Writing Python CORBA Servers

Brad Chapman – chapmanb@arches.uga.edu

Last Update–28 April 01

Contents

1 Writing Python CORBA Servers – The Basics

This tutorial is designed to help explain how to start writing CORBA servers in python. It is meant to be a companion piece to Alan Robinson's very nice tutorial setup about writing CORBA clients (http://industry.ebi.ac.uk/~alan/CORBA/Tutorials/Introduction/). It would be a good idea to read that set of Tutorials first, since it deals with a lot of basic CORBA concepts you need to be familiar with.

Writing servers is the hard part of CORBA, but definitely not that hard! In this section, we'll just look at a very simple example. The example code in this section is written using omniORBpy, a very nice CORBA ORB implementation for python. omniORBpy is available from http://www.uk.research.att.com/omniORB/omniORBpy/ and omniORB itself is available from http://www.uk.research.att.com/omniORB/.

For our first example, we are going to look at the SecretMessage IDL we talked about in the CORBA client examples. The purpose of this interface (besides being really simple!) is to return a secret message through CORBA.

There are five steps involved in writing our server:

- Get the IDL for the CORBA server.
- Compile the IDL to get helper classes which will make it easy for us to write the server.

- Write a class in python which implements the functionality defined in the interface. This will actually do the server work.
- Write the code necessary to create servers.
- Make the server available by writing its IOR to someplace readily available.

1.1 The Secret Message IDL

The IDL (Interface Definition Language) describes the interface to an object, or what things you can do with it. The IDL for our example looks like:

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

There are just three things in this IDL definition:

- The "namespace" for the entire thing, which is specified by `module messenger`. This makes everything be inside the messenger namespace.
- The object we are going to be dealing with. In this case, we are using a SecretMessage object, defined by `interface SecretMessage`.
- The function we are going to be calling. This function is called `get_message()` and will return a string.

1.2 Compiling the IDL file

As in the client examples, we need to compile the IDL to produce "skeleton" and "stub" classes. For the server, we'll be using the stub classes to help write our implementation. To compile the IDL file with omniORBpy, you just need to type in a UNIX shell or MS-DOS window:

```
omniidl -bpython secret_message.idl
```

This will call the IDL compiler from omniORB (called omniidl), which will generate two directories:

1. messenger – These are the skeleton classes, which contain the basic definition of the interface and any defined constants (we don't have any in this case.
2. `messenger__POA` – These are the stub classes, which contain basic (stub) descriptions of the SecretMessage object. We'll use these to help create our server implementation.

1.3 Writing an implementation of the server

Since we are writing the server, the first thing we'll need to do is write some python code that actually does the work of the server. In our case, we'll need to write a SecretMessage object that provides the `get_message()` function.

First, let's look at the implementation of the server, then we'll explain how it works:

```
# stubs and skeletons
import messenger, messenger__POA

class OurSecretMessage(messenger__POA.SecretMessage):
```

```
    def __init__(self, secret_message):
        """Intialize with the secret message we're going to use.
        """
        self.secret_message = secret_message

    def get_message(self):
        return self.secret_message
```

Let's start at the beginning and walk through the server step by step. First, you need to import the stubs and skeletons that were generated by the IDL compiler. This gets us all of the helper classes we'll need to write our server.

After importing the relevant stuff we are ready to dive right in and write the class to do all of the work. If you look back at the IDL defining the interface, you'll see that there is a SecretMessage interface containing the function. In python, interfaces are mapping into classes, so we need to write a class to implement the SecretMessage interface. Here, we name the class `OurSecretMessage` – it can be named anything you want, although I normally name my classes after the relevant IDL names, so that I can remember what they do!

The most important thing about the class is that it needs to inherit from the stub class definition. All of the stub classes are in the `messenger__POA` module. These stub classes do all of the tricky stuff that needs to be done to work with CORBA. By deriving from these classes, we save ourselves all of the work of implementing this CORBA specific code ourselves.

