The omniORB version 3.0 User's Guide

Sai-Lai Lo
(*email: slo@uk.research.att.com*)

David Riddoch
(*email: djr@uk.research.att.com*)

Duncan Grisby
(*email: dgrisby@uk.research.att.com*)

AT&T Laboratories Cambridge

May 2000

Changes and Additions, May 2000

- Updated to omniORB 3.

Contents

Chapter 1

Introduction

omniORB is an Object Request Broker (ORB) that implements the 2.3 specification of the Common Object Request Broker Architecture (CORBA) [OMG99][1]. It has passed the Open Group CORBA compliant testsuite and is one of the three ORBs to have been 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 CORBA 2.3 compliant

omniORB implements the Internet Inter-ORB Protocol (IIOP). This protocol provides omniORB the means of achieving interoperability with the ORBs implemented by other vendors. In fact, this is the native protocol used by omniORB for the communication amongst its objects residing in different address spaces. Moreover, the IDL to C++ language mapping provided by omniORB conforms to the latest revision of the CORBA specification. Type Any and TypeCode are now supported (introduced in version 2.5.0). DynAny is supported since 2.6.0. The Dynamic Invocation Interface and Dynamic Skeleton Interface are supported since 2.7.0. The C++ mapping has been updated to the CORBA 2.3 specification since 2.8.0. The POA was introduced with 3.0.

[1]Most of the 2.3 features have been implemented. The features still missing in this release are listed in section 1.1.4. 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`

1.1.2 Multithreading

omniORB is fully multithreaded. To achieve low IIOP call overhead, unnecessary call-multiplexing is eliminated. At any time, there is at most one call in-flight in each communication channel between two address spaces. To do so without limiting the level of concurrency, new channels connecting the two address spaces are created on demand and cached when there are more 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.

1.1.3 Portability

At AT&T Laboratories, the ability to target a single source tree to multiple platforms is very important. This is difficult to achieve if the IDL to C++ mapping for these platforms are different. We avoid this problem by making sure that only one IDL to C++ mapping is used. We run several flavours of Unix, Windows NT, Windows 95 and our in-house developed systems for our own hardware. omniORB has been ported to all these platforms. **The IDL to C++ mapping for all these targets is the same**.

omniORB uses real C++ exceptions and nested classes. We keep to the CORBA specification's standard mapping as much as possible and do not use the alternative mappings for C++ dialects. The only exception is the mapping of **modules**.

Starting with 2.6.0, the code generated by the IDL compiler of omniORB can be compiled using **C++ classes** or **namespaces** to represent IDL **modules** depending on the availability of namespace support in the compiler.

omniORB relies on the native thread libraries to provide the multithreading capability. A small class library (omnithread [Ric96b]) is used to encapsulate the (possibly different) 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.

1.1.4 Missing features

omniORB is not (yet) a complete implementation of the CORBA 2.3 core. The following is a list of the missing features.

- omniORB does not have its own Interface Repository. However, it can act as a client to an IR.

- The IDL types wchar, wstring, fixed, and valuetype are not supported in this release.

- The `PortableServer::Current` interface is not yet supported.
- The DSI interface has changed significantly. Old code will have to be rewritten to the new interface. There is no support for using the DSI with the BOA. This part of the ORB is incomplete, since it depends in part on `PortableServer::Current`. See chapter 12 for a way to workaround this problem for now.

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://www.uk.research.att.com/omniORB/`).

1.2 Setting Up Your Environment

At AT&T Laboratories Cambridge, you should use the OMNI Development Environment (ODE) [Ric96a] and the OMNI tree version 5.0 or above to compile your programs. If this is the case, there is no extra setup you have to do other than those described in the ODE documentation.

If you are running omniORB at other sites, you (or your system administrator) should install omniORB by following the instructions in the installation notes.

- 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.
- On ARM/ATMos, the omniORB runtime looks for configuration information in the file `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\ORL\omniORB\2.0`.

The configuration file is used to obtain an object reference for the Naming Service, via either a stringified IOR or an Interoperable Naming Service URI. The entry in the configuration file should be specified in the form:

```
ORBInitRef NameService=<URI for the Naming Service>
```

The easiest way of specifying the Naming Service is with a `corbaname:` URI, as described in section 4.1.2.

Comments in the configuration file should be prefixed with a '#' character.

On Win32 platforms, the naming service reference can be placed in the system registry, in the (string) value `ORBInitRef`, under the key `HKEY_LOCAL_MACHINE\SOFTWARE\ORL\omniORB\2.0`.

1.2.1 Deprecated configuration file entries

`ORBInitRef` is new with omniORB 3.0. The omniORB 2 formats are still supported, so you can specify the Naming Service IOR using a line of the form:

```
NAMESERVICE <IOR for the Naming Service>
```

Since 2.6.0, two other entries have been supported, but are now made obsolete by the Interoperable Naming Service URIs:

```
ORBInitialHost <hostname string>
ORBInitialPort <port number (1-65535)>
```

The corresponding entries under the Win32 system registry are the keys named `ORBInitialHost` and `ORBInitialPort`.

The two entries provide information to the ORB to locate a (proprietary) bootstrap service at runtime. The bootstrap service is able to return the initial object reference for the Naming Service and others.

1.3 Platform specific variables

To compile omniORB programs correctly, several C++ preprocessor defines **must** be specified to identify the target platform.

Platform	CPP defines			
Sun Solaris 2.5	__sparc__	__sunos__	__OSVERSION__=5	
Digital Unix 3.2	__alpha__	__osf1__	__OSVERSION__=3	
HPUX 10.x	__hppa__	__hpux__	__OSVERSION__=10	
HPUX 11.x	__hppa__	__hpux__	__OSVERSION__=11	
IBM AIX 4.x	__aix__	__powerpc__	__OSVERSION__=4	
Linux 2.0 (x86)	__x86__	__linux__	__OSVERSION__=2	
Linux 2.0 (alpha)	__alpha__	__linux__	__OSVERSION__=2	
Linux 2.0 (powerpc)	__powerpc__	__linux__	__OSVERSION__=2	
Windows/NT 3.5	__x86__	__NT__	__OSVERSION__=3	__WIN32__
Windows/NT 4.0	__x86__	__NT__	__OSVERSION__=4	__WIN32__
Windows/95	__x86__	__WIN32__		
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. Alternately, you could create a sitedef.h file in the same directory as `omniORB3/CORBA.h`. Write into the file the appropriate set of preprocessor defines and add

```
#include <omniORB3/sitedef.h>
```

at the beginning of `omniORB3/CORBA_sysdep.h`.

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. Now that omniORB supports 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.

The source code of this example is included in the last section of this chapter. A makefile written to be used under the OMNI Development Environment (ODE) [Ric96a] is also included.

[1]The stub code is the C++ code that provides the object mapping as defined in the CORBA 2.3 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 specification 2.3 [OMG99]. For the moment, you only need to know that the interface consists of a single operation, `echoString()`, which takes a string as an 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. If you are using ODE, the IDL files should be placed in the `idl` directory of your export tree. This is done so that the stub code will be generated automatically and kept up-to-date with your IDL file.

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 3 uses distinct C++ types for object references and servants, so the compiler will complain if you use a servant when an object reference is expected, or vice-versa.

Warning

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. If you have code which relies on this, it will not compile with omniORB 3, even under omniORB 3's BOA compatibility mode.

2.5 A Quick Tour of the C++ stubs

The C++ stubs conform to the mapping defined in the CORBA 2.3 specification (chapter 23). 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 test 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==`.

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.

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 widened 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`.

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

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

Once you have called `CORBA::release()` on an object reference, you should 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. For details, please read the CORBA specification [OMG99], chapter 23.

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

[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.

```
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 3 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 3.2 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 _impl_Echo :
  public virtual omniServant
{
public:
  virtual char * echoString(const char*  mesg) = 0;
  // ...
};

class POA_Echo :
  public virtual _impl_Echo,
  public virtual PortableServer::ServantBase
{
public:
  Echo_ptr _this();
  // ...
};
```

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 section 23.22 of the CORBA 2.3 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. Not 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 which 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.

```
void
hello(CORBA::Object_ptr obj)
{
  Echo_var e = Echo::_narrow(obj);

  if (CORBA::is_nil(e)) {
    cerr << "cannot invoke on a nil object reference.\n"
         << endl;
    return;
  }

  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 function `hello()` 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 it was explained in the previous section, `T_var` types take care of storage allocation and release automatically when variables of the type are assigned to 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 said earlier, `Echo_var` type can be used interchangeably with `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 leak 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

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

[5] Please refer to the CORBA specification section 23.7 for the details of the String_var mapping. Other `T_var` types are also covered in chapter 23.

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,"omniORB3");
5
6    CORBA::Object_var       obj = orb->resolve_initial_references("RootPOA");
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 3rd argument to determine which ORB should be returned. To use omniORB 3, this argument must either be 'omniORB3' or NULL. If it is NULL, there must be an argument, -ORBid 'omniORB3', in `argv`[6]. Like all command-line arguments understood by the ORB, it will be removed from `argv` when `CORBA::ORB_init()` returns. Therefore, an application is not required to handle any command-line arguments it does not understand. If the ORB identifier is not 'omniORB3', the initialisation will fail and a nil `ORB_ptr` will be returned. If supplied, omniORB also reads the configuration file `omniORB.cfg`. Among other things, the file provides a list of initial object references. One example of these object references is the Naming service. Its use will be discussed in section 2.10.1. If any error occurs during the processing of the configuration file, the system exception

[6] For backwards compatibility, the ORB identifier 'omniORB2' is also accepted.

`CORBA::INITIALIZE` 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.

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 2.3 specification[OMG99] for details of all the POA policies which are available.

2.8.3 Object initialisation

Line 9

An instance of the Echo object 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 IIOP 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.

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 object's reference count drops to zero, so the object 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, "omniORB3");
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[7]. 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]);
```

[7]Notice that the object key is not globally unique across address spaces.

```
  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[8]. 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[9], a single catch of `CORBA::SystemException` will catch all the different system exceptions thrown by omniORB.

When omniORB detects an internal error that is most likely to be caused by a bug in the runtime, 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. If this occurs, you are strongly encouraged to file a bug report and point out the location.

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[10]. 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.

---

[8] 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.

[9] Run Time Type Identification

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

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

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[11]. 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,"omniORB3");
```

[11] There are escaping rules to cope with id and kind fields which contain '.' and '/' characters. See chapter 4 of this manual, and chapter 3 of the CORBA services specification, as updated for the Interoperable Naming Service [OMG00].

```
CORBA::Object_var initServ;
initServ = orb->resolve_initial_references("NameService");

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

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

Since 2.6.0, omniORB has supported *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 section 23.36.7 of the CORBA 2.3 C++ mapping specification [OMG99].

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, "omniORB3");

    // 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)
{
  return CORBA::string_dup(mesg);
}

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

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);

    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();
    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;
}
```

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, "omniORB3");

    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;
    }
    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, "omniORB3");
```

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

    myecho->_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;
}

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

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, "omniORB3");
```

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

    Echo_var echoref = Echo::_narrow(obj);
    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, "omniORB3");

    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();
    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;
}

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

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;
}
```

2.12.7 dir.mk

```
CXXSRCS = eg1.cc eg2_impl.cc eg2_clt.cc eg3_impl.cc eg3_clt.cc

DIR_CPPFLAGS = $(CORBA_CPPFLAGS)

CORBA_INTERFACES = echo

ifdef OSF1
ifeq ($(notdir $(CXX)),cxx)
NoTieExample = 1
endif
endif

ifndef NoTieExample
# -Wbtp tells the compiler to generate tie implementation template
OMNIORB_IDL += -Wbtp
eg3_tieimpl = $(patsubst %,$(BinPattern),eg3_tieimpl)
endif

eg1        = $(patsubst %,$(BinPattern),eg1)
eg2_impl   = $(patsubst %,$(BinPattern),eg2_impl)
eg2_clt    = $(patsubst %,$(BinPattern),eg2_clt)
eg3_impl   = $(patsubst %,$(BinPattern),eg3_impl)
eg3_clt    = $(patsubst %,$(BinPattern),eg3_clt)

all:: $(eg1) $(eg2_impl) $(eg2_clt)  $(eg3_impl) $(eg3_clt) $(eg3_tieimpl)

clean::
        $(RM) $(eg1) $(eg2_impl) $(eg2_clt) $(eg3_impl) $(eg3_clt) \
                $(eg3_tieimpl)

export:: $(eg1) $(eg2_impl) $(eg2_clt) $(eg3_impl) $(eg3_clt) $(eg3_tieimpl)
        @(module="echoexamples"; $(ExportExecutable))

$(eg1): eg1.o $(CORBA_STATIC_STUB_OBJS) $(CORBA_LIB_DEPEND)
        @(libs="$(CORBA_LIB_NODYN)"; $(CXXExecutable))

$(eg2_impl): eg2_impl.o $(CORBA_STATIC_STUB_OBJS) $(CORBA_LIB_DEPEND)
        @(libs="$(CORBA_LIB_NODYN)"; $(CXXExecutable))

$(eg2_clt): eg2_clt.o $(CORBA_STATIC_STUB_OBJS) $(CORBA_LIB_DEPEND)
        @(libs="$(CORBA_LIB_NODYN)"; $(CXXExecutable))

