The omniORBpy version 1.3 User's Guide

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

AT&T Laboratories Cambridge

August 2000

Changes and Additions, August 2000

- New `omniORB.maxTcpConnectionPerServer` function.

Changes and Additions, August 2000

- New `omniORB.LOCATION_FORWARD` exception.
- New `omniORB.traceLevel()` function.

Changes and Additions, June 2000

- Brand new manual.

Contents

Chapter 1

Introduction

omniORBpy is an Object Request Broker (ORB) that implements the CORBA 2.3 Python mapping [OMG00b]. It is designed for use with omniORB 3, but it can also be used with omniORB 2.8. If you use omniORB 3.0, the full POA functionality is available; with omniORB 2.8 many POA functions are not supported.

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 CORBA 2.3 compliant

omniORB implements the Internet Inter-ORB Protocol (IIOP). This protocol provides omniORB the means of achieving interoperability with the ORBs implemented by other vendors. In fact, this is the native protocol used by omniORB for the communication amongst its objects residing in different address spaces.

Moreover, the IDL to Python language mapping provided by omniORBpy conforms to the 2.3 Python mapping specification[1].

1.1.2 Multithreading

omniORBpy is fully multithreaded[2]. To achieve low IIOP call overhead, unnecessary call-multiplexing is eliminated. At any time, there is at most one call in-flight

[1] As detailed in section 1.1.3, a number of features are missing.

[2] This means that your Python interpreter must have been built with thread support.

in each communication channel between two address spaces. To do so without limiting the level of concurrency, new channels connecting the two address spaces are created on demand and cached when there are more concurrent calls in progress. Each channel is served by a dedicated thread. This arrangement provides maximal concurrency and eliminates any thread switching in either of the address spaces to process a call. Furthermore, to maximise the throughput in processing large call arguments, large data elements are sent as soon as they are processed while the other arguments are being marshalled.

1.1.3 Missing features

omniORBpy is not a complete implementation of the CORBA 2.3 core. The following is a list of the missing features.

- omniORB does not have its own Interface Repository. However, it can act as a client to an IR[3].
- The IDL types wchar, wstring, fixed, and valuetype are not supported in this release.
- The `PortableServer.Current` interface is not yet supported.
- DII and DSI are not supported. However, since both Python code and IDL can be generated and used at run-time, this is not a significant restriction.

These features may be implemented in the short to medium term. It is best to check out the latest status on the omniORB home page (`http://www.uk.research.att.com/omniORB/`).

1.2 Setting up your environment

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

For Python to find omniORBpy, 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 `i586_linux_glibc` or `x86_win32`.

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

[3] See section 3.3 for a note about using an interface repository.

```
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.
- On Win32 platforms (Windows NT, 2000, 95, 98), omniORB first checks the environment variable `OMNIORB_CONFIG` to obtain the pathname of the configuration file. If this is not set, it then attempts to obtain configuration data in the system registry. It searches for the data under the key `HKEY_LOCAL_MACHINE\SOFTWARE\ORL\omniORB\2.0`.

The configuration file is used to obtain an object reference for the Naming Service, via either a stringified IOR or an Interoperable Naming Service URI. When using omniORB 3, the entry in the configuration file should be specified in the form:

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

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

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

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

1.2.2.1 omniORB 2.8 configuration file entries

`ORBInitRef` is new with omniORB 3. Under omniORB 2.8, you must use the older format (which is still supported by omniORB 3):

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

Two other entries are also supported, but with omniORB 3 are made obsolete by the Interoperable Naming Service URIs:

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

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

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

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 2.3 [OMG99]. For the moment, you only need to know that the interface consists of a single operation, `echoString()`, which takes a string as an argument and returns a copy of the same string.

The interface is written in a file, called `echo_example.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 6.12, but it is more efficient to generate them statically.

To generate the stubs, we use a command line like

```
omniidl -bpython echo_example.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 `echo_example_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 'omniORB3' or 'omniORB2', depending on which version of omniORB you are using. It is usually best to use `CORBA.ORB_ID`, which is initialised to a suitable string.