Once we define the class, we are ready to implement it. First, we use a standard python initializer and allow our `__init__` function to be passed the secret message we're going to make available. This brings up an important point – we can do anything inside python CORBA classes that we normally do in python classes. Just think of CORBA classes as having special "extensions" that allow them to work with CORBA.

Well, we've got the class, but we need to do our actual job and implement the function defined in the interface. To get this function called, it will need to be named the same thing as specified in the IDL. So, we write a `get_message` function. This function must return a string, as specified in the IDL definition. We return a python string object, and the inner CORBA workings will be kind enough to convert this string to be ready to pass through CORBA.

Well, that's it, now we've got our class. Now we just need to get an instance of that class, which we do in the normal way:

```
our_message = OurSecretMessage("I don't want any spam!")
```

Now, let's see how to create a CORBA server and register the `OurSecretMessage` object with it.

1.4 Setting up the server to be a CORBA server

You can think of a CORBA server as being similar to an http or ftp server – it's job is going to be to sit around and wait for requests, and then process those requests when they come. Of course, CORBA servers are much cooler than http or ftp servers since you can easily customize them for exactly what you are doing. Now, we need to set up a server to provide secret messages.

There are basically three steps we'll have to follow:

- Initialize the ORB (Object Request Broker) and POA (Portable Object Adaptor). These are the standard objects we'll need to do things with CORBA objects.
- Activate the POA Manager. This objects job will be to manage all objects registered with the POA, and make them available.
- Activate the Secret Message object. This will make it known to the POA and thus make it able to be accessed via CORBA.

1.4.1 Initializing the ORB and POA

As with the client, we need to start dealing with CORBA by getting the standard CORBA objects. For servers, there are two of these standard objects – the Object Request Broker (ORB) and the Portable Object Adapter (POA). The code to initialize them is:

```
# standard modules
import sys

# omniORB
import CORBA

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

The ORB initialization code should look similar to what you saw with clients. The new thing we get here is the POA. In this case we deal with the base POA object, the Root POA. The ORB will be started with an initial reference to this Root POA, so you use the standard ORB function `resolve_initial_references` to get this POA.

The POA has very important functionality for a server, since it is the object that will (transparently) manage all of the objects we have on the server, and make them available.

1.4.2 Activating the POA Manager

Now that we've got the POA, we need to get it ready. Each POA has a POA Manager, which, as it's name suggests, is the object that manages all objects registered with the POA.

It is very important to activate the POA manager before starting to deal with our own objects. You can do it with two lines of code:

```
# activate the POA manager
poa_manager = poa._get_the_POAManager()
poa_manager.activate()
```

Now everything is set up on the server, and we are ready to start setting up our own objects.

1.4.3 Activating the SecretMessage Object

We've got the `OurSecretMessage` object, and we've got the basic server components – now we need to combine them together to make our object available via CORBA. To do this, we need to activate the object, and register it with the POA manager. This is all done in one line using the `_this()` function of our object. We got this function "for free" earlier by inheriting from the base `messenger__POA.SecretMessage` class.

```
message_obj = our_message._this()
```

Now our message is ready and we've got the object. We're almost ready to go!

1.5 Making the server available

The final thing that we need to do is make this server available so that a client can get in. In this case, what we'll do is write the IOR to our local directory on the web server, which will make it available to anyone who can connect. This assumes that you've got an http server set up and everything. If you don't, or don't want to do this, you can get the IOR to other people any number of other ways (send it to them in an e-mail, put it on an anonymous ftp server).

What we'll do is write the IOR as `message.ior` into the standard `public_html` directory of our home directory. First, we'll do standard python stuff to create this directory if is doesn't exist, and get the file we are going to write:

```
output_dir = os.path.join(os.environ["HOME"], "public_html")
if not(os.path.exists(output_dir)):
    os.makedirs(output_dir)
output_file = os.path.join(output_dir, "message.ior")
output_handle = open(output_file, "w")
```

Now, we need to convert the object reference we got earlier into a string, so that we can write it out to the file. This is done via the ORB function `object_to_string`:

