http://www.dstc.edu.au/Fnorb

Version 1.0
12th April 1999

Martin Chilvers
CRC for Distributed Systems Technology
Level 7, GP South
Staff House Road
The University of Queensland
QLD 4072
Australia

fnorb@dstc.edu.au
tel: +61 7 3365 4310
fax: +61-7-3365-4311

0 Contents

1 Introduction

1.1 What is Fnorb?

Fnorb is an object request broker (ORB) compliant with the CORBA 2.0 specification[1] from the Object Management Group (OMG). Fnorb is written almost entirely in the Python programming language and, not surprisingly, implements a single language mapping from OMG IDL to Python. Because of the interpreted and interactive nature of Python, and the simplicity of the mapping (compared to C++ and Java), Fnorb is ideally suited as a tool for the rapid prototyping and scripting of CORBA systems and architectures.

1.2 What is Python?

Python is a portable, interpreted, object-oriented programming language that contains influences from a variety of other languages, most notably, ABC, C, Modula-3 and Icon. With an elegant syntax and powerful high-level datatypes it is easy to learn and ideal for CGI scripts, systems administration, and many other extension and integration tasks. More importantly, its support for rapid prototyping and object oriented programming make it a valuable tool for serious software engineering and product development.

The small, but not over simplified, core language provides the usual basic datatypes and flow control statements along with higher-level types such as strings, lists, tuples, and associative arrays. Object-oriented programming is supported by a class mechanism with a multiple inheritance model, and exception handling is provided using the familiar try/catch paradigm.

The real power of Python, however, lies in its extensibility. The language can be extended by writing modules in either Python itself, or compiled languages such as C and C++. These modules can define variables, functions, new datatypes and their methods, or simply provide a link to existing code libraries. It is also possible to embed the Python interpreter in another application for use as an extension language. The standard Python library includes modules for a wide range of tasks, from debuggers and profilers to Internet/Web services, and graphical user interfaces. In fact, if you need it, it is probably already there!

Python runs under many environments including most flavours of Unix, Windows 3.x, Windows 95 and NT, Macintosh, and OS/2. It is freely copyable and can be used without fee in commercial products. More information (including the source code!) can be obtained from http://www.python.org/.

1.3 Why do we need another IDL language mapping?

Ever since the seminal work “The Mythical Man Month”[2], many people have advocated that the only way to produce quality software is by employing exploratory methodologies such as prototyping (Brookes’ famous “build one to throw away”). Prototyping not only provides greater opportunity for end-user feedback than traditional “build and deliver” approaches, but also allows system architects and developers to experiment with critical design decisions before they really do become critical! In fact, when attempting to build complex systems, prototyping is arguably the *only* way to discover the subtle and often unexpected interactions between components.

Designing and deploying distributed systems *is* a complex problem and while CORBA goes some way to managing this complexity, the most commonly used language mappings, C++ and Java, are not simple enough to allow the rapid development of system prototypes, command scripts, and test harnesses etc. This is exactly where Python and Fnorb come in! Anecdotal evidence [3] suggests that, in terms of code size, Python programs are 3-10 shorter than C++ and 3-5 times shorter than Java. The language mapping from OMG IDL to Python maintains this advantage by allowing programmers to use familiar language data structures and idioms with minimal concessions to the notion of object distribution.

According to a recent ‘SunWorld’ article [4] Python is now considered one of the “big 3” scripting languages along with TCL and Perl. Compared with its rivals Python is easier to learn and is (much!) easier to read and maintain. The combination of Python and Fnorb provides the existing CORBA

community with a much needed tool for rapid prototyping and scripting, and gives those new to CORBA a great way to learn the fundamental concepts without being swamped by the intricacies of a 'heavyweight' language mapping.

1.4 About this document

This document is intended to be an introduction to using Fnorb and Python to build CORBA systems. It is *not* a CORBA tutorial. There are many excellent CORBA texts available, and the reader is expected to be familiar with at least the basic concepts including OMG's interface definition language (IDL). Similarly, this document will not teach you Python! The best way to learn this great language is to read the documentation provided in the standard distribution available at http://www.python.org.

2 Getting started with Fnorb ("Hello World" strikes again!)

The following sections assume that you have successfully installed Fnorb by following the instructions provided in the distribution (see the web page at http://www.dstc.edu.au/Fnorb for more details).

As dictated by tradition, we will start with a "Hello World" example! In this case it will be implemented as a client/server system, with a server object in one process offering the 'hello_world' operation, and a client in another process that locates the server object and invokes the operation on it.

2.1 Defining the interface in OMG IDL

As always, the first step in developing a CORBA system is to define the interface(s) in OMG IDL. In the "Hello World" example, this is trivial as we have a single interface containing a single operation! However, it is worth noting the use of the '#pragma prefix' directive, and the definition of the 'HelloWorld' module to contain our interface. These mechanisms ensure that we do not pollute the global namespace and should be adopted by every good CORBA citizen! The IDL shown below can be found in the file '…/Fnorb/examples/hello-world/HelloWorld.idl' (where '…' is the pathname of the Fnorb installation directory).

```
#pragma prefix "dstc.edu.au"

//
// "Hello World" example!
//
module HelloWorld {

    const string Message = "Hello CORBA World!";

    interface HelloWorldIF {
        //
        // The IDL version of the old faithful, 'Hello World' example!
        //
        // This operation takes no parameters and returns a string.
        //
        string hello_world();
    };
};
```

2.2 Writing the "Hello World" client.

Now lets take a look at the code for the "Hello World" client as found in the file '…/Fnorb/examples/hello-world/client.py'.