$(eg3_impl): eg3_impl.o $(CORBA_STATIC_STUB_OBJS) $(CORBA_LIB_DEPEND)
        @(libs="$(CORBA_LIB_NODYN)"; $(CXXExecutable))

$(eg3_clt): eg3_clt.o $(CORBA_STATIC_STUB_OBJS) $(CORBA_LIB_DEPEND)
```

```
	@(libs="$(CORBA_LIB_NODYN)"; $(CXXExecutable))

ifndef NoTieExample
$(eg3_tieimpl): eg3_tieimpl.o $(CORBA_STATIC_STUB_OBJS) $(CORBA_LIB_DEPEND)
	@(libs="$(CORBA_LIB_NODYN)"; $(CXXExecutable))
endif
```

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 [OMG99]. 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 CORBA 2.3 C++ mapping.

3.1 Incompatibilities with pre-2.8.0 releases

Before 2.8.0, omniORB implemented the CORBA 2.0 C++ mapping. From 2.8.0 onwards, the mapping has been updated to CORBA 2.3. Unfortunately, to comply with the CORBA 2.3 specification, it has been necessary to change the semantics of a few APIs in a way that is incompatible with older omniORB releases. The incompatible changes are limited to:

1. The mapping for some out and inout argument types is different.

2. The extraction of string, object reference and typecode from an Any has different ownership rules.

3. the DII interface now defaults to reporting a system exception by raising a C++ exception, instead of returning the exception as an environment value.

The changes are minor and require minimal changes to the application source code. The C++ compiler will complain about the first change. **However, it is not possible to detect the old usage for changes 2 and 3 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 omniORB configuration variable `omniORB::omniORB_27_CompatibleAnyExtraction` can be set to revert the any extraction operators to the old semantics. More information can be found in chapter 9.

3.2 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 3'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 3.
- 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 6.13). Code which used the old interfaces will have to be rewritten.

Chapter 4

Interoperable Naming Service

omniORB 3 supports the Interoperable Naming Service (INS), which will be part of CORBA 2.4. The following is a summary of the new facilities described in the INS edited chapters document [OMG00].

4.1 Object URIs

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

4.1.1 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, although omniORB only supports IIOP 1.0 at present:

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

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

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

4.1.2 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
```

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

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

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

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.

4.2 Configuring resolve_initial_references

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

4.2.1 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...
```

4.2.2 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
```

4.2.3 omniORB configuration file

As an extension to the standard facilities of the INS, omniORB supports configuration file entries named `ORBInitRef` and `ORBDefaultInitRef`. The syntax is identical to the command line arguments. `omniORB.cfg` might contain:

```
ORBInitRef NameService=corbaname::myhost.example.com
ORBDefaultInitRef corbaname:rir:#services
```

4.2.4 Resolution order

With all these options for specifying object references to be returned by `resolve_initial_references()`, it is important to understand the order in which the options are tried. The resolution order, as required by the CORBA specification, is:

1. Check for special names such as '`RootPOA`'[2].
2. Resolve with an `-ORBInitRef` argument.
3. Resolve with the `-ORBDefaultInitRef` prefix, if present.
4. Resolve with an `ORBInitRef` (or old-style `NAMESERVICE`) entry in the configuration file.
5. Resolve with the `ORBDefaultInitRef` entry in the configuration file, if present.
6. Resolve with the deprecated `ORBInitialHost` boot agent.

This order mostly has the expected consequences—in particular that command line arguments override entries in the configuration file. However, you must be careful with the default prefixes. Suppose you have configured a '`NameService`' entry in the configuration file, and you specify a default prefix on the command line with:

```
-ORBDefaultInitRef corbaname:rir:#services
```

expecting unknown services to be looked up in the configured naming service. Now, step 3 above means that `resolve_initial_references("MyService")` should be processed with the steps:

1. Construct the URI '`corbaname:rir:#services/MyService`' and give it to `string_to_object()`.
2. Resolve the first part of the `corbaname` URI by calling `resolve_initial_references("NameService")`.

[2] In fact, a strict reading of the specification says that it should be possible to override '`RootPOA`' etc. with `-ORBInitRef`, but since POAs are locality constrained that is ridiculous.

3. Construct the URI 'corbaname:rir:#services/NameService' and give it to string_to_object().
4. Resolve the first part of the corbaname URI by calling resolve_initial_references("NameService").
5. ... and so on for ever...

omniORB detects loops like this and throws either CORBA::ORB::InvalidName if the loop started with a call to resolve_initial_references(), or CORBA::BAD_PARAM if it started with a call to string_to_object(). To avoid the problem you must either specify the NameService reference on the command line, or put the DefaultInitRef in the configuration file.

4.3 omniNames

4.3.1 NamingContextExt

omniNames now supports the 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.

4.3.2 Use with corbaname

To make it easy to use omniNames with `corbaname` URIs, it now starts with the default port of 2809, and an object key of '`NameService`' for the root naming context. This is only possible when it is started 'fresh', rather than with a log file from an older omniNames version.

If you have a previous omniNames log, configured to run on a different port, and with a different object key for its root context, all is not lost. If the root context's object key is not '`NameService`', omniNames creates a forwarding agent with that key. Effectively, this means that there are two object keys which refer to the root context—'`NameService`' and whatever the original key was.

For the port number, there are two options. The first is to run omniNames with a command line argument like:

```
omniNames -logdir /the/log/dir -ORBpoa_iiop_port 2809
```

This causes it to listen on both port 2809 *and* whatever port it listened on before. The disadvantage with this is that the IORs to all naming contexts now contain two IIOP profiles, one for each port, which, amongst other things, increases the size of the omniNames log.

The second option is to use omniMapper, as described below.

4.4 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[3]. 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
```

4.5 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.

[3] You can also play the `-ORBpoa_iiop_port` trick to make it listen on more than one port.

Chapter 5

The IDL compiler

omniORB 3 has a brand new IDL compiler, named 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 earlier versions of omniORB, and with other ORBs.

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:

`-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.
`-b`*back-end*	Run the specified back-end. For the C++ ORB, use `-bcxx`.
`-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.
`-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 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.

5.2.2 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 '`_`'.

5.2.3 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
```

Compile two files with Any support:

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

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

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

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

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

Chapter 6

The omniORB API

In this chapter, we introduce the omniORB API. The purpose of this API is to provide access points to omniORB specific functionality that is not covered by the CORBA specification. Obviously, if you use this API in your application, that part of your code is not going to be portable to run unchanged on other vendors' ORBs. To make it easier to identify omniORB dependent code, this API is defined under the name space '`omniORB`'[1].

6.1 ORB initialisation options

`CORBA::ORB_init()` accepts the following standard command-line arguments:

`-ORBid omniORB3`	The identifier must be 'omniORB3'.
`-ORBInitRef` <*ObjectId*>=<*ObjectURI*>	See section 4.2.
`-ORBDefaultInitRef` <*Default URI*>	See section 4.2.

and the following omniORB-specific arguments:

`-ORBtraceLevel` <*level*>	See section 6.3.
`-ORBtraceInvocations`	See section 6.3.
`-ORBstrictIIOP`	See section 6.9.
`-ORBtcAliasExpand` <*0 or 1*>	See section 9.2.
`-ORBgiopMaxMsgSize` <*size in bytes*>	See section 6.5.
`-ORBobjectTableSize` <*number of entries*>	See section 6.6.
`-ORBserverName` <*string*>	See section 6.4.
`-ORBno_bootstrap_agent`	See section 6.8.
`-ORBdiiThrowsSysExceptions` <*0 or 1*>	See section 11.4.
`-ORBabortOnInternalError` <*0 or 1*>	See section 6.11.
`-ORBverifyObjectExistsAndType` <*0 or 1*>	See section 6.9.
`-ORBinConScanPeriod` <*0–max integer*>	See section 8.3.

[1] `omniORB` is a class name if the C++ compiler does not support the `namespace` keyword.

`-ORBoutConScanPeriod` <*0–max integer*>	See section 8.3.
`-ORBclientCallTimeOutPeriod` <*0–max integer*>	See section 8.3.
`-ORBserverCallTimeOutPeriod` <*0–max integer*>	See section 8.3.
`-ORBscanGranularity` <*0–max integer*>	See section 8.3.
`-ORBlcdMode`	See section 6.9.
`-ORBpoa_iiop_port` <*port no.*>	See section 6.2.
`-ORBpoa_iiop_name_port` <*hostname*[*:port no.]*>	See section 6.2.
`-ORBhelp`	Lists all ORB command line options.

and these two obsolete omniORB-specific arguments:

`-ORBInitialHost` <*string*>	See section 6.8.
`-ORBInitialPort` <*1–65535*>]	See section 6.8.

As defined in the CORBA specification, any command-line arguments understood by the ORB will be removed from `argv` when the initialisation functions return. Therefore, an application is not required to handle any command-line arguments it does not understand.

6.2 Hostname and port

Normally, omniORB lets the operating system pick which port number it should use to listen for IIOP calls. Alternatively, you can specify a particular port using `-ORBpoa_iiop_port`. If you specify `-ORBpoa_iiop_port` more than once, omniORB will listen on all the ports you specify.

By default, the ORB can work out the IP address of the host machine. This address is recorded in the object references of the local objects. However, when the host has multiple network interfaces and multiple IP addresses, it may be desirable for the application to control what address the ORB should use. This can be done by defining the environment variable `OMNIORB_USEHOSTNAME` to contain the preferred host name or IP address in dot-numeric form. Alternatively, the same can be achieved using the `-ORBpoa_iiop_name_port` option. You can optionally specify a port number too. Again, you can specify more than one host name by using `-ORBpoa_iiop_name_port` more than once.

6.3 Run-time Tracing and Diagnostic Messages

omniORB can output tracing and diagnostic messages to the standard error stream. To avoid conflicts between old-style `iostream.h` and new-style `iostream`, omniORB uses neither. Some or all of these messages can be turned on/off by setting the variable `omniORB::traceLevel`. The type definition of the variable is:

```
CORBA::ULong omniORB::traceLevel = 1; // The default value is 1
```

At the moment, the following trace levels are defined:

level 0	turn off all tracing and informational messages
level 1	informational messages only
level 2	the above plus configuration information
level 5	the above plus notifications when server threads are created or communication endpoints are shutdown
level 10–20	the above plus execution and exception traces
level 25	the above plus hex dumps of all data sent and received by the ORB via its network connections.

The variable can be changed by assignment inside your applications. It can also be changed by specifying the command-line option: `-ORBtraceLevel` <*level*>. For instance:

```
$ eg2_impl -ORBtraceLevel 5
```

New in omniORB 3, you can also trace operation invocations by setting `omniORB::traceInvocations` to true, or with the `-ORBtraceInvocations` command line argument.

6.4 Server Name

Applications can optionally specify a name to identify the server process. At the moment, this name is only used by the host-based access control module. See section 8.5 for details. The name is stored in the variable `omniORB::serverName`.

```
CORBA::String_var omniORB::serverName;
```

The variable can be changed by assignment inside your applications. It can also be changed by specifying the command-line option: `-ORBserverName` <*string*>.

6.5 GIOP Message Size

omniORB sets a limit on the GIOP message size that can be sent or received. The value can be obtained by calling:

```
size_t omniORB::MaxMessageSize();
```

and can be changed by:

```
void omniORB::MaxMessageSize(size_t newvalue);
```

or by the command-line option `-ORBgiopMaxMsgSize`. The exact value is somewhat arbitrary. The reason such a limit exists is to provide some way to protect the server side from resource exhaustion. Think about the case when the server receives a rogue GIOP(IIOP) request message that contains a sequence length field set to 2^{31}. With a reasonable message size limit, the server can reject this rogue message straight away.

6.6 Object table size

omniORB uses a hash table to store the mapping from object keys to servant objects. Normally, it dynamically re-sizes the hash table when it becomes too full or too empty. This is the most efficient trade-off between performance and memory usage. However, since all POA operations which add or remove objects from the table can (very occasionally) cause the object table to resize, the time spent in POA operations is much less predictable than if the table size was fixed.

To prevent omniORB from resizing its object table, set the variable `omniORB::objectTableSize` to the number of hash table entries you require *before* calling `CORBA::ORB_init()`. Alternatively, use the `-ORBobjectTableSize` argument. Note that omniORB uses an open hash table so you can have any number of objects active, no matter what size table you specify. If you have many more active objects than hash table entries, object look-up performance will become linear with the number of objects.

6.7 POA request holding timeout

POAs can be put into the *holding* state, which means that incoming requests are queued until the POA is set to a different state. Normally, queued requests are held for ever (or until the client times out as described in section 8.3). If you set `omniORB::poaHoldRequestTimeout` to a non-zero time in seconds, held requests will be cancelled after that time. When requests are cancelled in this way, the caller is sent a `TRANSIENT` exception.

