BioCorba How-To

Brad Chapman, Jason Stajich

Last Update–23 January 01

Contents

1 Overview of Main Topics

1.1 What are Bioperl, Biojava and Biopython?

1.2 What is CORBA?

1.3 What is BioCorba?

1.4 What can BioCorba do for you?

2 Quick Start for the Lazy and Impatient

This section is designed to get you going quickly with Biocorba. It assumes you have some knowledge of how CORBA works and a familiarity with some programming languages (perl and python are used in this example). It gives a big overview of biocorba, and then tries to walk you through how Biocorba can be used by demonstrating a "real-life" example of it in use.

2.1 Fast overview of the big picture

The main goals of the Biocorba interface are language interoperability and position independence of code. This means that you can write one half of a progam in one language and another half in another language and have them talk with each other. In addition, one part of the program could be located on a local computer, while the other part could be located on some remote computer. The design of CORBA in general, and Biocorba in particular, thus allows a great deal of flexibility in designing and implementing clients and servers that deal with biological sequence data.

2.2 The interface

The interface used for communicating in Biocorba is defined in the file `biocorba.idl`. This interface is written in Interface Definition Language (IDL) which is the standard for defining objects in CORBA. The Biocorba IDL interface is designed to be simple so that it can be implemented in multiple langauges without support for more complex CORBA specifications, and has the following basic sequence objects:

- `AnonymousSeq` - A base class for all other sequence types. This just defines operations on a sequence, without any support for identifying the name of the sequence.
- `PrimarySeq` - A derived class of AnonymousSeq with added support for identifying the sequence through a name.
- `Seq` - A derived class of PrimarySeq which allows feature information of a sequence to be obtained.

As mentioned, the `Seq` class can hold features. There are represented by the `SeqFeature` interface. BioCorba features have support for fuzzy locations, and can completely represent the information in GenBank/EMBL feature tables.

In addition to the basic sequence and feature objects, there are also higher level interfaces which allow you to deal with these objects:

- Iterators (`PrimarySeqIterator`, `SeqFeatureIterator`) – provide the ability to move item by item over a list of objects.
- Vectors (`PrimarySeqVector`, `SeqFeatureVector`) – hold a list of objects which can accessed by individual elements.
- Databases (`PrimarySeqDB`,`SeqDB`) – hold a group of objects which can be access by name (ie. by accession number)
- `BioEnv` – This is general interface which can be used as a "factory" to obtain instances of other classes.

Used together, these objects provide a set of tools to examine and query biological data. The following example shows how they can be used.

2.3 Usage Scenario

To illustrate how useful the Biocorba interfaces can be, let's invent an example application to illustate. Melinda works in a lab in New York studying drought resistance in corn. Since other plant scientists worked hard to sequence the entire genome of *Arabidopsis thaliana* (`http://www.arabidopsis.org/`), Melinda decides she is going to try and see what kind of information she could use from Arabidopsis to help advance her own research.

Towards this goal, she does an Entrez search of the GenBank nucleotide database (`http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Nucleotide`) to get a bunch of sequences putatively related to drought resistance in Arabidopsis. Melinda collaborates on her research with another maize researcher in Iowa, named Jacob. Jacob is really keen on the idea of trying to learn something from the Arabidopsis data, so wants to try and work along with Melinda on this bioinformatic aspect of their project.

Jacob and Melinda would both like to be able to work off of the exact same data, so that they don't end up duplicating efforts. But this is complicated by a couple of problems: First, they are far away from each other and would have trouble sharing the exact same data. They could try and work off copies of the data, but this could be a problem if they start editing this data. Secondly, Melinda does all of her programming in perl (and wisely uses the Bioperl libraries to help her) which Jacob only can program in python (and similarly uses the Biopython libraries).

Oh my! Are they stuck? Thankfully, Jacob and Melinda read about biocorba, and discover that it has all of the solutions to their problems. Below we'll follow the steps they would do to start doing some useful work with the data.

