The omniORBpy version 3 User's Guide

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

Apasphere Ltd.

July 2007

Changes and Additions, June 2007

- Updates for omniORBpy 3.1.

Changes and Additions, June 2005

- New omniORBpy 3 features.

Changes and Additions, July 2004

- Minor updates.

Changes and Additions, November 2002

- Per thread timeouts.
- Minor fixes.

Changes and Additions, August 2002

- Updated to omniORBpy 2.

Contents

Chapter 1

Introduction

omniORBpy is an Object Request Broker (ORB) that implements the CORBA 2.6 Python mapping [OMG01b]. It works in conjunction with omniORB for C++, version 4.1.

This user guide tells you how to use omniORBpy to develop CORBA applications using Python. It assumes a basic understanding of CORBA, and of the Python mapping. Unlike most CORBA standards, the Python mapping document is small, and quite easy to follow.

This manual contains all you need to know about omniORB in order to use omniORBpy. Some sections are repeated from the omniORB manual.

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

1.1 Features

1.1.1 Multithreading

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

From version 4.0 onwards, omniORB also supports a flexible thread pooling policy, and supports sending multiple interleaved calls on a single connection.

This policy leads to a small amount of additional call overhead, compared to the default thread per connection model, but allows omniORB to scale to extremely large numbers of concurrent clients.

1.1.2 Portability

omniORB has always been designed to be portable. It runs on many flavours of Unix, Windows, several embedded operating systems, and relatively obscure systems such as OpenVMS and Fujitsu-Siemens BS2000. It is designed to be easy to port to new platforms. The IDL to C++ mapping for all target platforms is the same.

1.1.3 Missing features

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

- omniORB does not have its own Interface Repository. However, it can act as a client to an IfR. The omniifr project (http://omniifr.sourceforge.net/) aims to create an IfR for omniORB.
- omniORB supports interceptors, but not the standard Portable Interceptor API. Interceptor facilities available from Python code are quite limited.

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

1.2 Setting up your environment

omniORBpy relies on the omniORB C++ libraries. If you are building from source, you must first build omniORB itself, as detailed in the omniORB documentation. After that, you can build the omniORBpy distribution, according to the instructions in the release notes.

1.2.1 Paths

With an Autoconf build (the norm on Unix platforms), omniORBpy is usually installed into a location that Python will find it.

Otherwise, you must tell Python where to find it. You must add two directories to the `PYTHONPATH` environment variable. The `lib/python` directory contains platform-independent Python code; the `lib/$FARCH` directory contains platform-specific binaries, where `FARCH` is the name of your platform, such as `x86_win32`.

On Unix platforms, set `PYTHONPATH` with a command like:

```
export PYTHONPATH=$PYTHONPATH:$TOP/lib/python:$TOP/lib/$FARCH
```

On Windows, use

```
set PYTHONPATH=%PYTHONPATH%;%TOP%\lib\python;%TOP%\lib\x86_win32
```

(Where the `TOP` environment variable is the root of your omniORB tree.)

You should also add the `bin/$FARCH` directory to your `PATH`, so you can run the IDL compiler, omniidl. Finally, add the `lib/$FARCH` directory to `LD_LIBRARY_PATH`, so the omniORB core library can be found.

1.2.2 Configuration file

- On Unix platforms, the omniORB runtime looks for the environment variable `OMNIORB_CONFIG`. If this variable is defined, it contains the pathname of the omniORB configuration file. If the variable is not set, omniORB will use the compiled-in pathname to locate the file (by default `/etc/omniORB.cfg`).
- On Win32 platforms (Windows NT, 2000, 95, 98), omniORB first checks the environment variable `OMNIORB_CONFIG` to obtain the pathname of the configuration file. If this is not set, it then attempts to obtain configuration data in the system registry. It searches for the data under the key `HKEY_LOCAL_MACHINE\SOFTWARE\omniORB`.

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

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

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

See section 6.1.2 for full details of corbaname URIs.

Chapter 2

The Basics

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

If you have experience with using other ORBs, you should still go through this chapter because it provides important information about the features and APIs that are necessarily omniORB specific.

2.1 The Echo example

We use an example which is similar to the one used in the omniORB manual. We define an interface, called `Example::Echo`, as follows:

```
// echo_example.idl
module Example {
  interface Echo {
    string echoString(in string mesg);
  };
};
```

The important difference from the omniORB Echo example is that our `Echo` interface is declared within an IDL module named `Example`. The reason for this will become clear in a moment.

If you are new to IDL, you can learn about its syntax in Chapter 3 of the CORBA specification 2.6 [OMG01a]. 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 `example_echo.idl`. It is part of the CORBA standard that all IDL files should have the extension '`.idl`', although omniORB does not enforce this.

2.2 Generating the Python stubs

From the IDL file, we use the IDL compiler, omniidl, to produce the Python stubs for that IDL. The stubs contain Python declarations for all the interfaces and types declared in the IDL, as required by the Python mapping. It is possible to generate stubs dynamically at run-time, as described in section 4.9, but it is more efficient to generate them statically.

To generate the stubs, we use a command line like

```
omniidl -bpython example_echo.idl
```

As required by the standard, that produces two Python packages derived from the module name `Example`. Directory `Example` contains the client-side definitions (and also the type declarations if there were any); directory `Example__POA` contains the server-side skeletons. This explains the difficulty with declarations at IDL global scope; section 2.7 explains how to access global declarations.

If you look at the Python code in the two packages, you will see that they are almost empty. They simply import the `example_echo_idl.py` file, which is where both the client and server side declarations actually live. This arrangement is so that omniidl can easily extend the packages if other IDL files add declarations to the same IDL modules.

2.3 Object References and Servants

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

Object references and servants are quite separate entities, and it is important not to confuse the two. Client code deals purely with object references, so there can be no confusion; object implementation code must deal with both object references and servants. You will get a run-time error if you use a servant where an object reference is expected, or vice-versa.

2.4 Example 1 — Colocated client and servant

In the first example, both the client and servant are in the same address space. The next sections show how the client and servant can be split between different address spaces.

First, the code:

```
#!/usr/bin/env python

import sys
from omniORB import CORBA, PortableServer
import Example, Example__POA
```

```
6
7  class Echo_i (Example__POA.Echo):
8      def echoString(self, mesg):
9          print "echoString() called with message:", mesg
10         return mesg
11
12 orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
13 poa = orb.resolve_initial_references("RootPOA")
14
15 ei = Echo_i()
16 eo = ei._this()
17
18 poaManager = poa._get_the_POAManager()
19 poaManager.activate()
20
21 message = "Hello"
22 result  = eo.echoString(message)
23
24 print "I said '%s'. The object said '%s'." % (message,result)
```

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

2.4.1 Imports

Line 3

Import the `sys` module to access `sys.argv`.

Line 4

Import omniORB's implementations of the `CORBA` and `PortableServer` modules. The standard requires that these modules are available outside of any package, so you can also do

```
import CORBA, PortableServer
```

Explicitly specifying omniORB is useful if you have more than one Python ORB installed.

Line 5

Import the client-side stubs and server-side skeletons generated for IDL module `Example`.