6.8 Obsolete Initial Object Reference Bootstrapping

Starting from 2.6.0, but superseded by the Interoperable Naming Service in omniORB 3, a mechanism is available for the ORB runtime to obtain the initial object references to CORBA services. The bootstrap service is a special object with the object key 'INIT' and the following interface[2]:

```
// IDL
module CORBA {
  interface InitialReferences {
    Object get(in ORB::ObjectId id);
    // returns the initial object reference of the service
    // identified by <id>. For example the id for the
```

[2]This interface was first defined by Sun's NEO and is in used in Sun's JavaIDL.

```
      // Naming service is "NameService".

      ORB::ObjectIdList list();
      // returns the list of service ids that this agent knows
   };
 };
```

By default, all omniORB servers contain an instance of this object and are able to respond to remote invocations. To prevent the ORB from instantiating this object, the command-line option `-ORBno_bootstrap_agent` should be specified.

In particular, the Naming Service omniNames is able to respond to a query through this interface and return the object reference of its root context. In effect, the bootstrap agent provides a level of indirection. All omniORB clients still have to be supplied with the address of the bootstrap agent. However, the information is much easier to specify than a stringified IOR! Another advantage of this approach is that it is completely compatible with JavaIDL. This makes it possible for programs written for JavaIDL to share a Naming Service with omniORB.

The address of the bootstrap agent is given by the `ORBInitialHost` and `ORBInitialPort` parameter in the omniORB configuration file (section 1.2). The parameters can also be specified as command-line options (section 6.1). The parameter `ORBInitialPort` is optional. If it is not specified, port number 900 will be used.

During initialisation, the ORB reads the parameters in the omniORB configuration file. If the parameter `NAMESERVICE` is specified, the stringified IOR is used as the object reference of the root naming context. If the parameter is absent and the parameter `ORBInitialHost` is present, the ORB contacts the bootstrap agent at the address specified to obtain the root naming context when the application calls `resolve_initial_references()`. If neither is present, `resolve_initial_references()` returns a nil object reference. Finally, the command line argument `-ORBInitialHost` overrides any parameters in the configuration file. The ORB always contacts the bootstrap agent at the address specified to obtain the root naming context.

Now we are ready to describe a simple way to set up omniNames.

1. Start omniNames for the first time on a machine (e.g. wobble):

   `$ omniNames -start 1234`

2. Add to omniORB.cfg:

   `ORBInitialHost wobble`

   `ORBInitialPort 1234`

3. All omniORB applications will now be able to contact omniNames.

   Alternatively, the command line options can be used, for example:

   `$ eg3_impl -ORBInitialHost wobble -ORBInitialPort 1234 &`

   `$ eg3_clt -ORBInitialHost wobble -ORBInitialPort 1234`

6.9 GIOP Lowest Common Denominator Mode

Sometimes, to cope with bugs in another ORB, it is necessary to disable various GIOP and IIOP features in order to achieve interoperability. If the command line option `-ORBlcdMode` is present or the function `omniORB::enableLcdMode()` is called, the ORB enters the so-called 'lowest common denominator mode'. It bends over backwards to cope with bugs in the ORB at the other end. This is purely a transitional measure. The long term solution is to report the bugs to the other vendors and ask them to fix them expediently.

In some (sloppy) IIOP implementations, the message size value in the IIOP header can be larger than the actual body size, i.e. there is garbage at the end. As the spec does not say the message size must match the body size exactly, this is not a clear violation of the spec. omniORB's default policy is to expect incoming messages to be formatted properly. Any messages that have garbage at the end will be rejected.

`-ORBlcdMode` and `omniORB::enableLcdMode()` set omniORB to silently skip the unread part of such invalid messages. Alternatively, you can just change this policy with the `-ORBstrictIIOP 0` command line, or by setting `omniORB::strictIIOP` to zero. The problem with doing this is that the header message size may actually be garbage, caused by a bug in the sender's code. The receiving thread may block forever as it tries to read more data from the connection. In this case the sender won't send any more as it thinks it has marshalled in all the data.

By default, omniORB uses the GIOP LOCATE_REQUEST message to verify the existence of an object prior to the first invocation. If another vendor's ORB is known not to be able to handle this GIOP message, set the variable `omniORB::verifyObjectExistsAndType` to 0 to disable this feature, and hence achieve interoperability. The command line option `-ORBverifyObjectExistsAndType` has the same effect.

6.10 GIOP Requesting Principal field

In versions 1.0 and 1.1 of the GIOP specification, request messages contain a 'principal' field which was intended to identify the client. The meaning of the principal field was never specified, and its use is now deprecated. The field is not present in GIOP 1.2. omniORB normally uses the string '`nobody`' in the principal field. However, some systems (e.g. the GNOME desktop environment) use the principal field as an authentication mechanism, so omniORB allows you to configure the principal by setting the `OMNIORB_PRINCIPAL` environment variable.

6.11 Trapping omniORB Internal Errors

```
class fatalException {
public:
```

```
    const char *file() const;
    int line() const;
    const char *errmsg() const;
};
```

When omniORB detects an internal problem that is most likely to be caused by a bug in the runtime, 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. You are strongly encouraged to file a bug report and point out the location.

It may help to cause a core-dump and look at the stack trace to locate where the exception was thrown. This can be done by setting the variable `omniORB::abortOnInternalError` to 1. The variable can also be set via the command line option `-ORBabortOnInternalError`.

6.12 System Exception Handlers

By default, all system exceptions which are raised during an operation invocation, with the exception 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 their operations are idempotent.

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.

6.12.1 CORBA::TRANSIENT handlers

When a `CORBA::TRANSIENT` exception is raised by the ORB runtime, the default behaviour of the proxy objects is to retry indefinitely until the operation succeeds. Successive retries will be delayed progressively by multiples of `omniORB::defaultTransientRetryDelayIncrement`. The delay will be limited to the maximum specified by `omniORB::defaultTransientRetryDelayMaximum`. The unit of both values are in seconds.

The ORB runtime will raise `CORBA::TRANSIENT` under the following conditions:

1. When a *cached* network connection is broken while an operation invocation is in progress. The ORB will try to open a new connection at the next invocation.

2. When the proxy object has been redirected by a location forward message by the remote object to a new location and the object at the new location cannot be contacted. In addition to the `CORBA::TRANSIENT` exception, the proxy object also resets its internal state so that the next invocation will be directed at the original location of the remote object.

3. When the remote object reports `CORBA::TRANSIENT`.

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

```
class omniORB {
public:

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

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

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

static CORBA::ULong defaultTransientRetryDelayIncrement;
static CORBA::ULong defaultTransientRetryDelayMaximum;
}
```

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.
}
```

6.12.2 CORBA::COMM_FAILURE

When the ORB runtime fails to establish a network connection to the remote object and none of the conditions listed above for raising a `CORBA::TRANSIENT` is applicable, it 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`.

6.12.3 CORBA::SystemException

To report an error condition, the ORB runtime may raise other system exceptions. If the exception is neither `CORBA::TRANISENT` nor `CORBA::COMM_FAILURE`, 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`.

6.13 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.

```
class LOCATION_FORWARD {
public:
  LOCATION_FORWARD(CORBA::Object_ptr objref);
};
```

The exception object consumes the object reference it is passed.

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

7.2 Basic Interface Type Checking

The ORB is provided with the interface information by the stubs via the `proxyObjectFactory` class. For an interface `A`, the stub of `A` contains a `_pof_A` class. This class is derived from the `proxyObjectFactory` class. The `proxyObjectFactory` is an abstract class which contains 3 functions of interest:

```
class proxyObjectFactory {
public:

  const char *irRepoId() const;

  virtual CORBA::Boolean is_a(const char *base_repoId) const = 0;

  virtual CORBA::Object_ptr newObjRef(const char* mostDerivedTypeId,
                                      IOP::TaggedProfileList* profiles,
                                      omniIdentity* id,
                                      omniLocalIdentity* lid) = 0;
};
```

- `irRepoId()` returns the RepoId of the interface.
- `is_a()` returns true(1) if the argument is the RepoId of the interface itself or it is that of its base interfaces.
- `newObjRef()` returns an object reference based on the information supplied in the arguments.

A single instance of every `_pof_*` is instantiated at runtime. The instances are entered into a list inside the ORB. The list constitutes all the interface information known to the ORB.

When the ORB receives an IOR from the network, it unmarshals and extracts the RepoId from the IOR. At this point, the ORB has two pieces of information in hand:

1. The RepoId of the object reference received from the network.
2. The RepoId the ORB is expecting. This comes from the unmarshal function that tells the ORB to receive the object reference.

Using the RepoId received, the ORB searches its proxyObjectFactory list for an exact match. If there is an exact match, all is well because the runtime can use the `is_a()` method of the proxyFactory to check if the expected RepoId is the same as the received RepoId or if it is one of its base interfaces. If the answer is positive, the IOR passes the type checking test and the ORB can proceed to create an object reference in its own address space to represent the IOR.

However, the ORB may fail to find a match in its proxyObjectFactory list. This means that the ORB has no local knowledge of the RepoId. There are three possible causes:

1. The remote end is another ORB and it sends a null string as the RepoId.
2. The ORB is expecting an object reference of interface A. The remote end sends the RepoId of B which is an interface that inherits from A. The stubs of A are linked into the executable but the stubs of B are not.
3. The remote end has sent a duff IOR.

To handle this situation, the ORB must find out the type information dynamically. This is explained in the next section.

7.3 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.

```
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 Management

This chapter describes how omniORB manages 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 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 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.

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.

On the client side of a connection, the thread that invokes on a proxy object drives the IIOP protocol directly and blocks on the connection to receive the reply. On the server side, a dedicated thread blocks on the connection. When it receives a request, it performs the up-call to the object and sends the reply when the up-call returns. There is no thread switching along the call chain.

With this design, there is at most one call in-flight at any time on a connection. If there is only one connection, concurrent invocations to the same remote address space would have to be serialised. To eliminate this limitation, omniORB implements a dynamic policy—multiple connections to the same remote address space are created on demand and cached when there are concurrent invocations in progress.

To be more precise, a network connection to another address space is only established when a remote invocation is about to be made. Therefore, there may be one or more object references in one address space that refer to objects in a different address space but unless the application invokes on these objects, no network connection is made. The maximum number of connections opened to another address space is 5 by default. Since 2.6.0, this parameter can be changed by setting the variable `omniORB::maxTcpConnectionPerServer` *before* calling `ORB_init()`.

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 to a server 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.

8.3 Idle Connection Shutdown and Remote Call Timeout

Inside the ORB, a thread is dedicated to scan for idle connections. The thread looks after both the outgoing connections and the incoming connections.

When a connection is idle for a period of time, the connection is shutdown. Similarly, if a remote call has not completed within a defined period of time, the connection is shutdown and the ORB will return `COMM_FAILURE` to the client.

How often the internal thread scans the connections is determined by the value of the *scan granularity*. This value is defaulted to 5 seconds and can be changed using the command-line option `-ORBscanGranularity` or using the `omniORB::`

`scanGranularity` call. Notice that this value determines the precision the ORB is able to keep to the value of the idle connection or remote call timeout.

How long the ORB will wait before it shuts down an idle connection is determined by the idleConnectionPeriods. There are separate values for incoming and outgoing connections. The default values are 180 and 120 seconds for incoming and outgoing connections respectively. These values can be changed using the command-line options `-ORBinConScanPeriod` and `-ORBoutConScanPeriod`. They can also be controlled by the `omniORB::idleConnectionScanPeriod()` call.

Similarly, how long the ORB will wait for a remote call to complete is determined by the parameter clientCallTimeOutPeriod for the client side and the serverCallTimeOutPeroid for the server side. By default calls will not timeout on either the client or server side.

The timeouts can be changed using the `omniORB::callTimeOutPeriod()` call, or with the command line options `-ORBclientCallTimeOutPeriod` and `-ORBserverCallTimeOutPeriod`.

The APIs are documented in `include/omniORB3/omniORB.h`.

```
class omniORB {
public:

  static void scanGranularity(CORBA::ULong sec);

  static CORBA::ULong scanGranularity();

  enum   idleConnType { idleIncoming, idleOutgoing };

  static void idleConnectionScanPeriod(idleConnType direction, CORBA::ULong sec);

  static CORBA::ULong idleConnectionScanPeriod(idleConnType direction);

  enum   callTimeOutType { clientSide, serverSide };

  static void callTimeOutPeriod(callTimeOutType direction, CORBA::ULong sec);

  static CORBA::ULong callTimeOutPeriod(callTimeOutType direction);
};
```

The scan can be disabled completely by setting the scan granularity to 0.

8.4 Interoperability Considerations

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

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

omniORB implements these semantics completely. However, it is known that some ORBs are not (yet) able to distinguish between an orderly and an abnormal shutdown. Usually this is manifested as the client in these ORBs seeing a `COMM_FAILURE` occasionally when connected to an omniORB server. The workaround is either to catch the exception in the application code and retry, or to turn off the idle connection shutdown inside the omniORB server.

8.5 Connection Acceptance

omniORB provides the hook to implement a connection acceptance policy. Inside the ORB runtime, a thread is dedicated to receive new connections. When the thread is given the handle of a new connection by the operating system, it calls the policy module to decide if the connection can be accepted. If the answer is yes, the ORB will start serving requests coming in from that connection. Otherwise, the connection is shutdown immediately.

There can be a number of policy module implementations. The basic one is a dummy module which just accepts every connection.