In typical Python fashion, we start by importing the modules and packages required by the application. *All* Fnorb programs, both clients *and* servers, must import the CORBA module from the package 'Fnorb.orb'. The client must also import the stub code for any CORBA interfaces that it wants to use. Stub code is generated by Fnorb's IDL compiler 'fnidl' (covered in detail in section 7.1), and in this example, the code we require is found in the package 'HelloWorld'. For the full scoop on the mapping from OMG IDL to Python see section 3.

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

""" Client for the "Hello World" example. """

# Standard/built-in modules.
import sys

# Fnorb modules.
from Fnorb.orb import CORBA

# Stubs generated by 'fnidl'.
import HelloWorld
```

The first thing that all Fnorb programs must do is initialise the ORB. This is achieved by calling the function 'ORB_init' found in the CORBA module. 'ORB_init' takes two parameters. The first is a list of command line options that can be used to configure the ORB (see section 4.2 for more details), and

the second is the vendor specific identity of the ORB, which in Fnorb must *always* be the value of the variable 'CORBA.ORB_ID'. 'ORB_init' can be called as many times as you like and always returns a reference to the previously initialised ORB (although you should be aware that command line arguments are only processed on the *first* call). **For convenience, 'ORB_init' can also be called with no parameters at all**.

```
def main(argv):
    """ Do it! """

    print 'Initialising the ORB...'

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

Having initialised the ORB, the client must now locate the server object. In this case when the server starts up it will write a 'stringified' version of its object reference to the file 'server.ref'. Obviously, this is not an ideal solution as it requires that the client and server share the same file system, but it will do for now! In practice the client would locate the server using more a flexible mechanism such as the CORBA naming and/or trading services.

```
    # Read the server's stringified IOR from a file (this is just a 'cheap
    # and cheerful' way of locating the server - in practise the client
    # would use the naming or trader services).
    stringified_ior = open('server.ref', 'r').read()
```

Unfortunately, the 'stringified' version of the object reference is not much use other than for passing around to your friends, and writing to databases etc, and so the client's next call is to the ORB method 'string_to_object. 'string_to_object' creates a 'proxy' object that has the same methods as the server object itself. When methods are invoked on the proxy, it forwards the request onto the server object in whatever process, on whatever machine it happens to be running at the time! In this way the client is insulated from the details of where the server object is physically located, and how operation requests are delivered to it.

```
    # Convert the stringified IOR into an active object reference.
    server = orb.string_to_object(stringified_ior)
```

Just because 'string_to_object' succeeds, it doesn't mean that we can use the server straight away. First of all, the reference could be a 'nil object reference'. Nil object references are how CORBA represents references to nothing at all – a pointer into space (cf. NULL pointer references in languages such as C and C++). In Python, nil object references are represented as the value 'None', and so the first check we have to make is as follows:-

```
    # Make sure that the server is not a 'nil object reference' (represented
    # in Python by the value 'None').
    if server is None:
        raise 'Nil object reference!'
```

Although, we now know that the object reference is not 'nil', it still doesn't mean that the server is actually up and running. The only way to make sure that this is the case is to attempt to invoke an operation on it (via the proxy object). A good choice is the '_is_a' operation which is available on *every* CORBA object. '_is_a' takes a CORBA interface repository identifier and asks the object if it implements an interface of that type. If the call succeeds then the client can be sure that a) the server object is alive and kicking, and b) the server object really does implement the expected interface. For a detailed description of interface repository identifiers see the CORBA 2.0 specification[1].

```
    # Make sure that the object implements the expected interface.
    if not server._is_a('IDL:dstc.edu.au/HelloWorld/HelloWorldIF:1.0'):
        raise 'This is not a "HelloWorldIF" server!'
```

Now that the client is sure that the server is what it says it is (i.e. an implementation of our 'HelloWorldIF' interface) it can make that all-important 'hello_world' call just as it would on a local Python instance!

```
    # Call the server!
    print server.hello_world()

    return 0
```

```
if __name__ == '__main__':
    # Do it!
    sys.exit(main(sys.argv))
```

2.3 Writing the "Hello World" server.

Writing the server for our example requires a little more work, but only a little! In this section we will discuss the server code as found in the file '…/Fnorb/examples/hello-world/server.py'.

As mentioned in the previous section *all* Fnorb programs, both clients *and* servers, must import the 'CORBA' module from the package 'Fnorb.orb'. Fnorb servers must also import the basic object adapter module 'BOA'. The BOA is an implementation of the basic object adapter interface defined in the CORBA 2.0 specification[1], and it is used to 'connect' interface implementations to the ORB. Unfortunately, the specification was hazy enough to allow every ORB vendor to implement the BOA in their own special way, and to address this problem, the OMG has approved the specification of a well defined and more portable object adapter known (not surprisingly) as the POA. It is expected that future releases of Fnorb will contain an implementation of the POA, and that use of the BOA may *eventually* be deprecated.

