Writing Python CORBA Clients

Brad Chapman

Last Update–15 April 01

Contents

1 Writing Python CORBA Clients – Part I

This section deals with how to write a Python CORBA client that will connect with a 'SecretMessage' CORBA object on a server at EBI and retrieve a secret message.

This tutorial assumes you are using omniORB and the omniORBpy python bindings. These are available from http://www.uk.research.att.com/omniORB/index.html and http://www.uk.research.att.com/omniORB/omniORBpy/. Make sure you have omniORB and omniORBpy installed before beginning.

There are 4 stages involved in building a Python CORBA client:

- Get the IDL for the objects on the CORBA server.
- Compile the IDL file to get helper classes which will use in creating a connection.
- Get the IOR for the CORBA object on the server
- Write a client that connects to the CORBA object on the server using it's IOR and fetches data from it.

1.1 Get the IDL for the 'SecretMessage' object

The IDL (Interface Definition Language) describes what you can do with an object; it describes the "interface" to the object on the CORBA server.

The IDL for the SecretMessage object is:

```
module messenger
{
  interface SecretMessage
  {
    string get_message();
  };
};
```

You'll want to copy these five lines into a file called `secret_message.idl` into a working directory. In this case we'll call the directory we're working in `MyFirstClient`.

The 'module' just describes a "box" into which an object is stored (there can be lots of objects in the same box).

The object is called `SecretMessage` and it has one method `get_message` that will return a string.

1.2 Compile the IDL file

We need to compile the IDL file to generate "stub" and "skeleton" classes. These classes are used to help implement clients and servers in CORBA. In either a UNIX shell or an MS-DOS window, type:

```
[MyFirstClient]$ omniidl -bpython secret_message.idl
```

omniidl is the name of the IDL compiler that comes with omniORB.

This creates two new directories and a new file. The directories are `messenger` and `messenger__POA`, which will need to be imported by python clients and servers, respectively. In our case, since we are just writing a client, we'll only need to import things from the `messenger` directory. The file that is created is `secret_message_idl.py`. This file actually contains all of the implementation information needed by the `messenger` and `messenger__POA` directories. You don't need to worry at all about the information in these files, but you might want to look at them if you are interested in how things are working under the covers.

1.3 Get the IOR for the CORBA object on the server

The client first needs to find where the 'SecretMessage' object is on the Internet. This is done in CORBA using an IOR (Interoperable Object Reference). In a string form, this is really just a complicated URL that describes the location, port and name of the object where the remote server is located.

In our example, we can get the stringified form of the IOR from http://industry.ebi.ac.uk/~alan/IOR/message.SecretMessage-Server.ior. To get the IOR in our python program, we'll just use the `urllib` built-in module to read it:

```
import urllib

SECRET_MESSAGE_URL = "http://industry.ebi.ac.uk/~alan/IOR/" + \
                     "message.SecretMessage-Server.ior"

ior_handle = urllib.urlopen(SECRET_MESSAGE_URL)
secret_message_ior = ior_handle.read()
ior_handle.close()
```

The IOR we need is held in the `secret_message_ior` variable. Don't forget this, as we'll need it in just a second to connect to the server.

1.4 Write the client...

Now that we've got the IOR, we are ready to connect to the server and get that secret message we've been talking so much about. First, we need to import the CORBA implementation. We can do this with:

```
import CORBA
```

If we want to be explicit about using omniORB, we can also do this import with:

```
# omniORB
from omniORB import CORBA
```

Now import the code produced by the IDL compiler:

```
# skeleton code generated by the IDL compiler
import messenger
```

Now we need to initialize a CORBA ORB, and use this to resolve the IOR we got into an actual object we can use. We do this will the following lines:

```
import sys
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
secret_message_server = orb.string_to_object(secret_message_ior)
```

Now that we've got the server, we can get the secret message:

```
print "Secret Message:", secret_message_server.get_message()
```

Excluding comments, this entire program only has 11 lines of code:

1. 5 lines to get the IOR string from its web location
2. 1 line to import the omniORB CORBA implementation
3. 1 line to import the IDL generated code
4. 1 line to import the standard library module sys, so that we can pass command line arguments to the ORB (this allows configuration of ORB behavior from the command line).
5. 1 line to initialize a CORBA ORB
6. 1 line to get the remote server
7. 1 line to get the message and print it out

If you run this program with:

```
[MyFirstClient]$ python secret_message_client.py
```

Then you'll get to see the secret message. (No, I'm not going to show it to you here – it's supposed to be secret, you know!).