2.4 A Perl server

XXX The Perl server section is not up to date! Need to wait until Bioperl-0.7 is out so that the bioperl-corba server can be updated to version 0.2 of the biocorba spec. So don't trust anything in this section right now.

Melinda first needs to set up a server using the data she got from GenBank, so that Jacob can get this data and work with it. So she gets to coding to write up a perl script to do just this.

The first step is to import the bioperl helper classes she'll need to make it easier to do this. She imports the Biocorba-perl module that will help her set up a sequence database, and the Bioperl module to help her parse her GenBank file:

```
# biocorba modules
use Bio::CorbaServer::SeqDB;

# bioperl modules
use Bio::Index::GenBank;
```

Since she is going to making the file available so that Jacob can query it like it was a database, she uses the Bio::Index::GenBank perl module, which parses a GenBank file and inputs it into a UNIX dbm database format. She has her file of sequences in GenBank format (`a_drought.gb`) and is going to create an index file `gb_index.dbm` that will contain the file in a database like format:

```
my $dir = `pwd`;
chomp($dir);

my $gb_file = "$dir/a_drought.gb";
my $index_file = "$dir/gb_index.dbm";

my $ind = Bio::Index::GenBank->new($index_file, 'WRITE');

$ind->make_index($gb_file);
# make sure we are done before doing ahead
$ind = undef;
```

Since we are going to use the CORBA communication protocol to make the data available, we now need to get CORBA started. Bioperl uses `CORBA::ORBit`, a set of perl bindings for the ORBit ORB (Object Request Broker) implementation, and the following code initializes the ORB, and sets up the Portable Object Adapter (POA), which will manage the entire server:

```
# CORBA::ORBit doesn't use a idl compiler, and instead imports the idl
# file containing the interfaces.
use CORBA::ORBit idl => [ 'biocorba.idl' ];

$orb = CORBA::ORB_init("orbit-local-orb");
$root_poa = $orb->resolve_initial_references("RootPOA");
```

Now that we are set up with our database file and initialized CORBA server, we are ready to combine them. First, we load up the index file that we've created earlier. This will server as the database.

```
my $gb_database = Bio::Index::GenBank->new($index_file);
```

Now, we want to make this database a CORBA database which implements the SeqDB interface. To do this, we create a SeqDB object using the Biocorba module:

```
$servant = Bio::CorbaServer::SeqDB->new($root_poa, 'gb_db', $gb_database);
```

Now that we've got a servant item, we need to register this with the POA that controls all of the objects. Once we do this, the POA will return the ID of the newly created object. This is an IOR (interoperable object reference) which other CORBA implementations can recognize, even those in different languages. So now we do the registration steps and get the object reference:

```
my $servant_id = $root_poa->activate_object($servant);
$object_id = $root_poa->id_to_reference($servant_id);
```

The problem with the object form of the IOR is that it is difficult to be passed to other computers. A much easier way to do it is to get a string representation of the object, which can be accomplished simply by:

```
my $string_ior = $orb->object_to_string($object_id);
```

Now that Melinda has got this string, she is ready to send it off to Jacob to he can use it. Since it is a string, Melinda could paste it into an e-mail message and send it to him, or put it on the web where he could get it, or generally do any creative thing she could think of to send a string. In this case, we'll assume that Melinda's computer has anonymous ftp set up, and we'll put the IOR in a file in anonymous ftp:

```
$ior_file = "/var/ftp/pub/my_gb_db.ior";
open (OUT, ">$ior_file") || die "Cannot open file for ior: $!";
print OUT "$string_ior";
close OUT;
```

Now we just need to start the ORB running to wait for Jacob to invoke requests on it:

```
$root_poa->_get_the_POAManager->activate;
$orb->run;
```

So we are all set. That little bit of code set up a full fledged database server that Jacob can now query and extract information from. Now that Melinda is all done, she drops Jacob an e-mail to let him know that her server is ready, and now it is his turn to get started.

2.5 A Python client

XXX This section is up to date with the current spec and biopython-corba implementation, but needs to be finished once we can make a perl server.

