The omniORB2 version 2.5 User's Guide

Sai-Lai Lo
(*email: slo@orl.co.uk*)
Olivetti & Oracle Research Laboratory

10 Feb, 1998

Contents

Chapter 1

Introduction

OmniORB2 is an Object Request Broker (ORB) that implements the 2.0 specification of the Common Object Request Broker Architecture (CORBA) [OMG96a]. This user guide tells you how to use omniORB2 to develop CORBA applications. It assumes a basic understanding of CORBA.

In this chapter, we give an overview of the main features of omniORB2 and what you need to do to setup your environment to run omniORB2.

1.1 Features

1.1.1 CORBA 2 compliant

OmniORB2 implements the Internet Inter-ORB Protocol (IIOP). This protocol provides omniORB2 the means of achieving interoperability with the ORBs implemented by other vendors. In fact, this is the native protocol used by omniORB2 for the communication amongst its objects residing in different address spaces. Moreover, the IDL to C++ language mapping provided by omniORB2 conforms to the latest revision of the CORBA specification. Type Any and TypeCode are now supported (introduced in version 2.5.0).

1.1.2 Multithreading

OmniORB2 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 ORL, 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 Unices, Windows NT, Windows 95 and our in-house developed systems for our own hardware. OmniORB2 have been ported to all these platforms. **The IDL to C++ mapping for these targets are all the same**.

OmniORB2 uses real C++ exceptions and nested classes. We stay with 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** to **C++ classes** instead of **namespaces**.

OmniORB2 relies on the native thread libraries to provide the multithreading capability. A small class library (omnithread [Richardson96a]) is used to encapsulated the (possibly different) APIs of the native thread libraries. In the 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

OmniORB2 is not (yet) a complete implementation of the CORBA core. The following is a list of the missing features.

- The BOA only support the persistent server activation policy. Other dynamic activation and deactivation policies are not supported.
- The Dynamic Invocation Interface is not supported.
- The Dynamic Skeleton Interface is not supported.
- OmniORB2 does not has its own Interface Repository.

These features may be implemented in the medium term. It is best to check out the latest status on the omniORB2 home page (`http://www.orl.co.uk/omniORB/omniORB.html`).

1.2 Setting Up Your Environment

After you have unpacked the distribution, read all the README files at the top level of the directory tree. These files contain essential information on installing, building and using omniORB2 on the supported platforms.

The following is a checklist of what you have to do:

1. Setup the naming service. An implementation of the COS Naming Service, called omniNames, is provided in this distribution. If you want to use the service, you have to start it up first. Consult the document "The OMNI Naming Service" for details. When omniNames starts up, it writes the stringified object reference for

its root context on standard error. This is needed by the omniORB2 runtime. See below for how to configure the runtime. You can also use other naming service implementations provided that you can obtain the stringified object reference for its root context.

2. Configure the omniORB2 runtime. At startup the omniORB runtime tries to read the configuration file `omniORB.cfg` to obtain the object reference to the root context of the Naming Service. This object reference is returned by the call `resolve_initial_references("NameService")`.

   (a) On Unix platforms, omniORB2 looks for the environment variable `OMNIORB_CONFIG`. If this variable is defined, it contains the pathname of the omniORB2 configuration file. If the variable is not set, omniORB2 will use the compiled-in pathname (`/etc/omniORB.cfg`) to locate the file.

   (b) On Win32 platforms (Windows NT, Windows '95), omniORB2 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 format of the entry is the word `NAMESERVICE` followed by space and the stringified IOR all on one line. For example:

```
NAMESERVICE IOR:00fff71c0000002849444c3a6f6d672e6f72672f436f734e616d696e
672f4e616d696e67436f6e746578743a312e300000000001000000000000002c00010000
00000012776962626c652e776f62626c652e636f6d0004d20000000c34c35a8305a931a2
00000001
```

   Aternatively, the stringified IOR can be placed in the system registry on Win32 platforms, in the (string) value `NAMESERVICE`, under the key `HKEY_LOCAL_MACHINE\SOFTWARE\ORL\omniORB\2.0`.

1.3 Compiler Flags

You should be able to build the whole distribution using the makefiles provided. The makefiles are configured to supply a set of preprocessor defines that are necessary to compile omniORB2 programs. The preprocessor defines are needed because the same set of header files are used for all platforms. If you are to incorporate omniORB2 into your own development environment, you **must** specify the following preprocessor defines to identify a 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
IBM AIX 4.x	__aix__ __powerpc__ __OSVERSION__=4
Linux 2.0 (x86)	__x86__ __linux__ __OSVERSION__=2
Linux 2.0 (alpha)	__alpha__ __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
ATMos 4.0	__arm__ __atmos__ __OSVERSION__=4
NextStep 3.x	__m68k__ __nextstep__ __OSVERSION__=3

You should also specify the preprocessor defines (e.g. -D_REENTRANT) for compiling multithreaded programs.

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 `omniORB2/CORBA.h`. Write into the file the appropriate set of preprocessor defines and add `#include <omniORB2/sitedef.h>` at the beginning of `omniORB2/CORBA_sysdep.h`.

Chapter 2

The Basics

In this chapter, we go through three examples to illustrate the practical steps to use omniORB2. 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 omniORB2 specific. For instance, the object implementation skeleton is covered in section 2.4.2.

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 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) [Richardson96b] is also included.

2.2 Specifying the Echo interface in IDL

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

[1] The stub code is the C++ code that provides the object mapping as defined in the CORBA 2.0 specification.

```
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 2.0 [OMG96a].

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`. If you are using ODE, all IDL files should have the same extension- `.idl` and 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. This practice should be avoided for the sake of object reusuability. 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 omniORB2 is called `omniidl2`. Given the IDL file, `omniidl2` 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`

If you are using ODE, you don't need to invoke omniidl2 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.

2.4 A Quick Tour of the C++ stubs

The C++ stubs conform to the mapping defined in the CORBA 2.0 specification (chapter 16-18). 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.

2.4.1 Object Reference

The use of an object interface denotes an object reference. 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:

```
class Echo;
typedef Echo* Echo_ptr;

class Echo : public virtual omniObject, public virtual CORBA::Object {
public:

  virtual char *  echoString ( const char *  mesg ) = 0;
  static Echo_ptr _nil();
  static Echo_ptr _duplicate(Echo_ptr);
  static Echo_ptr _narrow(CORBA::Object_ptr);

  ... // 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 `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 **operation==** must not be performed on the type.

In addition to `echoString`, the mapping also defines three static member functions in the class Echo: `_nil`, `_duplicate`, and `_narrow`. Note that these are operations on an object reference.

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 for `CORBA::Object`. Any 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 the 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.

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

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

You should not use an object reference once you have called `CORBA::release`. 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 object implementation**. Here is a concrete example, if the implementation of an object resides in a different address space, then a call to `CORBA::release` will only caused the resources associated with the object reference in the current address space to be deallocated. The object implementation in the other address space is unaffected.

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;
...            // initialised 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 [OMG96a] chapter 17.

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

a = p;               // a assumes ownership of p, must not use p anymore

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.
```

2.4.2 Object Implementation

Unlike the client side of an object, i.e. the use of object references, the CORBA specification 2.0 deliberately leave many of the necessary functionalities to implement an object unspecified. As a consequence, it is very unlikely the implementation code of an object on top of two different ORBs can be identical. However, most of the code are expected to be portable. In particular, the body of an operation implementation can normally be ported with no or little modification.

OmniORB2 uses C++ inheritance to provide the skeleton code for object implementation. For each object interface, a skeleton class is generated. In our example, the skeleton class `_sk_Echo` is generated for the Echo IDL interface. An object implementation can be written by creating an implementation class that derives from the skeleton class.

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

```
class _sk_Echo :  public virtual Echo {
public:
  _sk_Echo(const omniORB::objectKey& k);
  virtual char *  echoString ( const char *  mesg ) = 0;
  Echo_ptr        _this();
```

[2] In omniORB2, 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 casting when the compiler complains about the conversion being ambiguous.