2.4.2 Servant class definition

Lines 7–10

For interface `Example::Echo`, omniidl produces a skeleton class named `Example__POA.Echo`. Here we define an implementation class, `Echo_i`, which derives from the skeleton class.

There is little constraint on how you design your implementation class, except that it has to inherit from the skeleton class and must implement all of the operations declared in the IDL. Note that since Python is a dynamic language, errors due to missing operations and operations with incorrect type signatures are only reported when someone tries to call those operations.

2.4.3 ORB initialisation

Line 12

The ORB is initialised by calling the `CORBA.ORB_init()` function. `ORB_init()` is passed a list of command-line arguments, and an ORB identifier. The ORB identifier should be 'omniORB4', but it is usually best to use `CORBA.ORB_ID`, which is initialised to a suitable string, or leave it out altogether, and rely on the default.

`ORB_init()` processes any command-line arguments which begin with the string '`-ORB`', and removes them from the argument list. See section 4.1.1 for details. If any arguments are invalid, or other initialisation errors occur (such as errors in the configuration file), the `CORBA.INITIALIZE` exception is raised.

2.4.4 Obtaining the Root POA

Line 13

To activate our servant object and make it available to clients, we must register it with a POA. In this example, we use the *Root POA*, rather than creating any child POAs. The Root POA is found with `orb.resolve_initial_references()`.

A POA's behaviour is governed by its *policies*. The Root POA has suitable policies for many simple servers. Chapter 11 of the CORBA 2.6 specification [OMG01a] has details of all the POA policies which are available.

2.4.5 Object initialisation

Line 15

An instance of the Echo servant object is created.

Line 16

The object is implicitly activated in the Root POA, and an object reference is returned, using the `_this()` method.

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

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

2.4.6 Activating the POA

Lines 18–19

POAs are initially in the *holding* state, meaning that incoming requests are blocked. Lines 18 and 19 acquire a reference to the POA's POA manager, and use it to put the POA into the *active* state. Incoming requests are now served. **Failing to activate the POA is one of the most common programming mistakes. If your program appears deadlocked, make sure you activated the POA!**

2.4.7 Performing a call

Line 22

At long last, we can call the object's `echoString()` operation. Even though the object is local, the operation goes through the ORB and POA, so the types of the arguments can be checked, and any mutable arguments can be copied. This ensures that the semantics of local and remote calls are identical. If any of the arguments (or return values) are of the wrong type, a `CORBA.BAD_PARAM` exception is raised.

2.5 Example 2 — Different Address Spaces

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

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

2.5.1 Object Implementation: Making a Stringified Object Reference

```python
1  #!/usr/bin/env python
2
3  import sys
4  from omniORB import CORBA, PortableServer
5  import Example, Example__POA
6
7  class Echo_i (Example__POA.Echo):
8      def echoString(self, mesg):
9          print "echoString() called with message:", mesg
10         return mesg
11
12 orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
13 poa = orb.resolve_initial_references("RootPOA")
14
15 ei = Echo_i()
16 eo = ei._this()
17
18 print orb.object_to_string(eo)
19
20 poaManager = poa._get_the_POAManager()
21 poaManager.activate()
22
23 orb.run()
```

Up until line 18, this example is identical to the colocated case. On line 18, the ORB's `object_to_string()` operation is called. This results in a string starting with the signature 'IOR:' and followed by some hexadecimal digits. All CORBA 2 compliant ORBs are able to convert the string into its internal representation of a so-called Interoperable Object Reference (IOR). The IOR contains the location information and a key to uniquely identify the object implementation in its own address space[1]. From the IOR, an object reference can be constructed.

After the POA has been activated, `orb.run()` is called. Since omniORB is fully multi-threaded, it is not actually necessary to call `orb.run()` for operation dispatch to happen—if the main program had some other work to do, it could do so, and remote invocations would be dispatched in separate threads. However, in the absence of anything else to do, `orb.run()` is called so the thread blocks rather than exiting immediately when the end-of-file is reached. `orb.run()` stays blocked until the ORB is shut down.

2.5.2 Client: Using a Stringified Object Reference

```python
#!/usr/bin/env python

import sys
from omniORB import CORBA
import Example
```

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

```
6
7  orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
8
9  ior = sys.argv[1]
10 obj = orb.string_to_object(ior)
11
12 eo = obj._narrow(Example.Echo)
13
14 if eo is None:
15     print "Object reference is not an Example::Echo"
16     sys.exit(1)
17
18 message = "Hello from Python"
19 result  = eo.echoString(message)
20
21 print "I said '%s'. The object said '%s'." % (message,result)
```

The stringified object reference is passed to the client as a command-line argument[2]. The client uses the ORB's `string_to_object()` function to convert the string into a generic object reference (`CORBA.Object`).

On line 12, the object's `_narrow()` function is called to convert the `CORBA.Object` reference into an `Example.Echo` reference. If the IOR was not actually of type `Example.Echo`, or something derived from it, `_narrow()` returns `None`.

In fact, since Python is a dynamically-typed language, `string_to_object()` is often able to return an object reference of a more derived type than `CORBA.Object`. See section 3.1 for details.

2.5.3 System exceptions

The keep it short, the client code shown above performs no exception handling. A robust client (and server) should do, since there are a number of system exceptions which can arise.

As already mentioned, `ORB_init()` can raise the `CORBA.INITIALIZE` exception if the command line arguments or configuration file are invalid. `string_to_object()` can raise two exceptions: if the string is not an IOR (or a valid URI with omniORB 3), it raises `CORBA.BAD_PARAM`; if the string looks like an IOR, but contains invalid data, is raises `CORBA.MARSHAL`.

The call to `echoString()` can result in any of the CORBA system exceptions, since any exceptions not caught on the server side are propagated back to the client. Even if the implementation of `echoString()` does not raise any system exceptions itself, failures in invoking the operation can cause a number of exceptions. First, if the server process cannot be contacted, a `CORBA.TRANSIENT` exception is raised. Second, if the server process *can* be contacted, but the object in question does not exist there, a `CORBA.OBJECT_NOT_EXIST` exception is raised.

[2] The code does not check that there is actually an IOR on the command line!

As explained later in section 3.1, the call to `_narrow()` may also involve a call to the object to confirm its type. This means that `_narrow()` can also raise `CORBA.TRANSIENT`, `CORBA.OBJECT_NOT_EXIST`, and `CORBA.COMM_FAILURE`.

Section 4.7 describes how exception handlers can be installed for all the various system exceptions, to avoid surrounding all code with `try...except` blocks.

2.5.4 Lifetime of a CORBA object

CORBA objects are either *transient* or *persistent*. The majority are transient, meaning that the lifetime of the CORBA object (as contacted through an object reference) is the same as the lifetime of its servant object. Persistent objects can live beyond the destruction of their servant object, the POA they were created in, and even their process. Persistent objects are, of course, only contactable when their associated servants are active, or can be activated by their POA with a servant manager[3]. A reference to a persistent object can be published, and will remain valid even if the server process is restarted.

