The omniORB version 4.0 User's Guide

Duncan Grisby
(*email: dgrisby@apasphere.com*)

Apasphere Ltd.

Sai-Lai Lo

David Riddoch

AT&T Laboratories Cambridge

October 2004

Changes and Additions, October 2004

- Packaging stubs into DLLs.

Changes and Additions, July 2004

- Minor updates.

Changes and Additions, November 2002

- Per thread timeouts.
- Implement missing interceptors.
- Minor fixes.

Changes and Additions, June 2002

- Updated to omniORB 4.0.

Contents

1 Introduction **1**
- 1.1 Features . . . 1
  - 1.1.1 Multithreading . . . 1
  - 1.1.2 Portability . . . 2
  - 1.1.3 Missing features . . . 2
- 1.2 Setting Up Your Environment . . . 3
- 1.3 Platform specific variables . . . 3

2 The Basics **5**
- 2.1 The Echo Object Example . . . 5
- 2.2 Specifying the Echo interface in IDL . . . 6
- 2.3 Generating the C++ stubs . . . 6
- 2.4 Object References and Servants . . . 7
- 2.5 A Quick Tour of the C++ stubs . . . 7
  - 2.5.1 Servant Object Implementation . . . 10
- 2.6 Writing the servant implementation . . . 11
- 2.7 Writing the client . . . 12
- 2.8 Example 1 — Colocated Client and Implementation . . . 13
  - 2.8.1 ORB initialisation . . . 14
  - 2.8.2 Obtaining the Root POA . . . 14
  - 2.8.3 Object initialisation . . . 15
  - 2.8.4 Activating the POA . . . 15
  - 2.8.5 Performing a call . . . 16
  - 2.8.6 ORB destruction . . . 16
- 2.9 Example 2 — Different Address Spaces . . . 16
  - 2.9.1 Object Implementation: Generating a Stringified Object Reference . . . 16
  - 2.9.2 Client: Using a Stringified Object Reference . . . 17
  - 2.9.3 Catching System Exceptions . . . 18
  - 2.9.4 Lifetime of a CORBA object . . . 18
- 2.10 Example 3 — Using the Naming Service . . . 19
  - 2.10.1 Obtaining the Root Context Object Reference . . . 19
  - 2.10.2 The Naming Service Interface . . . 20

2.11 Example 4 — Using tie implementation templates . . . 20
2.12 Source Listings . . . 21
2.12.1 eg1.cc . . . 21
2.12.2 eg2_impl.cc . . . 24
2.12.3 eg2_clt.cc . . . 26
2.12.4 eg3_impl.cc . . . 28
2.12.5 eg3_clt.cc . . . 32
2.12.6 eg3_tieimpl.cc . . . 35

3 C++ language mapping **39**
3.1 omniORB 2 BOA compatibility . . . 39
3.2 omniORB 3.0 compatibility . . . 40

4 omniORB configuration and API **41**
4.1 Setting parameters . . . 41
4.1.1 Command line arguments . . . 41
4.1.2 ORB_init() parameter . . . 41
4.1.3 Environment variables . . . 42
4.1.4 Configuration file . . . 42
4.1.5 Windows registry . . . 42
4.2 Tracing options . . . 42
4.2.1 Tracing API . . . 43
4.3 Miscellaneous global options . . . 44
4.4 Client side options . . . 45
4.5 Server side options . . . 47
4.5.1 Main thread selection . . . 49
4.6 GIOP and interoperability options . . . 49
4.7 System Exception Handlers . . . 51
4.7.1 Minor codes . . . 51
4.7.2 CORBA::TRANSIENT handlers . . . 51
4.7.3 CORBA::COMM_FAILURE . . . 53
4.7.4 CORBA::SystemException . . . 54
4.8 Location forwarding . . . 54

5 The IDL compiler **57**
5.1 Common options . . . 57
5.1.1 Preprocessor interactions . . . 58
5.1.1.1 Windows 9x . . . 58
5.1.2 Forward-declared interfaces . . . 58
5.1.3 Comments . . . 58
5.2 C++ back-end options . . . 59
5.2.1 Stub / skeleton files . . . 59
5.2.2 Module splicing . . . 60
5.2.3 Flattened tie classes . . . 61

5.2.4 Generating example implementations . . . 61
5.3 Examples . . . 61

6 Interoperable Naming Service **63**
6.1 Object URIs . . . 63
6.1.1 corbaloc . . . 63
6.1.2 corbaname . . . 64
6.2 Configuring resolve_initial_references . . . 65
6.2.1 ORBInitRef . . . 65
6.2.2 ORBDefaultInitRef . . . 65
6.3 omniNames . . . 66
6.3.1 NamingContextExt . . . 66
6.3.2 Use with corbaname . . . 67
6.4 omniMapper . . . 67
6.5 Creating objects with simple object keys . . . 67

7 Interface Type Checking **69**
7.1 Introduction . . . 69
7.2 Interface Inheritance . . . 70

8 Connection and Thread Management **73**
8.1 Background . . . 73
8.2 The model . . . 74
8.3 Client side behaviour . . . 74
8.3.1 Client side timeouts . . . 75
8.4 Server side behaviour . . . 75
8.4.1 Thread per connection mode . . . 75
8.4.2 Thread pool mode . . . 76
8.4.3 Policy transition . . . 76
8.5 Idle connection shutdown . . . 77
8.5.1 Interoperability Considerations . . . 77
8.6 Transports and endpoints . . . 78
8.6.1 End point publishing . . . 79
8.7 Connection selection and acceptance . . . 79
8.7.1 Client transport rules . . . 79
8.7.2 Server transport rules . . . 80
8.8 Bidirectional GIOP . . . 81
8.9 SSL transport . . . 81

9 Code set conversion **83**
9.1 Native code sets . . . 83
9.2 Code set library . . . 83
9.3 Implementing new code sets . . . 84

10 Interceptors **85**
10.1 Interceptor registration 86
10.2 Available interceptors 86

11 Type Any and TypeCode **89**
11.1 Example using type Any 89
11.1.1 Type Any in IDL 90
11.1.2 Inserting and Extracting Basic Types from an Any 90
11.1.3 Inserting and Extracting Constructed Types from an Any . . . 91
11.2 Type Any in omniORB 93
11.3 TypeCode in omniORB 95
11.4 Source Listing 97
11.4.1 anyExample_impl.cc 97
11.4.2 anyExample_clt.cc 100

12 Packaging stubs into DLLs **103**
12.1 Dynamic loading and unloading 103
12.2 Windows DLLs 103
12.2.1 Exporting symbols 103
12.2.2 Importing constant symbols 104

Chapter 1

Introduction

omniORB is an Object Request Broker (ORB) that implements the 2.6 specification of the Common Object Request Broker Architecture (CORBA) [OMG01][1]. It has 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[2].