```
string_ior = orb.object_to_string(message_obj)
```

Finally, we'll just write the IOR to the file:

```
output_handle.write(string_ior)
output_handle.close()
```

Now that we've got the reference written out, we are at the final step – start up the server. Since we've worked hard so far, this is really easy:

```
orb.run()
```

Now the server will sit and wait for requests to come in, and process them when they do. Congratulations! We've got it all set up. Feel free to use any of the clients from the client tutorials to connect with this server and get the secret message.

2 Writing Python CORBA Servers – Getting more advanced

Now that we've got a basic server going, we're ready to attack some more advanced aspects of writing servers. In this section, we're going to tackle writing a CORBA Server to make biological sequences available. In addition to looking at a more realistic example, we'll also cover raising CORBA exceptions in the server, and using the standard CORBA naming service. Although passing string IORs works well in many cases, the naming service can be pretty handy to know about, so it is good to be familiar with how it works from python.

2.1 The BioSequence example

As in the client examples, we're going to use an IDL that allows us to make biological sequences available. The IDL definition for this interface is:

```
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);
  };
};
```

This is similar to the IDL we had before except for two things:

- The function `get_biosequence` now takes an input parameter – a string specifying the identifier of a sequence.
- The function can raise a CORBA exception, `DoesNotExist`.

As before, we need to compile this IDL file. Assuming that is is saved as `biosequence.idl` we can do this with:

```
omniidl -bpython biosequence.idl
```

Now we've got the stubs and skeletons generated, and we're ready to write the server.

2.2 Writing the Biosequence Server

Since we're already familiar with writing servers from the last example, we can dive right in and present a Biosequence server implementing the IDL interface:

```
# stubs and skeletons
import servers, servers__POA

class BiosequenceServer(servers__POA.BioSequenceServer):
    def __init__(self):
        """Initialize the server.

        Attributes:

        o sequences - A dictionary mapping IDs to sequence strings.
        Items should be added using the add_sequence function.
        """
        self._sequences = {}

    def add_sequence(self, identifier, sequence):
        """Add a sequence with the specified ID to the database of sequences.
        """
        self._sequences[identifier] = sequence

    def get_biosequence(self, identifier):
        """CORBA function which returns a sequence using an identifier.
        """
        try:
            return self._sequences[identifier]
        except KeyError:
            raise servers.DoesNotExist

bioseq_server = BiosequenceServer()
bioseq_server.add_sequence("HSERPG",
                           "agcttctgggcttccagacccagctactttgcggaac" +
                           "tcagcaacccaggcatctctgagtctccg")
```

This creates a Python object, called `BiosequenceServer`, which inherits from the stub CORBA classes, and implements the CORBA function `get_biosequence`. Notice that the `get_biosequence` function takes a single string argument, the identifier, this time. The CORBA stubs will do all of the work of converting a CORBA string into a python string, so all you have to worry about is doing something with this string.

In addition to taking an argument, the `get_biosequence` function also can raise an Exception. For python users, this should make a lot of sense, since exceptions are a natural way to express error conditions, such as trying to get a sequence that doesn't exist. In our example, we'll try to get the identifier from our internal dictionary of identifiers and sequences. If the identifier isn't found, python will raise a KeyError. We catch this KeyError, and use it as a signal to raise the appropriate CORBA exception.

You raise a CORBA exception exactly as you would a regular python exception, using the `raise` keyword. In our case, the DoesNotExist Exception is specified under the `servers` namespace. A CORBA exception

like this will be transmitted to the client, which then has a chance to handle it. So, raising exceptions is nearly identical to what you already know about them in python.

After defining the server, we then create a server object and add a sequence to it with the identifier `HSERPG`. Notice that to add sequences we've defined an addition function `add_sequence` in the `BiosequenceServer` class. It is perfectly okay to define functions not already defined in the interface, but these will only be available "locally" and not through CORBA.

Well, there we go, we've got our server – now we're ready to start attaching it to the naming service so it'll be available.

2.3 Setting up the naming service

Before we start messing around with the naming service, we need to be sure that we have one running. This section will describe the very basics of setting up the omniNames naming service which is supplied with omniORB. If you are not using omniORB, or if you want more specifics, the best place to consult is the specific documentation for your ORB implementation.

As mentioned previously, the naming service in omniORB is called omniNames. Getting it running is not too difficult. To run it for the first time you'll need to type:

```
omniNames -start
```

omniNames has a default directory where it tries to write it's log file (/var/omninames/ on UNIX, I think). If this directory doesn't exist, omniNames may complain – you can either create the directory or set the `OMNINAMES_LOGDIR` environmental variable to specify a new directory.

After you start it once, you can start it subsequent times with just `omniNames`. If everything is working right, you should see something like:

```
$ omniNames

Sat Apr 28 14:23:18 2001:

Read log file successfully.
Root context is
IOR:000000000000002b49444c3a6f6d672e6f72672f436f734e616d696e
672f4e616d696e67436f6e746578744578743a312e300000000000010000
00000000002700010000000000b3137322e31362e302e3500ff0af9ffff
0000000b4e616d6553657276696365
Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.
```

You'll have to be sure the naming service is running while you are running this example.

2.4 Making the Server available via the naming service

Now that we've got the naming service running, and we are ready to use it to make our object available.

2.5 Initializing the ORB and POA

As before, we first need to get the Object Request Broker and Portable Object Adaptor. We have one little extra wrinkle this time, though, we need to let the ORB know where the NameService is located at. This is very important, because the ORB has no way of knowing by itself how to contact the specific naming service you are planning to be using.

In our case, we're just learning, so we'll be running the NameService on our local machine. We're just going to do things easy, and having the naming service running on the default port on the machine and

everything. The CORBA specifications have defined a common naming convention to use for specifying the location of an object. For our naming service, this string is `corbaname::localhost`.

Now that we know how to specify where the NameService is, we'll pass this information as an argument to the ORB when initializing it. The parameter name that specifies where to look for initial references is `-ORBInitRef`.

Knowing all of this, we are now ready to look at the code sample:

```
# standard modules
import sys

# omniORB
import CORBA, PortableServer

# initialize the ORB and POA
sys.argv.extend(("-ORBInitRef", "NameService=corbaname::localhost"))
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")
```

This looks just like before, but now we have the extra line that adds the `-ORBInitRef` argument that will be passed to the ORB initializer. This gets us an ORB that is able to find the naming service. Let's use this to actually grab the naming service.

2.6 Getting the naming service

Since we did our job and passed the ORB the location of the NameService, we can now ask the ORB to do it's job and give us a naming service object. We do this with the following lines:

```
# Import the stubs for the Naming service
import CosNaming

name_service_obj = orb.resolve_initial_references("NameService")
name_service_root = name_service_obj._narrow(CosNaming.NamingContext)
assert name_service_root is not None, "Failed to narrow to NamingContext."
```

Initially, we get a naming service object, then we use the built in CORBA operation `_narrow` to make sure this object specifies the right interface. This is very important, since we want to be sure all of the objects we get are what we expect. In this case we are expecting a `CosNaming.NamingContext` object, which is just a fancy way of saying we expect a standard naming service object.

If we don't get the right kind of object, `_narrow` will return None, which indicates failure. We check for this on the final line, to be sure we've actually got the right thing. Once we've got it, we're now ready to use it.

2.7 Placing the Biosequence server in the naming service

Our goal now is to name our object and stick it in the naming service. The very first thing to do is get our CORBA object. As previously, we activate our object by using `_this()`.