In addition, a host-based access control module is available on Unix platforms. The module uses the IP address of the client to decide if the connection can be accepted. The module is implemented using *tcp_wrappers 7.6*. The access control policy can be defined as rules in two access control files: `hosts.allow` and `hosts.deny`. The syntax of the rules is described in the manual page `hosts_access(5)` which can be found in appendix A. The syntax defines a simple access control language that is based on client (host name/address, user name), and server (process name, host name/address) patterns. When searching for a match on the server process name, the ORB uses the value of `omniORB::serverName`. `ORB_init()` uses the argument `argv[0]` to set the default value of this variable. This can be overridden by the application with the `-ORBserverName` <*string*> command line argument

The default location of the access control files is `/etc`. This can be overridden by the extra options in `omniORB.cfg`. For instance:

```
# omniORB configuration file - extra options

GATEKEEPER_ALLOWFILE   /project/omni/var/hosts.allow

GATEKEEPER_DENYFILE    /project/omni/var/hosts.deny
```

As each policy module is implemented as a separate library, the choice of policy module is determined at program linkage time. For instance, if the host-based access control module is in use:

```
% eg1 -ORBtraceLevel 2
omniORB gateKeeper is tcpwrapGK 1.0 - based on tcp_wrappers_7.6
I said,"Hello!". The Object said,"Hello!"
```

Whereas if the dummy module is in use:

```
% eg1 -ORBtraceLevel 2
omniORB gateKeeper is not installed. All incoming are accepted.
I said,"Hello!". The Object said,"Hello!"
```

Chapter 9

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 section of the CORBA 2.3 specification [OMG99], and for more information on TypeCode, refer to the Interface Repository chapter in the CORBA core section of the CORBA 2.3 specification.

Warning

Since 2.8.0, omniORB has been updated to CORBA 2.3. In order to comply with the 2.3 specification, it is 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 variable `omniORB::omniORB_27_CompatibleAnyExtraction` can be set to revert the any extraction operators to the old semantics.

9.1 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.

9.1.1 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 _sk_anyExample {
public:
  anyExample_i() { }
  virtual ~anyExample_i() { }
  virtual CORBA::Any* testOp(const CORBA::Any& a);
};
```

9.1.2 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;
}
```

9.1.3 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 chapter of the CORBA 2.3 specification [OMG99] for further information.

Warning

In pre-omniORB 2.8.0 releases, it was unclear in the CORBA specification whether or not object references should be managed by an Any. The omniORB implementation leaves management of an extracted object reference to the caller. Therefore, the programmer should release object references and TypeCodes that have been extracted from an Any. The same also applies to string extraction. CORBA 2.3 has clarified this issue and decreed that the management of an extracted object reference still belongs to the Any! Since 2.8.0, the omniORB implementation conforms to the CORBA 2.3 specification. For backward compatibility, the runtime variable `omniORB::omniORB_27_CompatibleAnyExtraction` can be set to 1 to get back the old behaviour. Notice that this should be used as a transitional measure and in the long run, applications should be written to use the new behaviour.

9.2 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 {
```

[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.

```
  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 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 inter-operating 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;
};
```

Type-unsafe construction and insertion. If using the type-unsafe Any constructor, or the `CORBA::Any::replace()` member function, ensure that the value returned by the `CORBA::Any::value()` member function and the TypeCode returned by the `CORBA::Any::type()` member function are used as arguments to the constructor or function. Using other values or TypeCodes may result in a mismatch, and is undefined behaviour.

Note that a non-CORBA 2 function,

```
CORBA::ULong CORBA::Any::NP_length() const
```

is supplied. This member function returns the length of the value returned by the `CORBA::Any::value()` member function. It may be necessary to use this function if the Any's value is to be stored in a file.

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.

9.3 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 (see section 10.7 of the CORBA 2.3 specification [OMG99]) 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 2 specification [OMG96]). 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()`.

9.4 Source Listing

9.4.1 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;
}
```

9.4.2 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 10

Dynamic Management of Any Values

In CORBA specification 2.2, a new facility—*DynAny* was introduced. Previously, it was not possible to insert or extract constructed and other complex types from an Any without using the stub code generated by an idl compiler for these types. This makes it impossible to write generic servers (bridges, event channels supporting filtering, etc.) because these servers can not have static knowledge of all the possible data types that they have to handle.

To fill this gap, the DynAny facility is defined to enable traversal of the data value associated with an Any at runtime and extraction of its constituents. This facility also enables the construction of an Any at runtime, without having static knowledge of its types.

This chapter describes how DynAny may be used. For completeness, you should also read the DynAny specification defined in Chapter 9 of the CORBA 2.3 specification. Where possible, the implementation in omniORB adheres closely to the specification. However, there are areas in the specification that are ambiguous or lacking in details. A number of these issues are currently opened with the ORB revision task force. Until the issues are resolved, it is possible that a different implementation may choose to interpret the specification differently. This chapter provides clarifications to the specification, explains the interpretation used and offers some advice and warnings on potential portability problems.

Notice that the DynAny interface has been changed in CORBA 2.3, particularly with the addition of the support for the IDL type `valuetype`. Future releases of omniORB will be updated to implement the interface as defined in CORBA 2.3.

10.1 C++ mapping

```
// namespace CORBA

class ORB {
```

```
public:
  ...

  class InconsistentTypeCode : public UserException { ... };

  DynAny_ptr create_dyn_any(const Any& value);

  DynAny_ptr create_basic_dyn_any(TypeCode_ptr tc);

  DynStruct_ptr create_dyn_struct(TypeCode_ptr tc);

  DynSequence_ptr create_dyn_sequence(TypeCode_ptr tc);

  DynArray_ptr create_dyn_array(TypeCode_ptr tc);

  DynUnion_ptr create_dyn_union(TypeCode_ptr tc);

  DynEnum_ptr create_dyn_enum(TypeCode_ptr tc);

};

typedef DynAny* DynAny_ptr;
class DynAny_var { ... };

class DynAny {
public:

  class Invalid : public UserException { ... };
  class InvalidValue : public UserException { ... };
  class TypeMismatch : public UserException { ... };
  class InvalidSeq : public UserException { ... };

  typedef _CORBA_Unbounded_Sequence__Octet OctetSeq;

  TypeCode_ptr type() const;

  void assign(DynAny_ptr dyn_any) throw(Invalid,SystemException);
  void from_any(const Any& value) throw(Invalid,SystemException);
  Any* to_any() throw(Invalid,SystemException);
  void destroy();
  DynAny_ptr copy();

  DynAny_ptr current_component();
  Boolean next();
  Boolean seek(Long index);
  void rewind();

  void insert_boolean(Boolean value) throw(InvalidValue,SystemException);
  void insert_octet(Octet value) throw(InvalidValue,SystemException);
```

```
  void insert_char(Char value) throw(InvalidValue,SystemException);
  void insert_short(Short value) throw(InvalidValue,SystemException);
  void insert_ushort(UShort value) throw(InvalidValue,SystemException);
  void insert_long(Long value) throw(InvalidValue,SystemException);
  void insert_ulong(ULong value) throw(InvalidValue,SystemException);
  void insert_float(Float value) throw(InvalidValue,SystemException);
  void insert_double(Double value) throw(InvalidValue,SystemException);
  void insert_string(const char* value) throw(InvalidValue,SystemException);
  void insert_reference(Object_ptr v) throw(InvalidValue,SystemException);
  void insert_typecode(TypeCode_ptr v) throw(InvalidValue,SystemException);
  void insert_any(const Any& value) throw(InvalidValue,SystemException);

  Boolean get_boolean() throw(TypeMismatch,SystemException);
  Octet get_octet() throw(TypeMismatch,SystemException);
  Char get_char() throw(TypeMismatch,SystemException);
  Short get_short() throw(TypeMismatch,SystemException);
  UShort get_ushort() throw(TypeMismatch,SystemException);
  Long get_long() throw(TypeMismatch,SystemException);
  ULong get_ulong() throw(TypeMismatch,SystemException);
  Float get_float() throw(TypeMismatch,SystemException);
  Double get_double() throw(TypeMismatch,SystemException);
  char* get_string() throw(TypeMismatch,SystemException);
  Object_ptr get_reference() throw(TypeMismatch,SystemException);
  TypeCode_ptr get_typecode() throw(TypeMismatch,SystemException);
  Any* get_any() throw(TypeMismatch,SystemException);

  static DynAny_ptr _duplicate(DynAny_ptr);
  static DynAny_ptr _narrow(DynAny_ptr);
  static DynAny_ptr _nil();
};

// DynFixed is not supported.

typedef DynEnum* DynEnum_ptr;
class DynEnum_var { ... };

class DynEnum :  public DynAny {
public:

  char* value_as_string();
  void value_as_string(const char* value);
  ULong value_as_ulong();
  void value_as_ulong(ULong value);

  static DynEnum_ptr _duplicate(DynEnum_ptr);
  static DynEnum_ptr _narrow(DynAny_ptr);
  static DynEnum_ptr _nil();
};
```

```
typedef char* FieldName;
typedef String_var FieldName_var;

struct NameValuePair {
  String_member id;
  Any value;
};

typedef _CORBA_ConstrType_Variable_Var<NameValuePair> NameValuePair_var;
typedef _CORBA_Unbounded_Sequence<NameValuePair > NameValuePairSeq;

typedef DynStruct* DynStruct_ptr;
class DynStruct_var { ... };

class DynStruct :  public DynAny {
public:

  char*  current_member_name();
  TCKind current_member_kind();
  NameValuePairSeq* get_members();
  void set_members(const NameValuePairSeq& NVSeqVal)
                 throw(InvalidSeq,SystemException);

  static DynStruct_ptr _duplicate(DynStruct_ptr);
  static DynStruct_ptr _narrow(DynAny_ptr);
  static DynStruct_ptr _nil();
};

typedef DynUnion* DynUnion_ptr;
class DynUnion_var { ... };

class DynUnion :  public DynAny {
public:

  Boolean set_as_default();
  void set_as_default(Boolean value);
  DynAny_ptr discriminator();
  TCKind discriminator_kind();
  DynAny_ptr member();
  char*  member_name();
  void member_name(const char* value);
  TCKind member_kind();

  static DynUnion_ptr _duplicate(DynUnion_ptr);
  static DynUnion_ptr _narrow(DynAny_ptr);
  static DynUnion_ptr _nil();
};

typedef _CORBA_Unbounded_Sequence<Any > AnySeq;
```

```
typedef DynSequence* DynSequence_ptr;
class DynSequence_var { ... };

class DynSequence :  public DynAny {
public:

  ULong length();
  void length (ULong value);
  AnySeq* get_elements();
  void set_elements(const AnySeq& value) throw(InvalidValue,SystemException);

  static DynSequence_ptr _duplicate(DynSequence_ptr);
  static DynSequence_ptr _narrow(DynAny_ptr);
  static DynSequence_ptr _nil();
};

typedef DynArray* DynArray_ptr;
class DynArray_var { ... };

class DynArray : public DynAny {
public:

  AnySeq* get_elements();
  void set_elements(const AnySeq& value) throw(InvalidValue,SystemException);

  static DynArray_ptr _duplicate(DynArray_ptr);
  static DynArray_ptr _narrow(DynAny_ptr);
  static DynArray_ptr _nil();
};
```

10.2 The DynAny Interface

10.2.1 Example: extract data values from an Any

If an Any contains a value of one of the basic data types, its value can be extracted using the pre-defined operators in the Any interface. When the value is a struct or other non-basic types, one can use the DynAny interface to extract its constituent values.

In this section, we use a struct as an example to illustrate how the DynAny interface can be used.

The example struct is as follows:

```
// IDL
struct exampleStruct1 {
  string s;
  double d;
  long   l;
};
```

To create a DynAny from an Any value, one uses the `create_dyn_any()` method:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.

Any v;
...        // Initialise v to contain a value of type exampleStruct1.