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 setup your environment to run omniORB.

1.1 Features

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.

New in version 4.0, omniORB also supports a flexible thread pooling policy,

[1] Most of the 2.6 features have been implemented. The features still missing in this release are listed in section 1.1.3. Where possible, backward compatibility has been maintained up to specification 2.0.

[2] More information can be found at `http://www.opengroup.org/press/7jun99_b.htm`

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 has always been designed to be portable. It runs on many flavours of Unix, Windows, several embedded operating systems, and relatively obscure systems such as OpenVMS, 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 exception is the mapping of IDL modules, which can use either namespaces or nested classes.

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.

1.1.3 Missing features

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

- omniORB does not have its own Interface Repository. However, it can act as a client to an IfR. The omniifr project (`http://omniifr.sourceforge.net/`) aims to create an IfR for omniORB.
- Objects by value (IDL valuetype) is not supported in this release.
- Local interfaces are not supported.
- omniORB supports interceptors, but not the standard Portable Interceptor API.

These features may be implemented in the short to medium term. It is best to check out the latest status on the omniORB home page (`http://omniorb.sourceforge.net/`).

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. 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 set-up. 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 platforms (Windows NT, 2000, 95, 98), 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 6.1.2 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 NT 4.0,2000,XP	__x86__	__NT__	__OSVERSION__=4	__WIN32__
Windows NT 3.5	__x86__	__NT__	__OSVERSION__=3	__WIN32__
Windows 95	__x86__	__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. For instance, the platform configuration file for Sun Solaris 2.6 is in `mk/platforms/sun4_sosV_5.6.mk`. 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[1];
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 one directory above that in `src/examples`.

[1]The stub code is the C++ code that provides the object mapping as defined in the CORBA specification.

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 should have the extension '`.idl`', although omniORB does not enforce this.

For simplicity, the interface is defined in the global IDL namespace. You should 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
```

If you are using our make environment (ODE), 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 ODE to generate the stubs. Remember, 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 4 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.

Warning

omniORB 2.x *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. If you have code which relies on this, it will not compile with omniORB 3.x or 4.x, even under the BOA compatibility mode.

2.5 A Quick Tour of the C++ stubs

The C++ stubs conform to the mapping defined in the CORBA specification [OMG03]. It is important to understand the mapping before you start writing any serious CORBA applications. Before going any further, it is worth knowing what the mapping looks like.

For the example interface `Echo`, the C++ mapping for its object reference is `Echo_ptr`. The type is defined in `echo.hh`. The relevant section of the code is reproduced below. The stub code produced by other ORBs will be functionally equivalent to omniORB's, but will almost certainly look very different.

```
class Echo;
class _objref_Echo;
class _impl_Echo;
typedef _objref_Echo* Echo_ptr;

class Echo {
public:
  // Declarations for this interface type.
  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();

  // ... methods generated for internal use
```

```
};

class _objref_Echo :
  public virtual CORBA::Object, public virtual omniObjRef {
public:
  char * echoString(const char* mesg);

  // ... methods generated for internal use
};
```

In a compliant application, the operations defined in an object interface should **only** be invoked via an object reference. This is done by using arrow ('->') on an object reference. For example, the call to the operation `echoString()` would be written as `obj->echoString(mesg)`.

It should be noted that the concrete type of an object reference is opaque, i.e. you must not make any assumption about how an object reference is implemented. In our example, even though `Echo_ptr` is implemented as a pointer to the class `_objref_Echo`, it should not be used as a C++ pointer, i.e. conversion to `void*`, arithmetic operations, and relational operations including testing for equality using `operator==`, must not be performed on the type.

In addition to class `_objref_Echo`, the mapping defines three static member functions in the class `Echo`: `_nil()`, `_duplicate()`, and `_narrow()`.

The `_nil()` function returns a nil object reference of the Echo interface. The following call is guaranteed to return TRUE:

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

Remember, `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; *omniORB does not*.

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. Duplications are required to satisfy the C++ mapping's reference counting memory management.

All CORBA objects inherit from the generic object `CORBA::Object`. `CORBA::Object_ptr` is the object reference type for `CORBA::Object`. Any `_ptr` object reference is therefore conceptually inherited from `CORBA::Object_ptr`. In other words, an object reference such as `Echo_ptr` can be used in places where a `CORBA::Object_ptr` is expected.

The `_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 `COMM_FAILURE` or `OBJECT_NOT_EXIST`.

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

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

Once you have called `CORBA::release()` on an object reference, you must no longer use that reference. This is because the associated resources may have been deallocated. Notice 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.

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(B);
// Note: the above call is guaranteed to be TRUE
```

You have now been introduced to most of the operations that can be invoked via `Echo_ptr`. The generic object `CORBA::Object` provides a few more operations and all of them can be invoked via `Echo_ptr`. These operations deal mainly with CORBA's dynamic interfaces. You do not have to understand them in order to use the C++ mapping provided via the stubs.

Since object references must be released explicitly, their usage is prone to error and can lead to memory leakage. The mapping defines the *object reference variable* type to make life easier. In our example, the variable type `Echo_var` is defined[2].