Once Jorge gets the exciting e-mail that the server is set up, he decides he would just like to browse through the data that Melinda has. This also serves to illustrate the interfaces that are available in Biocorba.

To begin, he first imports the necessary python classes to set up his client. Since he knows Melinda set up a SeqDB object, he imports this client class:

```
>>> from BioCorba.Client.Seqcore.CorbaSeq import CorbaSeqDB
```

Additionally, since he wants to set up a connection to a perl client, he imports the class which will be used to make the connection:

```
>>> from BioCorba.Client.BiocorbaConnect import PerlCorbaClient
```

Now that we has imported everything he needs, he now sets up an object that can be used to retrieve and resolve references to remove SeqDB objects.

```
>>> server_retriever = PerlCorbaClient(CorbaSeqDB)
```

Melinda also sent him the ftp site containg the IOR of her sequence object. He uses this information to get the object reference and establish a connection:

```
>>> IOR_URL = "ftp://24.9.210.117/pub/my_gb_db.ior"
>>> gb_server = server_retriever.from_url_ior(IOR_URL)
```

This set of commands returns a connection with her server. Now looking through the SeqDB interface, we can make calls to query about what is available in the database. First, we might like to know all of the sequence ids, so we can use these to get actual sequences:

XXX Fill the following in once the Bioperl server is set up

```
>>> print gb_server.accession_numbers()
```

Let's just get a sequence object for an id from the database. The first argument to pass is the sequence to retrieve, and the second is the version of the sequence. Passing a zero indicates you want the most recent version available.

```
>>> my_seq = gb_server.get_Seq('U73781', 0)
```

You can now query information about the sequence:

```
>>> my_seq.seq()
'CAAGCAAAGCAACACTTTCACTAATCCTCTAATGGCAGAAGACAAGATCTTAAAGAAGACTCCGGCG
GCGAAGAAGCCGCGAAAACCGAAAACCACCACTCATCCTCCATACTTTCAGATGATAAAAGAGGCTTT
GATGGTCCTGAAAGAGAAGAACGGATCAAGCCCTTATGCTATAGCTAAGAAGATAGAGGAGAAACACA
AGTCTTTACTTCCAGAGAGTTTCCGTAAAACACTTTCTCTACAGCTTAAAAACTCTGTTGCTAAAGGT
AAGCTCGTGAAGATCAGAGCCTCTTACAAGCTCTCAGATACCACCAAGATGATAACGAGGCAGCAGGA
```

```
CAAGAAGAATAAGAAGAATATGAAGCAAGAAGATAAAGAGATCACAAAGAGGACTAGGTCTTCTTCGA
CAAGGCCTAAGAAGACTGTGTCTGTGAACAAACAAGAAAAGAAGAGGAAAGTGAAGAAGGCGAGACAG
CCTAAGTCTATCAAATCTTCAGTTGGTAAGAAGAAGGCCATGAAAGCTTCCGCTGCTTGATCACTGAG
GAGGAGGAGAAGAAGATGGAACTGTGTGTGAATATGAAGAATAAGGTTTTTTGTTGTCTCTGTAAATG
GATGATGACTTTGTTTTGGTTTGGTGTGGTCTCTTTTGGGTGTAAGAGTAAAGAGTCTTCGGATGTAT
AACAGTACTACCATGTAATATACTCTAAATAGATAGATTTTTACTTGGGTTGATAAAAAAAAAAAAAA
AAAAAAA'
>>> my_seq.type()
'DNA'
>>> my_seq.display_id()
'ATU73781'
```

You can also look at features associated with the sequence:

```
>>> feature_vector = my_seq.all_SeqFeatures(1)
```

In this way, Jacob can similarly explore any of the other data in the database.

Although scanning through the data is interesting, the major advantage of serving out the data using Biocorba is that it is in a pre-parsed format, ready to extract the information of interest. Since Jacob has ready access to pre-parsed data, he could readily do tasks like search for features matching some item of iterest, or search for particular sequence motifs in the data. Hopefully, this quick and dirty example gave you an idea of some of the things that can done with the Biocorba interface. Further sections will go into more depth on the overall design on the Biocorba interface and into how to use particular language implementation.