CORBA::DynAny_var dv = orb->create_dyn_any(v);
```

Like CORBA object and pseudo object references, a `DynAny_ptr` can be managed by a `_var` type (`DynAny_var`) which will release the `DynAny_ptr` automatically when the variable goes out of scope.

10.2.1.1 Iterate through the components

Once the DynAny object is created, we can use the DynAny interface to extract the individual components in `exampleStruct1`. The DynAny interface provides a number of functions to extract and insert component values. These functions are defined to operate on the component identified by the *current component* pointer.

A *current component* pointer is an internal state of a DynAny object. When a DynAny object is created, the pointer is initialised to point to the first component of the any value.

The pointer can be advanced to the next component with the `next()` operation. The function returns FALSE (0) if there are no more components. Otherwise it returns TRUE (1). When the any value in the DynAny object contains only one component, the `next()()` operation always returns FALSE(0).

Another way of adjusting the pointer is the `seek()` operation. The function returns FALSE (0) if there is no component at the specified index. Otherwise it returns TRUE (1). The index value of the first component is zero. Therefore, a `seek(0)` call rewinds the pointer to the first component, this is also equivalent to a call to the `rewind()` operation.

For completeness, we should also mention here the `current_component()` operation. This operation causes the DynAny object to return a reference to another DynAny object that can be used to access the current component. It is possible that the current component pointer is not pointing to a valid component, for instance, the `next()` operation has been invoked and there is no more component. Under this circumstance, the `current_component()` operation returns a nil DynAny object reference[1]. For components which are just basic data types, calling `current_component()` is an overkill because we can just use the basic type extraction and insertion functions directly.

[1]Testing a nil DynAny object with CORBA::is_nil() returns TRUE(1). The CORBA 2.2 specification does not specify what is the return value of this function when the current component pointer is invalid. To ensure portability, it is best to avoid calling `current_component()` under this condition.

10.2.1.2 Extract basic type components

In our example, the component values can be extracted as follows:

```
CORBA::String_var s = dv->get_string();
CORBA::Double     d = dv->get_double();
CORBA::Long       l = dv->get_long();
```

Each get basic type operation has the side-effect of advancing the current component pointer. For instance:

```
CORBA::String_var s = dv->get_string();
```

is equivalent to:

```
CORBA::DynAny_var temp = dv->current_component();
CORBA::String_var s = temp->get_string();
dv->next();
```

The get operations ensure that the current component is of the same type as requested. Otherwise, the object throws a `TypeMismatch` exception. If the current component pointer is invalid when a get operation is called, the object also throws a `TypeMismatch` exception[2].

To repeatedly access the components, one can use the `rewind()` or `seek()` operations to manipulate the current component pointer. For instance, to access the `d` member in `exampleStruct1` directly:

```
dv->seek(1);        // position current component to member d.
CORBA::Double d = dv->get_double();
```

10.2.1.3 Extract complex components

When a component is not one of the basic data types, it is not possible to extract its value using the get operations. Instead, a DynAny object has to be created from which the component is accessed.

Consider this example:

```
// IDL
struct exampleStruct2 {
  string m1;
  exampleStruct1 m2;
};
```

In order to extract the data members within `m2` (of type `exampleStruct1`), we use `current_component()` as follows:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.
```

[2]The CORBA 2.2 specification does not define the behavior of this error condition. To ensure portability, it is best to avoid calling the get operations when the current component pointer is known to be invalid.

```
Any v;
...       // Initialise v to contain a value of type exampleStruct2.

CORBA::DynAny_var dv = orb->create_dyn_any(v);

CORBA::String_var m1 = dv->get_string();  // extract member m1
CORBA::DynAny_var dm = dv->current_component(); // DynAny reference to m2
CORBA::String_var s = dm->get_string();   // m2.s
CORBA::Double     d = dm->get_double();   // m2.d
CORBA::Long       l = dm->get_long();     // m2.l
```

10.2.1.4 Clean-up

Now we finish off this example with a description on destroying DynAny objects. There are two points to remember:

1. A DynAny reference (`DynAny_ptr`) is like any CORBA object or psuedo object reference and should be handled in the same way. In particular, one has to call the `CORBA::release()` operation to indicate that a DynAny reference will no longer be accessed. In the example, this is done automatically by `DynAny_var`.

2. A DynAny object and its references are separate entities, just as a CORBA object implementation and its object references are different entities. While `CORBA::release()` will release any resources associated with a `DynAny_ptr`, one has to separately destroy the DynAny object to avoid any memory leak. This is done by calling the `destroy()()` operation.

In the example, the DynAny object can be destroyed as follows:

```
// C++
...
CORBA::DynAny_var dv = orb->create_dyn_any(v);
...
dv->destroy();

// From now on, one should not invoke any operation in dv.
// Otherwise the behaviour is undefined.
```

10.2.2 Example: insert data values into an Any

Using the DynAny interface, one can create an Any value from scratch. In this example, we are going to create an Any containing a value of the `exampleStruct1` type.

First, we have to create a DynAny to store the value using one of the `create_dyn()` functions. Because `exampleStruct1` is a struct, we use the `create_dyn_struct()` operation.

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.

// create the TypeCode for exampleStruct.
StructMemberSeq tc_members;
tc_members.length(3);
tc_members[0].name = (const char*)"s";
tc_members[0].type = CORBA::TypeCode::_duplicate(CORBA::_tc_string);
tc_members[0].type_def = CORBA::IDLType::_nil();
tc_members[1].name = (const char*)"d";
tc_members[1].type = CORBA::TypeCode::_duplicate(CORBA::_tc_double);
tc_members[1].type_def = CORBA::IDLType::_nil();
tc_members[2].name = (const char*)"l";
tc_members[2].type = CORBA::TypeCode::_duplicate(CORBA::_tc_long);
tc_members[2].type_def = CORBA::IDLType::_nil();
CORBA::TypeCode_var tc = orb->create_struct_tc("IDL:exampleStruct1:1.0",
                                               "exampleStruct1",
                                               tc_members);

// create the DynAny object to represent the any value
CORBA::DynAny_var dv = orb->create_dyn_struct(tc);
```

10.2.2.1 Insert basic type components

Once the DynAny object is created, we can use the DynAny interface to insert the components. The DynAny interface provides a number of insert operations to insert basic types into the any value. In our example, the component values can be inserted as follows:

```
CORBA::String_var s = (const char*)"Hello";
CORBA::Double     d = 3.1416;
CORBA::Long       l = 1;

dv->insert_string(s);
dv->insert_double(d);
dv->insert_long(l);
```

Each insert basic type operation has the side-effect of advancing the current component pointer. For instance:

```
dv->insert_string(s);
```

is equivalent to:

```
CORBA::DynAny_var temp = dv->current_component();
temp->insert_string(s);
dv->next();
```

The insert operations ensure that the current component is of the same type as the inserted value. Otherwise, the object throws an `InvalidValue` exception. If the current component pointer is invalid when an insert operation is called, the object also throws a `InvalidValue` exception[3].

Sometimes, one may just want to modify one component in an Any value. For instance, one may just want to change the value of the double member in `exampleStruct1`. This can be done as follows:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.

Any v;
...       // Initialise v to contain a value of type exampleStruct1.

CORBA::Double d = 6.28;

CORBA::DynAny_var dv = orb->create_dyn_any(v);

dv->seek(1);
dv->insert_double(d);    // Change the value of the member d.
```

Finally, the any value can be obtained from the DynAny object using the `to_any()` operation:

```
CORBA::Any_var v = dv->to_any();    // Obtain the any value.
```

10.2.2.2 Insert complex components

When a component is not one of the basic data types, it is not possible to insert its value using the insert operations. Instead, a DynAny object has to be created through which the component can be inserted.

In our example, one can insert component values into `exampleStruct2` as follows:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.

CORBA::TypeCode_var tc;
// create the TypeCode for exampleStruct2.
...
// create the DynAny object to represent the any value
CORBA::DynAny_var dv = orb->create_dyn_struct(tc);

CORBA::String_var m1  = (const char*)"Greetings";
CORBA::String_var m2s = (const char*)"Hello";
CORBA::Double     m2d = 3.1416;
CORBA::Long       m2l = 1;
```

[3]The CORBA 2.2 specification does not define the behavior of this error condition. To ensure portability, it is best to avoid calling the insert operations when the current component pointer is known to be invalid.

```
dv->insert_string(m1);   // insert member m1
CORBA::DynAny_var dm = dv->current_component(); // DynAny reference to m2
dm->insert_string(m2s);  // insert member m2.s
dm->insert_double(m2d);  // insert member m2.d
dm->insert_long(m2l);    // insert member m2.l

CORBA::Any_var v = dv->to_any();  // obtain the any value

dv->destroy();           // destroy the DynAny object.
                         // No operation should be invoked on dv
                         // from this point on except CORBA::release.
```

In addition to the DynAny interface, a number of derived interfaces are defined. These interfaces are specialisation of the DynAny interface to facilitate the handling of any values containing non-basic types: struct, sequence, array, enum and union[4]. The next few sections will provide more details on these interfaces.

10.3 The DynStruct Interface

When a DynAny object is created through the `create_dyn_any()` operation and the any value contains a struct type, a `DynStruct` object is created. The DynAny reference returned can be narrowed to a `DynStruct` reference using the `CORBA::DynStruct::_narrow()` operation.

In the previous example, the components are extracted using the get operations. Alternatively, the DynStruct interface provides an additional operation (`get_members()`) to return all the components in a single call. The returned value is a sequence of name value pairs. The member name is given in the name field and its value is returned as an Any value. For example, an alternative way to extract the components in the previous example is as follows:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.
Any v;
...       // Initialise v to contain a value of type exampleStruct1.
CORBA::DynAny_var dv = orb->create_dyn_any(v);

CORBA::DynStruct_var ds = CORBA::DynStruct::_narrow(dv);

CORBA::NameValuePairSeq* sq = ds->get_members();

char*         s;
CORBA::Double d;
CORBA::Long   l;

(*sq)[0].value >>= s;        // 1st element contains member s
```

[4] In the CORBA 2.2 specification, the DynFixed interface is defined to handle the fixed data type. This is not supported in this implementation.

```
(*sq)[1].value >>= d;        // 2nd element contains member d
(*sq)[2].value >>= l;        // 3rd element contains member l
```

Similarly, the DynStruct interface provides an additional operation (`set_members()`) to insert all the components in a single call. The following is an alternative way to insert the components of the type `exampleStruct1` into an Any value:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.

CORBA::TypeCode_var tc;
// create the TypeCode for exampleStruct1.
...
// create the DynAny object to represent the any value
CORBA::DynAny_var dv = orb->create_dyn_struct(tc);

CORBA::String_var s = (const char*)"Hello";
CORBA::Double     d = 3.1416;
CORBA::Long       l = 1;

CORBA::NameValuePairSeq sq;
sq.length(3);
sq[0].id = (const char*)"s";
sq[0].value <<= CORBA::Any::from_string(s,0);
                              // 1st element contains member s
sq[1].id = (const char*)"d";
sq[1].value <<= d;            // 2nd element contains member d
sq[2].id = (const char*)"l";
sq[2].value <<= l;            // 3rd element contains member l

dv->set_members(sq);
```

Notice that the name-value pairs in the argument to `set_members()` must match the members of the struct exactly or the object would throw the `InvalidSeq` exception.

In addition to the `current_component()` operation, the DynStruct interface provides two operations: `current_member_name()` and `current_member_kind()`, to return information about the current component.

10.4 The DynSequence Interface

Like struct values, sequence values can be traversed using the operations introduced in section 10.2. The first sequence element can be accessed as the first DynAny component, the second sequence element as the second DynAny component and so on.

To extract component values from an Any containing a sequence, the length of the sequence can be obtained using the get length operation in the DynSequence interface. Here is an example to extract the components of a sequence of long:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.
Any v;
...       // Initialise v to contain a value of a sequence of long
CORBA::DynAny_var dv = orb->create_dyn_any(v);

CORBA::DynSequence_var ds = CORBA::DynSequence::_narrow(dv);
CORBA::ULong len = ds->length();     // extract the length of the sequence
CORBA::ULong index;
for (index = 0; index < len; index++) {
  CORBA::Long v = ds->get_long();
  cerr << "[" << index << "] = " << v << endl;
}
```

Conversely, the set length operation is provided to set the length of the sequence. Here is an example to insert the components of a sequence of long:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.

CORBA::TypeCode_var tc;
// create the TypeCode for a sequence of long.
...
// create the DynAny object to represent the any value
CORBA::DynSequence_var ds = orb->create_dyn_sequence(tc);

CORBA::ULong len = 3;

ds->length(len);             // set the length of the sequence

CORBA::ULong index;
for (index = 0; index < len; index++) {
  ds->insert_long(index);    // insert a sequence element
}
```

Similar to the DynStruct interface, the `get_elements()` operation is provided to return all the sequence elements and the `set_elements()` operation is provided to insert all the sequence elements.

10.5 The DynArray Interface

Array values are handled by the DynArray interface. The DynArray interface is the same as the DynSequence interface except that the former does not provide the set length and get length operations.

10.6 The DynEnum Interface

Enum values are handled by the DynEnum interface. A DynEnum object contains a single component which is the enum value. This value cannot be extracted or inserted using the get and insert operations of the DynAny interface. Instead, two pairs of operations are provided to handle this value.

The `value_as_string()` operation allows the enum value to be extracted or inserted as a string. The `value_as_ulong()` operation allows the enum value to be extracted or inserted as an unsigned long.

10.7 The DynUnion Interface

Union values are handled by the DynUnion interface. Unfortunately, the CORBA 2.2 specification does not define the DynUnion interface in sufficient details to nail down its intended usage[5]. In this section, we try to fill in the gaps and describe a sensible way to use the DynUnion interface. Where necessary, the semantics of the operations is clarified. It is possible that the behavior of this interface in another ORB is different from this implementation. Where appropriate, we give warnings on usage that might cause problems with portability.

In relation to the current component pointer (section 10.2.1.1), a DynUnion object contains two components. The first component (with the index value equals 0) is the discriminator value, the second one is the member value. Therefore, one can use the `seek()` and `current_component()` operations to obtain a reference to the DynAny objects that handle the two components. However, it is better to use the operations defined in the DynUnion interface to manipulate these components as the semantics of the operations is easier to understand.

10.7.1 Three Categories of Union

Before we continue, it is important to understand that unions can be classified into the following categories:

1. One that has a default branch defined in the IDL. This will be called *explicit default union* in the rest of this section.

2. One that has no default branch and not all the possible values of the discriminator type are covered by the branch labels in the IDL. This will be called *implicit default union*.

3. One that has no default branch but all the possible values of the discriminator type are covered. This will be called *no default union*.

[5] This interface is currently an open issue with the ORB revision task force.

Of the three categories, the implicit default union is interesting because by definition if the discriminator value is not equal to any of the branch labels, the union has *no* member. That is, the union value consists solely of the discriminator value.

10.7.2 Example: extract data values from a union

10.7.2.1 Explicit default union

Consider a union of the following type:

```
// IDL
union exampleUnion1 switch(boolean) {
case TRUE: long l;
default:   double d;
};
```

The most straightforward way to extract the member value is as follows:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.

Any v;
...        // Initialise v to contain a value of type exampleUnion1.

CORBA::DynAny_var dv = orb->create_dyn_any(v);
CORBA::DynUnion_var du = CORBA::DynUnion::_narrow(dv);

CORBA::String_var di = du->member_name();
CORBA::DynAny_var dm = du->member();

if (strcmp((const char*)di,"l") == 0) {
  // branch label is TRUE
  CORBA::Long v = dm->get_long();
  cerr << "l = " << v << endl;
}

if (strcmp((const char*)di,"d") == 0) {
  // Is default branch
  CORBA::Double v = dm->get_double();
  cerr << "d = " << v << endl;
}
```

In the example, the operation `member_name()` is used to determine which branch the union has been instantiated. The operation `member()` is used to obtain a reference to the DynAny object that handles the member.