```
  void              _obj_is_ready(BOA_ptr);
  void              _dispose();
  BOA_ptr           _boa();
  omniORB::objectKey _key();
  ... // methods generated for internal use
};
```

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. **Unless specified otherwise, the description below is omniORB2 specific.** The functions are:

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. **The signature of this function is specified by the CORBA specification**.

_this this function returns an object reference for the target object. The returned value must be deallocated via `CORBA::release`. See 2.7 for an example of how this function is used.

_obj_is_ready this function tells the Basic Object Adaptor[4] (BOA) that the object is ready to serve. Until this function is called, the BOA would not serve any incoming calls to this object. See 2.7 for an example of how this function is used.

_dispose this function tells the BOA to dispose of the object. The BOA will stop serving incoming calls of this object and remove any resources associated with it. See 2.7 for an example of how this function is used.

_boa this function returns a reference to the BOA that serves this object.

_key this function returns the key that the ORB used to identify this object. The type `omniORB::objectKey` is opaque to application code. The function `omniORB::keyToOctetSequence` can be used to convert the key to a sequence of octets.

2.5 Writing the object implementation

You define an implementation class to provide the object 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 hooks for the ORB to perform upcalls to your implementation.

Here is a simple implementation of the Echo object.

[4] The interface of a BOA is described in chapter 8 of the CORBA specification.

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

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

There are three points to note here:

Storage Responsibilities A string, which is used 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 who's responsibility to allocate and release their associated storage. As a rule of thumb, the client (or the caller to the implementation functions) owns the storage of all IN arguments, the object implementation (or the callee) must copy the data if it wants to retain a copy. For OUT arguments and return values, the object implementation allocates the storage and passes the ownership to the client. The client must release the storage when the variables will no longer be used. For details, please refer to Table 24-27 of the CORBA specification.

Multi-threading As omniORB2 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.

Instantiation You must not instantiate an implementation as automatic variables. Instead, you should always instantiate an implementation using the new operator, i.e. its storage is allocated on the heap. The reason behind this restriction will become clear in section 2.7.

2.6 Writing the client

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

```
void
hello(CORBA::Object_ptr obj)
{
  Echo_var e = Echo::_narrow(obj);                // line 1

  if (CORBA::is_nil(e)) {                         // line 2
    cerr << "hello: cannot invoke on a nil object reference.\n" << endl;
    return;
  }
```

```
  CORBA::String_var src = (const char*) "Hello!";  // line 3
  CORBA::String_var dest;                           // line 4

  dest = e->echoString(src);                        // line 5

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

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 1, 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. Line 2 and 5 shows 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` variable `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 function returns. Line 5 shows how `CORBA::String_var` variables are used. They can be used in place of a string (for which the mapping is `char*`)[5]. As used in line 3, assigning a constant string (`const char*`) to a `CORBA::String_var` causes the string to be copied. On the otherhand, assigning a `char*` to a `CORBA::String_var`, as used in line 5, causes the latter to assume the ownership of the string[6].

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.

2.7 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. In this section, we describe an example in which both the client and the object implementation are in the same address space. In the next two sections, we shall describe the case where the two are in different address spaces.

The code for this example is reproduced below:

[5] A conversion operator() of CORBA::String_var converts a CORBA::String_var to a char*.

[6] Please refer to the CORBA specification 16.7 for the details of the String_var mapping. Other T_var types are also covered in chapter 16.

```
int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");   // line 1
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA"); // line 2

  Echo_i *myobj = new Echo_i();                                 // line 3
  myobj->_obj_is_ready(boa);                                    // line 4

  boa->impl_is_ready(0,1);                                      // line 5

  Echo_ptr myobjRef = myobj->_this();                           // line 6
  hello(myobjRef);                                              // line 7
  CORBA::release(myobjRef);                                     // line 8

  myobj->_dispose();                                            // line 9
  return 0;
}
```

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

2.7.1 ORB/BOA initialisation

line 1 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 omniORB2, this argument must either be "omniORB2" or NULL. If it is NULL, there must be an argument, -ORBid "omniORB2", in `argv`. Like any 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 "omniORB2", the initialisation will fail and a nil `ORB_ptr` will be returned. If supplied, omniORB2 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 2.9.1. If any error occurs during the processing of the configuration file, the system exception CORBA::INITIALIZE is raised.

line 2 The BOA is initialised by calling the ORB's `BOA_init`. The 3rd argument must either be "omniORB2_BOA" or NULL. If it is NULL, then `argv` must contain an argument, -BOAid "omniORB2_BOA". If the BOA identifier is not "omniORB2_BOA", the initialisation will fail and a nil BOA_ptr will be returned. Like `ORB_init`, any command-line arguments understood by `BOA_init` will be removed from `argv`.

2.7.2 Object initialisation

line 3 An instance of the Echo object is initialised using the `new` operator.

line 4 The object's `_obj_is_ready` is called. This function informs the BOA that this object is ready to serve. Until this function is called, the BOA will not accept any invocation on the object and will not perform any upcall to the object.

line 5 The BOA's `impl_is_ready` is called. This function tells the BOA the implementation is ready. After this call, the BOA will accept IIOP requests from other address spaces. There are 2 points to note here:

1. `boa→impl_is_ready` can be called any time after `BOA_init` is called (line 2). In other words, object instances can be initialised and advertised to the BOA before or after this function is called.
2. The 2nd argument[7] to `impl_is_ready` tells the ORB whether this call should be non-blocking. The default value of this argument is FALSE(0) and the call will block indefinitely within the ORB. If there are more things the main thread should do after it calls `impl_is_ready`, as it is the case in this example, the non-blocking option (TRUE=1) should be specified. Whether the main thread blocks in this call or not, the ORB is not affected because its functions are provided by other threads spawned internally. Notice that the signature of `impl_is_ready` in the CORBA specification does not have the 2nd argument[8]. Therefore, calling `impl_is_ready` with the non-blocking option is omniORB2 specific.

2.7.3 Client invocation

line 6 The object reference is obtained from the implementation by calling `_this`. Like any object reference, the return value of _this must be released by `CORBA::release` when it is no longer needed.

line 7 Call `hello` with this object reference. The argument is widened implicitly to the generic object reference `CORBA::Object_ptr`.

line 8 Release the object reference.

One of the important characteristic 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 object is colocated in the same address space or resided in a different address space.

In case of colocated client and object implementation, omniORB2 is able to short-circuit the client calls to direct calls on the implementation methods. The cost of an invocation is reduced to that of a function call. This optimisation is applicable **not only** to object references returned by the _this function but to any object references that are passed around within the same address space or received from other address spaces via IIOP calls.

[7] The 1st argument is a pointer to the implementation definition and is always ignored by omniORB2.

[8] The CORBA specification does not specify when `impl_is_ready` should return. Many ORB vendors choose to implement `impl_is_ready` as blocking until a certain time-out value is exceeded. In a single threaded implementation this is necessary to give the ORB the time to serve incoming requests.

2.7.4 Object disposal

line 9 To dispose of an object implementation and release all the resources associated with it, the `_dispose` function is called. In fact, this is the **only** clean way to get rid of an object implementation. Even though the object is created using the new operator in the application code, the application should never call the delete operator on the object directly.

Once an application calls `_dispose` on an object implementation, the pointer to the object should not be used any more. At the time the `_dispose` call is made, there may be other threads invoking on the object, omniORB2 ensures that all these calls are completed before removing the object from its internal tables and releasing the resources associated with it. The storage associated with the object is released by omniORB2 using the delete operator. This is why all object implementation should be initialised using the new operator (section 2.5).

The disposal of an object implementation by omniORB2 may also be deferred when **colocated** clients continue to hold on to copies of the object's reference[9]. This behavior is to prevent the short-circuited calls from the clients to fail unpredictably.

To summarise, an application can make no assumption as to when the object is disposed by omniORB2 after the `_dispose` call returns. If it is necessary to have better control on when to stop serving incoming requests, the work should be done by the object implementation itself, such as by keeping track of the current serving state.