Just as in the client, the server imports the generated stub code from the 'HelloWorld' package. Strictly speaking this is *not* necessary, *iff* the server does not use any user defined IDL types. However, in practice, all but the most trivial interfaces *will* use user-defined types, and it is therefore probably good practice to import stub packages at the outset (in our example the server uses the constant string 'HelloWorld.Message' and so the stub module *is* required). More importantly, the server must also include skeleton code for every interface that it implements. Just like the stub code in the client, the skeleton code is generated by Fnorb's IDL compiler 'fnidl', and in this case, is found in the 'HelloWorld_skel' package.

```
#!/usr/bin/env python
""" Implementation of the HelloWorldIF interface. """

# Standard/built-in modules.
import sys

# Fnorb modules.
from Fnorb.orb import BOA, CORBA

# Stubs and skeletons generated by 'fnidl'.
import HelloWorld_skel
```

Now we come to the Python class that implements the interface we defined in the IDL. The only difference between defining a Fnorb server class as opposed to a standard Python class is that it must inherit from the skeleton generated by the IDL compiler. The skeleton class is found in the skeleton package and has the same name as the IDL interface with the suffix '_skel'. The server class must contain implementations for *all* of the operations defined in the IDL, which in this case means that we have to implement the single operation 'hello _world'! Note that in IDL the operation is defined as taking no parameters, but in its Python implementation we have to explicitly add the current instance parameter traditionally known as 'self'.

```
class HelloWorldServer(HelloWorld_skel.HelloWorldIF_skel):
    """ Implementation of the 'HelloWorldIF' interface. """

    def hello_world(self):
        print HelloWorld.Message
        return HelloWorld.Message

##############################################################################
```

Now let's look at how we create an instance of the server, and make it available to prospective clients. As in the client, the first thing that a Fnorb server must do is initialise the ORB using the 'ORB_init' function.

```
def main(argv):
    """ Do it! """

    print 'Initialising the ORB...'
```

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

Next comes an additional step only required by Fnorb *servers* – the initialisation of the BOA that will be used to 'connect' interface implementations to the ORB. The BOA initialisation function is (not surprisingly) called 'BOA_init', and just like the 'ORB_init' function it can be called multiple times with the second and subsequent calls simply returning a reference to the initial BOA instance (for more details on using the BOA see section 5).

```
    print 'Initialising the BOA...'

    # Initialise the BOA.
    boa = BOA.BOA_init(argv, BOA.BOA_ID)
```

In the CORBA world, objects are identified by opaque data structures known as object references. To create an object reference for our implementation we use the 'create' method on the BOA. 'create' takes two arguments, the object key (chosen by the server implementer i.e. you!), and the type of the interface.

```
    print 'Creating object reference...'

    # Create an object reference ('fred' is the object key).
    obj = boa.create('fred', HelloWorldServer._FNORB_ID)
```

Next we simply create an instance of our implementation class, as we would any other Python class.

```
    print 'Creating implementation...'

    # Create an instance of the implementation class.
    impl = HelloWorldServer()
```

Before the ORB can route any operation requests to our implementation, we have to 'connect' it to the object reference that we created previously. Note that even after this step is complete the implementation will not receive any requests until we start Fnorb's event loop.

```
    print 'Activating the implementation...'

    # Activate the implementation (ie. connect the generated object reference
    # to the implementation).  Note that the implementation will not receive
    # any operation requests until we start the event loop (see below).
    boa.obj_is_ready(obj, impl)
```

As mentioned when we created the client, this example uses a file to advertise the server's object reference. Here we use the ORB method 'string_to_object' to turn the object reference into a 'stringified' format and then write it to the file 'server.ref'.

```
    # Write the stringified object reference to a file (this is just a 'cheap
    # and cheerful' way of making the object reference available to the
    # client!).
    open('server.ref', 'w').write(orb.object_to_string(obj))

    print 'Server created and accepting requests...'
```

The final step is to start the BOA's event loop. This tells the BOA to listen for operation requests and to pass them on to the appropriate object implementation. Note that the '_fnorb_mainloop' method will never return unless an exception is raised, or the method '_fnorb_quit' is called from within an operation implementation.

```
    # Start the event loop.
    boa._fnorb_mainloop()

    return 0

############################################################################

if __name__ == '__main__':
    # Do it!
    sys.exit(main(sys.argv))

############################################################################
```

2.4 Generating the stub and skeleton modules

Before we can use the client and server code that we have just written, we have to create the stub and skeleton code using the IDL compiler 'fnidl'. This is done at the command line with the simple command: -

```
$ fnidl HelloWorld.idl
```

When the command completes you will (hopefully) see that two Python packages have been generated; 'HelloWorld' which contains all of the type definitions and the stub code required by the client, and 'HelloWorld_skel' which contains the skeleton code required by the server.

2.5 Running the "Hello World" example

The easiest way to test the example is to run the client and server programs in two separate windows. In one window start the server using the command 'python server.py'. If the server starts up successfully you should see the following output:-

```
$ python server.py
Initialising the ORB...
Initialising the BOA...
Creating object reference...
Creating implementation...
Activating the implementation...
Server created and accepting requests...
```

In the other window, run the client using the command 'python client.py'. Once again, if everything goes to plan, you should see the following output:-

```
$ python client.py
Initialising the ORB...
Hello CORBA World!
$
```

3 OMG IDL to Python Language Mapping

This section describes the language mapping from OMG IDL into Python. The mapping is currently a "work in progress" within the Python community, and it has not yet been submitted to the OMG for standardisation. Fnorb will attempt to keep up with the latest draft of the standard, and will implement the final standard as soon as is practically possible.