Alternatively, the branch can be determined by reading the discriminator value:

```
// C++
CORBA::DynAny_var di = du->discriminator();
CORBA::DynAny_var dm = du->member();
```

```
CORBA::Boolean di_v = di->get_boolean();

switch (di_v) {
case 1:
  CORBA::Long v = dm->get_long();
  cerr << "l = " << v << endl;
  break;
default:
  CORBA::Double v = dm->get_double();
  cerr << "d = " << v << endl;
}
```

The operation `discriminator()` is used to obtain the value of the discriminator.

Finally, the third way to determine the branch is to test if the default is selected:

```
// C++
switch (dv->set_as_default()) {
case 1:
  CORBA::Double v = dm->get_double();
  cerr << "d = " << v << endl;
  break;
default:
  CORBA::Long v = dm->get_long();
  cerr << "l = " << v << endl;
}
```

The operation `set_as_default()()` returns TRUE (1) if the discriminator has been assigned a valid default value.

10.7.2.2 Implicit default union

Consider a union of the following type:

```
// IDL
union exampleUnion2 switch(long) {
case 1: long l;
case 2: double d;
};
```

This example is similar to the previous one but there is no default branch. The description above also applies to this example. However, the discriminator may be set to neither 1 nor 2. Under this condition, the implicit default is selected and the union value contains the discriminator only!

When the discriminator contains an implicit default value, one might ask what is the value returned by the `member_name()` and `member()` operation. Since there is no member in the union value, omniORB returns a null string and a nil DynAny reference respectively. This behavior is not specified in the CORBA 2.2 specification. To ensure that your application is portable, it is best to avoid calling these operations when the DynUnion object might contain an implicit default value.

10.7.2.3 No default union

This is the last union category. For instance:

```
// IDL
union exampleUnion3 switch(boolean) {
case TRUE: long l;
case FALSE: double d;
};
```

In this example, all the possible values of the discriminator are used as union labels. There is no default branch. The only difference between this category and the explicit default union is that the `set_as_default()` operation always returns FALSE (0).

10.7.3 Example: insert data values into a union

Writing into a union involves selecting the union branch with the appropriate discriminator value and then writing the member value. There are three ways to set the discriminator value:

1. Use the `member_name()()` write operation to specify the union branch by specifying the union member directly. This operation has the side effect of setting the discriminator to the label value of the branch.

2. Write the label value of a union branch into the DynAny object that handles the discriminator.

3. If the union has a default branch, either explicitly or implicitly, use the `set_as_default()()` write operation to set the discriminator to a valid default value.

The following example shows the three ways of writing into a union:

```
// C++
CORBA::ORB_ptr orb;  // orb initialised by CORBA::ORB_init.

CORBA::TypeCode_var tc;
// create the TypeCode for exampleUnion1.
...
// create the DynAny object to represent the any value
CORBA::DynUnion_var dv = orb->create_dyn_union(tc);

CORBA::Any_var v;
DynAny_ptr dm;
```

```
// Use member_name to select the union branch
dv->member_name("l");
dm = dv->member();
dm->insert_long(10);
v = dv->to_any();          // transfer to an Any
CORBA::release(dm);

// Setting the discriminator value to select the union branch
CORBA::DynAny_var di = dv->discriminator();
di->insert_boolean(1);     // set discriminator to label TRUE
dm = dv->member();
dm->insert_long(20);
v = dv->to_any();          // transfer to an Any
CORBA::release(dm);

// Use set_as_default to select the default union branch
dv->set_as_default(1);
dm = dv->member();
dm->insert_double(3.14);
v = dv->to_any();          // transfer to an Any
CORBA::release(dm);

dv->destroy();
```

10.7.3.1 Ambiguous usage

1. When the discriminator is set to a different value, a different member branch is selected. Suppose the application has previously obtained a DynAny reference to a union member when it changes the discriminator value. As a result of the value change, the union is now instantiated to another union branch, i.e. a call to the `member()` operation will now return a reference to a different DynAny object. If the application continues to access the DynAny object of the old union member, the behavior of the ORB under this condition is not defined by the CORBA 2.2 specification. With omniORB, the DynAny object of the old union member is detached from the union when a new union branch is selected. Therefore reading or writing this object will not have any relation to the current value of the union. To avoid this ambiguity, the reference to the old union member should be released before a different union branch is selected.

2. The write operation `set_as_default()` takes a boolean argument. It is ambiguous to call this function with the argument set to FALSE (0). With omniORB, such a call will be silently ignored.

3. It is also ambiguous to pass the value TRUE (1) to the `set_as_default()` operation when the union is a no default union (10.7.1). With omniORB, such a call will be silently ignored.

4. When the discriminator value is not set, calling the `member()` operation is ambiguous. With omniORB, such a call will return a nil DynAny reference. Similarly, a call to the `member_kind()` operation under this condition will return `tk_null`.

To ensure portability, it is best to avoid using the DynUnion interface and not to rely on the ORB to behave as omniORB does under these ambiguous conditions.

10.8 Duplicate DynAny References

Like any CORBA object and psuedo object references, a DynAny reference can be duplicated using the `_duplicate()()` operations. When an application has obtained multiple DynAny references to the same DynAny object, it should be noted that a change made to the object by invoking on one reference is also visible through the other references. In particular, if a call through one reference has caused the current component pointer to be changed, subsequent calls through other references will operate on the new current component pointer.

10.9 Other Operations

The following is a short summary of the other operations in the DynAny interface which have not been covered in previous sections:

`assign()` initialises a DynAny object with another DynAny object. The two objects must have the same typecode.

`from_any()` initialises a DynAny object from the value in an any. The typecode in the two objects must be the same.

`copy()` creates a new DynAny object whose value is a deep copy of the current object.

`type()` returns the typecode associated with the DynAny object.

Chapter 11

The Dynamic Invocation Interface

The Dynamic Invocation Interface (or DII) allows applications to invoke operations on CORBA objects about which they have no static information. That is to say the application has not been linked with stub code which performs the remote operation invocation. Thus using the DII applications may invoke operations on *any* CORBA object, possibly determining the object's interface dynamically by using an Interface Repository.

This chapter presents an overview of the Dynamic Invocation Interface. A toy example use of the DII can be found in the omniORB distribution in the `src/examples/dii` directory. The DII makes extensive use of the type Any, so ensure that you have read chapter 9. For more information refer to the Dynamic Invocation Interface and C++ Mapping sections of the CORBA specification [OMG99].

11.1 Overview

To invoke an operation on a CORBA object an application needs an object reference, the name of the operation and a list of the parameters. In addition the application must know whether the operation is one-way, what user-defined exceptions it may throw, any user-context strings which must be supplied, a 'context' to take these values from and the type of the returned value. This information is given by the IDL interface declaration, and so is normally made available to the application via the stub code. In the DII this information is encapsulated in the `CORBA::Request` pseudo-object.

To perform an operation invocation the application must obtain an instance of a `Request` object, supply the information listed above and call one of the methods to actually make the invocation. If the invocation causes an exception to be thrown then this may be retrieved and inspected, or the return value on success.

11.2 Pseudo Objects

The DII defines a number of psuedo-object types, all defined in the CORBA namespace. These objects behave in many ways like CORBA objects. They should only be accessed by reference (through foo_ptr or foo_var), may not be instantiated directly and should be released by calling CORBA::release()[1]. A nil reference should only be represented by foo::_nil().

These pseudo objects, although defined in pseudo-IDL in the specification do not follow the normal mapping for CORBA objects. In particular the memory management rules are different—see the CORBA 2.3 specification [OMG99] for more details. New instances of these objects may only be created by the ORB. A number of methods are defined in CORBA::ORB to do this.

11.2.1 Request

A Request encapsulates a single operation invocation. It may *not* be re-used—even for another call with the same arguments.

```
class Request {
public:
  virtual Object_ptr        target() const;
  virtual const char*       operation() const;
  virtual NVList_ptr        arguments();
  virtual NamedValue_ptr    result();
  virtual Environment_ptr   env();
  virtual ExceptionList_ptr exceptions();
  virtual ContextList_ptr   contexts();
  virtual Context_ptr       ctxt() const;
  virtual void              ctx(Context_ptr);

  virtual Any& add_in_arg();
  virtual Any& add_in_arg(const char* name);
  virtual Any& add_inout_arg();
  virtual Any& add_inout_arg(const char* name);
  virtual Any& add_out_arg();
  virtual Any& add_out_arg(const char* name);

  virtual void set_return_type(TypeCode_ptr tc);
  virtual Any& return_value();

  virtual Status  invoke();
  virtual Status  send_oneway();
  virtual Status  send_deferred();
  virtual Status  get_response();
  virtual Boolean poll_response();
```

[1] if not managed by a _var type.

```
  static Request_ptr _duplicate(Request_ptr);
  static Request_ptr _nil();
};
```

11.2.2 NamedValue

A pair consisting of a string and a value—encapsulated in an Any. The name is optional. This type is used to encapsulate parameters and returned values.

```
class NamedValue {
public:
  virtual const char* name() const;
  // Retains ownership of return value.

  virtual Any* value() const;
  // Retains ownership of return value.

  virtual Flags flags() const;

  static NamedValue_ptr _duplicate(NamedValue_ptr);
  static NamedValue_ptr _nil();
};
```

11.2.3 NVList

A list of `NamedValue` objects.

```
class NVList {
public:
  virtual ULong count() const;
  virtual NamedValue_ptr add(Flags);
  virtual NamedValue_ptr add_item(const char*, Flags);
  virtual NamedValue_ptr add_value(const char*, const Any&, Flags);
  virtual NamedValue_ptr add_item_consume(char*,Flags);
  virtual NamedValue_ptr add_value_consume(char*, Any*, Flags);
  virtual NamedValue_ptr item(ULong index);
  virtual Status remove (ULong);

  static NVList_ptr _duplicate(NVList_ptr);
  static NVList_ptr _nil();
};
```

11.2.4 Context

Represents a set of context strings.

```
class Context {
public:
```

```
  virtual const char* context_name() const;
  virtual CORBA::Context_ptr parent() const;
  virtual CORBA::Status create_child(const char*, Context_out);
  virtual CORBA::Status set_one_value(const char*, const CORBA::Any&);
  virtual CORBA::Status set_values(CORBA::NVList_ptr);
  virtual CORBA::Status delete_values(const char*);
  virtual CORBA::Status get_values(const char* start_scope,
                                   CORBA::Flags op_flags,
                                   const char* pattern,
                                   CORBA::NVList_out values);
  // Throws BAD_CONTEXT if <start_scope> is not found.
  // Returns a nil NVList in <values> if no matches are found.

  static Context_ptr _duplicate(Context_ptr);
  static Context_ptr _nil();
};
```

11.2.5 ContextList

A `ContextList` is a list of strings, and is used to specify which strings from the 'context' should be sent with an operation.

```
class ContextList {
public:
  virtual ULong count() const;
  virtual void add(const char* ctxt);
  virtual void add_consume(char* ctxt);
  // consumes ctxt

  virtual const char* item(ULong index);
  // retains ownership of return value

  virtual Status remove(ULong index);

  static ContextList_ptr _duplicate(ContextList_ptr);
  static ContextList_ptr _nil();
};
```

11.2.6 ExceptionList

`ExceptionLists` contain a list of TypeCodes—and are used to specify which user-defined exceptions an operation may throw.

```
class ExceptionList {
public:
  virtual ULong count() const;
  virtual void add(TypeCode_ptr tc);
  virtual void add_consume(TypeCode_ptr tc);
  // Consumes <tc>.
```

```
  virtual TypeCode_ptr item(ULong index);
  // Retains ownership of return value.

  virtual Status remove(ULong index);

  static ExceptionList_ptr _duplicate(ExceptionList_ptr);
  static ExceptionList_ptr _nil();
};
```

11.2.7 UnknownUserException

When a user-defined exception is thrown by an operation it is unmarshalled into a value of type Any. This is encapsulated in an `UnknownUserException`. This type follows all the usual rules for user-defined exceptions—it is not a pseudo object, and its resources may be released by using `delete`.

```
class UnknownUserException : public UserException {
public:
  UnknownUserException(Any* ex);
  // Consumes <ex> which MUST be a UserException.

  virtual ~UnknownUserException();

  Any& exception();

  virtual void _raise();
  static const UnknownUserException* _downcast(const Exception*);
  static UnknownUserException* _downcast(Exception*);
  static UnknownUserException* _narrow(Exception*);
  // _narrow is a deprecated function from CORBA 2.2,
  // use _downcast instead.
};
```

11.2.8 Environment

An `Environment` is used to hold an instance of a system exception or an `UnknownUserException`.

```
class Environment {
  virtual void exception(Exception*);
  virtual Exception* exception() const;
  virtual void clear();

  static Environment_ptr _duplicate(Environment_ptr);
  static Environment_ptr _nil();
};
```

11.3 Creating Requests

`CORBA::Object` defines three methods which may be used to create a `Request` object which may be used to perform a single operation invocation on that object:

```
class Object {
  ...
  Status _create_request(Context_ptr ctx,
                         const char* operation,
                         NVList_ptr arg_list,
                         NamedValue_ptr result,
                         Request_out request,
                         Flags req_flags);

  Status _create_request(Context_ptr ctx,
                         const char* operation,
                         NVList_ptr arg_list,
                         NamedValue_ptr result,
                         ExceptionList_ptr exceptions,
                         ContextList_ptr ctxlist,
                         Request_out request,
                         Flags req_flags);

  Request_ptr _request(const char* operation);
  ...
};
```

`operation` is the name of the operation—which is the same as the name given in IDL. To access attributes the name should be prefixed by `_get_` or `_set_`.