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

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

2.6 Example 3 — Using the Naming Service

In this example, the object implementation uses the Naming Service [OMG98] to pass on the object reference to the client. This method is far more practical than using stringified object references. The full listings of the server and client are below.

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

Sequences of name components can be represented as a flat string, using '.' to separate the id and kind fields, and '/' to separate name components from each

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

other[4]. In our example, the Echo object reference is bound to the stringified name 'test.my_context/ExampleEcho.Object'.

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

2.6.1 Obtaining the Root Context object reference

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

```
import CosNaming
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
obj = orb.resolve_initial_references("NameService");
cxt = obj._narrow(CosNaming.NamingContext)
```

Remember, omniORB constructs its internal list of initial references at initialisation time using the information provided in the configuration file omniORB.cfg, or given on the command line. If this file is not present, the internal list will be empty and resolve_initial_references() will raise a CORBA.ORB.InvalidName exception.

Note that, like string_to_object(),resolve_initial_references() returns base CORBA.Object, so we should narrow it to the interface we want. In this case, we want CosNaming.NamingContext[5].

2.6.2 The Naming Service interface

It is beyond the scope of this chapter to describe in detail the Naming Service interface. You should consult the CORBA services specification [OMG98] (chapter 3).

2.6.3 Server code

Hopefully, the server code is self-explanatory:

```
#!/usr/bin/env python
import sys
```

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

[5] If you are on-the-ball, you will have noticed that we didn't call _narrow() when resolving the Root POA. The reason it is safe to miss it out is given in section 3.1.

```
from omniORB import CORBA, PortableServer
import CosNaming, Example, Example__POA

# Define an implementation of the Echo interface
class Echo_i (Example__POA.Echo):
    def echoString(self, mesg):
        print "echoString() called with message:", mesg
        return mesg

# Initialise the ORB and find the root POA
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")

# Create an instance of Echo_i and an Echo object reference
ei = Echo_i()
eo = ei._this()

# Obtain a reference to the root naming context
obj         = orb.resolve_initial_references("NameService")
rootContext = obj._narrow(CosNaming.NamingContext)

if rootContext is None:
    print "Failed to narrow the root naming context"
    sys.exit(1)

# Bind a context named "test.my_context" to the root context
name = [CosNaming.NameComponent("test", "my_context")]
try:
    testContext = rootContext.bind_new_context(name)
    print "New test context bound"

except CosNaming.NamingContext.AlreadyBound, ex:
    print "Test context already exists"
    obj = rootContext.resolve(name)
    testContext = obj._narrow(CosNaming.NamingContext)
    if testContext is None:
        print "test.mycontext exists but is not a NamingContext"
        sys.exit(1)

# Bind the Echo object to the test context
name = [CosNaming.NameComponent("ExampleEcho", "Object")]
try:
    testContext.bind(name, eo)
    print "New ExampleEcho object bound"

except CosNaming.NamingContext.AlreadyBound:
    testContext.rebind(name, eo)
    print "ExampleEcho binding already existed -- rebound"
```

```
# Activate the POA
poaManager = poa._get_the_POAManager()
poaManager.activate()

# Block for ever (or until the ORB is shut down)
orb.run()
```

2.6.4 Client code

Hopefully the client code is self-explanatory too:

```
#!/usr/bin/env python
import sys
from omniORB import CORBA
import CosNaming, Example

# Initialise the ORB
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)

# Obtain a reference to the root naming context
obj         = orb.resolve_initial_references("NameService")
rootContext = obj._narrow(CosNaming.NamingContext)

if rootContext is None:
    print "Failed to narrow the root naming context"
    sys.exit(1)

# Resolve the name "test.my_context/ExampleEcho.Object"
name = [CosNaming.NameComponent("test", "my_context"),
        CosNaming.NameComponent("ExampleEcho", "Object")]
try:
    obj = rootContext.resolve(name)

except CosNaming.NamingContext.NotFound, ex:
    print "Name not found"
    sys.exit(1)

# Narrow the object to an Example::Echo
eo = obj._narrow(Example.Echo)

if eo is None:
    print "Object reference is not an Example::Echo"
    sys.exit(1)

# Invoke the echoString operation
message = "Hello from Python"
result  = eo.echoString(message)

print "I said '%s'. The object said '%s'." % (message,result)
```

2.7 Global IDL definitions

As we have seen, the Python mapping maps IDL modules to Python packages with the same name. This poses a problem for IDL declarations at global scope. Global declarations are generally a bad idea since they make name clashes more likely, but they must be supported.

Since Python does not have a concept of a global scope (only a per-module global scope, which is dangerous to modify), global declarations are mapped to a specially named Python package. By default, this package is named `_GlobalIDL`, with skeletons in `_GlobalIDL__POA`. The package name may be changed with omniidl's `-Wbglobal` option, described in section 5.2. The omniORB C++ Echo example, with IDL:

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

can therefore be supported with code like

```
#!/usr/bin/env python

import sys
from omniORB import CORBA
import _GlobalIDL

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)

ior = sys.argv[1]
obj = orb.string_to_object(ior)
eo  = obj._narrow(_GlobalIDL.Echo)

message = "Hello from Python"
result  = eo.echoString(message)
print "I said '%s'. The object said '%s'" % (message,result)
```

Chapter 3

Python language mapping issues

omniORBpy adheres to the standard Python mapping [OMG01b], so there is no need to describe the mapping here. This chapter outlines a number of issues which are not addressed by the standard (or are optional), and how they are resolved in omniORBpy.

3.1 Narrowing object references

As explained in chapter 2, whenever you receive an object reference declared to be base `CORBA::Object`, such as from `NamingContext::resolve()` or `ORB::string_to_object()`, you should narrow the reference to the type you require. You might think that since Python is a dynamically typed language, narrowing should never be necessary. Unfortunately, although omniORBpy often generates object references with the right types, it cannot do so in all circumstances.

The rules which govern when narrowing is required are quite complex. To be totally safe, you can *always* narrow object references to the type you are expecting. The advantages of this approach are that it is simple and that it is guaranteed to work with all Python ORBs.

The disadvantage with calling narrow for all received object references is that much of the time it is guaranteed not to be necessary. If you understand the situations in which narrowing *is* necessary, you can avoid spurious narrowing.

3.1.1 The gory details

When object references are transmitted (or stored in stringified IORs), they contain a single type identifier string, termed the *repository id*. Normally, the repository id represents the most derived interface of the object. However, it is also permitted to be the empty string, or to refer to an interface higher up the inheritance hierarchy. To give a concrete example, suppose there are two IDL files:

```
// a.idl
module M1 {
```

```
  interface A {
    void opA();
  };
};

// b.idl
#include "a.idl"
module M2 {
  interface B : M1::A {
    void opB();
  };
};
```

A reference to an object with interface `B` will normally contain the repository id '`IDL:M2/B:1.0`'[1]. It is also permitted to have an empty repository id, or the id '`IDL:M1/A:1.0`'. '`IDL:M1/A:1.0`' is unlikely unless the server is being deliberately obtuse.