2.8 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 need 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 COS Naming Service.

2.8.1 Object Implementation: Generating a Stringified Object Reference

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

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

[9] Object references held by clients in other address spaces will not prevent the object implementation from being disposed of. If these clients invoke on the object after it is disposed, the system exception INV_OBJREF is raised.

```
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  Echo_i *myobj = new Echo_i();
  myobj->_obj_is_ready(boa);

  {
    Echo_var myobjRef = myobj->_this();
    CORBA::String_var p;

    p = orb->object_to_string(myobjRef);            //line 1

    cerr << "'" << (char*)p << "'" << endl;
  }

  boa->impl_is_ready();    // block here indefinitely
                           // See the explanation in example 1
  return 0;
}
```

The stringified object reference is obtained by calling the ORB's function `_object_to_string` (line 1). This is a sequence starting with the signature "IOR:" and followed by a hexadecimal string. All CORBA 2.0 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[10]. From the IOR, an object reference can be constructed.

2.8.2 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 function `string_to_object` to convert the string into a generic object reference (`CORBA::Object_ptr`). The relevant section of the code is reproduced below. The full listing (`eg2_clt.cc`) can be found at the end of this chapter.

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

2.8.3 Catching System Exceptions

When omniORB2 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

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

and recover from the error condition[11]. For instance, the code fragment, shown in section 2.8.2, catches the system exception COMM_FAILURE which indicates that communication with the object implementation in another address space has failed.

All system exceptions inherit from the class `CORBA::SystemException`. With compilers that support RTTI[12][13], a single catch `CORBA::SystemException` will catch all the different system exceptions thrown by omniORB2.

When omniORB2 detects an internal inconsistency 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 carries by `omniORB::fatalException` contains the exact location (the file name and the line number) where the exception is raised. You are strongly encourage to file a bug report and point out the location.

2.8.4 Lifetime of an Object Implementation

It may be obvious but it has to stated that an object implementation exists only for the duration of the process's lifetime. When the same program is run again, a different instance of the object implementation is created. More significantly, **the IOR, and hence the object reference, of this instance is different from that of the previous run**.

For instance, if you look at the stringified object reference produced by the program `eg2_impl` in different runs, they are all different. The implication is that you cannot store away the stringified object reference and expect to be able to use it again later when the original program run has terminated.

For system services and other applications, it may be desirable to have "persistent" object implementations. The objects are "persistent" in the sense that they can be contacted using the same IOR when they are instantiated in different program runs. To provide this functionality, omniORB2 needs to be provided with two pieces of information: the (network) location and the object key. The details of how this can be done will be described in the later part of this manual.

Alternatively, an indirection from textual pathnames to object references can be used. Applications can register object implementations at runtime to 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 naming service, which is a component of the Common Object Services (COS) [OMG96b], that can be used for this purpose. The next section describes an example of how to use the COS Naming Service.

---

[11] 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 system will cause a core file to be produced.

[12] Run Time Type Identification

[13] A noticeable exception is the GNU C++ compiler (version 2.7.2). It doesn't support RTTI unless the compilation flag -frtti is specified. The omniORB2 runtime is not compiled with the -frtti flag. It is said that RTTI will be properly supported in the upcoming version 2.8.

2.9 Example 3 - Using the COS Naming Service

In this example, the object implementation uses the COS Naming Service [OMG96b] to pass on the object reference to the client. This method is by-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 object reference is bound to the pathname "**test**/**Echo**"[14]. The pathname consists of the context **test** and the object name **Echo**. Both the context and the object name has an attribute **kind**. This attribute is a string that is intended to be used 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 for there is no standardised set of kind values.

2.9.1 Obtaining the Root Context Object Reference

The initial contact with the Naming Service can be established via what we called 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,"omniORB2");

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

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

Remember, omniORB2 constructs its internal list of initial references at initialisation time using the information provided in the configuration file `omniORB.cfg`. If this file is not present, the internal list will be empty and `resolve_initial_references` will raise a CORBA::ORB::InvalidName exception.

2.9.2 The Naming Service Interface

It is beyond the scope of this chapter to describe in detail the Naming Service interface. You should consult the CORBAservices specification [OMG96b] (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.

[14] A pathname, or in the Naming Service's terminology- a *compound name*, is a sequence of textual names. Each name component except the last one is 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. Note: '/' is purely a notation to separate two components in the pathname. It does not appear in the *compound name* that is registered with the Naming Service.

2.10 Source Listing

2.10.1 echo_i.cc

```
// echo_i.cc - This source code demonstrates an implmentation of the
//             object interface Echo. It is part of the three examples
//             used in Chapter 2 "The Basics" of the omniORB2 user guide.
//
#include <string.h>
#include "echo.hh"

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

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

2.10.2 greeting.cc

```
// greeting.cc - This source code demonstrates the use of an object
//               reference by a client to perform an operation on an
//               object. It is part of the three examples used
//               in Chapter 2 "The Basics" of the omniORB2 user guide.
//
#include <iostream.h>
#include "echo.hh"

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

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

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

  dest = e->echoString(src);

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

2.10.3 eg1.cc

```
// eg1.cc - This is the source code of example 1 used in Chapter 2
//          "The Basics" of the omniORB2 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"

#include "echo_i.cc"
#include "greeting.cc"

int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  Echo_i *myobj = new Echo_i();
  // Note: all implementation objects must be instantiated on the
  // heap using the new operator.

  myobj->_obj_is_ready(boa);
  // Tell the BOA the object is ready to serve.
  // This call is omniORB2 specific.
  //
  // This call is equivalent to the following call sequence:
  //     Echo_ptr myobjRef = myobj->_this();
  //     boa->obj_is_ready(myobjRef);
  //     CORBA::release(myobjRef);

  boa->impl_is_ready(0,1);
  // Tell the BOA we are ready and to return immediately once it has
  // done its stuff. It is omniORB2 specific to call impl_is_ready()
  // with the extra 2nd argument- CORBA::Boolean NonBlocking,
  // which is set to TRUE (1) in this case.

  Echo_ptr myobjRef = myobj->_this();
  // Obtain an object reference.
  // Note: always use _this() to obtain an object reference from the
  //       object implementation.

  hello(myobjRef);

  CORBA::release(myobjRef);
  // Dispose of the object reference.

  myobj->_dispose();
  // Dispose of the object implementation.
```

```
  // This call is omniORB2 specific.
  // Note: *never* call the delete operator or the dtor of the object
  //       directly because the BOA needs to be informed.
  //
  // This call is equivalent to the following call sequence:
  //     Echo_ptr myobjRef = myobj->_this();
  //     boa->dispose(myobjRef);
  //     CORBA::release(myobjRef);

  return 0;
}
```

2.10.4 eg2_impl.cc

```
// eg2_impl.cc - This is the source code of example 2 used in Chapter 2
//               "The Basics" of the omniORB2 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 "omnithread.h"
#include "echo.hh"

#include "echo_i.cc"

int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  Echo_i *myobj = new Echo_i();
  myobj->_obj_is_ready(boa);

  {
    Echo_var myobjRef = myobj->_this();
    CORBA::String_var p = orb->object_to_string(myobjRef);
    cerr << "'" << (char*)p << "'" << endl;
  }

  boa->impl_is_ready();
  // Tell the BOA we are ready. The BOA's default behaviour is to block
  // on this call indefinitely.

  return 0;
}
```

2.10.5 eg2_clt.cc

```
// eg2_clt.cc - This is the source code of example 2 used in Chapter 2
//              "The Basics" of the omniORB2 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"

#include "greeting.cc"

extern void hello(CORBA::Object_ptr obj);

int
main (int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

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

  try {
    CORBA::Object_var obj = orb->string_to_object(argv[1]);
    hello(obj);
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "object." << endl;
  }
  catch(omniORB::fatalException& ex) {
    cerr << "Caught omniORB2 fatalException. This indicates a bug is caught "
         << "within omniORB2.\nPlease send a bug report.\n"
         << "The exception was thrown in file: " << ex.file() << "\n"
         << "                            line: " << ex.line() << "\n"
         << "The error message is: " << ex.errmsg() << endl;
  }
  catch(...) {
    cerr << "Caught a system exception." << endl;
  }

  return 0;
}
```

