this document or in the FAQ. Subscription information and archives are at https://www.omniorb.net/list.html
- Commercial support is available from https://www.omniorb-support.com/

Bibliography

[BLFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396, August 1998.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley professional computing series, 1999.

[OMG98] Object Management Group. *CORBAServices: Common Object Services Specification*, December 1998.

[OMG00] Object Management Group. *Interoperable Naming Service revised chapters*, August 2000. From http://www.omg.org/cgi-bin/doc?ptc/00-08-07.

[OMG01] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, December 2001. From http://www.omg.org/cgi-bin/doc?formal/01-12-01.

[OMG03] Object Management Group. *C++ Language Mapping*, 1.1 edition, 2003. From http://www.omg.org/cgi-bin/doc?formal/03-06-03.

[OMG08] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.1 edition, January 2008. From http://www.omg.org/cgi-bin/doc?formal/08-01-04.

[Ric96] Tristan Richardson. *The OMNI Thread Abstraction*. AT&T Laboratories Cambridge, October 1996.