3 Overview of BioCorba design

3.1 The design of the BioCorba idl

Would be nice to have a UML diagram of the classes or something.

3.2 The power of BioCorba – language interoperability

Nice diagram of how two different languages could do something using BioCorba.

4 Using BioCorba from Bioperl

Bioperl stuff–Jason

5 Using BioCorba from Biojava

Biojava stuff–someone from Biojava

6 Using Biocorba from Biopython

The Biopython project also has a biocorba implementation. This implementation is written in the object oriented, interpreted language python (http://www.python.org), and is meant to be interoperable with the biopython libraries (http://www.biopython.org).

The interoperability of CORBA and the standard Biocorba interface allows you to write servers or clients in python, and have them be fully functional with components written in perl or java. This part of the manual describes the python interface to biocorba and highlights the strengths of python and biopython for dealing with sequence information.

6.1 CORBA and python

6.1.1 The standard mapping

The mapping of IDL to python has been standardized, and documentation describing this official mapping is available (http://www.omg.org/cgi-bin/doc?ptc/00-04-08). Due to recent acceptance of this standard, the degree to which different python ORB implementations follow it vary, unfortunately. The biopython-corba package attempts to standardize "non-compliant" ORB implementations so that they can be used through the same interface. This makes it possible to use any supported ORB with the biopython-corba interfaces without needed to change any code.

6.1.2 ORB implementations usable from python

You can currently access CORBA from python through four different ORB implementations:

1. omniORB - Python bindings are available from http://www.uk.research.att.com/omniORB/omniORBpy/ and omniORB itself is available from http://www.uk.research.att.com/omniORB/index.html. Of the available python choices, only omniORB currently follows the mapping standard to the letter.
2. Fnorb - This is an ORB implementation written almost entirely in python, and is available from http://www.fnorb.org/.
3. ILU - The Inter-Language Unification system (ILU) supports python (as well as a number of other languages, and is available from: ftp://ftp.parc.xerox.com/pub/ilu/ilu.html
4. ORBit - The ORBit ORB implementation (http://www.labs.redhat.com/orbit/), utilized by the Gnome project, has a set of bindings available from the orbit-python page at http://sourceforge.net/project/?group_id=3561.

6.2 CORBA and Biopython

6.2.1 Obtaining and Installing the necessary packages

The Biopython interfaces to Biocorba are available in the package `biopython-corba` which is distributed separately from the main Biopython libraries. This package is currently available via anonymous CVS, with instructions at http://cvs.biopython.org/, and the latest releases are available at http://www.biopython.org/Download/.

Installation and dependencies for running biopython-corba are described in the `README` file available in the biopython-corba distribution. If anything in this file is unclear or missing, please don't hesitate to let us know so we can update this file to make it better.

A very important thing when installing is to remember to edit the file `BioCorba/biocorbaconfig.py`. Most importantly, this file contains the option for which CORBA ORB implementation you are using.

6.2.2 Supported ORB Implmentations

Currently, both omniORB and Fnorb are usable for Biocorba interoperability. Both ORB implementations work through the same interface thanks to a small set of derived classes which make Fnorb act like it is compliant with the standard python mapping. Having a standard interface means that code written using the biopython-corba can be used for multiple ORB implementations. This would allow you to develop in Fnorb and deploy in omniORB, if you chose to do this. In addition, it helps to buffer your code from small changes in the mapping, and hopefully makes CORBA easier to use.

XXX Right now I'm having trouble with the Fnorb implementation – it only works about 95% of the time. I need to work more on this.

Support for other ORB implementations may be forthcoming if there is sufficient demand. Currently, the ILU stub generators have some problems with the biocorba IDL. Both of the ORBit python bindings are also at an early stage in development, and aren't really ready for a larger scale project like biocorba. We are following development on both of these fronts and will try to support these ORBs more in the future. Please feel free to bug us if you would like this support sooner.