The `Echo_var` is more convenient to use because it will automatically release its object reference when it is deallocated 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 castings[3]. For instance, the operation `echoString()` 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;
```

[2]In omniORB, all object reference variable types are instantiated from the template type `_CORBA_ObjRef_Var`.

[3]However, the implementation of the type conversion operator between `Echo_var` and `Echo_ptr` varies slightly among different C++ compilers; you may need to do an explicit cast if the compiler complains about the conversion being ambiguous.

```
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.
```

2.5.1 Servant Object Implementation

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. 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. See section 3.1 for details of the BOA compatibility, and problems you may encounter.

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()`
: 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 an implementation 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 stubs' skeleton class 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 PortableServer::RefCountServantBase
{
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
: A string, which is used both as an in argument and the return value of `echoString()`, is a variable size data type. Other examples of variable size 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, please refer to 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.

Reference Counting

As well as inheriting from the Echo skeleton class, the servant class is also derived from `PortableServer::RefCountServantBase` which, as the name suggests, is a mixin class which provides reference counting for the servant object. This 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 that derive from `PortableServer::RefCountServantBase` must not be instantiated as automatic variables (i.e. on the stack). Instead, you should always instantiate them using the `new` operator, i.e. their storage is allocated on the heap. Otherwise, the POA may attempt to delete an object on the stack.

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   cerr << "I said,\"" << src << "\"."
18        << " The Object said,\"" << dest <<"\"" << endl;
19 }
```

Briefly, 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 `cerr`.

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*`)[4]. 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[5].

Under the C++ mapping, `T_var` types are provided for all the non-basic data types. It is obvious that 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 Implementation

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 {
4   CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB4");
5
6   CORBA::Object_var       obj = orb->resolve_initial_references("RootPOA");
```

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

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

```
7    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);
8
9    Echo_i *myecho = new Echo_i();
10   PortableServer::ObjectId_var myechoid = poa->activate_object(myecho);
11
12   Echo_var myechoref = myecho->_this();
13   myecho->_remove_ref();
14
15   PortableServer::POAManager_var pman = poa->the_POAManager();
16   pman->activate();
17
18   hello(myechoref);
19
20   orb->destroy();
21   return 0;
22 }
```

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 4.0, this argument must be 'omniORB4'[6].

`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 creating 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.

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

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.

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 `_this()`. 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.

Line 13

The server code releases the reference it holds to the servant object. The only reference to that object is now held by the POA (it gained the reference on the call to `activate_object()`), so when the object is deactivated (or the POA is destroyed), the servant object will be deleted automatically. After this point, the code must no longer use the `myecho` pointer.

2.8.4 Activating the POA

Lines 15–16

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 18

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 20

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.

This call is particularly important when writing a CORBA DLL on Windows NT that is to be used from ActiveX. If this call is absent, the application will hang when the CORBA DLL is unloaded.

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 Object Implementation: Generating 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    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 << "'" << (char*)sior << "'" << endl;
15
16   myecho->_remove_ref();
17
18   PortableServer::POAManager_var pman = poa->the_POAManager();
19   pman->activate();
20
21   orb->run();
22   orb->destroy();
23   return 0;
24 }
```