Whenever omniORBpy receives an object reference from somewhere—either as a return value or as an operation argument—it has a particular *target* interface in mind, which it compares with the repository id it has received. A target of base `CORBA::Object` is just one (common) case. For example, in the following IDL:

```
// c.idl
#include "a.idl"
module M3 {
  interface C {
    Object getObj();
    M1::A  getA();
  };
};
```

the target interface for `getObj`'s return value is `CORBA::Object`; the target interface for `getA`'s return value is `M1::A`.

omniORBpy uses the result of comparing the received and target repository ids to determine the type of the object reference it creates. The object reference has either the type of the received reference, or the target type, according to this table:

Case		Objref Type
1.	The received id is the same as the target id	received
2.	The received id is not the same as the target id, but the ORB knows that the received interface is derived from the target interface	received
3.	The received id is unknown to the ORB	target
4.	The received id is not the same as the target id, and the ORB knows that the received interface is *not* derived from the target interface	target

[1]It is possible to change the repository id strings associated with particular interfaces using the `ID`, `version` and `prefix` pragmas.

Cases 1 and 2 are the most common. Case 2 explains why it is not necessary to narrow the result of calling `resolve_initial_references("RootPOA")`: the return is always of the known type `PortableServer.POA`, which is derived from the target type of `CORBA.Object`.

Case 3 is also quite common. Suppose a client knows about IDL modules `M1` and `M3` from above, but not module `M2`. When it calls `getA()` on an instance of `M3::C`, the return value may validly be of type `M2::B`, which it does not know. By creating an object reference of type `M1::A` in this case, the client is still able to call the object's `opA()` operation. On the other hand, if `getObj()` returns an object of type `M2::B`, the ORB will create a reference to base `CORBA::Object`, since that is the target type.

Note that the ORB *never* rejects an object reference due to it having the wrong type. Even if it knows that the received id is not derived from the target interface (case 4), it might be the case that the object actually has a more derived interface, which is derived from both the type it is claiming to be *and* the target type. That is, of course, extremely unlikely.

In cases 3 and 4, the ORB confirms the type of the object by calling `_is_a()` just before the first invocation on the object. If it turns out that the object is not of the right type after all, the `CORBA.INV_OBJREF` exception is raised. The alternative to this approach would be to check the types of object references when they were received, rather than waiting until the first invocation. That would be inefficient, however, since it is quite possible that a received object reference will never be used. It may also cause objects to be activated earlier than expected.

In summary, whenever your code receives an object reference, you should bear in mind what omniORBpy's idea of the target type is. You must not assume that the ORB will always correctly figure out a more derived type than the target. One consequence of this is that you must always narrow a plain `CORBA::Object` to a more specific type before invoking on it[2]. You *can* assume that the object reference you receive is of the target type, or something derived from it, although the object it refers to may turn out to be invalid. The fact that omniORBpy often *is* able figure out a more derived type than the target is only useful when using the Python interactive command line.

3.2 Support for Any values

In statically typed languages, such as C++, Anys can only be used with built-in types and IDL-declared types for which stubs have been generated. If, for example, a C++ program receives an Any containing a struct for which it does not have static knowledge, it cannot easily extract the struct contents. The only solution is to use the inconvenient DynAny interface.

Since Python is a dynamically typed language, it does not have this difficulty. When omniORBpy receives an Any containing types it does not know, it is able to

[2]Unless you are invoking pseudo operations like `_is_a()` and `_non_existent()`.

create new Python types which behave exactly as if there were statically generated stubs available. Note that this behaviour is not required by the Python mapping specification, so other Python ORBs may not be so accommodating.

The equivalent of DynAny creation can be achieved by dynamically writing and importing new IDL, as described in section 4.9.

There is, however, a minor fly in the ointment when it comes to receiving Anys. When an Any is transmitted, it is sent as a TypeCode followed by the actual value. Normally, the TypeCodes for entities with names—members of structs, for example—contain those names as strings. That permits omniORBpy to create types with the corresponding names. Unfortunately, the GIOP specification permits TypeCodes to be sent with empty strings where the names would normally be. In this situation, the types which omniORBpy creates cannot be given the correct names. The contents of all types except structs and exceptions can be accessed without having to know their names, through the standard interfaces. Unknown structs, exceptions and valuetypes received by omniORBpy have an attribute named '`_values`' which contains a sequence of the member values. This attribute is omniORBpy specific.

Similarly, TypeCodes for constructed types such as structs and unions normally contain the repository ids of those types. This means that omniORBpy can use types statically declared in the stubs when they are available. Once again, the specification permits the repository id strings to be empty[3]. This means that even if stubs for a type received in an Any are available, it may not be able to create a Python value with the right type. For example, with a struct definition such as:

```
module M {
  struct S {
    string str;
    long   l;
  };
};
```

The transmitted TypeCode for `M::S` may contain only the information that it is a structure containing a string followed by a long, not that it is type `M::S`, or what the member names are.

To cope with this situation, omniORBpy has an extension to the standard interface which allows you to *coerce* an Any value to a known type. Calling an Any's `value()` method with a TypeCode argument returns either a value of the requested type, or `None` if the requested TypeCode is not *equivalent* to the Any's TypeCode. The following code is guaranteed to be safe, but is not standard:

```
a = # Acquire an Any from somewhere
v = a.value(CORBA.TypeCode(CORBA.id(M.S)))
if v is not None:
    print v.str
else:
    print "The Any does not contain a value compatible with M::S."
```

[3]The use of empty repository id strings is deprecated as of GIOP 1.2.

3.2.1 Any helper module

omniORBpy provides an alternative, non-standard way of constructing and deconstructing Anys that is often more convenient to use in Python programs. It uses Python's own dynamic typing to infer the TypeCodes to use. The `omniORB.any` module contains two functions, `to_any()` and `from_any()`.

`to_any()` takes a Python object and tries to return it inside an Any. It uses the following rules:

- Python strings are represented as CORBA strings.
- Python unicode objects are represented as CORBA wstrings.
- Python integers are represented as CORBA longs.
- Python long integers are represented as a CORBA integer type taken from long, unsigned long, long long, unsigned long, depending on what size type the Python long integer will fit in. If the value is too large for any of these, `CORBA.BAD_PARAM` is raised.
- Python lists and tuples of the types above are represented as sequences of the corresponding CORBA types.
- Python lists and tuples of mixed types are represented as sequences of Anys.
- Python dictionaries with string keys are represented as CORBA structs, using the dictionary keys as the member names, and the types of the dictionary values as the member types.
- Instances of CORBA types (structs, unions, enums, etc.) generated by the IDL compiler are represented as themselves.

All other Python types result in a `CORBA.BAD_PARAM` exception.

The `from_any()` function works in reverse. It takes an Any as its argument and extracts its contents using the same rules as `to_any()`. By default, CORBA structs are extracted to dictionaries; if the optional `keep_structs` argument is set true, they are instead left as instances of the CORBA struct classes.

3.3 Interface Repository stubs

The Interface Repository interfaces are declared in IDL module `CORBA` so, according to the Python mapping, the stubs for them should appear in the Python `CORBA` module, along with all the other CORBA definitions. However, since the stubs are extremely large, omniORBpy does not include them by default. To do so would unnecessarily increase the memory footprint and start-up time.