6.3 General overview of what is going on.

The following few sections give the specific details of how to implement clients and servers for the Biocorba interface using python. This section just gives a quick overview of what you got when you installed `biopython-corba` and where you should look for information.

All of the python classes dealing with Biocorba get installed under the namespace `BioCorba`. For a traditional installation, this will be located in your `site-packages` directory (it is normally somewhere like `lib/pythonX.X/site-packages`). If you look inside here, you will find the Biopython directory, which is where all of the Biopython related code dealing with Biocorba is found.

The most important thing to know about biopython-corba is that it is designed to isolate you from dealing with CORBA as much as possible. This makes it possible to utilize the power of CORBA without knowing that much about it, and also makes it possible to write code that can be used with a number of ORB implementations. Just as the regular Biopython library is designed to make dealing with biological data-files and sequences easier, Biopython-corba is designed to make dealing with CORBA easier and more general. Enjoy your tour throught the world of Biocorba in python!

6.4 An example of making a useful Biocorba client in python

The idea behind the biopython client code for Biocorba is that you should be able to access Biocorba objects through the same interfaces that is used for accessing regular biopython objects. The major difference, of course, if that queries about the nature of an object occur through a (possibly remote) Biocorba interface, and not to local Biopython objects. The second difference is that the Biocorba IDL interface does not provide support for changing remote objects, only for querying information about them.

Biopython provides support for a 'Bio' interface, which accomplishes the above goals of making Biocorba code work similar to Biopython code. It also provides a interface which is identical to that defined in the `biocorba.idl` file, so that you can program against this interface if you choose to. We'll first delve into the Biocorba interface to give an idea of what is going on "behind the scenes" and then we'll show how to use the 'Bio' interface for maximum interchangability with Biopython code.

6.4.1 Writing a client with the Biocorba interface

The major reference guide for using the Biocorba interface is the `biocorba.idl` IDL file describing the interface. This file defines all the objects and methods on them, all of which are exactly mapped to corresponding classes and functions in python. In biopython-corba, these are all implemented in the module `BioCorba.Client.Seqcore`. If you look into this module, you will see files implementing each of the interfaces defined in the IDL file. They are prefixed with `Corba` to help me (oh yeah, also you) remember that these implement the straight Corba interface. So we've got files in there like `CorbaBioEnv.py`, `CorbaSeqDB.py`, etc.

When you start implementing a client, the first thing to determine is what type of interface the server you are using will be implementing. A Biocorba server object will implement one interface. From this "top-level" interface, you'll then be able to retrieve other objects and access them. Once you know what type of object you'll recieving, then you'll know what type of class you'll need to call. For now, let's pretend we'll be getting a `BioEnv` interface, which means we'll want to deal with `CorbaBioEnv.py`. Nice and simple!

The second thing to figure out is how you are going to get the object reference for the server. An object reference in CORBA is referred to an an Interoperable Object Reference (IOR). You'll need to get an instance of an IOR to be able to start working with the client. The easiest and most straightforward way to get an IOR for a server is to make the server IOR into a string and then get this. A typical IOR string would look something like:

```
IOR:015f69782300000049444c3a6f72672f42696f636f7262612f536571636f726
52f53657144423a312e30005f02000000caaedfba5000000001010000280000002f
746d702f6f726269742d726f6f742f6f72622d313737333534373438323832383032313
337343737000000000018000000000000019efbd6dc8d725a401000000d1669f08
ed5cc3db0000000034000000010100000b0000003137322e31362e302e310072820
6742f18000000000000019efbd6dc8d725a401000000d1669f08ed5cc3db
```

So now you know what to look for; a big mess! This string object can be converted back into a object reference so that you can access the `BioEnv` server, so the IOR is so messy because it needs to hold all of the information about where the server is located and how to find it. Biopython currently supports three ways to convert the object reference string into a live object, as shown in Table 1

Location of the IOR string	Method to Call
Just the string itself	`from_string_ior(your_string)`
In a file	`from_file_ior(file_with_the_IOR)`
On a remote URL	`from_url_ior(url_location)`

Table 1: Methods to convert a string IOR into a live object.