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 some hexadecimal digits. All CORBA 2 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::COMM_FAILURE& ex) {
  ... // code to handle communication failure
}
```

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[7]. For instance, the code fragment, shown in section 2.9.2, catches the `COMM_FAILURE` system exception which indicates that communication with the object implementation in another address space has failed.

All system exceptions inherit from `CORBA::SystemException`. With compilers that properly support RTTI[8], a single catch of `CORBA::SystemException` will catch all the different system exceptions thrown by omniORB.

When omniORB detects an internal error such as corrupt data or invalid conditions, it raises the exception `omniORB::fatalException`. When this exception is raised, it is not sensible to proceed with any operation that involves the ORB's runtime. It is best to exit the program immediately. The exception structure carried by `omniORB::fatalException` contains the exact location (the file name and the line number) where the exception is raised. In most cases, `fatalException`s occur due to incorrect behaviour by the application code, but they may be caused by bugs in omniORB.

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 servants are active, or can be activated by their POA with a servant manager[9]. A reference to a persistent object can be published, and will remain valid even if the server process is restarted.

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

[7] 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.

[8] Run Time Type Identification

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

Naming Service.

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 far 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 a naming context. 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[10]. 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 initServ;
initServ = orb->resolve_initial_references("NameService");

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

[10]There are escaping rules to cope with id and kind fields which contain '.' and '/' characters. See chapter 6 of this manual, and chapter 3 of the CORBA services specification, as updated for the Interoperable Naming Service [OMG00].

Remember, 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, 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. Please spend time to study the examples carefully.

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 stub 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 <iostream.h>
#include <echo.hh>


// This is the object implementation.

class Echo_i : public POA_Echo,
               public PortableServer::RefCountServantBase
{
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);
}

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

// 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!\n" << 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);

  cerr << "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 object on the heap.  Since this is a reference
      // counted object, it will be deleted by the POA when it is no
      // longer needed.
    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();

    // Decrement the reference count of the object implementation, so
    // that it will be properly cleaned up when the POA has determined
    // that it is no longer needed.
    myecho->_remove_ref();

    // 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::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE -- unable to contact the "
         << "object." << endl;
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught CORBA::SystemException." << endl;
  }
  catch(CORBA::Exception&) {
    cerr << "Caught CORBA::Exception." << endl;
  }
  catch(omniORB::fatalException& fe) {
    cerr << "Caught omniORB::fatalException:" << endl;
    cerr << "  file: " << fe.file() << endl;
    cerr << "  line: " << fe.line() << endl;
    cerr << "  mesg: " << fe.errmsg() << endl;
  }
  catch(...) {
    cerr << "Caught unknown exception." << 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 cerr as a
//        stringified IOR. This string should be used as the argument to
//        eg2_clt.
//

#include <iostream.h>
#include <echo.hh>


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


char* Echo_i::echoString(const char* mesg)
{
  cerr << "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);

    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));
    cerr << "'" << (char*)sior << "'" << endl;

    myecho->_remove_ref();

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

    orb->run();
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught CORBA::SystemException." << endl;
  }
  catch(CORBA::Exception&) {
    cerr << "Caught CORBA::Exception." << endl;
  }
  catch(omniORB::fatalException& fe) {
    cerr << "Caught omniORB::fatalException:" << endl;
    cerr << "  file: " << fe.file() << endl;
    cerr << "  line: " << fe.line() << endl;
    cerr << "  mesg: " << fe.errmsg() << endl;
  }
  catch(...) {
    cerr << "Caught unknown exception." << 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 <iostream.h>
#include <echo.hh>


static void hello(Echo_ptr e)
{
  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);

    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::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE -- unable to contact the "
         << "object." << endl;
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught a CORBA::SystemException." << endl;
  }
  catch(CORBA::Exception&) {
    cerr << "Caught CORBA::Exception." << endl;
  }
  catch(omniORB::fatalException& fe) {
    cerr << "Caught omniORB::fatalException:" << endl;
    cerr << "  file: " << fe.file() << endl;
    cerr << "  line: " << fe.line() << endl;
    cerr << "  mesg: " << fe.errmsg() << endl;
  }
  catch(...) {
    cerr << "Caught unknown exception." << 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 <iostream.h>
#include <echo.hh>


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


class Echo_i : public POA_Echo,
               public PortableServer::RefCountServantBase
{
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);

    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 x;
    x = orb->object_to_string(obj);
    cerr << x << "\n";

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

    myecho->_remove_ref();

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

    orb->run();
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught CORBA::SystemException." << endl;
  }
  catch(CORBA::Exception&) {
    cerr << "Caught CORBA::Exception." << endl;
  }
  catch(omniORB::fatalException& fe) {
    cerr << "Caught omniORB::fatalException:" << endl;
    cerr << "  file: " << fe.file() << endl;
    cerr << "  line: " << fe.line() << endl;
    cerr << "  mesg: " << fe.errmsg() << endl;
  }
  catch(...) {
    cerr << "Caught unknown exception." << 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;
    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::ORB::InvalidName& ex) {
    // 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
    // Note on kind: The kind field is used to indicate the type
    // of the object. This is to avoid conventions such as that used
    // by files (name.type -- e.g. test.ps = postscript etc.)

    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;
      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.

    // Amendment: When using OrbixNames, it is necessary to first try bind
    // and then rebind, as rebind on it's own will throw a NotFoundexception if
    // the Name has not already been bound. [This is incorrect behaviour -
    // it should just bind].
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE -- unable to contact the "
         << "naming service." << endl;
    return 0;
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught a CORBA::SystemException 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 COSS 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 <iostream.h>
#include <echo.hh>

static CORBA::Object_ptr getObjectReference(CORBA::ORB_ptr orb);

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);
```

```
    CORBA::Object_var obj = getObjectReference(orb);

    Echo_var echoref = Echo::_narrow(obj);

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

    orb->destroy();
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE -- unable to contact the "
         << "object." << endl;
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught CORBA::SystemException." << endl;
  }
  catch(CORBA::Exception&) {
    cerr << "Caught CORBA::Exception." << endl;
  }
  catch(omniORB::fatalException& fe) {
    cerr << "Caught omniORB::fatalException:" << endl;
    cerr << "  file: " << fe.file() << endl;
    cerr << "  line: " << fe.line() << endl;
    cerr << "  mesg: " << fe.errmsg() << endl;
  }
  catch(...) {
    cerr << "Caught unknown exception." << endl;
  }

  return 0;
}

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

static CORBA::Object_ptr
getObjectReference(CORBA::ORB_ptr orb)
{
  CosNaming::NamingContext_var rootContext;

  try {
    // Obtain a reference to the root context of the Name service:
    CORBA::Object_var obj;
    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 CORBA::Object::_nil();
    }
  }
  catch(CORBA::ORB::InvalidName& ex) {
    // This should not happen!
    cerr << "Service required is invalid [does not exist]." << endl;
    return CORBA::Object::_nil();
  }

  // Create a name object, containing the name test/context:
  CosNaming::Name name;
  name.length(2);

  name[0].id   = (const char*) "test";       // string copied
  name[0].kind = (const char*) "my_context"; // string copied
  name[1].id   = (const char*) "Echo";
  name[1].kind = (const char*) "Object";
  // Note on kind: The kind field is used to indicate the type
  // of the object. This is to avoid conventions such as that used
  // by files (name.type -- e.g. test.ps = postscript etc.)

  try {
    // Resolve the name to an object reference.
    return rootContext->resolve(name);
  }
  catch(CosNaming::NamingContext::NotFound& ex) {
    // This exception is thrown if any of the components of the
    // path [contexts or the object] aren't found:
    cerr << "Context not found." << endl;
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE -- unable to contact the "
         << "naming service." << endl;
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught a CORBA::SystemException while using the naming service."
         << endl;
  }

  return CORBA::Object::_nil();
}
```

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 <iostream.h>
#include <echo.hh>


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


// This is the object implementation.  Notice that it does not
// inherit from any stub class.

class Echo_i {
public:
  inline Echo_i() {}
  inline ~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);

    // Note that the <myecho> object is constructed on the stack here.
    // This is because tie implementations do not inherit from the
    // PortableServer::RefCountServantBase mixin class -- and so are
    // not automatically deleted by the POA.
    // However, 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 objects are not deleted
    // whilst still activated.
    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&) {
    cerr << "Caught CORBA::SystemException." << endl;
  }
  catch(CORBA::Exception&) {
    cerr << "Caught CORBA::Exception." << endl;
  }
  catch(omniORB::fatalException& fe) {
    cerr << "Caught omniORB::fatalException:" << endl;
    cerr << "  file: " << fe.file() << endl;
    cerr << "  line: " << fe.line() << endl;
    cerr << "  mesg: " << fe.errmsg() << endl;
  }
  catch(...) {
    cerr << "Caught unknown exception." << 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;
    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::ORB::InvalidName& ex) {
    // 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
    // Note on kind: The kind field is used to indicate the type
    // of the object. This is to avoid conventions such as that used
    // by files (name.type -- e.g. test.ps = postscript etc.)