`ORB_init()` processes any command-line arguments which begin with the string '`-ORB`', and removes them from the argument list. See section 6.1 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 2.3 specification [OMG99] has details of all the POA policies which are available.

When omniORBpy is used with omniORB 2.8, *only* the Root POA is available (and is mapped to the omniORB BOA). You cannot create child POAs or alter policies.

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.

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: Generating a Stringified Object Reference

```
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

```
#!/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.COMM_FAILURE` 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. Various events can also cause `CORBA.TRANSIENT` to be raised. If that oc-

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

curs, omniORB's default behaviour is to automatically retry the invocation, with exponential back-off.

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.COMM_FAILURE`, `CORBA.OBJECT_NOT_EXIST`, and `CORBA.TRANSIENT`.

Section 6.10 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. (Note that since only the Root POA is available when using omniORBpy with omniORB 2.8, it is not possible to create persistent objects in that environment.)

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.

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

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

[4]There are escaping rules to cope with id and kind fields which contain '.' and '/' characters. See chapter 4 of this manual, and chapter 3 of the CORBA services specification, as updated for the Interoperable Naming Service [OMG00a].

[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.

```
#!/usr/bin/env python
import sys
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 [OMG00b], 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 6.12.

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[3]. 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 and exceptions 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[4]. 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:
```

[3]This is now deprecated, but some ORBs may still send empty strings. No version of omniORB has ever sent empty strings.

[4]And once again, this practice is deprecated.

```
    print v.str
else:
    print "The Any does not contain a value compatible with M::S."
```

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.

3.4 Using omniORBpy with omniORB 2.8

omniORBpy is designed to work with omniORB 3. When it is used with omniORB 2.8, many facilities are not available. This section describes the limitations.

If you require facilities which are not available with omniORB 2.8, you can cause your program to bail-out gracefully by detecting the omniORB version at start-up. The `omniORB.coreVersion()` function returns a string of the form *major.minor.micro* which indicates what version of omniORB is in use. The two possible values are currently '`2.8.0`' and '`3.0.0`'. Versions which differ only in micro version number are guaranteed to be compatible with each other.

3.4.1 POA functions

Under 2.8, only the root POA is available. It supports only the following functions:

- `destroy()`
- `_get_the_name()`
- `_get_the_POAManager()`
- `activate_object()`
- `deactivate_object()`
- `servant_to_id()`
- `servant_to_reference()`
- `reference_to_servant()`
- `reference_to_id()`

- `id_to_servant()`
- `id_to_reference()`

The `POAManager` interface only supports:

- `activate()`
- `get_state()`

For both interfaces, all other operations fail, raising the `CORBA.NO_IMPLEMENT` exception.

3.4.2 Local / remote transparency

omniORB 3 goes to great lengths to make sure that the semantics of local objects are identical to those for remote objects. omniORB 2.8 does not. omniORBpy also tries very hard to keep local and remote semantics identical, but on the foundation of omniORB 2.8 it is not always possible.

In most cases, you will not notice a difference between local and remote operations. The cases where local/remote transparency is broken under omniORB 2.8 are:

- An object is not deactivated until all local references to it have been released.
- When invoking on an object reference to a local object of the wrong type, as described in section 3.1, `CORBA.INV_OBJREF` is not raised. Instead, if the operation name does not exist on the object, `CORBA.NO_IMPLEMENT` is raised; if the operation name *does* exist but the argument types are wrong, `CORBA.BAD_PARAM` is raised; if the operation exists and the argument types are correct, the operation is executed.
- Exception handlers (described later in section 6.10) are not executed when local objects raise system exceptions. Exceptions are always propagated to the caller.

In all of these cases, omniORB 3 properly preserves local/remote transparency.

Chapter 4

Interoperable Naming Service

omniORB 3 supports the Interoperable Naming Service (INS), which will be part of CORBA 2.4. The following is a summary of the new facilities described in the INS edited chapters document [OMG00a]. These facilities are not available when using omniORBpy with omniORB 2.8.