So in our mini-example, let's assume we have our `BioEnv` object stored in the local file `/tmp/my_server.ior`. Then to get our live BioEnv object, we would need to do the following:

```
from BioCorba.Client.Seqcore.CorbaBioEnv import CorbaBioEnv
from BioCorba.Client.BiocorbaConnect import PerlCorbaClient

bioenv_connector = PerlCorbaClient(CorbaBioEnv)
bioenv_server = bioenv_connector.from_file_ior("/tmp/my_server.ior")
```

Now you've got a corba object and are ready to go! Basically, we've just done two things to resolve this server reference:

1. Create an object that resolves BioEnv objects implemented by Perl based servers. This is done in the line:

   ```
   bioenv_connector = PerlCorbaClient(CorbaBioEnv)
   ```

   There are two things to notice in this line. The first is that we specifically specify we are making a client that connects to a perl server, by using the class PerlCorbaClient. The reason for this is that sometimes there are interoperability issues that we can work around by knowing the type of the server. For instance, omniORBpy doesn't normally get along well with the CORBA::ORBit server used in Bioperl. However, if we know the server is in perl and we are using omniORB, we can get around this by passing a flag to omniORB. The BiocorbaConnect module also contains PythonCorbaClient and JavaCorbaClients, for connecting to python and java servers, respectively, and a GenericCorbaClient, in case you aren't worried about interoperability with the server.

   The second thing to notice about the line is that we pass the constructor the type of objects that we want to recieve instances of. This requires that you know the interface that the server is implementing, which is definately not an unreasonable requirement, since you will need to know this later to know what methods to call!

2. Use the object we created to actually retrieve the server. As mentioned previously, the server can be retrieved via an IOR string either passed in, located in a file, or located at some URL.

Those four lines of code above were all you needed to get a Biocorba client running. Congratulations! Now that you've got your object primed and ready, you can begin to make calls on it to do something interesting. Since we have a `BioEnv` object, we first take a look at the `biocorba.idl` file to see what we can do with this type of object:

```
interface BioEnv
{
  // Return a PrimarySeqVector object generated by this BioEnv
  // factory from the sequences in the local file specified in
```

```
        // the given format. The format may be an empty string, in
        // which case the BioEnv factory guesses the format, but
        // throws an exception if it cannot determine it.
        PrimarySeqIterator get_PrimarySeqIterator_from_file(in string format,
                                                            in string filename)
          raises (UnableToProcess);

        // Return a PrimarySeq object generated by this BioEnv factory
        // from the sequence in the local file specified in the given
        // format.
        PrimarySeq get_PrimarySeq_from_file(in string format,
                                            in string filename)
          raises (UnableToProcess);

        // Return a Seq object generated by this BioEnv factory from
        // the sequence in the local file specified in the given
        // format.
        Seq get_Seq_from_file(in string format,
                              in string filename)
          raises (UnableToProcess);

        // Return a list of the names of the databases available from
        // this BioEnv server.
        StringList get_SeqDB_names();

        // Return a list of the available versions of a database with
        // the given name.
        LongList get_SeqDB_versions (in string name)
          raises (DoesNotExist);

        // Return a SeqDB reference for the database with the given
        // name and version. The 'version' may be '0', in which case
        // the latest version of the database is returned.
        SeqDB get_SeqDB_by_name(in string name,
                                in long version)
          raises (DoesNotExist);
      };
```

IDL files have a C++ like syntax, so they can take a bit o' getting used to if you spend a lot of time looking at beautiful python code. The IDL to python mapping for something this is pretty straightforward: interface objects map to python classes and functions map to functions in the class. So the `BioEnv` object that you got above is an instance of a class that looks basically like:

```
class CorbaBioEnv:
    def __init__(self, corba_object):
        spam

    def get_PrimarySeqIterator_from_file(self, format, filename):
        spam

    def get_PrimarySeq_from_file(self, format, filename):
        spam

    def get_Seq_from_file(self, format, filename):
```

```
        spam

    def get_SeqDB_names(self):
        spam

    def get_SeqDB_versions(self, name):
        spam

    def get_SeqDB_by_name(self, db_name, version = 0):
        spam
```

So, the correspondance between a interface defined in the IDL and the python implemented Corba class for it is very good, so you can really understand the methods you can call by reading the IDL file.