In the first two cases above the list of parameters may be supplied. If the parameters are not supplied in these cases, or `_request()` is used then the parameters (if any) may be specified using the `add_*_arg()` methods on the `Request`. You must use one method or the other—not a mixture of the two. For *in*/*inout* arguments the value must be initialised, for *out* arguments only the type need be given. Similarly the type of the result may be specified by passing a `NamedValue` which contains an Any which has been initialised to contain a value of that type, or it may be specified using the `set_return_type()` method of `Request`.

When using `_create_request()`, the management of any pseudo-object references passed in remains the responsibility of the application. That is, the values are not consumed—and must be released using `CORBA::release()`. The CORBA specification is unclear about when these values may be released, so to be sure of portability do not release them until after the request has been released. Values which are not needed need not be supplied—so if no parameters are specified then it defaults to an empty parameter list. If no result type is specified then it defaults to void. A `Context` need only be given if a non-empty `ContextList` is specified. The `req_flags` argument is not used in the C++ mapping.

11.3.1 Examples

An operation might be specified in IDL as:

```
short anOpn(in string a);
```

An operation invocation may be created as follows:

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB3");
...
CORBA::NVList_var args;
orb->create_list(1, args);
*(args->add(CORBA::ARG_IN)->value()) <<= (const char*) "Hello World!";

CORBA::NamedValue_var result;
orb->create_named_value(result);
result->value()->replace(CORBA::_tc_short, 0);

CORBA::Request_var req = obj->_create_request(CORBA::Context::_nil(),
                                              "anOpn", args, result, 0);
```

or alternatively and much more concisely:

```
CORBA::Request_var req = obj->_request("anOpn");
req->add_in_arg() <<= (const char*) "Hello World!";
req->set_return_type(CORBA::_tc_short);
```

11.4 Invoking Operations

Once the `Request` object has been properly constructed, the operation may be invoked by calling one of the following methods on the request object:

`invoke()` blocks until the request has completed. The application should then test to see if an exception was raised. Since the CORBA spec is not clear about whether or not system exceptions should be thrown from this method, a runtime configuration variable is supplied so that you can specify the behavior:

```
namespace omniORB {
  ...
  CORBA::Boolean diiThrowsSysExceptions;
  ...
};
```

If this is FALSE, and the application should call the `env()` method of the request to retrieve an exception (it returns 0 (nil) if no exception was generated). If it is TRUE then system exceptions will be thrown out of `invoke()`. User-defined exceptions are always passed via `env()`, which will return a pointer to a `CORBA::UnknownUserException`. The application can determine which type of exception was returned by `env()()` by calling the `_narrow()` method defined for each exception type.

Warning

In pre-omniORB 2.8.0 releases, the default value of `diiThrowsSysExceptions` is FALSE. From omniORB 2.8.0 onwards, the default value is TRUE.

After determining that no exception was thrown the application may retrieve any returned values by calling `return_value()` and `arguments()`.

`send_oneway()` has the same semantics as a *oneway* IDL operation. It is important to note that oneway operations have at-most-once semantics, and it is not guaranteed that they will not block. Any operation may be invoked 'oneway' using the DII, even if it was not declared as 'oneway' in IDL. A system exception may be generated, in which case it will either be thrown or may be retrieved using `env()` depending on `diiThrowsSysExceptions` as above.

`send_deferred()` initiates the invocation, and then returns without waiting for the result. At some point in the future the application must retrieve the result of the operation—but other than testing for completion of the operation the application must not call any of the request's methods in the meantime.

- `get_response()` blocks until the reply is received.
- `poll_response()` returns TRUE if the reply has been received, and FALSE if not. It does not block.

Once `poll_response()` has returned TRUE, or `get_response()` has been called and returned, the application may test for an exception and retrieve returned values as above. If `diiThrowsSysExceptions` is true, then a system exception may be thrown from `get_response()`. From omniORB 2.8.0 onwards, `poll_response()` will raise a system exception if one has occurred during the invocation. Previously, `poll_response()` would not raise an exception, so if polling, the application also had to call another method to give the request an opportunity to raise the exception. This could be one of the methods to retrieve values from the request, or `get_response()`.

11.5 Multiple Requests

The following methods are provided by the ORB to enable multiple requests to be invoked asynchronously.

```
namespace CORBA {
  ...
  class ORB {
  public:
    ...
```

```
    Status send_multiple_requests_oneway(const RequestSeq&);
    Status send_multiple_requests_deferred(const RequestSeq&);
    Boolean poll_next_response();
    Status get_next_response(Request_out);
    ...
  };
  ...
};
```

send_multiple_requests_oneway() is used to invoke a number of oneway requests. An attempt will be made to invoke each of the requests, even if one or more of the early requests fails. The application may check for failure of any of the requests by testing the request's `env()` method. System exceptions are never raised by this method.

send_multiple_requests_deferred() will initiate an invocation of each of the given requests, and return without waiting for the reply. At some point in the future the application must retrieve the reply by calling `get_next_response()`, which returns a completed request. If no requests have yet completed it will block. This method never throws exceptions—the request's `env()` method must be used to determine if an exception was generated. If not then any returned values may then be queried.

`poll_next_response()` returns TRUE if there are any completed requests, and FALSE otherwise, without blocking. If this returns true then the next call to `get_next_response()` will not block. However, if another thread may also be calling `get_next_response()` then it could retrieve the completed message first—in which case this thread might block.

There are no guarantee as to the order in which replies will be received. If multiple threads are using this interface then it is not even guaranteed that a thread will receive replies to the requests it sent. Any thread may receive replies to requests sent by any other thread. It is legal to call `get_next_response()` even if no requests have yet been invoked—in which case the calling thread blocks until another thread invokes a request and the reply is received.

Chapter 12

The Dynamic Skeleton Interface

The Dynamic Skeleton Interface (or DSI) allows applications to provide implementations of the operations on CORBA objects without static knowledge of the object's interface. It is the server-side equivalent of the Dynamic Invocation Interface.

This chapter presents the Dynamic Skeleton Interface and explains how to use it. A toy example use of the DSI can be found in the omniORB distribution in the `src/examples/dsi` directory. For further information refer to the Dynamic Skeleton Interface and C++ Mapping sections of the CORBA 2.3 specification.

The DSI interface has changed in CORBA 2.3. omniORB 3 uses the new mapping, but since the mapping depends on `PortableServer::Current`, which is not yet implemented, not all facilities are available. This chapter describes an approach to building DSI servers which works with omniORB 3.

12.1 Overview

When an ORB receives an invocation request, the information includes the object reference and the name of the operation. Typically this information is used by the ORB to select a servant object and call into the implementation of the operation (which knows how to unmarshal the parameters etc.). The Dynamic Skeleton Interface however makes this information directly available to the application—so that it can implement the operation (or pass it on to another server) without static knowledge of the interface. In fact it is not even necessary for the server to always implement the same interface on any particular object!

To provide an implementation for one or more objects an application must subclass `PortableServer::DynamicImplementation` and override the method `invoke()`. An instance of this class is registered with a POA and is assigned an object reference (see below). When the ORB receives a request for that object the `invoke()` method is called and will be passed a `CORBA::ServerRequest` object which provides:

- the operation name

- context strings
- access to the parameters
- a way to set the returned values
- a way to throw user-defined exceptions.

12.2 DSI Types

12.2.1 PortableServer::DynamicImplementation

This class must be sub-classed by the application to provide an implementation for DSI objects. The method `invoke()` will be called for each operation invocation.

```
namespace PortableServer {
  ...

  class DynamicImplementation : public virtual ServantBase {
  public:
    virtual ~DynamicImplementation();

    CORBA::Object_ptr _this();
    // Must only be called from within invoke(). Caller must release
    // the reference returned.

    virtual void invoke(CORBA::ServerRequest_ptr request) = 0;
    virtual char* _primary_interface(const ObjectId& oid, POA_ptr poa) = 0;

    virtual CORBA::Boolean _is_a(const char* logical_type_id);
    // The default implementation uses _primary_interface(),
    // but may be overridden by subclasses.
  };
  ...
};
```

12.2.2 ServerRequest

A `ServerRequest` object provides the interface between a dynamic implementation and the ORB.

```
namespace CORBA {
  ...

  class ServerRequest {
  public:
    virtual const char* operation() = 0;
    virtual void        arguments(NVList_ptr& parameters) = 0;
    virtual Context_ptr ctx() = 0;
```

```
      virtual void        set_result(const Any& value) = 0;
      virtual void        set_exception(const Any& value) = 0;

    protected:
      inline ServerRequest() {}
      virtual ~ServerRequest();
    };
    ...
  };
```

12.3 Creating Dynamic Implementations

The application must override the `invoke()` method of `DynamicImplementation` to provide an implementation for DSI objects. This method must behave as follows:

- It may be called concurrently by multiple threads of execution, and so must be thread-safe.
- It may not throw any exceptions. Both user-defined and system exceptions are passed in a value of type Any via a call to `ServerRequest::set_exception()`.
- The operations on the `ServerRequest` object must be carried out in the correct order, as described below.

12.3.1 Operations on the ServerRequest

`operation()` will return the name of the operation, and may be called at any time. For attribute access the operation name is the IDL name of the attribute, prefixed by `_get_` or `_set_`. If the operation name is not recognised a `CORBA::BAD_OPERATION` exception should be passed back through `set_exception()`. This will allow the ORB to then see if it is one of the standard object operations.

Firstly `arguments()` must be called passing a `CORBA::NVList`[1] which must be initialised to contain the type and mode of the parameters. The ORB consumes this value and will release it when the operation is complete. At this point any *in*/*inout* arguments will be unmarshalled, and when this operation returns, their values will be in the `NVList`. The application may set the value of *inout*/*out* arguments by modifying this parameter list.

If the operation has user-context information, then `ctx()` must be called after `arguments()` to retrieve it.

`set_result()` must then be called exactly once if the operation has a non-void return value (unless an exception is thrown). The value passed should be an Any allocated with `new`, and will be freed by the ORB.

[1] obtained by calling `CORBA::ORB::create_list()`

At any point in the above sequence `set_exception()` may be called to set a user-defined exception or a system exception. If this happens then no further operations should be invoked on the `ServerRequest` object, and the `invoke()` method should return.

Within the `invoke()` method `_this()` may be called to obtain the object reference. This method may not be used at any other time.

12.4 Registering Dynamic Objects

To use a `DynamicImplementation` servant, a CORBA object must be created and associated with it, just as for any other servant. Dynamic servants can also be created on demand by Servant Managers, just like static servants.

12.5 Example

This implementation of `DynamicImplementation::invoke()` is taken from an example which can be found in the omniORB distribution. The `echoString()` operation is declared in IDL as:

```
string echoString(in string mesg);
```

Here is the Dynamic Implementation Routine:

```
void
MyDynImpl::invoke(CORBA::ServerRequest_ptr request)
{
  try {
    if( strcmp(request->operation(), "echoString") )
      throw CORBA::BAD_OPERATION(0, CORBA::COMPLETED_NO);

    CORBA::NVList_ptr args;
    orb->create_list(0, args);
    CORBA::Any a;
    a.replace(CORBA::_tc_string, 0);
    args->add_value("", a, CORBA::ARG_IN);

    request->arguments(args);

    const char* mesg;
    *(args->item(0)->value()) >>= mesg;