The Interface Repository stubs are automatically included if you define the `OMNIORBPY_IMPORT_IR_STUBS` environment variable. Alternatively, you can import the stubs at run-time by calling the `omniORB.importIRStubs()` function. In both cases, the stubs become available in the Python `CORBA` module.

Chapter 4

omniORB configuration and API

omniORB 4.1, and thus omniORBpy 3, has a wide range of parameters that can be configured. They can be set in the configuration file / Windows registry, as environment variables, or on the command line. A few parameters can be configured at run time. This chapter lists all the configuration parameters, and how they are used.

4.1 Setting parameters

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

1. Command line arguments
2. Environment variables
3. Configuration file / Windows registry
4. Built-in defaults

4.1.1 Command line arguments

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

4.1.2 Environment variables

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

```
export ORBtraceLevel=10
```

4.1.3 Configuration file

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

```
traceLevel = 10
```

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

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

Note how command line arguments and environment variables prefix parameter names with '-ORB' and 'ORB' respectively, but the configuration file does not use a prefix.

4.1.4 Windows registry

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

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

4.2 Tracing options

The following options control debugging trace output.

`traceLevel` *default* = `1`

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

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

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

`traceExceptions` *default* = `0`

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

`traceInvocations` *default* = `0`

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

`traceInvocationReturns` *default* = `0`

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

`traceThreadId` *default* = `0`

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

`traceTime` *default* = `0`

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

4.2.1 Tracing API

The tracing parameters can be inspected or modified at runtime with the following functions in the `omniORB` module:

```
traceLevel()
traceInvocations()
traceInvocationReturns()
traceThreadId()
traceTime()
```

Calling one of the functions with no arguments returns the current value; calling it with a single integer argument sets the value.

4.3 Miscellaneous global options

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

`dumpConfiguration` *default* = `0`

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

`scanGranularity` *default* = `5`

As explained in chapter 7, omniORB regularly scans incoming and outgoing connections, so it can close unused ones. This value is the granularity in seconds at which the ORB performs its scans. A value of zero turns off the scanning altogether.

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

The native code set the application is using for `char` and `string`. See chapter 8.

`copyValuesInLocalCalls` *default* = `1`

Determines whether valuetype parameters in local calls are copied or not. See chapter 10.

`abortOnInternalError` *default* = `0`

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

4.4 Client side options

These options control aspects of client-side behaviour.

`InitRef` *default* = *none*

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

`DefaultInitRef` *default* = *none*

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

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

Used to specify the way the client contacts a server, depending on the server's address. See section 7.7.1 for details.

`clientCallTimeOutPeriod` *default* = `0`

Call timeout in milliseconds for the client side. If a call takes longer than the specified number of milliseconds, the ORB closes the connection to the server and raises a `TRANSIENT` exception. A value of zero means no timeout; calls can block for ever. See section 7.3.1 for more information about timeouts.

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

`clientConnectTimeOutPeriod` *default* = `0`

The timeout that is used in the case that a new network connection is established to the server. A value of zero means that the normal call timeout is used. See section 7.3.1 for more information about timeouts.

`supportPerThreadTimeOut` *default* = `0`

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

`outConScanPeriod` *default* = `120`

Idle timeout in seconds for outgoing (i.e. client initiated) connections. If a connection has been idle for this amount of time, the ORB closes it. See section 7.5.

`maxGIOPConnectionPerServer` *default* = `5`

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

`oneCallPerConnection` *default* = `1`

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

`offerBiDirectionalGIOP` *default* = `0`

If set true, the client will indicate to servers that it is willing to accept callbacks on client-initiated connections using bidirectional GIOP, provided the relevant POA policies are set. See section 7.8.

`verifyObjectExistsAndType` *default* = `1`

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

`giopTargetAddressMode` *default* = `0`

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

`bootstrapAgentHostname` *default* = *none*

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

`bootstrapAgentPort` *default* = `900`

The port number for the obsolete Sun bootstrap agent.

`principal` *default* = *none*

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

4.5 Server side options

These parameters affect server-side operations.

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

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

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

Configure the rules about whether a server should accept an incoming connection from a client. See section 7.7.2 for details.

`serverCallTimeOutPeriod` *default* = `0`

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

`inConScanPeriod` *default* = `180`

Idle timeout in seconds for incoming. If a connection has been idle for this amount of time, the ORB closes it. See section 7.5.

`threadPerConnectionPolicy` *default* = `1`

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

`maxServerThreadPerConnection` *default* = `100`

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

`maxServerThreadPoolSize` *default* = `100`

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

`threadPerConnectionUpperLimit` *default* = `10000`

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

`threadPerConnectionLowerLimit` *default* = `9000`

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

`threadPoolWatchConnection` *default* = `1`

If non-zero, threads from the pool temporarily behave a bit like thread per connection after dispatching a call. See section 7.4.2 for details.

`acceptBiDirectionalGIOP` *default* = `0`

Determines whether a server will ever accept clients' offers of bidirectional GIOP connections. See section 7.8.

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

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

`unixTransportPermission` *default* = `0777`

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

`supportCurrent` *default* = `1`

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

`objectTableSize` *default* = `0`

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

`poaHoldRequestTimeout` *default* = `0`

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

`supportBootstrapAgent` *default* = `0`

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

4.6 GIOP and interoperability options

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

`maxGIOPVerson` *default* = `1.2`

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

`giopMaxMsgSize` *default* = `2097152`

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

`strictIIOP` *default* = `1`

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

`lcdMode` *default* = `0`

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

`tcAliasExpand` *default* = `0`

This flag is used to indicate whether TypeCodes associated with Anys should have aliases removed. This functionality is included because some ORBs will not recognise an Any containing a TypeCode with aliases to be the same as the actual type contained in the Any. Note that omniORB will always remove top-level aliases, but will not remove aliases from TypeCodes that are members of other TypeCodes (e.g. TypeCodes for members of structs etc.), unless `tcAliasExpand` is set to 1. There is a performance penalty when inserting into an Any if `tcAliasExpand` is set to 1.

`useTypeCodeIndirections` *default* = `1`

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

`acceptMisalignedTcIndirections` *default* = `0`

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

4.7 System Exception Handlers

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

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

4.7.1 Minor codes

omniORB makes extensive use of exception minor codes to indicate the specific circumstances surrounding a system exception. The C++ file `include/omniORB4/minorCode.h` contains definitions of all the minor codes used in omniORB, covering codes allocated in the CORBA specification, and ones specific to omniORB.

Applications can use minor codes to adjust their behaviour according to the condition. You can receive a string format of a minor code by calling the `omniORB.minorCodeToString()` function, passing an exception object as its argument.

4.7.2 CORBA::TRANSIENT handlers

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

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

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

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

You can override the default behaviour by installing your own exception handler. The function to call has signature:

```
omniORB.installTransientExceptionHandler(cookie, function [, object])
```

The arguments are a cookie, which is any Python object, a call-back function, and optionally an object reference. If the object reference is present, the exception handler is installed for just that object; otherwise the handler is installed for all objects with no handler of their own.