So lets use our `BioEnv` object to get a PrimarySeqIterator and iterate through a GenBank file. To do this, we would need to write code like the following:

```
import os

GB_FILE = os.path.join(os.getcwd(), "a_drought.gb")
pseq_iterator = bioenv_server.get_PrimarySeqIterator_from_file('GenBank', GB_FILE)
```

Now that we've got our PrimarySeqIterator, let's start getting down and dirty and actually doing some interesting sequence manipulation. As before we glance at the IDL file to see what methods to call:

```
interface PrimarySeqIterator : GNOME::Unknown
{
  // Return a reference to the next PrimarySeq object in this
  // PrimarySeqIterator.
  PrimarySeq next()
    raises (EndOfStream);

  // Return whether this PrimarySeqIterator can return another
  // PrimarySeq objevt.
  boolean has_more();
};
```

So we've got a simple iterator object – we can keep calling `next()` to get PrimarySeq interface objects until `has_more()` becomes false.

With this object, we are set up to do some real work. Let's do something really simple, search through all of the sequences in the file for a particular sub-sequence. This could be useful if you were looking for primer binding sites or for a particular sequence motif. To do this, we'll have to be able to get information about the PrimarySeq objects that our iterator is returning to us. Once again, we look at the `biocorba.idl` file to see what methods to call.

First we check out the PrimarySeq interface:

```
interface PrimarySeq : AnonymousSeq
{
  // Return the ID to be used for display purposes.
  string display_id();

  // Return the ID to used as a unique ID for this sequence,
  // e.g. the accession number or the byte position/file munged
  // into a string.
  string primary_id();
```

```
    // Return the unique ID for the PrimarySeq in its biological
    // database.
    string accession_number();

    // Return the (unstable) version number for the sequence. This
    // is 0 for PrimarySeq objects that do not have a version
    // number.
    long version();
  };
```

Then, we look at the AnonymousSeq interface. The reason for this is that the `PrimarySeq` object that we are getting inherits from `AnonymousSeq` (this is represented in the IDL by `PrimarySeq : AnonymousSeq`) so we can call functions from both interfaces.

```
  interface AnonymousSeq : GNOME::Unknown
  {
    // Return the type of the biological sequence, e.g. PROTEIN,
    // RNA, DNA, etc. as defined using the type codes declared in
    // the SeqType interface.
    short type();

    // Return whether the biological sequence is circular or
    // linear.
    boolean is_circular();

    // Return the length of the biological sequence.
    long length();

    // Return the biological sequence as a string.
    string seq()
      raises (RequestTooLarge);

    // Return a sub-string of the biological sequence. The start
    // and end values are in biological coordinates, ie, 1-2 are
    // the first two bases.
    string subseq(in long start,
                  in long end)
      raises (OutOfRange,
              RequestTooLarge);
  };
```

Glancing over the interfaces indicates that we can get a bunch of interesting features of each entry, such as the sequence and id. Let's use this to do our search:

```
import string

SEARCH_STRING = "CAGAATG"

print "Searching for %s..." % SEARCH_STRING

while pseq_iterator.has_more():

    pseq = pseq_iterator.next()
```

```
    location = string.find(string.upper(pseq.seq()), SEARCH_STRING)
    if location != -1:
        print 'Id:', pseq.display_id()
```

So our simple search is all ready to go! Running it with a biopython-corba implemented BioEnv server gives:

```
$ python gb_bioenv.py
Searching for CAGAATG...
Id: AB04H01 AB Arabidopsis thaliana cDNA 5' similar to drought-induced protein, mRNA sequence.
Id: AA09A03 AA Arabidopsis thaliana cDNA 5' similar to drought-induced protein di19, mRNA sequence.
```