Looking at the code again, you'll notice that there are only 4 lines of actual CORBA specific code:

- Load the CORBA libraries with `from omniORB import CORBA`.
- Load the IDL produced files with `import messenger`.
- Create a client ORB object that will connect with the remote CORBA server: `orb = CORBA.ORB_init([], CORBA.ORB_`
- Use the IOR to get the reference to the remote CORBA object: `secret_message_server = orb.string_to_object(se`

Once we get the `secret_message_server` object, then we can start calling methods on it, just as we would with any other python object. This means, all we should need to do is understand how to get objects, which we just did, and then understand IDL so we know what methods to call. Not too hard!

1.5 About the CORBA SecretMessage Server

The CORBA server is written in Java – so we are easily communicating with a remote Java program without any trickery on our part at all. Pretty cool.

The CORBA server that serves out the message is only about 15 lines of code:

1. 2 to create the ORB server
2. 1 to create the 'SecretMessage' object.
3. 2 to create the IOR
4. 1 to attach the 'SecretMessage' object to the ORB server
5. 2 to activate the server and have it wait for requests
6. 7 to write out the IOR (File I/O in Java sucks!)

2 Writing Python CORBA Clients – Part II

Now that we've got the basics of writing a client down, we are ready to start writing CORBA servers that will deal with biological sequences. This example allows us to connect to another server at the EBI, and retrieve DNA sequences. In addition, we also get to learn all about the 'in' IDL keyword, and how exceptions are handled in CORBA.

As in the first section, we're going to use the omniORB python bindings to write our CORBA client. We follow the same steps as before, but now we are going to be using a brand new client. Ready for the excitement? Here we go...

2.1 Compile the BioSequenceServer IDL file

Once again we need an IDL file that will define the interface for the server object. This will tell us all of the methods that we can call on the server we retrieve.

In this case, the CORBA object we're going to be getting is a server at the EBI that takes an identifier for a biological sequence (ie. a GenBank or EMBL accession number) and returns the DNA sequence corresponding to that identifier. This is pretty nifty, and can save you a lot of time if you normally do this by loading up EMBL, searching for your sequence, and then pasting it into the file you are working.

Let's think about the type of IDL we could use to set up a simple server like this. A first try could be something like this:

```
module servers {
  interface BioSequenceServer
  {
    string get_biosequence(in string identifier);
  };
};
```

This IDL is a little more complicated then the one we had previously, since we now have an input to the function. The `in` keyword specifies that we need to pass an argument to the server. Other keywords which could also be here are `out` which specifies an argument that is returned from the server; and `inout` which specifies that an argument is sent, modified and returned. These are more complicated (and less often used), so we're just going to be simple here and stick with the `in` case.

Additionally, since IDL can be used for connecting to languages like Java and C++, it is very important that arguments specify some kind of strict type. This is kind of foreign to python programmers, since we don't have to worry about type in our functions, but don't worry too much about it – as long as you do your part and actually pass the function a string, everything will be just fine!

Looking again at this IDL, we can see a bit of a problem. What if someone calls the `get_biosequence` function with a string that is not a valid accession number found in the EMBL database? Right now, it is undefined what exactly will happen, and likely we'll get some kind of random error message and traceback that doesn't tell us much about what really happened. The way we deal with this is very similar to the way we would deal with the same problem in python – we raise an Exception that details what the problem was. In IDL we need to define what exceptions we are going to throw (for the same reasons that we had to define a type of the function arguments). So the IDL now looks like this:

```
module servers {
  // Exception thrown when a given ID does not exist in a database.
  exception DoesNotExist { };

  interface BioSequenceServer
  {
    string get_biosequence(in string identifier) raises (DoesNotExist);
  };
};
```

Now we know that the server should raise a DoesNotExist exception if we pass in an identifier which doesn't exist in the database. As we'll see, we can catch these exceptions just like normal python exceptions.