3.1 Scoped Names

Fnorb uses Python packages and classes to mirror the namespaces and scoping mechanisms found in IDL.

E.g. consider the following IDL: -

```
module Example {
    interface Foo {
        struct Bar {
            string baz;
        };
    };
};
```

The IDL scoped names are: -

- 'Example' (the module)
- 'Example::Foo' (the interface)
- 'Example::Foo::Bar' (the user defined structure)

To get the Python equivalent of the scoped names simply replace all of the '::' separators with '.'.

```
>>> import Example
>>> Example
<module Example>
>>> Example.Foo
<class Example.Foo at 132f68>
>>> Example.Foo.Bar
<class Example.Bar at 132d88>
```

If an IDL identifier clashes with a Python keyword it is *prefixed* with an underscore ('_').

E.g. In the following IDL:-

```
module Example {
    interface class {
        void while ();
     };
};
```

The IDL identifiers 'class' and 'while' are Python keywords and are therefore mapped to the Python identifiers '_class' and '_while'.

```
>>> import Example
>>> Example._class
<class Example._class at 133fa0>
>>> Example._class._while
<unbound method _class._while>
>>>
```

For a list of all Python keywords see Appendix A.

3.2 Mapping for Modules

All IDL modules (top level *and* nested) are mapped to Python *packages*. If there are any IDL definitions at the global scope (i.e. outside of a module definition) then the corresponding Python stubs

and skeletons are placed in the packages '_GlobalIDL' and '_GlobalIDL_skel' respectively (the names of these packages can be overriden using the '--globals=' option on the IDL compiler).

In order to prevent name clashes with other CORBA applications it is strongly recommended that you do not define any IDL types at the global scope!

E.g. Consider the following IDL: -

```
interface NastyGlobal {};

module TopLevel {
    interface Foo {};
    module Nested {
        interface Bar {};
    };
};
```

The IDL compiler (''fnidl') will generate the following packages (for more details of the IDL compilation process see section 7.1): -

'_GlobalIDL' and '_GlobalIDL_skel'

- Containing the stub and skeleton code for the 'NastyGlobal' interface.

'TopLevel' and 'TopLevel_skel'

- Containing the stub and skeleton code for the 'Foo' interface.

'TopLevel/Nested' and 'TopLevel_skel/Nested_skel'

- Containing the stub and skeleton code for the 'Bar' interface.

To use the packages simply import them as normal: -

```
>>> import _GlobalIDL
>>> import TopLevel
>>> import TopLevel.Nested
>>> _GlobalIDL.NastyGlobal
<class _GlobalIDL.NastyGlobal at bab40>
>>> TopLevel.Foo
<class TopLevel.Foo at 131e10>
>>> TopLevel.Nested.Bar
<class TopLevel.Nested.Bar at 9b188>
```

3.3 Mapping for Basic Data Types

Fnorb currently supports all CORBA 2.0 data types. Future releases will add support for the new CORBA 2.1 data types such as 'long long' and 'wchar' etc.

The mapping for IDL basic data types is given in the following table. As there are no type names for built-in Python types, the Python equivalent is given as the result of applying the built-in function 'type'! For the boolean type two predefined values 'CORBA.TRUE' and 'CORBA.FALSE' are available.

IDL	**Python (result of the 'type' function)s**
boolean	<type 'int'>
char	<type 'string'> of length 1
double	<type float>
float	<type 'float'>
long	<type 'int'>
long long	<type 'long'>

octet	<type 'int'>
short	<type 'int'>
unsigned long	<type 'long'>
unsigned long long	<type 'long'>
unsigned short	<type 'int'>

3.4 Mapping for Strings

IDL strings (both bounded and unbounded) map directly into Python strings. If an attempt is made to pass a string that exceeds the length of a bounded string a 'BAD_PARAM' system exception will be raised.

3.5 Mapping for Constants

An IDL constant is mapped to a Python variable with an equivalent name and scope. The Python variable is initialised with the value of the constant. Of course, Python does not have real 'constants', and nothing actually stops you assigning to these variables, but doing so should, at the very least, cost you your job ;^)

E.g. consider the following IDL: -

```
#pragma prefix "dstc.edu.au"

//
// Constant example.
//
module Example {

    const long MAX_LENGTH = 999;
    const string NAME = "Martin Chilvers";

    interface Foo {
        const float PI = 3.14;
    };
};
```

The constants can be used in Python as follows:-

```
>>> import Example
>>> Example.MAX_LENGTH
999
>>> Example.NAME
'Martin Chilvers'
>>> Example.Foo.PI
3.14
>>>
```

3.6 Mapping for Enumerations

An IDL enumeration is mapped into a number of Python objects in an equivalent scope. There is one Python object for each member of the enumeration, and a single Python object with the same name as the enumeration type itself.

E.g. consider the following IDL: -

```
Module Example {
    enum color {red, green, blue};
};
```

In Python the enumeration can be used as follows: -

```
>>> import Example
>>> Example.red
red
```

```
>>> Example.blue
blue
>>> Example.green
green
>>> Example.red == Example.blue
0
>>> for c in Example.color:
...     print c
...
red
green
blue
>>>
```

Note that enumerations can be compared and printed out, but you can make no other assumptions about how they are implemented (i.e. they *cannot* be treated as Python integers, or used in arithmetic operations etc).

One concession has been made to allow the **int** function to be applied to enumeration members. This is useful to allow them to be used to index sequences.