2.10.6 eg3_impl.cc

```
// eg3_impl.cc - This is the source code of example 3 used in Chapter 2
//               "The Basics" of the omniORB2 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]
//               |
//              text  [context] kind [my_context]
//               |
//              Echo  [object]  kind [Object]
//

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

#include "echo_i.cc"

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

int
main(int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  Echo_i *myobj = new Echo_i();
  myobj->_obj_is_ready(boa);

  {
    Echo_var myobjRef = myobj->_this();
    if (!bindObjectToName(orb,myobjRef)) {
      return 1;
    }
  }

  boa->impl_is_ready();
  // Tell the BOA we are ready. The BOA's default behaviour is to block
  // on this call indefinitely.

  return 0;
}
```

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

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

    // Narrow the object returned by resolve_initial_references()
    // to a CosNaming::NamingContext object:
    rootContext = CosNaming::NamingContext::_narrow(initServ);
    if (CORBA::is_nil(rootContext))
      {
        cerr << "Failed to narrow naming context." << endl;
        return 0;
      }
  }
  catch(CORBA::ORB::InvalidName& ex) {
    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, and assign testContext to it:
      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 tmpobj;
      tmpobj = rootContext->resolve(contextName);
      testContext = CosNaming::NamingContext::_narrow(tmpobj);
      if (CORBA::is_nil(testContext)) {
        cerr << "Failed to narrow naming context." << endl;
        return 0;
      }
```

```
    }

    // Bind the object (obj) to testContext, naming it Echo:
    CosNaming::Name objectName;
    objectName.length(1);
    objectName[0].id   = (const char*) "Echo";   // string copied
    objectName[0].kind = (const char*) "Object"; // string copied


    // Bind obj with name Echo to the testContext:
    try {
      testContext->bind(objectName,obj);
    }
    catch(CosNaming::NamingContext::AlreadyBound& ex) {
      testContext->rebind(objectName,obj);
    }
    // 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 (omniORB::fatalException& ex) {
    throw;
  }
  catch (...) {
    cerr << "Caught a system exception while using the naming service."<< endl;
    return 0;
  }
  return 1;
}
```

2.10.7 eg3_clt.cc

```
// eg3_clt.cc - This is the source code of example 3 used in Chapter 2
//                "The Basics" of the omniORB2 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"

#include "greeting.cc"

extern void hello(CORBA::Object_ptr obj);

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

int
main (int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  try {
    CORBA::Object_var obj = getObjectReference(orb);
    hello(obj);
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "object." << endl;
  }
  catch(omniORB::fatalException& ex) {
    cerr << "Caught omniORB2 fatalException. This indicates a bug is caught "
         << "within omniORB2.\nPlease send a bug report.\n"
         << "The exception was thrown in file: " << ex.file() << "\n"
         << "                            line: " << ex.line() << "\n"
         << "The error message is: " << ex.errmsg() << endl;
  }
  catch(...) {
```

```
    cerr << "Caught a system 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 initServ;
    initServ = orb->resolve_initial_references("NameService");

    // Narrow the object returned by resolve_initial_references()
    // to a CosNaming::NamingContext object:
    rootContext = CosNaming::NamingContext::_narrow(initServ);
    if (CORBA::is_nil(rootContext))
      {
        cerr << "Failed to narrow naming context." << endl;
        return CORBA::Object::_nil();
      }
  }
  catch(CORBA::ORB::InvalidName& ex) {
    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.)


  CORBA::Object_ptr obj;
  try {
    // Resolve the name to an object reference, and assign the reference
    // returned to a CORBA::Object:
    obj = 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;
      return CORBA::Object::_nil();
    }
  catch (CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "naming service." << endl;
    return CORBA::Object::_nil();
  }
  catch(omniORB::fatalException& ex) {
    throw;
  }
  catch (...) {
    cerr << "Caught a system exception while using the naming service."<< endl;
    return CORBA::Object::_nil();
  }
  return obj;
}
```

Chapter 3

IDL to C++ Language Mapping

Now that you are familiar with the basics, it is important to familiar yourselves with the IDL to C++ language. The mapping is described in detail in [OMG96a]. If you have not done so, you should obtain a copy of the document and use that as the programming guide to omniORB2.

Chapter 4

The omniORB2 API

In this chapter, we introduce the omniORB2 API. The purpose of this API is to provide access points to omniORB2 specific functionalities that are 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 omniORB2 dependent code, this API is defined under the name space "omniORB"[1].

4.1 ORB and BOA initialisation options

`CORBA::ORB_init` accepts the following command-line arguments:

`-ORBid ''omniORB2''` The identifier supplied must be "omniORB2".

`-ORBtraceLevel <level>` See section 4.2.

`-ORBserverName <string>` See section 4.3.

`-ORBtcAliasExpand <0 or 1>` See section 9.2.

`BOA_init` accepts the following command-line arguments:

`-BOAid ''omniORB2_BOA''` The identifier supplied must be "omniORB2_BOA".

`-BOAiiop_port <port number>` This option tells the BOA which TCP/IP port to use to accept IIOP calls. If this option is not specified, the BOA will use an arbitrary port assigned by the operating system.

By default, the BOA 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 BOA should use. This can be done by defining the environment variable `OMNIORB_USEHOSTNAME_VAR` to contain the preferred host name or IP address in dot-numeric form.

As defined in the CORBA specification, any command-line arguments understood by the ORB/BOA 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.

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

4.2 Run-time Tracing and Diagnostic Messages

OmniORB2 uses the C++ iostream `cerr` to output any tracing and diagnostic messages. 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 end-points are shutdown

level 10-20 the above plus execution traces

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
```

4.3 Server Name

Applications can optionally specified a name to identify the server process. At the moment, this name is only used by the host-based access control module. See section 7.5 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>`.

4.4 Object Keys

OmniORB2 uses a data type `omniORB::objectKey` to uniquely identify each object implementation. This is an opaque data type and can only be manipulated by the following functions:

```
void omniORB::generateNewKey(omniORB::objectKey &k);
```

`omniORB::generateNewKey` returns a new `objectKey`. The return value is guaranteed to be unique among the keys generated during this program run. On the platforms that have a realtime clock and unique process identifiers, a stronger assertion can be made, i.e. the keys are guaranteed to be unique among all keys ever generated on the same machine.

```
const unsigned int omniORB::hash_table_size;
int omniORB::hash(omniORB::objectKey& k);
```

`omniORB::hash` returns the hash value of an `objectKey`. The value returned by this function is always between 0 and `omniORB:hash_table_size - 1` inclusively.

```
omniORB::objectKey omniORB::nullkey();
```

`omniORB::nullkey` always returns the same `objectKey` value. This key is guaranteed to hash to 0.

```
int operator==(const omniORB::objectKey &k1,const omniORB::objectKey &k2);
int operator!=(const omniORB::objectKey &k1,const omniORB::objectKey &k2);
```

`ObjectKeys` can be tested for equality using the overloaded `operator==` and `operator!=`.

```
omniORB::seqOctets*
omniORB::keyToOctetSequence(const omniORB::objectKey &k1);

omniORB::objectKey
omniORB::octetSequenceToKey(const omniORB::seqOctets& seq);
```

`omniORB::keyToOctetSequence` takes an `objectKey` and returns its externalised representation in the form of a sequence of octets. The same sequence can be converted back to an `objectKey` using `omniORB::octetSequenceToKey`. If the supplied sequence is not an `objectKey`, `omniORB::octetSequenceToKey` raises a `CORBA::MARSHAL` exception.