    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;
      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.

    // Amendment: When using OrbixNames, it is necessary to first try bind
    // and then rebind, as rebind on it's own will throw a NotFoundexception if
    // the Name has not already been bound. [This is incorrect behaviour -
    // it should just bind].
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE -- unable to contact the "
         << "naming service." << endl;
    return 0;
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught a CORBA::SystemException while using the naming service."
         << endl;
    return 1;
  }

  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 that has begun to appear. For instance, Henning and Vinoski's 'Advanced CORBA Programming with C++' [HV99] includes many example code bits to illustrate how to use the C++ mapping.

3.1 omniORB 2 BOA compatibility

If you use the `-WbBOA` option to omniidl, it will generate skeleton code with 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 4.0'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 4.0.

- 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 4.0 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 4.0 uses the new mapping. The changes are largely syntax changes, rather than semantic differences.

Chapter 4

omniORB configuration and API

omniORB 4.0 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 arrays of strings, 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
```

Note how 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`.

`traceThreadId` *default* = `0`

If `traceThreadId` is set true, all trace messages are prefixed with the id of the thread outputting the message. This can be handy for tracking down race conditions, but it adds significant overhead to the logging function so it is turned off by default.

4.2.1 Tracing API

The four 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 traceThreadId;
};
```

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 8, 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.

`omniORB_27_CompatibleAnyExtraction` *default* = `0`

In omniORB 2.7 and earlier, the Any extraction operators for strings and object references returned a copy/duplicate of the value stored in the Any, meaning the caller had to release the extracted value. The C++ mapping was clarified to state that the Any retains ownership of the returned value, so the caller does not need to release it. To aid the transition for ancient programs from omniORB 2.7 or earlier, setting this flag to true selects the old copying behaviour.

`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.

4.4 Client side options

These options control aspects of client-side behaviour.

`InitRef` *default* = *none*

Specify the objects available from `ORB::resolve_initial_references()`. The arguments take the form <*key*>=<*uri*>, where the *key* is the name given to `resolve_initial_references()` and *uri* is a valid CORBA object reference URI, as detailed in chapter 6.

`DefaultInitRef` *default* = *none*

Specify the default URI prefix for `resolve_initial_references()`, as explained in chapter 6.

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

Used to specify the way the client contacts a server, depending on the server's address. See section 8.7.1 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 and raises a `TRANSIENT` exception. A value of zero means no timeout; calls can block for ever. See section 8.3.1 for more information about timeouts.

Note: omniORB 3 had timeouts specified in seconds; omniORB 4.0 uses milliseconds for 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.

`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 8.5.

`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 `maxGIOPConnection PerServer`. 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 calls on a single connection.

`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 8.8.

`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`.

`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::`
`endPointNoListen`
`endPointNoPublish`
`endPointPublishAllIFs`

These options determine the end-points the ORB should listen on, and the details that should be published in IORs. See chapter 8 for details.

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

Configure the rules about whether a server should accept an incoming connection from a client. See section 8.7.2 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. If a connection has been idle for this amount of time, the ORB closes it. See section 8.5.

`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`

If true, threads from the pool temporarily behave a bit like thread per connection after dispatching a call. See section 8.4.2 for details.

`acceptBiDirectionalGIOP` *default* = `0`

Determines whether a server will ever accept clients' offers of bidirectional GIOP connections. See section 8.8.

`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 4.0 supports the `PortableServer::Current` interface that provides thread context information to a servant. Supporting current has a small but no-

ticeable 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 `TRANSIENT` exception. Zero means no timeout.

`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.

`maxGIOPVerson` *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.

`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. Note that omniORB will always remove top-level aliases, but will not remove aliases from TypeCodes that are members of other TypeCodes (e.g. TypeCodes for members of structs etc.), unless `tcAliasExpand` is set to 1. 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 ORBs do not support them. Setting this flag to false prevents the use of indirections (and, therefore, 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 versions of Visibroker's Java ORB.

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::COMM_FAILURE` and `CORBA::SystemException`. This last handler covers all system exceptions other than the two covered by the first two 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

A new feature in omniORB 4.0 is 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. In compilers with namespace support, the minor code constants appear in namespace `omni`; otherwise 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 function installTransientExceptionHandler() can be 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 installTransientException Handler(). 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 returns a boolean value. If the return value is TRUE(1), the proxy object retries the operation. If the return value is FALSE(0), the `CORBA::TRANSIENT` exception is propagated into the application code.

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 1;           // retry immediately.
}

CORBA::Boolean my_transient_handler2 (void* cookie,
                                      CORBA::ULong retries,
                                      const CORBA::TRANSIENT& ex)
{
   cerr << "transient handler 2 called." << endl;
   return 1;           // retry immediately.
}


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::COMM_FAILURE

If the ORB has successfully contacted an object 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:

```
class omniORB {
public:
```

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

static void installCommFailureExceptionHandler(void* cookie,
                                               commFailureExceptionHandler_t fn);

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

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

4.7.4 CORBA::SystemException

If a system exception other than `CORBA::TRANISENT` or `CORBA::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:

```
class omniORB {
public:

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

static void installSystemExceptionHandler(void* cookie,
                                          systemExceptionHandler_t fn);

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

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

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 1*> <*file 2*> ...

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. Unlike the old IDL compiler, which used different C preprocessors on different platforms, 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.

5.1.1.1 Windows 9x

The output from the C preprocessor is normally fed to the omniidl parser through a pipe. On some Windows 98 machines (but not all!) the pipe does not work, and the preprocessor output is echoed to the screen. When this happens, the omniidl parser sees an empty file, and produces useless stub files with strange long names. To avoid the problem, use the '`-T`' option to create a temporary file between the two stages.

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.
`-Wbinline`	Output stubs for `#included` IDL files in line with the main file.
`-Wbtp`	Generate 'tie' implementation skeletons.
`-Wbtf`	Generate flattened 'tie' implementation skeletons.
`-Wbsplice-modules`	Splice together multiply-opened modules into one.
`-Wbexample`	Generate example implementation code.
`-WbF`	Generate code fragments (for experts only).
`-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_`'.
`-Wbkeep_inc_path`	Preserve IDL '`#include`' paths in generated '`#include`' directives.
`-Wbuse_quotes`	Use quotes in '`#include`' directives (e.g. `"foo"` rather than `<foo>`.)

Again, most of these are self-explanatory.

5.2.1 Stub / skeleton files

By default, omniidl separates the normal stub and skeleton file (the `SK.cc` file) from the '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. New in omniORB 4.0, 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
```

5.2.2 Module splicing

On 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 {};
};
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.