4.1 Object URIs

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

4.1.1 corbaloc

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

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

for example:

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

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

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

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

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

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

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

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

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

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

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

You can also specify an IIOP version number, although omniORB only supports IIOP 1.0 at present:

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

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

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

4.1.2 corbaname

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

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

for example:

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

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

[1]Not 2089 as printed in [OMG00a]!

```
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.

4.2 Configuring resolve_initial_references

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

4.2.1 ORBInitRef

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

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

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

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

4.2.2 ORBDefaultInitRef

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

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

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

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

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

4.2.3 omniORB configuration file

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

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

4.2.4 Resolution order

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

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

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

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

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

1. Construct the URI '`corbaname:rir:#services/MyService`' and give it to `string_to_object()`.

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

2. Resolve the first part of the `corbaname` URI by calling `resolve_initial_references("NameService")`.

3. Construct the URI '`corbaname:rir:#services/NameService`' and give it to `string_to_object()`.

4. Resolve the first part of the `corbaname` URI by calling `resolve_initial_references("NameService")`.

5. … and so on for ever…

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

4.3 omniNames

4.3.1 NamingContextExt

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

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

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

    exception InvalidAddress {};

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

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

`to_string()` and `to_name()` convert from `CosNaming::Name` sequences to flattened strings and vice-versa. Note that calling these operations involves remote calls to the naming service, so they are not particularly efficient. 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.

4.3.2 Use with corbaname

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

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

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

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

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

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

4.4 omniMapper

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

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

The -port option allows you to choose a port other than 2809 to listen on[3]. The -config option specifies a location for the configuration file. The default name is /etc/omniMapper.cfg, or C:\omniMapper.cfg on Windows. omniMapper does not normally print anything; the -v option makes it verbose so it prints configuration information and a record of the redirections it makes, to standard output.

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

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

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

```
ORBDefaultInitRef corbaloc::localhost
```

4.5 Creating objects with simple object keys

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

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

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

Chapter 5

The IDL compiler

omniORBpy uses a new IDL compiler, named omniidl, which is also used by omniORB 3. It consists of a generic front-end parser written in C++, and a number of back-ends written in Python. omniidl is very strict about IDL validity, so you may find that it reports errors in IDL which compiles fine with earlier versions of omniORB, and with other ORBs.

The general form of an omniidl command line is:

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

5.1 Common options

The following options are common to all back-ends:

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

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

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

5.1.1 Preprocessor interactions

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

5.1.1.1 Windows 9x

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

5.1.2 Forward-declared interfaces

If you have an IDL file like:

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

then omniidl will normally issue a warning:

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

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

5.1.3 Comments

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

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

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

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

5.2 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 6.12.

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:

```
// echo_example.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 (requires omniORB 3):

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

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

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

Chapter 6

The omniORBpy API

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

6.1 ORB initialisation options

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

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

and the following omniORB-specific arguments:

`-ORBtraceLevel` <*level*>	See section 6.3.
`-ORBtraceInvocations`	See section 6.3.
`-ORBstrictIIOP`	See section 6.8.
`-ORBgiopMaxMsgSize` <*size in bytes*>	See section 6.5.
`-ORBobjectTableSize` <*number of entries*>	See section 6.6.
`-ORBserverName` <*string*>	See section 6.4.
`-ORBno_bootstrap_agent`	See section 6.7.
`-ORBverifyObjectExistsAndType` <*0 or 1*>	See section 6.8.
`-ORBinConScanPeriod` <*0–max integer*>	See section 7.3.
`-ORBoutConScanPeriod` <*0–max integer*>	See section 7.3.
`-ORBclientCallTimeOutPeriod` <*0–max integer*>	See section 7.3.
`-ORBserverCallTimeOutPeriod` <*0–max integer*>	See section 7.3.
`-ORBscanGranularity` <*0–max integer*>	See section 7.3.

`-ORBlcdMode`	See section 6.8.
`-ORBpoa_iiop_port` <*port no.*>	See section 6.2.
`-ORBpoa_iiop_name_port` <*hostname*[*:port no.*]>	See section 6.2.
`-ORBhelp`	Lists all ORB command line options.

and these two obsolete omniORB-specific arguments:

`-ORBInitialHost` <*string*>	See section 6.7.
`-ORBInitialPort` <*1–65535*>]	See section 6.7.