To get started, let's create a directory `MySecondClient` and copy the above IDL into the file `biosequence.idl`. Now we need to compile the IDL file to generate the necessary helper classes just as before:

```
[MySecondClient]$ omniidl -bpython biosequence.idl
```

Now we're ready to write the client, so let's get to it!

2.2 Writing the Client

Now we're going to write up a little script that takes an accession number as input and prints out the sequence for the accession number, or a message if the identifier doesn't exist. This works much the same as the previous example, so hopefully it is relatively simple to follow with the comments:

```
# standard modules
import urllib
import sys

# omniORB
import CORBA

# generated stubs
import servers

# check to make sure we've got good arguments
if len(sys.argv) != 2:
    print "Usage:"
    print "python biosequence_client.py <Identifier>"
    print "<Identifier> is an accession number to retrieve the sequence for."
    sys.exit()

# First grab the sting IOR
BIOSEQUENCE_URL = "http://industry.ebi.ac.uk/~alan/IOR/" + \
                  "servers.BioSequenceServer-Server.ior"

ior_handle = urllib.urlopen(BIOSEQUENCE_URL)
```

```
biosequence_ior = ior_handle.read()
ior_handle.close()

# Now start up the ORB and get the server
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
biosequence_server = orb.string_to_object(biosequence_ior)

# Try to query the accession, and catch errors if they occur
try:
    seq = biosequence_server.get_biosequence(sys.argv[1])
    print "Query:", sys.argv[1]
    print "Sequence:", seq
except servers.DoesNotExist:
    print "The identifier %s does not exist in the database." % sys.argv[1]
```

Now we can run this script with some different inputs and see how it reacts. With a good identifier, we get back the sequence:

```
[MySecondClient]$ python biosequence_client.py HSERPG
Query: HSERPG
Sequence: agcttctgggcttccagacccagctactttgcggaactcagcaacccaggcatctctgagtctccg
....
```

With a fake identifier, we get an appropriate error message:

```
[MySecondClient]$ python biosequence_client.py X12345
The identifier X12345 does not exist in the database.
```

Very nice – now we've got CORBA working for something useful. If you want to make this even fancier, how about modifying the above script so it takes multiple identifiers on the command line and spits back out a file in FASTA format? That sounds like a good project for you!

3 Writing Python CORBA Clients – Part IV

The last section had a nice IDL describing all of the different types you could use in an IDL. Once again, there is a demo server at the EBI that we can use, so we're going to implement a python client to test it all out. This will help demonstrate how we use each of the different types in python. Since we normally don't worry too much about types in python, even dealing with all of these different types can be very easy. See the official python mapping specification at http://cgi.omg.org/cgi-bin/doc?ptc/00-04-08 for more details on how different types are mapped to python objects.

As in the other python examples, we're going to use the omniORB python bindings, so make sure you've got 'em running, and off we go.

3.1 Compiling the IDL file

You should copy the demo IDL file from the previous section into a file called `idl_demo.idl` and then generate the CORBA stubs and skeletons using the omniORB idl compiler:

```
[MyThirdClient]$ omniidl -bpython idl_demo.idl
omniidl: omniORBpy does not support the wchar type.
```

You'll notice this generates a warning message from the omniidl compiler. Wide character support is not yet standard across ORBs, and there in fact aren't any available ORBs for python that support wide characters. This is changing, and hopefully soon python will have full Unicode support through CORBA.

Anyways, it isn't a good thing that we are getting a warning, so we need to get rid of the wchar part of the IDL file. We can do this by commenting it out – so you get your first chance to modify an IDL file. Pretty exciting, eh? Anyways, you need to find the wchar part of the IDL and comment it out (IDL uses C++ style comments), like:

```
// wchar get_wchar();
```

Then recompile with `omniidl` and you shouldn't get any warnings. Then you're all ready to go.