5.2.3 Flattened tie classes

Another problem with mapping IDL modules to C++ classes arises 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 with compilers without namespace support.

The standard solution is to produce 'flattened' tie class names, using the `-Wbtf` command line argument. With that flag, the template class is declared at global scope with the name `POA_M_I_tie`. i.e. all occurrences of '`::`' are replaced by '`_`'.

5.2.4 Generating 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.

5.3 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

Interoperable Naming Service

omniORB 4.0 supports the Interoperable Naming Service (INS). The following is a summary of its facilities.

6.1 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. IOR-format strings are now also considered URIs.

6.1.1 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.

Alternatively, to use `resolve_initial_references()`, you use a URI of the form:

```
corbaloc:rir:/NameService
```

6.1.2 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:

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

```
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.

6.2 Configuring resolve_initial_references

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

6.2.1 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...
```

6.2.2 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
```

6.3 omniNames

6.3.1 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. 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.

6.3.2 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.

6.4 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 IORs. 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 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
```

6.5 Creating objects with simple 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 with a special POA with the name '`omniINSPOA`', acquired from `resolve_initial_references()`. This POA has the `USER_ID` and `PERSISTENT` policies, and the special property that the object keys it creates contain only the object ids given to the POA, and no other data. It is a normal POA in all other respects, so you can activate/deactivate it, create children, and so on, in the usual way.

Chapter 7

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.

7.1 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 {
   void op(in A arg);
};
```

[1]For further details of the repository ID formats, see section 10.6 in the CORBA 2.3 specification.

```
// 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 RepoId, but it may receive null RepoIds from other ORBs. In that case, it will use the mechanism described below to ensure type safety.

7.2 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(1) 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;
3  A_ptr objA;
4  CORBA::Object_ptr objR;
5
6  objA =  objC->op1();
7  (void) 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. The ORB raises a `CORBA::MARSHAL` exception at line 1 because it knows it has received an interface of the wrong type.

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. The application will also see a `CORBA::INV_OBJREF` at line 10.

Chapter 8

Connection and Thread Management

This chapter describes how omniORB manages threads and network connections.

8.1 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 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.

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

8.2 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 ORB 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.

8.3 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 8.7.1). 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 to multiplex calls on a single connection, set the `oneCallPerConnection` parameter to zero.

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.

8.3.1 Client side timeouts

omniORB can associate a timeout with a call, meaning that if the call takes too long a `TRANSIENT` exception 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);
};
```

`setClientCallTimeout()` sets either the global timeout or the timeout for a specific object reference. `setClientThreadCallTimeout()` sets the timeout for the calling thread. The calling thread must have an `omni_thread` associated with it. Setting any timeout value to zero disables 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.

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.

8.4 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.

8.4.1 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 multiplexed calls.

8.4.2 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 selected with the `maxServerThreadPool Size` 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 `threadPoolWatch Connection` parameter is set to zero, connection watching is disabled and threads return to the pool immediately after finishing a single request.

8.4.3 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 `threadPerConnection UpperLimit` and `threadPerConnectionLowerLimit` parameters. The former must always be larger than the latter. 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.

8.5 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 it runs out of spare communication channels. For example, most Unix platforms have a limit on the number of file handles a process can open. 64 is the usual default limit. The value can 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.

8.5.1 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 work-

around is either to catch the exception in the application code and retry, or to turn off the idle connection shutdown inside the omniORB server.

8.6 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 an SSL 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. End point 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 end points have the format:

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

The host must be a valid host name 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 published 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 a port.

Multiple TCP end points 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 network interface, the ORB listens on all interfaces, and the OS chooses a port number.

SSL end points have the same format as TCP ones, except '`tcp`' is replaced with '`ssl`'. Unix domain socket end points 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.

8.6.1 End point publishing

To publish an end point in IORs, without actually listening on that end point, the `endPointNoListen` parameter can be set. This can be useful in fault-tolerant applications where replicas of an object can be contacted at more than one server. `endPointNoListen` does not check that the transport specified is sensible for the current machine, so it allows the address of a different machine to be specified.

Similarly, but less likely to be useful, it is possible to ask the server to listen on an end point, but not publish the details in IORs, using the `endPointNoPublish` parameter. This should *not* be used for security by obscurity!

If a machine has multiple TCP network interfaces, it may be useful to publish all interfaces, instead of just the first one. This is necessary if different interfaces are on separate non-gatewayed subnets, for example. Publishing all addresses could be achieved with lots of `endPoint` parameters, but a short-hand is to set the `endPointPublishAllIFs` parameter to 1. That (in conjunction with a '`giop:tcp::`' transport selection without a specific hostname) causes all the machine's non-loopback interfaces to be published in IORs.

8.7 Connection selection and acceptance

In the face of IORs containing details about multiple different end points, 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*.

8.7.1 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 the bits selected by the mask, e.g. 172.16.0.0/255.240.0.0.
3. `*` — 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. `bidir` Any connection to this address should be used bidirectionally (see section 8.8)

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

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

8.7.2 Server transport rules

The server transport rules gave 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 only accept 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
```

8.8 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`.

8.9 SSL transport

omniORB 4.0 supports an 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.

Chapter 9

Code set conversion

omniORB 4.0 supports full code set negotiation, used to select and translate between different character code sets, for the transmission of 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 most applications, the defaults are fine. Some applications may need 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 first plane without having to use UTF-16 surrogates[1].