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

6.2 Hostname and port

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

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

When using omniORB 2.8, these two options are named `-BOAiiop_port` and `-BOAiiop_name_port`.

6.3 Run-time Tracing and Diagnostic Messages

omniORB can output tracing and diagnostic messages to the standard error stream. You can vary the amount of tracing using the `-ORBtraceLevel` <*level*> command line argument. For instance:

```
$ example_echo_srv.py -ORBtraceLevel 5
```

You can also inspect or modify the trace level at run-time. A call to `omniORB.traceLevel(`*level*`)` sets the level; calling the function with no argument returns the current level.

At the moment, the following trace levels are defined:

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

With omniORB 3, you can also use `-ORBtraceInvocations` to trace all operation invocations.

6.4 Server Name

Applications can optionally specify a name to identify the server process. At the moment, this name is only used by the host-based access control module. See section 7.5 for details. The server name can be changed by specifying the command-line option: `-ORBserverName` <*string*>.

6.5 GIOP Message Size

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

6.6 Object table size

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

The `-ORBobjectTableSize` argument allows you to choose a fixed size for the object table. This prevents omniORB from resizing it. Note that omniORB uses an open hash table so you can have any number of objects active, no matter what size table you specify. If you have many more active objects than hash table entries, object look-up performance will become linear with the number of objects.

6.7 Obsolete Initial Object Reference Bootstrapping

Starting from 2.6.0, but superseded by the Interoperable Naming Service in omniORB 3, a mechanism is available for the ORB runtime to obtain the initial object references to CORBA services. The bootstrap service is a special object with the object key 'INIT' and the following interface[1]:

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

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

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

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

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

During initialisation, the ORB reads the parameters in the omniORB configuration file. If the parameter `NAMESERVICE` is specified, the stringified IOR is used as the object reference of the root naming context. If the parameter is absent and the parameter `ORBInitialHost` is present, the ORB contacts the bootstrap

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

agent at the address specified to obtain the root naming context when the application calls `resolve_initial_references()`. If neither is present, `resolve_initial_references()` returns a nil object reference. Finally, the command line argument `-ORBInitialHost` overrides any parameters in the configuration file. The ORB always contacts the bootstrap agent at the address specified to obtain the root naming context.

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

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

   ```
   $ omniNames -start 1234
   ```

2. Add to omniORB.cfg:

   ```
   ORBInitialHost wobble
   ORBInitialPort 1234
   ```

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

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

   ```
   $ eg3_impl -ORBInitialHost wobble -ORBInitialPort 1234 &
   $ eg3_clt -ORBInitialHost wobble -ORBInitialPort 1234
   ```

6.8 GIOP Lowest Common Denominator Mode

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

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

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

By default, omniORB uses the GIOP LOCATE_REQUEST message to verify the existence of an object prior to the first invocation. If another vendor's ORB is known not to be able to handle this GIOP message, you can disable this feature with the `-ORBverifyObjectExistsAndType` option, and hence achieve interoperability.

6.9 GIOP Requesting Principal field

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

6.10 System Exception Handlers

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

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.

6.10.1 CORBA.TRANSIENT handlers

When a `CORBA.TRANSIENT` exception is raised by the ORB runtime, the default behaviour of the proxy objects is to retry indefinitely until the operation succeeds, with an exponentially increasing delay (up to a limit) between retries.

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

1. When a *cached* network connection is broken while an operation invocation

is in progress. The ORB will try to open a new connection at the next invocation.

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

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

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 TRANSIENT exception is raised in the application.

6.10.2 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.

6.11 Location forwarding

Any CORBA operation invocation can return a `LOCATION_FORWARD` message to the caller, indicating that it should retry the invocation on a new object reference. The standard allows ServantManagers to trigger `LOCATION_FORWARD`s by raising the `PortableServer::ForwardRequest` exception, but it does not provide a similar mechanism for normal servants. omniORBpy provides the `omniORB.LOCATION_FORWARD` exception for this purpose. The exception object is initialised with the target object reference. It can be thrown by any operation implementation. `omniORB.LOCATION_FORWARD` is not supported on omniORB 2.8.0.