3.2 Writing the Client

Now we're ready to write a CORBA client that demonstrates all of this fun type stuff. Now that you're an expert at all of this, we can dive right into the code:

```
# standard modules
import urllib
import sys

# omniORB
import CORBA

# generated stubs
import idl_demo

# First grab the sting IOR
IDL_DEMO_URL = "http://industry.ebi.ac.uk/~alan/IOR/" + \
               "idl_demo.DemoFactory.ior"

ior_handle = urllib.urlopen(IDL_DEMO_URL)
idl_demo_ior = ior_handle.read()
ior_handle.close()

# Now start up the ORB and get the server
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
demo_factory = orb.string_to_object(idl_demo_ior)

# --- BasicDataTypes demo
print "BasicDataTypes demo:"

# get a reference to the BasicTypes object from the factory
basic_types = demo_factory.get_basic_data_types_demo()

# now get all of the different types
print "An IDL short:", basic_types.get_short()
print "An IDL long:", basic_types.get_long()
print "An IDL longlong:", basic_types.get_longlong()
print "An IDL char:", basic_types.get_char()

# wide chars aren't supported by python ORBs yet
# print "An IDL wchar:", basic_types.get_wchar()

print "An IDL string:", basic_types.get_string()
print "An IDL boolean (true):", basic_types.get_true()
print "An IDL boolean (false):", basic_types.get_false()
```

```
# --- Constants demo
print "\nConstants demo:"

# constants inside an interface are available as class variables of the
# Constants class, so we do not need a reference to a class object
# to get a constant
print "PI is:", idl_demo.Constants.PI
print "ZERO is:", idl_demo.Constants.ZERO

# --- enum demo
print "\nenum demo:"

# get a reference to the MultipleChoice object from the factory
multiple_choice = demo_factory.get_enum_demo()

# get back 5 different enum objects by calling get_answer on 5 different
# question numbers
for question_number in range(1, 6):
    answer = multiple_choice.get_answer(question_number)
    print "The answer to question number %s is %s" % (question_number, answer)

# --- struct demo
print "\nstruct demo:"

# get a reference to the TestStruct object from the factory
test_struct = demo_factory.get_struct_demo()

# send data to the TestStruct object, and we'll get back a struct with the
# given info
personal_info = test_struct.get_struct(2, 0.80, "Mary")

# the returned "struct" is a python class with the items as attributes
# of the class
print "Name is:", personal_info.name
print "Height is:", personal_info.height
print "Age is:", personal_info.age

# --- Collections demo
print "\nCollections demo:"

# get a reference to the Collections object from the factory
collections = demo_factory.get_collection_demo()

# get the IDL sequence, which is returned as a python list
# (in this case, it is a list of doubles)
double_seq_list = collections.get_double_sequence()

# we can deal with this just like a regular ol' python list
print "Sequence length:", len(double_seq_list)

for item in double_seq_list:
    print "Item:", item
```

```
# get the IDL array, which also is returned as a python list
double_seq_array = collections.get_double_array()

print "Array length:", len(double_seq_array)

for item in double_seq_array:
    print "Item:", item
```

You copy this script into a file called `idl_demo_client.py` and run the script using:

```
[MyThirdClient]$ python idl_demo_client.py
```

You should look through the output, the script and the IDL to get an idea of how to work with CORBA objects using python. It is really simple, and most of the mapping is very intuitive. The python CORBA mapping specification, mentioned earlier is a good reference to look at if something seems confusing. It is only about 20 pages of writing, and is definately a good read if you are serious about python and CORBA. Also, I found it useful to compare this script to Perl and Java scripts in the other sections of this Tutorial, to see how the mappings vary depending on the programming language used.

Now you've been exposed to lots about CORBA and python. Since you know as much as I do now, the next section will be devoted to getting you started writing your own CORBA clients by directing to you to useful resources. Good luck!