9.2 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

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

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.

9.3 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, and 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 4.0 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:
      giopStream&              giopstream;
      const omniIOR&           ior;
      const char*              opname;
      CORBA::Boolean           oneway;
      CORBA::Boolean           response_expected;
      IOP::ServiceContextList  service_contexts;

      info_T(giopStream& s, const omniIOR& i, const char* op,
             CORBA::Boolean ow, CORBA::Boolean re) :
        giopstream(s),ior(i),opname(op),oneway(ow),response_expected(re),
        service_contexts(5) {}

    private:
      info_T();
      info_T(const info_T&);
      info_T& operator=(const info_T&);
    };

    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 several omniORB internal data types. The definitions of these 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 that singleton with the `omniORB::getInterceptors()` function, which must be called after `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 `extractSupportedComponents()` in `src/lib/omniORB/orbcore/ior.cc` for an example.

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 `setCodeSetService Context()` 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.

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.

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;
};
```

The interceptor function 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.

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.

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.

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

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

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. This 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.

Warning

Since 2.8.0, omniORB has been updated to CORBA 2.3 and above. In order to comply with the 2.3 specification, it was necessary to change the semantics of *the extraction of string, object reference and typecode from an Any*. The memory of the extracted values of these types now belongs to the Any value. The storage is freed when the Any value is deallocated. Previously the extracted value was a copy and the application was responsible for releasing the storage. It is not possible to detect the old usage at compile time. In particular, unmodified code that uses the affected Any extraction operators will most certainly cause runtime errors to occur. To smooth the transition from the old usage to the new, an ORB configuration parameter `omniORB_27_CompatibleAnyExtraction` can be set to revert the Any extraction operators to the old semantics.

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. The source code for this example is included in the omniORB distribution, in the directory `src/examples/anyExample`. A listing of the source code is provided at the end of this chapter.

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;     // From CORBA 2.3 onwards, uses const char*
                     // instead of char*.

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;
    // Since 2.8.0 the storage of the extracted string is still
    // owned by the any.
    // In pre-omniORB 2.8.0 releases, the string returned is a copy.
}
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;    // From CORBA 2.3 onwards, use
                         // const testStruct* instead of testStruct*

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.

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.

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[1].

Examples:

```
// 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.

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. The treatment of top-level aliases from omniORB 2.8.0 onwards is different from pre-omniORB 2.8.0 releases.

In pre-omniORB 2.8.0 releases, if there are any top-level `tk_alias` TypeCodes in the TypeCode, they will be removed from the TypeCode stored in the Any. Note

[1] In pre-omniORB 2.8.0 releases, omniORB performs an equality test and will ignore any alias TypeCodes (`tk_alias`) when making this comparison. The semantics is similar to the `equivalent()` test in the TypeCode interface of CORBA 2.3.

that this does not affect the _tc_ TypeCode generated to represent the type (see section on TypeCode, below). This behaviour is necessary, as two types that differ only by a top-level alias can use the same insertion and extraction operators. If the tk_alias is not removed, one of the types could be transmitted with an incorrect tk_alias TypeCode. Example:

```
// IDL 3
typedef sequence<double> seqDouble1;
typedef sequence<double> seqDouble2;
typedef seqDouble2       seqDouble3;
```

If either seqDouble1 or seqDouble2 is inserted into an Any, the TypeCode stored in the Any will be for a sequence<double>, and not for an alias to a sequence<double>.

From omniORB 2.8.0 onwards, there are two changes. Firstly, in the example, seqDouble1 and seqDouble2 are now distinct types and therefore each has its own set of C++ operators for Any insertion and extraction. Secondly, the top level aliases are not removed. For example, if seqDouble3 is inserted into an Any, the insertion operator for seqDouble2 is invoked (because seqDouble3 is just a C++ typedef of seqDouble2). Therefore, the TypeCode in the Any would be that of seqDouble2. If this is not desirable, one can use the new member function 'void type(TypeCode_ptr)' of the Any interface to explicitly set the TypeCode to the correct one.

Removing aliases from TypeCodes. Some ORBs (such as 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 stored in Anys, supply the -ORBtcAliasExpand 1 command-line flag when running an omniORB executable. There will be some (small) performance penalty when inserting values into an Any.

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.

Recursive TypeCodes. omniORB (as of version 2.7) 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;
};
```

Threads and type Any. Inserting and extracting simultaneously from the same Any (in 2 different threads) is undefined behaviour.

Extracting simultaneously from the same Any (in 2 or more different threads) also leads to undefined behaviour. It was decided not to protect the Any with a mutex, as this condition should rarely arise, and adding a mutex would lead to performance penalties.

11.3 TypeCode in omniORB

This section contains some notes on the use and behaviour of TypeCode in omniORB

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;
};
```

orb.idl Inclusion of the file `orb.idl` in IDL using `CORBA::TypeCode` is optional. An empty `orb.idl` file is provided for compatibility purposes.

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 (as per the CORBA 2.3 specification). 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.

TypeCode equality. The behaviour of `CORBA::TypeCode::equal()` member function from omniORB 2.8.0 onwards is different from pre-omniORB 2.8.0 releases. In summary, the pre-omniORB 2.8.0 is close to the semantics of the new `CORBA::TypeCode::equivalent()` member function. Details are as follows:

The `CORBA::TypeCode::equal()` member function will now return true only if the two TypeCodes are *exactly* the same. `tk_alias` TypeCodes are included in this comparison, unlike the comparison made when values are extracted from an Any (see section on Any, above).

In pre-omniORB 2.8.0 releases, equality test would ignore the optional fields when one of the fields in the two typecodes is empty. For example, if one of the TypeCodes being checked is a `tk_struct`, `tk_union`, `tk_enum`, or `tk_alias`, and has an empty repository ID parameter, then the repository ID parameter will be ignored when checking for equality. Similarly, if the `name` or `member_name` parameters of a TypeCode are empty strings, they will be ignored for equality checking purposes. This is because a CORBA 2 ORB does not have to include these parameters in a TypeCode (see the Interoperability section of the CORBA specification). Note that these (optional) parameters are included in TypeCodes generated by omniORB.