4.5 GIOP Message Size

omniORB2 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);
```

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.

4.6 Trapping omniORB2 Internal Errors

```
class fatalException {
public:
    const char *file() const;
    int line() const;
    const char *errmsg() const;
};
```

When omniORB2 detects an internal inconsistency 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 carries by `omniORB::fatalException` contains the exact location (the file name and the line number) where the exception is raised. You are strongly encourage to file a bug report and point out the location.

Chapter 5

The Basic Object Adaptor (BOA)

This chapter describes the BOA implementation in omniORB2. The CORBA specification defines the Basic Object Adaptor as the entity that mediates between object implementations and the ORB. Unfortunately, the BOA specification is incomplete and does not address the multi-threading issues appropriately. The end result is that different ORB vendors implement different extensions to their BOAs. Worse, the implementation of the operations defined in the specification are different in different ORBs. Recently, a new Object Adaptor specification (the Portable Object Adaptor- POA) has been adopted and will replace the BOA as the standard Object Adaptor in CORBA. The new specification recognises the compatibility problems of BOA and recommends that all BOAs should be considered propriety extensions. OmniORB2 will support POA in future releases. Until then, you have to use the BOA to attach object implementations to the ORB.

The rest of this chapter describes the interface of the BOA in detail. It is important to recognise that the interface described below is omniORB2 specific and hence the code using this interface is unlikely to be portable to other ORBs.

Unless it is stated otherwise, the term "object" will be used below to refer to object implementations. This should not be confused with "object references" which are handles held by clients.

5.1 BOA Initialisation

It takes two steps to put the BOA into service. The BOA has to be initialised using `BOA_init` and activated using `impl_is_ready`.
`BOA_init` is a member of the `CORBA::ORB` class. Its signature is:

```
BOA_ptr BOA_init(int & argc,
                 char ** argv,
                 const char * boa_identifier);
```

Typically, it is used in the startup code as follows:

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");   // line 1
CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA"); // line 2
```

The `argv` parameters may contain BOA options. These options will be removed from the `argv` list when `BOA_init` returns. Other parameters in `argv` will remain. The supported options are:

-BOAiiop_port `<port number (0-65535)>` Use the port number to receive IIOP requests. This option can be specified multiple times in the command line and the BOA would be initialised to listen on all of the ports.

-BOAid `<id (string)>` If this option is used the id must be "omniORB2_BOA".

If the third argument of `BOA_init` is non-nil, it must be the string constant "omniORB2_BOA". If the argument is nil, -BOAid must be present in `argv`.

If there is any problem in the initialisation process, a `CORBA::INITIALIZE` exception would be raised.

To register an object with the BOA, the method `_obj_is_ready` should be called with the return value of `BOA_init` as the argument.

`BOA_init` is thread-safe. It can be called multiple times and the same `BOA_ptr` will be returned. However, only the `argv` in the first call will be scanned, the argument is ignored in subsequent calls.

`BOA_init` returns a pseudo object of type `CORBA::BOA_ptr`. Similar to `CORBA::Object_ptr`, the pointer can be managed using `CORBA::BOA_var`, `BOA::_duplicate` and `CORBA::release`. The pointer can be tested using `CORBA::is_nil` which returns true if the pointer is equivalent to the return value of `BOA::_nil`.

After `BOA_init` is called, objects can be registered. However, incoming IIOP requests would not be despatched until `impl_is_ready` is called.

```
class BOA {
public:
   impl_is_ready(CORBA::ImplementationDef_ptr p = 0,
                 CORBA::Boolean NonBlocking = 0);
};
```

One of the common pitfall in using the BOA is to forget to call impl_is_ready. Until this call returns, there is no thread listening on the port from which IIOP requests are received. The remote client may hang because of this.

When `impl_is_ready` is called with no argument. The calling thread would be blocked indefinitely in the function until `impl_shutdown` (see below) is called. The thread that is calling `impl_is_ready` is not used by the BOA to perform its internal functions. The BOA has its own set of threads to process incoming requests and general housekeeping. Therefore, it is not necessary to have a thread blocked in the call if it can be put into use elsewhere. For example, the main thread may call `impl_is_ready` once in non-blocking mode (see below) and then enter the event loop to handle the GUI frontend.

If non-blocking behaviour is needed, the `NonBlocking` argument should be set to 1. For instance, if you creates a callback object, you might call impl_is_ready in non-blocking mode to tell the BOA to start receiving IIOP requests before sending the callback object to the remote object. The first argument `ImplementationDef_ptr` is ignored by the BOA. Just set the argument to nil.

`impl_is_ready` is thread safe and can be called multiple times. Multiple threads can be blocked in `impl_is_ready`.

5.2 Object Registration

Once the BOA is initialised, objects can be registered. The purpose of object registration is to let the BOA know of the existence of the object and to dispatch requests for the object as upcalls into the object.

To register an object, the `_obj_is_ready` function should be called. `_obj_is_ready` is a member function of the implementation skeleton class. The function should be called only once for each object. The call should be made only after the object is fully initialised.

The member function `obj_is_ready` of the BOA may also be used to register an object. However, this function has been superseded by `_obj_is_ready` and should not be used in new application code.

5.3 Object Disposal

Once an object is registered, it is under the management of the BOA. To remove the object from the BOA and to delete it (when it is safe to do so), the `_dispose` function should be called. `_dispose` is a member function of the implementation skeleton class. The function should be called only once for each object.

Notice the asymmetry in object instantiation and destruction. To instantiate an object, the application code has to call the **new** operator. To remove the object, the application should never call the delete operator on the object directly.

At the time the `_dispose` call is made, there may be other threads invoking on the object, the BOA ensures that all these calls are completed before removing the object from its internal tables and calling the **delete** operator.

Internally, the BOA keeps a reference count on each object. Initially, the reference count is 0. After a call to `_obj_is_ready`, the reference count is 1. The BOA increases the reference count by 1 before an upcall into the object is made. The count is decreased by 1 when the upcall returns. `_dispose` decreases the reference count by 1, if the reference count is 0, the delete operator is called. If the count is non-zero, the object is marked as disposed. The object will be deleted when the reference count eventually goes to zero.

The reference count is also increased by 1 for each object reference held in the same address space. Hence, the **delete** operator will not be called when there are outstanding object references in the same address space. To ensure that an object is deleted, all its object references in the same address space should be released using `CORBA::release`.

Unlike colocated object references, references held by clients in other address spaces would not prevent the deletion of objects. If these clients invoke on the object after it is disposed, the system exception INV_OBJREF would be raised. The difference in semantics is an undesirable side-effect of the current BOA implementation. In future, colocated references will have the same semantics as remote references, i.e. their presence will not delay the deletion of the objects.

Instead of `_dispose`, it may be useful to have a method to deactivate the object but not deleting it. This feature is not supported in the current BOA implementation.

5.4 BOA Shutdown

The BOA can be withdrawn from service using member functions `impl_shutdown` and `destroy`.

```
class BOA {
public:
   void impl_shutdown();
   void destroy();
};
```

`impl_shutdown` and `destroy` are the inverse of `impl_is_ready` and `BOA_init` respectively.

`impl_shutdown` deactivates the BOA. When the call returns, all the internal threads and network connections will be shutdown. Any thread blocking in `impl_is_ready` would be unblocked. After the call, no request from other address spaces will be processed. In other words, the BOA will be in the same state as it was in before `impl_is_ready` was called. For example, a remote client may hang if it tries to connect to the server after `impl_shutdown` was called because no thread is listening on the IIOP port.

`impl_shutdown` does not wait for incoming requests to complete before it closes the network connections. The remote clients will see the network connections shutdown and the replies may not reach them even if the upcalls have been completed. Therefore, if the application is to define an operation in an IDL interface to shutdown the BOA, the operation should be defined as an oneway operation.

`impl_shutdown` is thread-safe and can be called multiple times. The call is silently ignored if the BOA has already been shutdown. After `impl_shutdown` is called, the BOA can be reactivated by another call to `impl_is_ready`.

It should be noted that `impl_shutdown` does not affect outgoing network connections. That is, clients in the same address space will still be able to make calls to objects in other address spaces.

While remote requests are not delivered after `impl_shutdown` is called, the current implementation does not stop colocated clients from calling the objects. In future, colocated clients will exhibit the same behaviour as remote clients.