    CORBA::Any* result = new CORBA::Any();
    *result <<= CORBA::Any::from_string(mesg, 0);
    request->set_result(*result);
  }
  catch(CORBA::SystemException& ex){
    CORBA::Any a;
```

```
      a <<= ex;
      request->set_exception(a);
    }
    catch(...){
      cout << "echo_dsiimpl: MyDynImpl::invoke - caught an unknown exception."
           << endl;
      CORBA::Any a;
      a <<= CORBA::UNKNOWN(0, CORBA::COMPLETED_NO);
      request->set_exception(a);
    }
  }
```

Appendix A

hosts_access(5)

DESCRIPTION

This manual page describes a simple access control language that is based on client (host name/address, user name), and server (process name, host name/address) patterns. Examples are given at the end. The impatient reader is encouraged to skip to the EXAMPLES section for a quick introduction.

An extended version of the access control language is described in the hosts_options(5) document. The extensions are turned on at program build time by building with -DPROCESS_OPTIONS.

In the following text, *daemon* is the process name of a network daemon process, and *client* is the name and/or address of a host requesting service. Network daemon process names are specified in the inetd configuration file.

ACCESS CONTROL FILES

The access control software consults two files. The search stops at the first match:

- Access will be granted when a (daemon,client) pair matches an entry in the `/etc/hosts.allow` file.
- Otherwise, access will be denied when a (daemon,client) pair matches an entry in the `/etc/hosts.deny` file.
- Otherwise, access will be granted.

A non-existing access control file is treated as if it were an empty file. Thus, access control can be turned off by providing no access control files.

ACCESS CONTROL RULES

Each access control file consists of zero or more lines of text. These lines are processed in order of appearance. The search terminates when a match is found.

- A newline character is ignored when it is preceded by a backslash character. This permits you to break up long lines so that they are easier to edit.

- Blank lines or lines that begin with a # character are ignored. This permits you to insert comments and whitespace so that the tables are easier to read.

- All other lines should satisfy the following format, things between [] being optional: `daemon_list : client_list [ : shell_command ]`

`daemon_list` is a list of one or more daemon process names (argv[0] values) or wildcards (see below).

`client_list` is a list of one or more host names, host addresses, patterns or wildcards (see below) that will be matched against the client host name or address.

The more complex forms `daemon@host` and `user@host` are explained in the sections on server endpoint patterns and on client username lookups, respectively.

List elements should be separated by blanks and/or commas.

With the exception of NIS (YP) netgroup lookups, all access control checks are case insensitive.

PATTERNS

The access control language implements the following patterns:

- A string that begins with a `.` character. A host name is matched if the last components of its name match the specified pattern. For example, the pattern `.tue.nl` matches the host name `wzv.win.tue.nl`.

- A string that ends with a `.` character. A host address is matched if its first numeric fields match the given string. For example, the pattern `131.155.` matches the address of (almost) every host on the Eindhoven University network (`131.155.x.x`).

- A string that begins with an character is treated as an NIS (formerly YP) netgroup name. A host name is matched if it is a host member of the specified netgroup. Netgroup matches are not supported for daemon process names or for client user names.

- An expression of the form `n.n.n.n/m.m.m.m` is interpreted as a 'net/mask' pair. A host address is matched if 'net' is equal to the bitwise AND of the address and the 'mask'. For example, the net/mask pattern `131.155.72.0/255.255.254.0` matches every address in the range `131.155.72.0` to `131.155.73.255`.

WILDCARDS

The access control language supports explicit wildcards:

`ALL`
: The universal wildcard, always matches.

`LOCAL`
: Matches any host whose name does not contain a dot character.

`UNKNOWN`
: Matches any user whose name is unknown, and matches any host whose name or address are unknown. This pattern should be used with care: host names may be unavailable due to temporary name server problems. A network address will be unavailable when the software cannot figure out what type of network it is talking to.

`KNOWN`
: Matches any user whose name is known, and matches any host whose name and address are known. This pattern should be used with care: host names may be unavailable due to temporary name server problems. A network address will be unavailable when the software cannot figure out what type of network it is talking to.

`PARANOID`
: Matches any host whose name does not match its address. When tcpd is built with -DPARANOID (default mode), it drops requests from such clients even before looking at the access control tables. Build without -DPARANOID when you want more control over such requests.

OPERATORS

`EXCEPT`
: Intended use is of the form: `list_1 EXCEPT list_2`; this construct matches anything that matches `list_1` unless it matches `list_2`. The `EXCEPT` operator can be used in `daemon_lists` and in `client_lists`. The `EXCEPT` operator can be nested: if the control language would permit the use of parentheses, `a EXCEPT b EXCEPT c` would parse as `(a EXCEPT (b EXCEPT c))`.

SHELL COMMANDS

If the first-matched access control rule contains a shell command, that command is subjected to `%<letter>` substitutions (see next section). The result is executed by a /bin/sh child process with standard input, output and error connected to

/dev/null. Specify an `&` at the end of the command if you do not want to wait until it has completed.

Shell commands should not rely on the PATH setting of the inetd. Instead, they should use absolute path names, or they should begin with an explicit `PATH=whatever` statement.

The hosts_options(5) document describes an alternative language that uses the shell command field in a different and incompatible way.

% EXPANSIONS

The following expansions are available within shell commands:

`%a (%A)` The client (server) host address.

`%c` Client information: user@host, user@address, a host name, or just an address, depending on how much information is available.

`%d` The daemon process name (argv[0] value).

`%h (%H)` The client (server) host name or address, if the host name is unavailable.

`%n (%N)` The client (server) host name (or "unknown" or "paranoid").

`%p` The daemon process id.

`%s` Server information: daemon@host, daemon@address, or just a daemon name, depending on how much information is available.

`%u` The client user name (or "unknown").

`%%` Expands to a single `%` character.

Characters in % expansions that may confuse the shell are replaced by underscores.

SERVER ENDPOINT PATTERNS

In order to distinguish clients by the network address that they connect to, use patterns of the form:

```
process_name@host_pattern :  client_list ...
```

Patterns like these can be used when the machine has different internet addresses with different internet hostnames. Service providers can use this facility to offer FTP, GOPHER or WWW archives with internet names that may even belong to different organisations. See also the 'twist' option in the hosts_options(5) document. Some systems (Solaris, FreeBSD) can have more than one internet address on one physical interface; with other systems you may have to resort to SLIP or PPP pseudo interfaces that live in a dedicated network address space.

The `host_pattern` obeys the same syntax rules as host names and addresses in `client_list` context. Usually, server endpoint information is available only with connection-oriented services.

CLIENT USERNAME LOOKUP

When the client host supports the RFC 931 protocol or one of its descendants (TAP, IDENT, RFC 1413) the wrapper programs can retrieve additional information about the owner of a connection. Client username information, when available, is logged together with the client host name, and can be used to match patterns like:

```
daemon_list :  ...  user_pattern@host_pattern ...
```

The daemon wrappers can be configured at compile time to perform rule-driven username lookups (default) or to always interrogate the client host. In the case of rule-driven username lookups, the above rule would cause username lookup only when both the `daemon_list` and the `host_pattern` match.

A user pattern has the same syntax as a daemon process pattern, so the same wildcards apply (netgroup membership is not supported). One should not get carried away with username lookups, though.

- The client username information cannot be trusted when it is needed most, i.e. when the client system has been compromised. In general, ALL and (UN)KNOWN are the only user name patterns that make sense.
- Username lookups are possible only with TCP-based services, and only when the client host runs a suitable daemon; in all other cases the result is 'unknown'.
- A well-known UNIX kernel bug may cause loss of service when username lookups are blocked by a firewall. The wrapper README document describes a procedure to find out if your kernel has this bug.
- Username lookups may cause noticeable delays for non-UNIX users. The default timeout for username lookups is 10 seconds: too short to cope with slow networks, but long enough to irritate PC users.

Selective username lookups can alleviate the last problem. For example, a rule like:

```
daemon_list :  @pcnetgroup ALL@ALL
```

would match members of the pc netgroup without doing username lookups, but would perform username lookups with all other systems.

DETECTING ADDRESS SPOOFING ATTACKS

A flaw in the sequence number generator of many TCP/IP implementations allows intruders to easily impersonate trusted hosts and to break in via, for example, the remote shell service. The IDENT (RFC931 etc.) service can be used to detect such and other host address spoofing attacks.

Before accepting a client request, the wrappers can use the IDENT service to find out that the client did not send the request at all. When the client host provides IDENT service, a negative IDENT lookup result (the client matches `UNKNOWN@host`) is strong evidence of a host spoofing attack.

A positive IDENT lookup result (the client matches `KNOWN@host`) is less trustworthy. It is possible for an intruder to spoof both the client connection and the IDENT lookup, although doing so is much harder than spoofing just a client connection. It may also be that the client's IDENT server is lying.

Note: IDENT lookups don't work with UDP services.

EXAMPLES

The language is flexible enough that different types of access control policy can be expressed with a minimum of fuss. Although the language uses two access control tables, the most common policies can be implemented with one of the tables being trivial or even empty.

When reading the examples below it is important to realise that the allow table is scanned before the deny table, that the search terminates when a match is found, and that access is granted when no match is found at all.

The examples use host and domain names. They can be improved by including address and/or network/netmask information, to reduce the impact of temporary name server lookup failures.

MOSTLY CLOSED

In this case, access is denied by default. Only explicitly authorised hosts are permitted access.

The default policy (no access) is implemented with a trivial deny file:

```
/etc/hosts.deny:
    ALL: ALL
```

This denies all service to all hosts, unless they are permitted access by entries in the allow file.

The explicitly authorised hosts are listed in the allow file. For example:

```
/etc/hosts.allow:
   ALL: LOCAL @some_netgroup
   ALL: .foobar.edu EXCEPT terminalserver.foobar.edu
```

The first rule permits access from hosts in the local domain (no . in the host name) and from members of the `some_netgroup` netgroup. The second rule permits access from all hosts in the `foobar.edu` domain (notice the leading dot), with the exception of `terminalserver.foobar.edu`.

MOSTLY OPEN

Here, access is granted by default; only explicitly specified hosts are refused service.

The default policy (access granted) makes the allow file redundant so that it can be omitted. The explicitly non-authorised hosts are listed in the deny file. For example:

```
/etc/hosts.deny:
   ALL: some.host.name, .some.domain
   ALL EXCEPT in.fingerd: other.host.name, .other.domain
```

The first rule denies some hosts and domains all services; the second rule still permits finger requests from other hosts and domains.

BOOBY TRAPS

The next example permits tftp requests from hosts in the local domain (notice the leading dot). Requests from any other hosts are denied. Instead of the requested file, a finger probe is sent to the offending host. The result is mailed to the superuser.

```
/etc/hosts.allow:
   in.tftpd: LOCAL, .my.domain

/etc/hosts.deny:
   in.tftpd: ALL: (/some/where/safe\_finger -l @%h | \
       /usr/ucb/mail -s %d-%h root) &
```

The `safe_finger` command comes with the tcpd wrapper and should be installed in a suitable place. It limits possible damage from data sent by the remote finger server. It gives better protection than the standard finger command.

The expansion of the %h (client host) and %d (service name) sequences is described in the section on shell commands.

Warning: do not booby-trap your finger daemon, unless you are prepared for infinite finger loops.

On network firewall systems this trick can be carried even further. The typical network firewall only provides a limited set of services to the outer world. All other services can be "bugged" just like the above tftp example. The result is an excellent early-warning system.

DIAGNOSTICS

An error is reported when a syntax error is found in a host access control rule; when the length of an access control rule exceeds the capacity of an internal buffer; when an access control rule is not terminated by a newline character; when the result of expansion would overflow an internal buffer; when a system call fails that shouldnt́. All problems are reported via the syslog daemon.

FILES

`/etc/hosts.allow`, (daemon,client) pairs that are granted access.
`/etc/hosts.deny`, (daemon,client) pairs that are denied access.

SEE ALSO

tcpd(8) tcp/ip daemon wrapper program.
tcpdchk(8), tcpdmatch(8), test programs.

BUGS

If a name server lookup times out, the host name will not be available to the access control software, even though the host is registered.

Domain name server lookups are case insensitive; NIS (formerly YP) netgroup lookups are case sensitive.

AUTHOR

Wietse Venema (wietse@wzv.win.tue.nl)
Department of Mathematics and Computing Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513,
5600 MB Eindhoven, The Netherlands

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.

[OMG96] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, 1996. From `http://cgi.omg.org/corba/cichpter.html`.

[OMG98] Object Management Group. *CORBAServices: Common Object Services Specification*, December 1998.

[OMG99] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.3.1 edition, October 1999. From `http://cgi.omg.org/corba/cichpter.html`.

[OMG00] Object Management Group. *Interoperable Naming Service revised chapters*, August 2000. From `http://cgi.omg.org/cgi-bin/doc?ptc/00-08-07`.

[Ric96a] Tristan Richardson. *The OMNI Development Environment Version 4.0*. AT&T Laboratories Cambridge, November 1996.

[Ric96b] Tristan Richardson. *The OMNI Thread Abstraction*. AT&T Laboratories Cambridge, October 1996.