Since CORBA 2.3, the issue of TypeCode equality has been clarified. There is now a new member `CORBA::TypeCode::equivalent()` which provides the semantics of the `CORBA::TypeCode::equal()` as implemented in omniORB releases prior to 2.8.0. So from omniORB 2.8.0 onwards, the `CORBA::TypeCode::equal()` function has been changed to enforce strict equality. The pre-2.8.0 behaviour can be obtained with `equivalent()`.

11.4 Source Listing

11.4.1 anyExample_impl.cc

```
// anyExample_impl.cc - This is the source code of the example used in
//                      Chapter 9 "Type Any and TypeCode" of the omniORB
//                      users guide.
//
//                      This is the object implementation.
//
// Usage: anyExample_impl
//
//        On startup, the object reference is printed to cerr as a
//        stringified IOR. This string should be used as the argument to
//        anyExample_clt.
//

#include <iostream.h>
#include <anyExample.hh>

class anyExample_i : public POA_anyExample {
public:
  inline anyExample_i() {}
  virtual ~anyExample_i() {}
  virtual CORBA::Any* testOp(const CORBA::Any& a);
};

CORBA::Any* anyExample_i::testOp(const CORBA::Any& a)
{
  cout << "Any received, containing: " << endl;

#ifndef NO_FLOAT
  CORBA::Double d;
#endif

  CORBA::Long l;
  const char* str;

  testStruct* tp;

  if (a >>= l) {
    cout << "Long: " << l << endl;
  }
#ifndef NO_FLOAT
  else if (a >>= d) {
    cout << "Double: " << d << endl;
  }
#endif
  else if (a >>= str) {
```

```
    cout << "String: " << str << endl;
  }
  else if (a >>= tp) {
    cout << "testStruct: l: " << tp->l << endl;
    cout << "            s: " << tp->s << endl;
  }
  else {
    cout << "Unknown value." << endl;
  }

  CORBA::Any* ap = new CORBA::Any;

  *ap <<= (CORBA::ULong) 314;

  cout << "Returning Any containing: ULong: 314\n" << endl;
  return ap;
}

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

int main(int argc, char** argv)
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB3");

    CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);

    anyExample_i* myobj = new anyExample_i();

    PortableServer::ObjectId_var myobjid = poa->activate_object(myobj);

    obj = myobj->_this();
    CORBA::String_var sior(orb->object_to_string(obj));
    cerr << "'" << (char*)sior << "'" << endl;

    myobj->_remove_ref();

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

    orb->run();
    orb->destroy();
  }
  catch(CORBA::SystemException&) {
    cerr << "Caught CORBA::SystemException." << endl;
  }
  catch(CORBA::Exception&) {
    cerr << "Caught CORBA::Exception." << endl;
```

```
    }
    catch(omniORB::fatalException& fe) {
      cerr << "Caught omniORB::fatalException:" << endl;
      cerr << "  file: " << fe.file() << endl;
      cerr << "  line: " << fe.line() << endl;
      cerr << "  mesg: " << fe.errmsg() << endl;
    }
    catch(...) {
      cerr << "Caught unknown exception." << endl;
    }
    return 0;
  }
```

11.4.2 anyExample_clt.cc

```
// anyExample_clt.cc -  This is the source code of the example used in
//                      Chapter 9 "Type Any and TypeCode" of the omniORB
//                      users guide.
//
//                      This is the client.
//
// Usage: anyExample_clt <object reference>
//

#include <iostream.h>
#include <anyExample.hh>

static void invokeOp(anyExample_ptr& tobj, const CORBA::Any& a)
{
  CORBA::Any_var bp;

  cout << "Invoking operation." << endl;
  bp = tobj->testOp(a);

  cout << "Operation completed. Returned Any: ";
  CORBA::ULong ul;

  if (bp >>= ul) {
    cout << "ULong: " << ul << "\n" << endl;
  }
  else {
    cout << "Unknown value." << "\n" << endl;
  }
}

static void hello(anyExample_ptr tobj)
{
  CORBA::Any a;

  // Sending Long
  CORBA::Long l = 100;
  a <<= l;
  cout << "Sending Any containing Long: " << l << endl;
  invokeOp(tobj,a);

  // Sending Double
#ifndef NO_FLOAT
  CORBA::Double d = 1.2345;
  a <<= d;
  cout << "Sending Any containing Double: " << d << endl;
  invokeOp(tobj,a);
#endif
```

```
  // Sending String
  const char* str = "Hello";
  a <<= str;
  cout << "Sending Any containing String: " << str << endl;
  invokeOp(tobj,a);

  // Sending testStruct  [Struct defined in IDL]
  testStruct t;
  t.l = 456;
  t.s = 8;
  a <<= t;
  cout << "Sending Any containing testStruct: l: " << t.l << endl;
  cout << "                                   s: " << t.s << endl;
  invokeOp(tobj,a);
}

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

int main(int argc, char** argv)
{
  try {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB3");

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

    CORBA::Object_var obj = orb->string_to_object(argv[1]);
    anyExample_var ref = anyExample::_narrow(obj);
    if( CORBA::is_nil(ref) ) {
      cerr << "Can't narrow reference to type anyExample (or it was nil)."
           << endl;
      return 1;
    }
    hello(ref);

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

```
    }
    catch(omniORB::fatalException& fe) {
      cerr << "Caught omniORB::fatalException:" << endl;
      cerr << "  file: " << fe.file() << endl;
      cerr << "  line: " << fe.line() << endl;
      cerr << "  mesg: " << fe.errmsg() << endl;
    }
    catch(...) {
      cerr << "Caught unknown exception." << endl;
    }
    return 0;
  }
```

Chapter 12

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.

12.1 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`.

12.2 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.

12.2.1 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.

12.2.2 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, 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.

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://cgi.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`.

[Ric96] Tristan Richardson. *The OMNI Thread Abstraction*. AT&T Laboratories Cambridge, October 1996.