6.12 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; 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 which have been imported. The modules themselves can be accessed through `sys.modules`. For example:

```
// 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'
```

Chapter 7

Connection Management

This chapter describes how omniORB manages network connections.

7.1 Background

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

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

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

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

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.

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

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

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

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

7.3 Idle Connection Shutdown and Remote Call Timeout

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

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

How often the internal thread scans the connections is determined by the value of the *scan granularity*. This value is defaulted to 5 seconds and can be changed using the command-line option `-ORBscanGranularity`. Notice that this value determines the precision the ORB is able to keep to the value of the idle connection or remote call timeout.

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

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

The timeout can be changed with the `-ORBclientCallTimeOutPeriod` and `-ORBserverCallTimeOutPeriod` options. The scan can be disabled completely by setting the scan granularity to 0.

7.4 Interoperability Considerations

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

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

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.

7.5 Connection Acceptance

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

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

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

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

```
# omniORB configuration file - extra options

GATEKEEPER_ALLOWFILE   /project/omni/var/hosts.allow

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

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

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

Whereas if the dummy module is in use:

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

Appendix A

hosts_access(5)

DESCRIPTION

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

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

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

ACCESS CONTROL FILES

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

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

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

ACCESS CONTROL RULES

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

- A newline character is ignored when it is preceded by a backslash character. This permits you to break up long lines so that they are easier to edit.
- Blank lines or lines that begin with a # character are ignored. This permits you to insert comments and whitespace so that the tables are easier to read.
- All other lines should satisfy the following format, things between [] being optional: `daemon_list : client_list [ : shell_command ]`

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

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

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

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

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

PATTERNS

The access control language implements the following patterns:

- A string that begins with a `.` character. A host name is matched if the last components of its name match the specified pattern. For example, the pattern `.tue.nl` matches the host name `wzv.win.tue.nl`.
- A string that ends with a `.` character. A host address is matched if its first numeric fields match the given string. For example, the pattern `131.155.` matches the address of (almost) every host on the Eindhoven University network (`131.155.x.x`).
- A string that begins with an character is treated as an NIS (formerly YP) netgroup name. A host name is matched if it is a host member of the specified netgroup. Netgroup matches are not supported for daemon process names or for client user names.
- An expression of the form `n.n.n.n/m.m.m.m` is interpreted as a 'net/mask' pair. A host address is matched if 'net' is equal to the bitwise AND of the address and the 'mask'. For example, the net/mask pattern `131.155.72.0/255.255.254.0` matches every address in the range `131.155.72.0` to `131.155.73.255`.

WILDCARDS

The access control language supports explicit wildcards:

`ALL`
: The universal wildcard, always matches.

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

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

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

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

OPERATORS

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

SHELL COMMANDS

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

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

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

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

% EXPANSIONS

The following expansions are available within shell commands:

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

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

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

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

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

`%p` The daemon process id.

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

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

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

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

SERVER ENDPOINT PATTERNS

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

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

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

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

CLIENT USERNAME LOOKUP

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

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

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

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

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

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

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

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

DETECTING ADDRESS SPOOFING ATTACKS

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

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

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

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

EXAMPLES

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

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

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

MOSTLY CLOSED

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

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

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

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

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

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

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

MOSTLY OPEN

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

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

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

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

BOOBY TRAPS

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

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

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

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

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

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

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

DIAGNOSTICS

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

FILES

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

SEE ALSO

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

BUGS

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

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

AUTHOR

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

Bibliography

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

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

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

[OMG00a] Object Management Group. *Interoperable Naming Service revised chapters*, August 2000. From `http://cgi.omg.org/cgi-bin/doc?ptc/00-08-07`.

[OMG00b] Object Management Group. *Python Language Mapping Specification*, April 2000. From `http://cgi.omg.org/cgi-bin/doc?ptc/00-04-08`.