3.7 Mapping for Unions

IDL unions map to Python classes with two attributes, the discriminant ‘d’, and the value ‘v’. The constructor of the class expects the discriminant and value as parameters (in that order).

There are 3 possible states that a union can be in: -

1. If the discriminant is explicitly listed in a case statement, then the value must be of the type associated with that case.
2. If the discriminant is not explicitly listed in a case statement *and* there is a default case, then the value must be of the type associated with the default case.
3. If the discriminant is not listed in case statement and there is no default case then the value must be the distinguished Python value, `None`.

E.g. considering each possible state in turn: -

```
module Example1 {
    union MyUnion switch(boolean) {
        true: string s;
        false: long n;
};

>>> import Example1
>>> u = Example1.MyUnion(CORBA.TRUE, “Weh Hey”)
>>> u.d
1
>>> u.v
‘WeyHey’
>>> u = Example1.MyUnion(CORBA.FALSE, 123)
>>> u.d
0
>>> u.v
123
>>>

module Example2 {
    union MyUnion switch(boolean) {
        true: string s;
        default: long n;
};

>>> import Example2
>>> u = Example2MyUnion(CORBA.FALSE, 123)
>>> u.d
0
>>> u.v
123
>>>
```

```
module Example3 {
    union MyUnion switch(boolean) {
        true: string s;
};
```

```
>>> import Example3
>>> u = Example.MyUnion(CORBA.FALSE, None)
>>> u.d
0
>>> u.v
None
>>>
```

3.8 Mapping for Structures

An IDL structure is mapped into a Python class. The Python class contains a public attribute for each member of the structure. The class's constructor takes a parameter for each member of the structure in the order listed in the IDL.

E.g. consider the following IDL: -

```
module Example {
    struct Foo {
        string bar;
        long baz;
    };
};
```

In Python the structure can be used as follows: -

```
>>> import Example
>>> struct = Example.Foo("Hello", 99)
>>> struct.bar
'Hello'
>>> struct.baz
99
>>> struct.bar = "Goodbye"
>>> struct.bar
'Goodbye'
>>>
```

3.9 Mapping for Sequences and Arrays

IDL sequences (both bounded and unbounded) and arrays map directly into Python lists!

E.g. consider the following IDL: -

```
module Example {
    interface Foo {
        typedef sequence<long> sequenceVector;
        void bar(in sequenceVector v);

        typedef long arrayVector[10];
        void baz(in arrayVector v);
    };
};
```

In Python the interface 'Foo' can be used as follows: -

```
>>> import Example
>>> # Assume the variable 'server' contains a reference to an object supporting the
>>> # interface 'Example.Foo'.
>>> server.bar([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> server.baz([100, 101, 102, 103, 104, 105, 106, 107, 108, 109])
```

If an attempt is made to pass a bounded sequence that exceeds its maximum length, or an array that does not contain the appropriate number of elements, a 'BAD_PARAM' system exception is raised.

3.10 Mapping for Exceptions

User defined IDL exceptions are mapped to Python classes that inherit from the base class 'CORBA.UserException'. The classes behave in the same way as the classes that represent IDL structure types (see section 3.8).

3.11 Mapping for TypeCodes

The TypeCode interface is defined in the CORBA 2.0 specification[1] and is mapped to Python in the usual manner. Similarly, typecode constants for all of the standard IDL datatypes are defined in the 'CORBA' module. The typecodes for user-defined types are accessed using the function 'CORBA.typecode'. 'CORBA.typecode' takes a single parameter that is the interface repository identifier of a user-defined type, and returns its corresponding typecode.

E.g. Using the enumeration example from section 3.6: -

```
Module Example {
    enum color {red, green, blue};
};
```

The typecode for the 'color' enumeration can be accessed and used as follows: -

```
>>> from Fnorb.orb import CORBA
>>> import Example
>>> tc = CORBA.typecode('IDL:dstc.edu.au/Example/color:1.0')
>>> tc
<TypeCode.EnumTypeCode instance at adae0)
>>> for i in range(tc.member_count()):
...      print tc.member_name(i)
...
red
green
blue
>>>
```

3.12 Mapping for 'any'

The IDL type 'any' is mapped to the Python class 'CORBA.Any'. The constructor for this class takes a typecode and a value that conforms to that type. The class also has two corresponding accessor methods 'typecode' and 'value.

E.g. Using the structure example from section 3.8: -

```
module Example {
    struct Foo {
        string bar;
        long baz;
    };
};
```

An 'Any' containing an instance of the structure 'Foo' can be constructed and used as follows: -

```
>>> from Fnorb.orb import CORBA
>>> import Example
>>> # Create an instance of the structure.
... s = Example.Foo('Hello', 123)
>>> # Get the structure's typecode.
... tc = CORBA.typecode('IDL:dstc.edu.au/Example/Foo:1.0')
>>> # Now create the 'Any'.
... a = CORBA.Any(tc, s)
>>> a.typecode()
<TypeCode.StructTypeCode instance at adc30>
>>> a.value()
<Example.Foo instance at adc60>
>>>
```

3.13 Client Side Mapping

3.13.1 Mapping for Interfaces

IDL interfaces map to Python stub classes that contain methods for every operation and attribute defined on the interface. As in IDL, the Python class introduces a new naming scope, and any types defined within the IDL interface are also defined inside the Python stub class.