The call-back function must have the signature

```
function(cookie, retries, exc) -> boolean
```

When a `TRANSIENT` exception occurs, the function is called, passing the cookie object, a count of how many times the operation has been retried, and the TRANSIENT exception object itself. If the function returns true, the operation is retried; if it returns false, the original exception is raised in the application. In the case of a `TRANSIENT` exception due to a failed location forward, the exception propagated to the application is the *original* exception that caused the `TRANSIENT` (e.g. a `COMM_FAILURE` or `OBJECT_NOT_EXIST`), rather than the `TRANSIENT` exception[1].

[1] This is a change from omniORB 4.0 / omniORBpy 2 and earlier, where it was the `TRANSIENT` exception that was propagated to the application.

4.7.3 CORBA.COMM_FAILURE and CORBA.SystemException

There are two other functions for registering exception handlers: one for CORBA.COMM_FAILURE, and one for all other exceptions. For both these cases, the default is for there to be no handler, so exceptions are propagated to the application.

```
omniORB.installCommFailureExceptionHandler(cookie, function [, object])
omniORB.installSystemExceptionHandler(cookie, function [, object])
```

In both cases, the call-back function has the same signature as for TRANSIENT handlers.

4.8 Location forwarding

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

4.9 Dynamic importing of IDL

omniORBpy is usually used with pre-generated stubs. Since Python is a dynamic language, however, it is possible to compile and import new stubs at run-time.

Dynamic importing is achieved with omniORB.importIDL() and omniORB.importIDLString(). Their signatures are:

```
importIDL(filename [, args ]) -> tuple
importIDLString(string [, args ]) -> tuple
```

The first function compiles and imports the specified file; the second takes a string containing the IDL definitions. The functions work by forking omniidl and importing its output[2]; they both take an optional argument containing a list of strings which are used as arguments for omniidl. For example, the following command runs omniidl with an include path set:

```
m = omniORB.importIDL("test.idl", ["-I/my/include/path"])
```

Instead of specifying omniidl arguments on each import, you can set the arguments to be used for all calls using the omniORB.omniidlArguments() function.

Both import functions return a tuple containing the names of the Python modules that have been imported. The modules themselves can be accessed through sys.modules. For example:

[2]omniidl must therefore be available on your path.

```
// test.idl
const string s = "Hello";
module M1 {
  module M2 {
    const long l = 42;
  };
};
module M3 {
  const short s = 5;
};
```

From Python:

```
>>> import sys, omniORB
>>> omniORB.importIDL("test.idl")
('M1', 'M1.M2', 'M3', '_GlobalIDL')
>>> sys.modules["M1.M2"].l
42
>>> sys.modules["M3"].s
5
>>> sys.modules["_GlobalIDL"].s
'Hello'
```

Note that each time `importIDL()` or `importIDLString()` is called, the IDL definitions are compiled and imported into the associated Python declarations. The new declarations overwrite any old ones with the same names. This can cause confusing situations where different modules see different definitions of the same objects. Although the objects appear identical, they are not, and comparisons within applications and within omniORBpy unexpectedly fail. You should not use these functions in more than one module to import the same IDL files.

4.10 C++ API

omniORBpy has a C++ API that can be used by programs that embed Python in C++, or by C++ extension modules to Python. The API has functions to convert object references between their Python representation and their C++ representation. For extensions to omniORBpy itself, it has a mechanism for adding pseudo object types to omniORBpy.

The definitions used by the C++ API are in the `omniORBpy.h` header. An example of its use is in `examples/embed/`.

The API is accessed through a singleton structure containing function pointers. A pointer to the API struct is stored as a `PyCObject` in the `_omnipy` module with the name `API`. It can be accessed with code like:

```
PyObject*     omnipy = PyImport_ImportModule((char*)"_omnipy");
PyObject*     pyapi  = PyObject_GetAttrString(omnipy, (char*)"API");
omniORBpyAPI* api    = (omniORBpyAPI*)PyCObject_AsVoidPtr(pyapi);
Py_DECREF(pyapi);
```

The structure has this definition:

```
struct omniORBpyAPI {

  PyObject* (*cxxObjRefToPyObjRef)(const CORBA::Object_ptr cxx_obj,
                                   CORBA::Boolean hold_lock);
  // Convert a C++ object reference to a Python object reference.
  // If <hold_lock> is true, caller holds the Python interpreter lock.

  CORBA::Object_ptr (*pyObjRefToCxxObjRef)(PyObject* py_obj,
                                           CORBA::Boolean hold_lock);
  // Convert a Python object reference to a C++ object reference.
  // Raises BAD_PARAM if the Python object is not an object reference.
  // If <hold_lock> is true, caller holds the Python interpreter lock.
};
```

Chapter 5

The IDL compiler

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

The general form of an omniidl command line is:

`omniidl` [*options*] `-b`<*back-end*> [*back-end options*] <*file 1*> <*file 2*> ...

5.1 Common options

The following options are common to all back-ends:

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

`-v` Verbose: trace compilation stages.

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

5.1.1 Preprocessor interactions

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

5.1.1.1 Windows 9x

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

5.1.2 Forward-declared interfaces

If you have an IDL file like:

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

then omniidl will normally issue a warning:

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

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

5.1.3 Comments

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

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

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

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

5.2 Python back-end options

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

`-Wbstdout`	Send the generated stubs to standard output, rather than to a file.
`-Wbinline`	Output stubs for #included files in line with the main file.
`-Wbglobal=`*g*	Use *g* as the name for the global IDL scope (default `_GlobalIDL`).
`-Wbpackage=`*p*	Put both Python modules and stub files in package *p*.
`-Wbmodules=`*p*	Put Python modules in package *p*.
`-Wbstubs=`*p*	Put stub files in package *p*.

The `-Wbstdout` option is not really useful if you are invoking omniidl yourself. It is used by `omniORB.importIDL()`, described in section 4.9.

When you compile an IDL file which #includes other IDL files, omniidl normally only generates code for the main file, assuming that code for the included files will be generated separately. Instead, you can use the `-Wbinline` option to generate code for the main IDL file *and* all included files in a single stub file.

Definitions declared at IDL global scope are normally placed in a Python module named '`_GlobalIDL`'. The `-Wbglobal` allows you to change that.

As explained in section 2.2, when you compile an IDL file like:

```
// example_echo.idl
module Example {
  interface Echo {
    string echoString(in string mesg);
  };
};
```

omniidl generates directories named `Example` and `Example__POA`, which provide the standard Python mapping modules, and also the file `example_echo_idl.py` which contains the actual definitions. The latter file contains code which inserts the definitions in the standard modules. This arrangement means that it is not possible to move all of the generated code into a Python package by simply placing the files in a suitably named directory. You may wish to do this to avoid clashes with names in use elsewhere in your software.

You can place all generated code in a package using the `-Wbpackage` command line option. For example,

```
omniidl -bpython -Wbpackage=generated echo_example.idl
```

creates a directory named '`generated`', containing the generated code. The stub module is now called '`generated.Example`', and the actual stub definitions are in '`generated.example_echo_idl`'. If you wish to split the modules and the stub definitions into different Python packages, you can use the `-Wbmodules` and `-Wbstubs` options.