```
bioseq_obj = bioseq_server._this()
```

Now that we've got our object, we need to think about what we want to call it. You can think of the `name_service_root` as being like the root directory in UNIX (or the `C:/` directory in windows). We can put objects in this directory like in a filesystem. We create names, and then use these names to identify specific objects. You can place things into their own "directories" to further separate them, and this is probably a good choice as you get more advanced. In our case, we are going to just go for the simplest case and put our server into the naming service in the base directory. You can place things into their own "directories" to further separate them, and this is probably a good choice as you get more advanced.

We're going to name our object `MyServers.BiosequenceServer`. Now we need to create this name:

```
bioseq_name = [CosNaming.NameComponent("MyServers", "BiosequenceServer")]
```

Well, now we've got everything we need, our name and our object. It's time to put them into the naming service:

```
try:
    name_service_root.bind(bioseq_name, bioseq_obj)
    print "Bound the Biosequence to the naming service."
except CosNaming.NamingContext.AlreadyBound, msg:
    print "Biosequence already bound, rebinding the new object."
    name_service_root.rebind(bioseq_name, bioseq_obj)
```

What we are doing here is "binding" the object to the naming service. All this does is associate our name with the object.

Notice that we have two choices when binding. We first try to `bind` the object, but the problem with just doing this is that we might have already bound an object with the same name, perhaps in a previous run of the server. In this case, we would get an exception raised, which helps protect against accidentally overwriting objects.

To make sure we always bind without a problem, we catch the exception, if it is raised, and instead `rebind` the object. This does the same thing as `bind` but will overwrite the old object bound to `bioseq_name` with our new object.

Either way, we've now got our `bioseq_obj` in the naming service, identified by the name `bioseq_name`. We're almost there!

2.8 Starting up the server

Whew! Almost done, but... we don't want to forget some of the most important things. Remember from our previous example that we need a POA manager to manage our objects. Additionally, we need to actually start the server running so it is ready to receive requests. We do both of these things exactly as before:

```
# activate the POA manager
poa_manager = poa._get_the_POAManager()
poa_manager.activate()

# start the server to wait for requests
print "Server is running!"
orb.run()
```

And, that's it folks! We've got the server running. Very cool. Now you'll probably want to get some extra satisfaction by making a client that uses this server. No problem, read on to the next section.

2.9 A client using the naming service

Since we're using the naming service for this example, we'll need to modify our client from the client examples so that it gets the biosequence object from the naming service instead of from an IOR. Here's the complete code, which is very similar to before:

```
# standard modules
import sys

# omniORB
import CORBA

# CORBA naming service
import CosNaming
```

```
# 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()

# Now start up the ORB and get the server
sys.argv.extend(("-ORBInitRef", "NameService=corbaname::localhost"))
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)

# get the naming service
name_service_obj = orb.resolve_initial_references("NameService")
name_service_root = name_service_obj._narrow(CosNaming.NamingContext)
assert name_service_root is not None, "Failed to narrow to NamingContext."

# set up the name of the BioSequence server
bioseq_name = [CosNaming.NameComponent("MyServers", "BiosequenceServer")]

# get the server object from the naming service
try:
    biosequence_server = name_service_root.resolve(bioseq_name)
except CosNaming.NamingContext.NotFound, msg:
    print "Could not find Biosequence object"
    sys.exit(1)


# 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]
```

The only thing we haven't seen before is the use of `resolve`. This is how the naming service can be used to get an object – you just have to specify it's name (in this case `MyServers.BiosequenceServer`), and then call resolve on the name, which gives you back the server object. This is done instead of getting an IOR (which you can think of as just another name, albeit a long ugly one!) and resolving the object from it.

Now that we've got the server, we can run it and we'll get the same results as before:

```
[IntroServer]$ python biosequence_client.py HSERPG
Query: HSERPG
Sequence: agcttctgggcttccagacccagctactttgcggaactcagcaacccaggcatctctgagtctccg
[IntroServer]$ python biosequence_client.py X1234
The identifier X1234 does not exist in the database.
```

Congratulations! Now you've got a real-life server up and running. You're ready now to go out and get some serious work done using the powerful CORBA framework. Enjoy, and be sure to send me a cut of the extra money you make with this new CORBA knowledge!