3.13.2 Mapping for Operations

Each operation on an interface maps to a method on the corresponding Python stub class. The method takes a parameter for each 'in' and 'inout' parameter defined in the IDL, and, in the spirit of the ILU mapping, returns the result, 'inout', and 'out' parameters according to the following rules:-

- If there is no result value, the method returns the special Python value `None`.
- If there is exactly one result value, the method returns that value.
- If there is more than one result value, the method returns a tuple, with the result (if specified) as the first element followed by any 'inout' and 'out' parameters in the order that they were specified in the IDL.

3.13.3 Mapping for Attributes

Interface attributes map to two methods on the corresponding Python stub class, an accessor and a modifier. The accessor has the same name as the attribute but prefixed with '_get_', and similarly, the modifier is prefixed with '_set_'. Obviously, IDL attributes defined as 'readonly' only have the accessor method!

The accessor takes no parameters and returns a value of the attribute type, while the modifier takes a single parameter of the attribute type and has no return value.

E.g. consider the following IDL: -

```
module Example {
    interface Foo {
        attribute string change_me;
        readonly attribute long read_me;
};
```

These attributes can be used in Python follows:-

```
>>> import Example
>>> # Assume the variable 'server' contains a reference to an object supporting the
>>> # interface 'Example.Foo'.
>>> print server._get_change_me()
'Some string value'
>>> server._set_change_me('This is the new value!')
>>> print server._get_change_me()
'This is the new value'
>>> print server._get_read_me()
1234
>>>
```

3.14 Server-side Mapping

3.14.1 Mapping for Interfaces

IDL interfaces map to Python skeleton classes that contain methods for every operation and attribute and defined on the interface. The skeleton class has the name of the IDL interface suffixed with '_skel'. Implementations use the skeleton via inheritance.

E.g. consider the following IDL: -

```
module Example {
```

```
    interface Foo {};
};
```

The server implementor must make sure that the implementation class inherits from the skeleton class as follows: -

```
import Example_skel
class FooImpl(Example_skel.Foo_skel):
    pass
```

4 Mapping for the ORB

Every Fnorb program must create an instance of the class CORBA.ORB. The following sections describe how to create and initialise the ORB, and how it can be configured using command line options and environment variables.

4.1 Initialising the ORB

The ORB is a singleton i.e. there is at most one ORB instance per process. The ORB is created and initialised using the function 'CORBA.ORB_init' which is declared as follows: -

```
def ORB_init(argv=[], orb_id=ORB_ID):
    """ Initialise the ORB.

    This is a factory function for the ORB class (the ORB is a singleton (ie.
    there can only be one ORB instance per process)).

    """
```

The first parameter, 'argv', is a list of command line configuration options (often passed into an application via the Python variable 'sys.argv'). The second parameter is the ORB identifier which in Fnorb must *always* be the value 'CORBA.ORB_ID'. 'ORB_init' can be called as many times as you like, and always returns a reference to the active ORB. Note that for convenience 'ORB_init' can be called with no parameters at all.

4.2 Configuring the ORB

Before describing the configuration options available for the ORB, we must first introduce the notion of what we call Universal Object Locators (UOLs)

4.2.1 Universal Object Locators

Universal Object Locators (or UOLs for short) are Fnorb specific and are an attempt to provide a more flexible mechanism for exchanging CORBA object references UOLs are a bit like URLs on the web in that they allow objects to be referenced by a number of different 'protocols'. Currently, four 'protocols' are supported, traditional CORBA stringified object references (protocol 'IOR'), files in the local file system (protocol 'file'), web documents (protocol 'http') and naming service names (protocol 'name').

UOLs can therefore contain: -

1. A traditional CORBA stringified object reference:-

   E.g. 'IOR:a stringified object reference'

2. A reference to a file on the local file system that contains a UOL:-

   E.g. 'file:/the/pathname/of/a/file/containing/a/UOL'

3. The URL of a web document that contains a UOL:-

   E.g. 'http://www.dstc.edu.au/Fnorb/etc/UOL'

4. A naming service name:-

   E.g. 'name:/a/path/through/the/naming/service'

4.2.1.1 Configuration Options

The ORB can be configured via the following 3 mechanisms (in order of increasing precedence):-

1. A configuration file.
2. Environment variables.
3. The command line

The following sections describe the options that are currently recognised by the ORB.

4.2.1.2 The name of a configuration file

Environment variable **FNORB_CONFIG_FILE**

Command line **--ORBconfig-file='filename'**

This option specifies the name of a configuration file that contains any of the remaining ORB options. For an example see the file '…/Fnorb/etc/fnorb.cfg'.

4.2.1.3 The UOL of the naming service

Configuration file **Naming Service**

Environment variable **FNORB_NAMING_SERVICE**

Command line **--ORBnaming-service='UOL'**

This option specifies the UOL of the naming service.

4.2.1.4 The UOL of the interface repository

Configuration file **Interface Repository**

Environment variable **FNORB_INTERFACE_REPOSITORY**

Command line **--ORBinterface-repository='UOL'**

This option specifies the UOL of the interface repository.

4.2.1.5 The threading model

Configuration file **Threading Model**

Environment variable **FNORB_THREADING_MODEL**

Command line **--ORBthreading-model='value'**

This option specifies the threading model used by the ORB. Possible values are:-

- 'Reactive' to use the ORB in single-threaded mode.
- 'Threaded' to use the ORB in multi-threaded mode

4.2.1.6 The size of the thread pool

Configuration file **Thread Pool Size**

Environment variable **FNORB_THREAD_POOL_SIZE**

Command line **--ORBthreading-pool-size=n**

This option specifies the size of the thread pool that is used to service operation requests (the default value is 10). This option only takes effect when the threading model is 'Threaded'!