`destroy` permanently removed the BOA. This function will call `impl_shutdown` implicitly if it has not been called. When this call returns, the IIOP port(s) held by the BOA will be freed. Remote clients will see their requests refused by the operating system when they try to open a connection to the IIOP port(s).

After `destroy` is called, the BOA should not be used. If there is any objects still registered with the BOA, the objects should not be invoked afterwards. The objects are not disposed. Invoking on the objects after `destroy` would result in undefined behaviour. Initialisation of another BOA using `BOA_init` is not supported. The behaviour of `BOA_init` after this call is undefined.

5.5 Unsupported functions

The following member functions are not implemented. Calling these functions do not have any effect.

- `Object_ptr create(...)`
- `ReferenceData* get_id(Object_ptr)`
- `Principal_ptr get_principal(Object_ptr,Environment_ptr)`
- `void change_implementation(Object_ptr, ImplementationDef_ptr)`
- `void deactivate_impl(ImplementationDef_ptr)`
- `void deactivate_obj(Object_ptr)`

Chapter 6

Interface Type Checking

This chapter describes the mechanism used by omniORB2 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.

6.1 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);
   };

// C++
   C_ptr server;
   B_ptr objB;
   A_ptr objA = objB;
   server->op(objA);      // Send B as A
```

[1]For further details of the repository ID formats, see section 6.6 in the CORBA specification.

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. OmniORB2 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.

6.2 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 A_proxyObjectFactory class. This class is derived from the proxyObjectFactory class. The proxyObjectFactory is an abstract class which contains 3 virtual functions.

```
class proxyObjectFactory {
public:

  virtual const char *irRepoId() const = 0;

  virtual _CORBA_Boolean  is_a(const char *base_repoId) const = 0;

  virtual CORBA::Object_ptr newProxyObject(Rope *r,
                                           CORBA::Octet *key,
                                           size_t keysize,
                                           IOP::TaggedProfileList
                                           *profiles,
                                           CORBA::Boolean release) = 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.
- `newProxyObject` returns an object reference based on the information supplied in the arguments.

A single instance of every *_proxyObjectFactory 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 that 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 is linked into the executable but the stubs of B is 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.

6.3 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 mode 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();
   };

\\ C++
```

```
C_ptr objC;
A_ptr objA;
CORBA::Object_ptr objR;

objA =  objC->op1();              // line 1
(void) objA->_non_existent();     // line 2

objR =  objC->op2();              // line 3
objA =  A::_narrow(objR);         // line 4
```

If the stubs of A,B,C,D are linked into the executable and:

Case 1 `C::op1` and `C::op2` returns a B. Line 1-4 complete successful. The remote object is only contacted at line 2.

Case 2 `C::op1` and `C::op2` returns 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 stub of A is linked into the executable and:

Case 1 C::op1 and C::op2 returns a B. Line 1-4 completes successful. When line 2 and 4 is executed, the object is contacted to ask if it is a A.

Case 2 C::op1 and C::op2 returns a D. This condition only occurs if the runtime of the remote end is buggy. Line 1 completes and no exception is raised. At line 2, the object is contacted to ask if it is a A. If the answer is no, a CORBA::INV_OBJREF exception is raised. The application will also see a CORBA::INV_OBJREF at line 4.

Chapter 7

Connection Management

This chapter describes how omniORB2 manages network connections.

7.1 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 previously generated by the same ORB. 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 omniORB2 manages network connections and the programming interface to fine tune the management policy.

7.2 The Model

OmniORB2 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 upcall returns. There is no thread switching along the call chain.

With this design, there is at most one call in-flight at any time in 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, omniORB2 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 refers to objects in a different address space but unless the application invokes on these objects, no network connection is made.

It is wasteful to leave a connection opened 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 has 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.

7.3 Idle Connection Shutdown

Inside the ORB, two separate threads are dedicated to scan for idle connections. One thread is responsible for outgoing connections and the other looks after incoming connections. The thread for incoming connections is only created when the BOA is initialised because only then will there be any incoming connections.

The threads scan all opened connections once every "scan period". If a connection is found to be idle for two consecutive periods, it will be closed. The threads use mark-and-swipe to detect if a connection is idle. When a connection is checked, a status flag attached to the connection is set. Every remote invocation using that connection would clear the flag. So if a connection's status flag is found to be set in two consecutive scans, the connection has been idled during the scan period.

The scan period for incoming and outgoing connections can be individually controlled by the following API:

```
class omniORB {

public:

  enum idleConnType { idleIncoming, idleOutgoing };

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

  static CORBA::ULong idleConnectionScanPeriod(idleConnType direction);

};
```

The current value of the scan period (in seconds) is returned by the read-only `idleConnectionScanPeriod`. The scan period can be changed by the write-only `idleConnectionScanPeriod`. The default value (30 seconds) is compiled into the ORB runtime. The scan can be disabled completely by setting the scan period to 0. The scan period can be changed at any time. The write function is non-thread safe. Concurrent calls to this function could results in undefined behaviour.

7.4 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.

OmniORB2 implements this 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 omniORB2 server. The workaround is either to catch the exception in the application code and retries or to turn off the idle connection shutdown inside the omniORB2 server.

7.5 Connection Acceptance

OmniORB2 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 by passing the option: `-ORBserverName <string>` to `ORB_init`.

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
omniORB2 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
omniORB2 gateKeeper is not installed. All incoming are accepted.
I said,"Hello!". The Object said,"Hello!"
```

Chapter 8

Proxy Objects

When a client acquires a reference to an object in another address space, omniORB2 creates a local representation of the object and returns a pointer to this object as its object reference. The local representation is known as the proxy object.

The proxy object maps each IDL operation into a method to deliver invocations to the remote object. The method implements argument marshalling using the ORB runtime. When the ORB runtime detects an error condition, it may raise a system exception. These exceptions will normally be propagated by the proxy object to the application code. However, there may be applications that prefer to have the system exceptions trapped in the proxy object. For these applications, it is possible to install exception handlers for individual proxy object or all proxy objects. The API to do this will be explained in this chapter.

As described in section 6.2, proxy objects are created by instances of the proxyObjectFactory class. For each IDL interface A, the stubs of A contains a derived class of proxyObjectFactory (A_proxyObjectFactory). This derived class is responsible for creating proxy objects for A. This process is completely transparent to the application. However, there may be applications that require greater control on the creation of proxy objects or even want to change the behavior of the proxy objects. To cater for this requirement, applications can override the default proxyObjectFactories and install their own versions of proxyObjectFactories. The way to do this will be explained in this chapter.

8.1 System Exception Handlers

By default, all system exceptions, with the exception of CORBA::TRANSIENT, are propagated by the proxy objects 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.

OmniORB2 provides a set of functions to install exception handlers. Once they are installed, proxy objects will call these handlers when the target system exceptions are raised by the ORB runtime. Exception handlers can be installed for CORBA::TRANSIENT, CORBA::COMM_FAILURE and CORBA::SystemException. The 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 object or it can be installed for all proxy objects in the address space.

8.1.1 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 a maximum specified by `omniORB::defaultTransientRetryDelayMaximum`. The unit of both values are in seconds.

The ORB runtime will raised CORBA::TRANSIENT under the following conditions:

1. When a **cached** network connection is broken while an 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 functions `installTransientExceptionHandler` can be used to install the exception handlers for CORBA::TRANSIENT.

Two overloaded forms are available. The first form install an exception handler for all object references except for those which have an exception handler installed by the second form, which takes an addition argument to identify the target object reference. The argument `cookie` is an opaque pointer which will be passed on by the ORB when it calls the exception handler.