Note that if you use these options to change the module package, the interface to the generated code is not strictly-speaking CORBA compliant. You may have to change your code if you ever use a Python ORB other than omniORBpy.

5.3 Examples

Generate the Python stubs for a file `a.idl`:

```
omniidl -bpython a.idl
```

As above, but put the stubs in a package called '`stubs`':

```
omniidl -bpython -Wbstubs=stubs a.idl
```

Generate both Python and C++ stubs for two IDL files:

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

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

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

Chapter 6

Interoperable Naming Service

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

6.1 Object URIs

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

6.1.1 corbaloc

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

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

for example:

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

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

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

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

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

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

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

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

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

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

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

You can also specify an IIOP version number:

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

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

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

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

6.1.2 corbaname

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

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

for example:

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

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

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

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

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

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

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

6.2 Configuring resolve_initial_references

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

6.2.1 ORBInitRef

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

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

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

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

6.2.2 ORBDefaultInitRef

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

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

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

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

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

6.3 omniNames

6.3.1 NamingContextExt

omniNames supports the `CosNaming::NamingContextExt` interface:

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

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

    exception InvalidAddress {};

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

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

`to_string()` and `to_name()` convert from `CosNaming::Name` sequences to flattened strings and vice-versa. Calling these operations involves remote calls to the naming service, so they are not particularly efficient. The `omniORB.URI` module contains equivalent `nameToString()` and `stringToName()` functions, which do not involve remote calls.

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

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

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

6.3.2 Use with corbaname

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

6.4 omniMapper

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

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

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

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

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

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

```
ORBDefaultInitRef corbaloc::localhost
```

6.5 Creating objects with simple object keys

In normal use, omniORB creates object keys containing various information including POA names and various non-ASCII characters. Since object keys are supposed to be opaque, this is not usually a problem. The INS breaks this opacity and requires servers to create objects with human-friendly keys.

If you wish to make your objects available with human-friendly URIs, there are two options. The first is to use omniMapper as described above, in conjunction with a `PERSISTENT` POA. The second is to create objects with the required keys yourself. You do this with a special POA with the name '`omniINSPOA`', acquired from `resolve_initial_references()`. This POA has the `USER_ID` and `PERSISTENT` policies, and the special property that the object keys it creates contain only the object ids given to the POA, and no other data. It is a normal POA in all other respects, so you can activate/deactivate it, create children, and so on, in the usual way.

Children of the omniINSPOA do not inherit its special properties of creating simple object keys. If the omniINSPOA's policies are not suitable for your application, you cannot create a POA with different policies (such as single threading, for example), and still generate simple object keys. Instead, you can activate a servant in the omniINSPOA that uses location forwarding to redirect requests to objects in a different POA.

Chapter 7

Connection and Thread Management

This chapter describes how omniORB manages threads and 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 factory object) or by conversion from a stringified representation. If the object is in a different address space, the binding process involves the ORB building a proxy object in the client's address space. The ORB arranges for invocations on the proxy object to be transparently mapped to equivalent invocations on the implementation object.

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

Notice that in CORBA, as in most distributed systems, remote bindings are established implicitly without application intervention. This provides the illusion that all objects are local, a property known as 'location transparency'. CORBA does not specify when such bindings should be established or how they should be multiplexed over the underlying network connections. Instead, ORBs are free to implement implicit binding by a variety of means.

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

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

7.2 The model

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

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

7.3 Client side behaviour

On the client side of a connection, the thread that invokes on a proxy object drives the GIOP protocol directly and blocks on the connection to receive the reply. The first time the client makes a call to a particular address space, the ORB opens a suitable connection to the remote address space (based on the client transport rule as described in section 7.7.1). After the reply has been received, the ORB caches the open network connection, ready for use by another call.

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

In the default one call per connection mode, there is a limit to the number of concurrent connections that are opened, set with the `maxGIOPConnectionPerServer` parameter. To tell the ORB that it may multiplex calls on a single connection, set the `oneCallPerConnection` parameter to zero. If the `oneCallPerConnection` parameter is set to the default value of one, and there are more concurrent calls than specified by `maxGIOPConnectionPerServer`, calls block waiting for connections to become free.

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

7.3.1 Client side timeouts

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

Timeouts are set using functions in the `omniORB` module:

```
omniORB.setClientCallTimeout(millisecs)
omniORB.setClientCallTimeout(objref, millisecs)
omniORB.setClientThreadCallTimeout(millisecs)
omniORB.setClientConnectTimeout(millisecs)
```

`setClientCallTimeout()` sets either the global timeout or the timeout for a specific object reference. `setClientThreadCallTimeout()` sets the timeout for the calling thread. Setting any timeout value to zero disables it.

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

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

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

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

Connect timeout > usual timeout

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

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

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

Connect timeout < usual timeout

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

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

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

7.4 Server side behaviour

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

7.4.1 Thread per connection mode

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

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

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

7.4.2 Thread pool mode

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

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

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

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

7.4.3 Policy transition

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

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

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

7.5 Idle connection shutdown

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

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

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

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

7.5.1 Interoperability Considerations

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

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

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

7.6 Transports and endpoints

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

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

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

TCP endpoints have the format:

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

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

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

If no `endPoint` parameters are set, the ORB assumes a single parameter of `giop:tcp::`, meaning IORs contain the address of the first non-loopback network interface, the ORB listens on all interfaces, and the OS chooses a port number.

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

`giop:unix:`<*filename*>

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

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

7.6.1 IPv6

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

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

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

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

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

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

7.6.2 Endpoint publishing

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

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

The following core rules are supported:

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

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

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

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

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

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

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

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

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

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

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

7.7 Connection selection and acceptance

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

7.7.1 Client transport rules

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

<*address mask*> *[action]+*

The address mask can be one of

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

The action is one or more of the following:

1.	`none`	Do not use this address
2.	`tcp`	Use a TCP transport
3.	`ssl`	Use an SSL transport
4.	`unix`	Use a Unix socket transport
5.	`bidir`	Connections to this address can be used bidirectionally (see section 7.8)

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

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

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

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

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

7.7.2 Server transport rules

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

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

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

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

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

7.8 Bidirectional GIOP

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

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

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

On the server side, these conditions must be met:

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

7.9 SSL transport

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

To use the SSL transport from Python you must import and set parameters in the `omniORB.sslTP` module before calling `CORBA.ORB_init()`. To initialise the module, you must call the `certificate_authority_file()`, `key_file()` and `key_file_password()` functions, providing the file names of the certificate authority and encryption keys, and the key file password.

Chapter 8

Code set conversion

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

8.1 Native code set

For the ORB to know how to handle strings given to it by the application, it must know what code set they are represented with, so it can properly translate them if need be. The default is ISO 8859-1 (Latin 1). A different code sets can be chosen at initialisation time with the `nativeCharCodeSet` parameter. The supported code sets are printed out at initialisation time if the ORB traceLevel is 15 or greater.