4.3 Using the ORB

The CORBA module defines the interface to the ORB itself. In Fnorb, the following ORB methods are available: -

```
class CORBA:

    def object_to_string(self, object):
        “”” Create a stringified object reference.

        This method takes an object reference and ‘stringifies’ it i.e. turns it into
        a sequence of ASCII characters suitable for storing in a file sytem/database
        etc.

        “””

    def string_to_object(self, stringified_ior):
        “”” Convert a stringified object reference into a live object!

        The inverse of ‘object_to_string’,  this method takes a stringified object
        reference and returns an object reference that can be used to invoke
        operations.

        “””

    def list_initial_services(self):
        “”” List the names of the available object services.

        This method returns a list of strings containing the names of the CORBA
        services configured for the ORB.

        “””

    def resolve_initial_references(self, identifier):
        “”” Return a reference to the object service named by ‘identifier’.

        This method takes the name of a CORBA service and,  if the service is
        configured for the ORB,   returns the object reference of the service.  If the
        service is not configured for the ORB the method raise the
        ‘CORBA.ORB.InvalidName’ exception.

        “””

    def nil(self):
        “”” Generate a NIL object reference.

        A NIL object reference is one that is correctly initialised but does not
        ‘point’ to an implementation object (i.e. it is the CORBA equivalent of the
         NULL pointer in C/C++.

        “””
```

5 Mapping for the BOA

All Fnorb servers must create an instance of the class BOA.BOA. The following sections describe how to create and initialise the BOA, and how it can be configured using command line options.

5.1 Initialising the BOA

Like the ORB, the BOA is a singleton i.e. there is at most one BOA instance per process. The BOA is created and intialised using the function ‘BOA.BOA_init’ which is declared as follows: -

```
def BOA_init(argv=[], boa_id=BOA_ID):
    """ Initialise the BOA.

    This is a factory function for the BOA class (the BOA is a singleton (ie.
    there can only be one BOA instance per process)).

    """
```

The fist parameter, ‘argv’, is a list of command line configuration options (often passed into an application via the Python variable ‘sys.argv’). The second parameter is the BOA identifier which in Fnorb must always be the value ‘BOA..BOA_ID’. ‘BOA_init’ can be called as many times as you like, and always returns a reference to the active BOA. Note that for convenience, ‘BOA_init’ can be called with no parameters at all.

5.2 Configuring the BOA

The configuration options currently recognised by the BOA are:-

--OAport=PortNumber

- To specify the port number that the BOA should start on. This is useful if you want particular CORBA objects to always be available using the same object reference (when Fnorb has an implementation of the POA there will be better ways to achieve this!)

--OAhost=’hostname’

- To specify the hostname to use for the BOA’s socket (only really useful if the current host has more than one name, and you want to specify which name is used).

5.3 Using the BOA

The BOA module defines the interface to the BOA itself. In Fnorb, the following BOA methods are available: -

```
class BOA:

    def create(self, object_key, intrep_id):
        “”” Generate an object reference.

        This method creates an object reference containing the specified object key
        and for an interface of the specified type id.

        “””

    def obj_is_ready(self, obj, impl):
        “”” Connect an object reference to an implementation instance.

        This method ‘connects’ an object reference with a Python implementation
        instance.  Note that the implementation will not receive any operation
        requests until the event loop is started.

        “””

    def deactivate_obj(self, obj):
        “”” Tell the BOA not to invoke any more requests for this object reference.

        This method prevents any further requests being passed on to the
        implementation connected to the specified object reference.

        “””
```

The BOA also implements a Fnorb specific interface as follows:-

```
    def _fnorb_mainloop(self):
        “”” Start the Fnorb event loop.

        This method tells the BOA to start listening for operation requests.
        It will not return unless the ‘_fnorb_quit’ method is invoked from
        within an operation.

        “””

    def fnorb_quit(self):
        “”” Exit the Fnorb event loop.

        This method makes the BOA exit the event loop.  Currently,  the BOA does not
        wait for active requests to complete so it is up to the application to make
        sure that the server only allows the shutdown to take place in appropriate
```

```
        places.

        """
```

6 Using Fnorb with 'Tkinter' (Unix only)

This section describes how to use Fnorb in conjunction with the GUI toolkit 'Tkinter'.

In a nutshell, here is the problem. In server processes, Fnorb uses an event loop to listen for and dispatch operation requests. Unfortunately, Tk also uses an event loop to dispatch window events e.g. mouse movement, button up, button down. Obviously, if we want both Fnorb and Tk to respond in a sensible and timely manner, we must somehow combine the two event loops and have a single dispatcher delivering events to both systems.