An exception handler will be called by proxy objects with three arguments. The `cookie` is the opaque pointer registered by `installTransientExceptionHandler`. The argument `n_retries` is the number of times the proxy has called this handler for the same invocation. The argument `ex` is the value of the exception caught. The exception handler is expected to do whatever is appropriate and returns a boolean value. If the return value is TRUE(1), the proxy object would retry the operation again. If the return value is FALSE(0), the CORBA::TRANSIENT exception would be 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.
}
```

8.1.2 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.

8.1.3 CORBA::SystemException

To report an error condition, the ORB runtime may raise other SystemExceptions. 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.

Application 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.

8.2 Proxy Object Factories

This section describes how an application can control the creation or change the behaviour of proxy objects.

8.2.1 Background

For each interface A, its stub contains a proxy factory class- `A_proxyObjectFactory`. This class is derived from `proxyObjectFactory` and implements three virtual functions:

```
class A_proxyObjectFactory : public virtual proxyObjectFactory {
```

```
public:

  virtual const char *irRepoId() const;

  virtual _CORBA_Boolean  is_a(const char *base_repoId) const;

  virtual CORBA::Object_ptr newProxyObject(Rope *r,
                                           CORBA::Octet *key,
                                           size_t keysize,
                                           IOP::TaggedProfileList
                                           *profiles,
                                           CORBA::Boolean release);

};
```

As described in chapter 6, the functions allow the ORB runtime to perform type checking. The function `newProxyObject` creates a proxy object for A based on its input arguments. The return value is a pointer to the class `_proxy_A` which is automatically re-casted into a `CORBA::Object_ptr`. `_proxy_A` implements the proxy object for A:

```
class _proxy_A :  public virtual A {
public:

  _proxy_A (Rope *r,
            CORBA::Octet *key,
            size_t keysize,IOP::TaggedProfileList *profiles,
            CORBA::Boolean release);
  virtual ~_proxy_A();

  // plus other internal functions.

};
```

The stub of A guarantees that exactly **one** instance of `A_proxyObjectFactory` is instantiated when an application is executed. The constructor of `A_proxyObjectFactory`, via its base class `proxyObjectFactory` links the instance into the ORB's proxy factory list.

Newly instantiated proxy object factories are always entered at the front of the ORB's proxy factory list. Moreover, when the ORB searches for a match on the type, it always stops at the first match. In other words, when additional instances of `A_proxyObjectFactory` or derived classes of it are created, the last instantiation will override earlier instantiations to be the proxy factory selected to create proxy objects of A. This property can be used by an application to install its own proxy object factories.

8.2.2 An Example

Using the `Echo` example in chapter 2 as the basis, one can tell the ORB to use a modified proxy object class to create proxy objects. The steps involved are as follows:

8.2.2.1 Define a new proxy class

We define a new proxy class to cache the result of the last invocation of `echoString`.

```
class _new_proxy_Echo : public virtual _proxy_Echo {
public:
  _new_proxy_Echo (Rope *r,
                   CORBA::Octet *key,
                   size_t keysize,IOP::TaggedProfileList *profiles,
                   CORBA::Boolean release)
         : _proxy_Echo(r,key,keysize,profiles,release) {}
   virtual ~_new_proxy_echo() {}


   virtual char* echoString(const char* mesg) {
     //
     // Only calls the remote object if the argument is different from the
     // last invocation.

     omni_mutex_lock sync(lock);
     if ((char*)last_arg) {
       if (strcmp(mesg,(char*)last_arg) == 0) {
          return CORBA::string_dup(last_result);
       }
     }
     char* res = _proxy_Echo::echoString(mesg);
     last_arg = mesg;
     last_result = (const char*) res;
     return res;
   }

private:
  omni_mutex        lock;
  CORBA::String_var last_arg;
  CORBA::String_var last_result;
};
```

8.2.2.2 Define a new proxy factory class

Next, we define a new proxy factory class to instantiate _new_proxy_Echo as proxy objects for Echo.

```
class _new_Echo_proxyObjectFactory : public virtual Echo_proxyObjectFactory
{
public:
   _new_Echo_proxyObjectFactory () {}
   virtual ~_new_Echo_proxyObjectFactory() {}

   // Only have to override newProxyObject
   virtual CORBA::Object_ptr newProxyObject(Rope *r,
                                            CORBA::Octet *key,
                                            size_t keysize,
                                            IOP::TaggedProfileList *profiles,
                                            CORBA::Boolean release) {
      _new_proxy_Echo *p = new _new_proxy_Echo(r,key,keysize,profiles,release);
      return p;
   }
```

```
};
```

Finally, we have to instantiate a single instance of the new proxy factory in the application code.

```
int main(int argc, char** argv)
{
   // Other initialisation steps

   _new_Echo_proxyObjectFactory* f =  new _new_Echo_proxyObjectFactory;

   // Use the new operator to instantiate the proxy factory and never
   // call the delete operator on this instance.

   // From this point onwards, _new_proxy_Echo will be used to create
   // proxy objects for Echo.

}
```

8.2.3 Further Considerations

Notice that the ORB may call `newProxyObject` multiple times to create proxy objects for the same remote object. In other words, the ORB does not guarantee that only one proxy object is created for each remote object. For applications that require this guarantee, it is necessary to check within `newProxyObject` whether a proxy object has already been created for the current request. If the argument `Rope* r` points to the same structure and the content of the sequence `CORBA::Octet* key` is the same, then an existing proxy object can be returned to satisfy the current request. Do not forget to call `CORBA::duplicate()` before returning the object reference.

`newProxyObject` may be called concurrently by different threads within the ORB. Needless to say, the function must be thread-safe.

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 omniORB2. For further information on type Any, refer to the C++ Mapping section of the CORBA 2 specification [OMG96a], and for more information on TypeCode, refer to the Interface Repository chapter in the CORBA core section of the CORBA 2 specification.

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 omniORB2 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 `>>=` .

Two 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 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;
char* str;

if (an_any >>= l) {
    cout << "Long: " << l << endl;
}
else if (an_any >>= d) {
    cout << "Double: " << d << endl;
}
else if (an_any >>= str) {
    cout << "String: " << str << endl;
}
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. `omniidl2` will generate insertion and extraction operators for the constructed type. Note that it is necessary to specify the `-a` command-line flag when running omniidl2 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 `omniidl2 -a`, 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&, 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 used 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:

```
testStruct* tp;

if (an_any >>= tp) {
    cout << "testStruct: l: " << tp->l << endl;
    cout << "            s: " << tp->s << endl;
}
else {
    cout << "Unknown value contained in Any." << endl;
}
```

As with basic types, if an attempt is made to extract a type from an Any that does not contain a value of that type, the extraction operator returns False. If the Any does contain that type, the extraction operator returns True. If the extraction is successful, the caller's pointer will point to memory managed by the Any. The caller must not delete or otherwise change this storage, and should not use this storage after the contents of the Any are replaced (either by insertion or assignment), or after the Any has been destroyed [1]. 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 `_forany` types), and for inserting and extracting booleans, octets, chars, and bounded strings. Please refer to section 16.14 of the C++ Mapping section of the CORBA 2 specification [OMG96a] for further information.

9.2 Type Any in omniORB2

This section contains some notes on the use and behaviour of type Any in omniORB2.

[1] It is unclear in the CORBA specification whether or not object references should be managed by an Any. The omniORB2 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.

Generating Insertion and Extraction Operators. To generate type Any insertion and extraction operators for constructed types and interfaces, the `-a` command line flag should be specified when running `omniidl2`.

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 equality with the TypeCode of the type stored by the Any. omniORB2 will ignore any alias TypeCodes (`tk_alias`) when making this comparison. Examples:

```
// IDL 1

typedef double Double1;

struct Test1 {
Double1 a;
};


------


// IDL 2

typedef double Double2;

struct Test1 {
  Double2 a;
};
```

If an attempt is made to extract the type `Test1` defined in IDL 1 from an Any containing the `Test1` defined in IDL 2, this will succeed (and vice-versa), as the two types differ only by an alias.

Object references. Note that type Any does not manage object reference types - it is unclear in the CORBA specification whether this is required or not. Therefore, the programmer should release object references and pseudo-objects (such as TypeCode) that have been extracted from an Any. Type Any will, however, manage constructed types (as per the CORBA 2 specification) - constructed types extracted from an Any should not be deleted, as they will be deleted by the Any when it is destroyed.

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. 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;
```

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

Removing aliases from TypeCodes. Some ORBs (e.g. Orbix) will not accept TypeCodes containing `tk_alias` TypeCodes. When using type Any while interoperating with these ORBs, it is necessary to remove `tk_alias` TypeCodes from throughout the TypeCode representing a constructed type.