For most applications, the default is fine. Some applications may need to set the native char code set to UTF-8, allowing the full Unicode range to be supported in strings.

In omniORBpy, wchar and wstring are always represented by the Python Unicode type, so there is no need to select a native code set for wchar.

8.2 Code set library

To save space in the main ORB core library, most of the code set implementations are in a separate library. To load it from Python, you must import the `omniORB.codesets` module before calling `CORBA.ORB_init()`.

8.3 Implementing new code sets

Code sets must currently be implemented in C++. See the omniORB for C++ documentation for details.

Chapter 9

Interceptors

omniORBpy has limited interceptor support. Interceptors permit the application to insert processing at various points along the call chain, as requests are processed. The Portable Interceptors API is not yet supported.

Interceptors are registered using functions in the `omniORB.interceptors` module:

```
addClientSendRequest()
addClientReceiveReply()
addServerReceiveRequest()
addServerSendReply()
addServerSendException()
```

To register an interceptor function, call the relevant registration function with a callable argument. The callable will be called with two or three arguments. The first argument is the name of the operation being invoked; the second is the set of service contexts to be retrieved or filled in. `ServerSendException` has a third argument, the repository id of the exception being thrown.

When receiving service contexts (in the `ClientReceiveReply` and `ServerReceiveRequest` interceptors), the second argument is a tuple of 2-tuples. In each 2-tuple, the first item is the service context id and the second item is the CDR encapsulation of the service context. The encapsulation can be decoded with `omniORB.cdrUnmarshal()` (but only if you know the type to decode it to).

When sending service contexts (`ClientSendRequest`, `ServerSendReply`, and `ServerSendException`), the second argument is an empty list. The interceptor function can choose to add one or more service context tuples, with the same form described above, by appending to the list. Encapsulations are created with `omniORB.cdrMarshal()`.

Interceptor registration functions may only be called before the ORB is initialised. Attempting to call them later results in a `BAD_INV_ORDER` exception.

Chapter 10

Objects by value

omniORBpy 3 supports objects by value, declared with the `valuetype` keyword in IDL. This chapter outlines some issues to do with using valuetypes in omniORB. You are assumed to have read the relevant parts of the CORBA specification, specifically chapters 4 and 5 of the CORBA 2.6 specification, and section 1.3.10 of the Python language mapping, version 1.2.

10.1 Features

omniORB supports the complete objects by value specification, with the exception of custom valuetypes. All other features including value boxes, value sharing semantics, abstract valuetypes, and abstract interfaces are supported.

10.2 Value sharing and local calls

When valuetypes are passed as parameters in CORBA calls (i.e. calls on CORBA objects declared with `interface` in IDL), the structure of related values is maintained. Consider, for example, the following IDL definitions (which are from the example code in `src/examples/valuetype/simple`:

```
module ValueTest {
  valuetype One {
    public string s;
    public long   l;
  };

  interface Test {
    One op1(in One a, in One b);
  };
};
```

If the client to the `Test` object passes the same value in both parameters, just one value is transmitted, and the object implementation receives a copy of the single value, with references to it in both parameters.

In the case that the object is remote from the client, there is obviously a copying step involved. In the case that the object is in the same address space as the client, the same copying semantics must be maintained so that the object implementation can modify the values it receives without the client seeing the modifications. To support that, omniORB must copy the entire parameter list in one operation, in case there is sharing between different parameters. Such copying is a rather more time-consuming process than the parameter-by-parameter copy that takes place in calls not involving valuetypes.

To avoid the overhead of copying parameters in this way, applications can choose to relax the semantics of value copying in local calls, so values are not copied at all, but are passed by reference. In that case, the client to a call *will* see any modifications to the values it passes as parameters (and similarly, the object implementation will see any changes the client makes to returned values). To choose this option, set the `copyValuesInLocalCalls` configuration parameter to zero.

10.3 Value factories

As specified in section 1.3.10 of the Python language mapping (version 1.2), factories are automatically registered for values with no operations. This means that in common usage where values are just used to hold state, the application code does not need to implement and register factories. The application may still register different factories if it requires.

If the IDL definitions specify operations on values, the application is supposed to provide implementations of the operations, meaning that it must register suitable factories. If the application chooses to ignore the operations and just manipulate the data inside the values, omniidl can be asked to register factories for *all* values, not just ones with no operations, using the `-Wbfactories` option.

The Python language mapping says a value factory should be "a class instance with a `__call__` method taking no arguments". omniORBpy is less restrictive than that, and permits the use of *any* callable object, in particular the value implementation class itself.

10.4 Standard value boxes

The standard `CORBA.StringValue` and `CORBA.WStringValue` value boxes are available to application code. To make the definitions available in IDL, #include the standard `orb.idl`.

10.5 Values inside Anys

Valuetypes inserted into Anys cause a number of interesting issues. Even when inside Anys, values are required to support complete sharing semantics. Take this IDL for example:

```
module ValueTest {
  valuetype One {
    public string s;
    public long   l;
  };

  interface AnyTest {
    void op1(in One v, in Any a);
  };
};
```

Now, suppose the client behaves as follows:

```
v = One_impl("hello", 123)
a = CORBA.Any(ValueTest._tc_One, v)
obj.op1(v, a)
```

then on the server side:

```
class AnyTest_impl:
    ...
    def op1(self, v, a):
        v2 = a.value()
        assert v2 == v
```

This is all very well in this kind of simple situation, but problems can arise if truncatable valuetypes are used. Imagine this derived value:

```
module ValueTest {
  valuetype Two : truncatable One {
    public double d;
  };
};
```

Now, suppose that the client shown above sends an instance of valuetype `Two` in both parameters, and suppose that the server has not seen the definition of valuetype `Two`. In this situation, as the first parameter is unmarshalled, it will be truncated to valuetype `One`, as required. Now, when the Any is unmarshalled, it refers to the same value, which has been truncated. So, even though the TypeCode in the Any indicates that the value has type `Two`, the stored value actually has type `One`. If the receiver of the Any tries to pass it on, transmission will fail because the Any's value does not match its TypeCode.

In the opposite situation, where an Any parameter comes before a valuetype parameter, a different problem occurs. In that case, as the Any is unmarshalled, there is no type information available for valuetype `Two`, so omniORBpy constructs a suitable type based on the transmitted TypeCode. Because omniORBpy is unable to know how (and indeed if) the application has implemented valuetype `One`, the generated class for valuetype `Two` is not derived from the application's `One` class. When the second parameter is unmarshalled, it is given as an indirection to the previously-marshalled value inside the Any. The parameter is therefore set to the constructed `Two` type, rather than being truncated to an instance of the application's registered `One` type.

Because of these issues, it is best to avoid defining interfaces that mix valuetypes and Anys in a single operation, and certainly to avoid trying to share plain values with values inside Anys.

Bibliography

[BLFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396, August 1998.

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

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

[OMG01a] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, December 2001. From http://www.omg.org/cgi-bin/doc?formal/01-12-01.

[OMG01b] Object Management Group. *Python Language Mapping Specification*, February 2001. http://www.omg.org/technology/documents/formal/python.htm.