The solution to this problem is to initialise Fnorb to use a special event dispatcher called the 'TkReactor'. Here is an example taken from the file '…/Fnorb/examples/tkinter/server.py': -

First of all, we have to import the 'TkReactor' module!

```
# Fnorb modules.
from Fnorb.orb import BOA, CORBA, TkReactor

def main(argv):
    """ Do it! """

    print 'Initialising the ORB...'

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

    print 'Initialising the Tk Reactor...'
```

And here is the single extra line of code that we have to add to make Fnorb and Tkinter work in perfect harmony ;^)

```
    # Because we are using Tk, we must use the 'TkReactor' which allows Tk
    # and Fnorb events to be handled from within a single event loop.
    #
    # The 'TkReactor' MUST be be initialised BEFORE the BOA.
    TkReactor.TkReactor_init()

    print 'Initialising the BOA...'

    # Initialise the BOA.
    boa = BOA.BOA_init(sys.argv, BOA.BOA_ID)
```

The rest of the program is the same for any other Fnorb server!

7 Fnorb tools

7.1 fnidl

'fnidl' is Fnorb's IDL compiler. It takes interface and type definitions in IDL files, and generates the stub and skeleton code required by Fnorb clients and servers. 'fnidl' currently assumes that you have a pre-processor named 'gcc', 'cpp', or 'cl.exe' somewhere on your path.

To compile an IDL file use the following command (note that the CORBA specification says that IDL files *must* have the '.idl' extension): -

```
$ fnidl IDLFileName.idl
```

Options currently allowed are those used by your pre-processor plus the following :-

'—directory=*pathname*'

- All Python module and packages generated by 'fnidl' will be placed in the directory specified by *pathname* which must already exist. The default is the current working directory.

‘—package=*packagename*’

- All Python module and packages generated by 'fnidl' will be placed in the specified package under the output directory (see above). The package name is in dotted notation (e.g. 'foo.bar.baz') and 'fnidl' will create any packages in the package name that do not already exist.

‘—globals=*basename*’

- Stubs and skeletons for IDL definitions at the global scope are put in the packages 'basename' and 'basename_skel' instead of the default '_GlobalIDL' and '_GlobalIDL_skel'.

7.2 fnior

‘fnior’ dumps the contents of a stringified object reference.

Usage: -

```
$ fnior IOR:12A4C......^D
$ fnior –f AFileContainingAnIOR
$ fnior `cat AFileContainingAnIOR`
$ fnior < AFileContainingAnIOR
```

7.3 fnmkior

‘fnmkior’ constructs an object reference and writes the stringified version onto stdout.

Usage: -

```
fnmkior Hostname Port TypeId ObjectKey
```

```
$ fnmkior sundial.dstc.edu.au 9999 IDL:dstc.edu.au/HelloWorld/HelloWorldIf:1.0 fred
```

7.4 fnfeed

‘fnfeed’ is used to parse IDL and store the type definitions in a CORBA compliant interface repository.

E.g. To parse the definitions in the file 'Example.idl': -

```
$ fnfeed Example.idl
```

This example assumes that you have configured Fnorb to use the appropriate interface repository (see section 4.2 for more details).

7.5 fngen

‘fngen’ is used to generate Python stub and skeleton code when the interface and type definitions are stored in a CORBA compliant interface repository rather than an IDL file.

E.g. To generate the code for a IDL module ‘Example’ stored in an interface repository: -

```
$ fngen IDL:dstc.edu.au/Example:1.0
```

This example assumes that you have configured Fnorb to use the appropriate interface repository (see section 4.2 for more details).

7.6 fnping

‘fnping’ is used (not surprisingly) to ‘ping’ objects to see if they are alive. ‘fnping’ takes a UOL as its input (see section 4.2.1 for more information about UOLs).

Usage: -

```
$ fnping UOL
$ fnping –f AFileContainingAUOL
$ fnping `cat AFileContainingAUOL`
```

```
$ fnping < AFileContainingAUOL
```

7.7 fnoptions

Given that Fnorb now takes options from 3 locations (a configuration file, environment variables and the command line), we figured that it might be nice to have a tool that shows you what options you have actually specified. This can be very useful, particularly when compared against the options that you *think* you have specified ;^)

Usage: -

```
$ fnoptions
```

8 Fnorb CORBA Object Services

8.1 fnifr

'fnifr' is Fnorb's implementation of the CORBA Interface Repository. The command line options currently available are as follows:-

'—ior'

- The IOR of the naming service is written to 'stdout'.

For more information on using the interface repository, see the CORBA 2.0 specification[1]

8.2 fnaming

'fnaming' is Fnorb's implementation of the CORBA Naming Service. The command line options currently available are as follows:-

'—ior'

- The IOR of the naming service is written to 'stdout'.

'—database=*directory*'

- Configures the Name Service to use a persistent database that is stored in the specified *directory*.

9 References

[1] The Common Object Request Broker Architecture and Specification Revision 2.0,
Object Management Group (July 1995, Updated July 1996)

[2] "The Mythical Man Month"
Frederick P. Brookes Jr,
Addison-Wesley ISBN 0-201-00650-2

[3] "Comparing Python to Other Languages"
Guido Van Rossum
http://www.cwi.nl/www.python.org/doc/essays/comparisons.html

[4] "Choosing a scripting language"
Cameron Laird and Kathryn Soraiz
SunWorld Online October 1997
http://www.sun.com/sunworldonline/swol-10-1997/swol-10-scripting.html

Appendix A: Python Keywords

Here is a list of all of the keywords in Python 1.5. Note that the module 'keyword' contains a list and a dictionary of the keywords, and also the very useful function 'iskeyword'.

A	and, assert
B	break
C	class, continue
D	def, del
E	elif, else, except, exec
F	finally, for, from
G	global
I	if, import, in, is
L	lambda
N	not
O	or
P	pass, print
R	raise, return
T	try
W	while