While omniORB2 will always remove top-level aliases, it doesn't remove aliases contained in a constructed type (for example, `Double1` in the struct `Test1` in example 1, above). To remove all `tk_alias` TypeCodes from TypeCodes stored in Anys, supply the `-ORBtcAliasExpand 1` command-line flag when running an omniORB2 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. omniORB2 does not yet support recursive TypeCodes. This means that types such as the following can not be inserted or extracted from an Any:

```
// IDL 4

struct Test4 {
  sequence<Test4> a;
};
```

If these types are specified in IDL, omniidl2 will not generate Any insertion / extraction operators, so an attempt to insert / extract them will result in compile-time errors. Similarly, it won't generate a `_tc_` TypeCode for the type. If another type contains a type which is recursive, operators won't be generated for that type either, and so on. Recursive TypeCodes will be supported in a future release of omniORB2.

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) may result in a memory leak if the type being extracted is a constructed type. It was decided not to mutex the Any, as this condition should rarely arise, and adding a mutex would lead to performance penalties.

9.3 TypeCode in omniORB2

This section contains some notes on the use and behaviour of TypeCode in omniORB2

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 `-a` command-line flag when running omniidl2. This will generate a `_tc_` TypeCode describing the type, at the same scope as the type (as per the CORBA 2 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 6.7.1 of the CORBA 2 specification [OMG96a]) can be used to query the TypeCode about the type it represents.

TypeCode equality. The `CORBA::TypeCode::equal()` member function will 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).

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 section 12.3.4 of the Interoperability section of the CORBA 2 specification [OMG96a]). Note that these (optional) parameters are included in TypeCodes generated by omniORB2.

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 omniORB2
//                      users guide.
//
//               This is the object implementation.
//
// Usage: anyExample_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]
//               |
//              text  [context] kind [my_context]
//               |
//              anyExample  [object]  kind [Object]
//

#include <iostream.h>

#include "anyExample.hh"


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


class anyExample_i : public virtual _sk_anyExample {
public:
  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;
  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)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");

  anyExample_i *myobj = new anyExample_i();
  myobj->_obj_is_ready(boa);

  {
    anyExample_var myobjRef = myobj->_this();
    if (!bindObjectToName(orb,myobjRef)) {
      return 1;
    }
  }

  boa->impl_is_ready();
  // Tell the BOA we are ready. The BOA's default behaviour is to block
  // on this call indefinitely.

  return 0;
}
```

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

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

    // Narrow the object returned by resolve_initial_references()
    // to a CosNaming::NamingContext object:
    rootContext = CosNaming::NamingContext::_narrow(initServ);
    if (CORBA::is_nil(rootContext))
      {
        cerr << "Failed to narrow naming context." << endl;
        return 0;
      }
  }
  catch(CORBA::ORB::InvalidName& ex) {
    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, and assign testContext to it:
      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 tmpobj;
      tmpobj = rootContext->resolve(contextName);
      testContext = CosNaming::NamingContext::_narrow(tmpobj);
      if (CORBA::is_nil(testContext)) {
        cerr << "Failed to narrow naming context." << endl;
        return 0;
```

```
      }
    }

    // Bind the object (obj) to testContext, naming it anyExample:
    CosNaming::Name objectName;
    objectName.length(1);
    objectName[0].id   = (const char*) "anyExample";   // string copied
    objectName[0].kind = (const char*) "Object"; // string copied


    // Bind obj with name anyExample to the testContext:
    try {
      testContext->bind(objectName,obj);
    }
    catch(CosNaming::NamingContext::AlreadyBound& ex) {
      testContext->rebind(objectName,obj);
    }
    // Note: Using rebind() will overwrite any Object previously bound
    //       to /test/anyExample with obj.
    //       Alternatively, bind() can be used, which will raise a
    //       CosNaming::NamingContext::AlreadyBound exception if the name
    //       supplied is already bound to an object.
  }
  catch (CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "naming service." << endl;
    return 0;
  }
  catch (omniORB::fatalException& ex) {
    throw;
  }
  catch (...) {
    cerr << "Caught a system exception while using the naming service."<< endl;
    return 0;
  }
  return 1;
}
```

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 omniORB2
//                      users guide.
//
//              This is the client. It uses the COSS naming service
//              to obtain the object reference.
//
// Usage: anyExample_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]
//               |
//              anyExample  [object]  kind [Object]
//

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

static CORBA::Object_ptr getObjectReference(CORBA::ORB_ptr orb);
static void invokeOp(anyExample_ptr& tobj, const CORBA::Any& a);

int
main (int argc, char **argv)
{
  CORBA::ORB_ptr orb = CORBA::ORB_init(argc,argv,"omniORB2");
  CORBA::BOA_ptr boa = orb->BOA_init(argc,argv,"omniORB2_BOA");
  CORBA::Object_var obj;

  try {
    obj = getObjectReference(orb);
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "object." << endl;
    return -1;
  }
  catch(omniORB::fatalException& ex) {
    cerr << "Caught omniORB2 fatalException. This indicates a bug is caught "
         << "within omniORB2.\nPlease send a bug report.\n"
         << "The exception was thrown in file: " << ex.file() << "\n"
         << "                            line: " << ex.line() << "\n"
         << "The error message is: " << ex.errmsg() << endl;
    return -1;
  }
  catch(...) {
```

```
    cerr << "Caught a system exception." << endl;
    return -1;
  }


  anyExample_ptr tobj = anyExample::_narrow(obj);

  if (CORBA::is_nil(tobj)) {
    cerr << "Can't narrow object reference to type anyExample." << endl;
    return -1;
  }


  CORBA::Any a;

  try {
    // 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);
  }
  catch(CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "object." << endl;
    return -1;
  }
  catch(omniORB::fatalException& ex) {
    cerr << "Caught omniORB2 fatalException. This indicates a bug is caught "
         << "within omniORB2.\nPlease send a bug report.\n"
```

```
         << "The exception was thrown in file: " << ex.file() << "\n"
         << "                                line: " << ex.line() << "\n"
         << "The error message is: " << ex.errmsg() << endl;
    return -1;
  }
  catch(...) {
    cerr << "Caught a system exception." << endl;
    return -1;
  }

  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 initServ;
    initServ = orb->resolve_initial_references("NameService");

    // Narrow the object returned by resolve_initial_references()
    // to a CosNaming::NamingContext object:
    rootContext = CosNaming::NamingContext::_narrow(initServ);
    if (CORBA::is_nil(rootContext))
      {
        cerr << "Failed to narrow naming context." << endl;
        return CORBA::Object::_nil();
      }
  }
  catch(CORBA::ORB::InvalidName& ex) {
    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*) "anyExample";
  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.)


  CORBA::Object_ptr obj;
```

```
  try {
    // Resolve the name to an object reference, and assign the reference
    // returned to a CORBA::Object:
    obj = 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;
      return CORBA::Object::_nil();
    }
  catch (CORBA::COMM_FAILURE& ex) {
    cerr << "Caught system exception COMM_FAILURE, unable to contact the "
         << "naming service." << endl;
    return CORBA::Object::_nil();
  }
  catch(omniORB::fatalException& ex) {
    throw;
  }
  catch (...) {
    cerr << "Caught a system exception while using the naming service."<< endl;
    return CORBA::Object::_nil();
  }
  return obj;
}


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;
  }
}
```

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 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` through `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. .sp 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

[OMG96a] *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, OMG, Updated July 1996.

[OMG96b] *CORBAservices: Common Object Services Specification*, OMG, Updated July 1996.

[Richardson96a] *The OMNI Thread Abstraction*, Tristan Richardson, ORL, 22 October 1996.

[Richardson96b] *The OMNI Development Environment Version 4.0*, Tristan Richardson, ORL, 5 November 1996.