So we've gone all of the way from getting an object to doing something useful with it. The Biocorba interface is very powerful, and contains a number of interfaces to access biological data. However, there are a couple of disadvantages to using it. The first is that your code isn't very "pythonic" since it is derived straight from the IDL. In addition, code written against the Biocorba interface can't be used interchangably with normal Biopython code. To help deal with these problems, a 'Bio' interface also exists which allows you to deal with Biocorba objects using similar code to that used to deal with normal Biopython objects.

6.4.2 Writing a client with the Biopython 'Bio' interface

As mentioned, the 'Bio' interface is designed to work relatively interchangably with the normal Biopython code. So a good reference to look at in addition to this is the Biopython documentation. The biopython-corba implementations of these interfaces can be found in the module `BioCorba.Bio`. If you peek in there, you can see it resembles the Biopython 'Bio' module.

One major advantage of Biocorba for Biopython is that we can use the implemented code in say, Bioperl, to provide functionality for designing our own interfaces. This can allow interface design and testing without having to worry about underlying implementation details (which can be filled in later). But, let's focus now on what you can do in the current 'Bio' implementation.

The 'Bio' implementation provides a thin wrapper around the straight Corba interface described previously, so the access methods are very similar to those previously described. As before, the first thing to look at is what kind of server object you will recieve. The server object you get will determine what Bio object you can start with. Table 2 details how the Biocorba interfaces match up with Bio objects.

'Bio' object	Biocorba interface object
Bio.Seq ⇒ Seq	AnonymousSeq
Bio.SeqRecord ⇒ SeqRecord	PrimarySeq
Bio.SeqFeature ⇒ SeqFeature	SeqFeature
Bio.Fasta ⇒ Dictionary	PrimarySeqDB, SeqDB
Bio.Fasta ⇒ Iterator	PrimarySeqIterator
Bio.Fasta	BioEnv (for loading dictionaries and iterators)
Bio.GenBank ⇒ Dictionary	SeqDB
Bio.GenBank	BioEnv (for loading dictionaries)

Table 2: Mapping between the Bio-like interface and Biocorba objects.

So, for example, to load up a FASTA iterator from a file of our choosing we would need to have an IOR corresponding to an implemented BioEnv object. So to load up a FastaIterator, we would first need to resolve a reference to a BioEnv server:

```
from BioCorba.Client.BiocorbaConnect import GenericCorbaClient
from BioCorba.Client.Seqcore.CorbaBioEnv import CorbaBioEnv

server_retriever = GenericCorbaClient(CorbaBioEnv)
bioenv_server = server_retriever.from_file_ior("/tmp/my_server.ior")
```

This is done in the exact same way as before. Now that we've got a BioEnv server, we can use the utilities on this server to create a Fasta iterator for a file of our choice. To do this, we use the BioCorba.Bio.Fasta specific function `iterator_from_bioenv`, which uses a BioEnv object to load up a file:

```
from BioCorba.Bio import Fasta
import os

FASTA_FILE = os.path.join(os.getcwd(), "a_drought.fasta")

seq_parser = Fasta.SequenceParser()
fasta_iterator = Fasta.iterator_from_bioenv(bioenv_server, FASTA_FILE,
                                            seq_parser)
```

What we are doing here is analagous to creating a Fasta Iterator using the standard Biopython code. To create the Iterator in Biopython, we would have done the following:

```
from Bio import Fasta
import os

FASTA_FILE = os.path.join(os.getcwd(), "a_drought.fasta")
fasta_handle = open(FASTA_FILE, 'r')

seq_parser = Fasta.SequenceParser()
fasta_iterator = Fasta.Iterator(fasta_handle, seq_parser)
```

Both methods are accomplishing the same goal – loading an Iterator that will return Sequence objects from a Fasta file. The specifics are different since we are creating the object from two very different sources, but the ideas are similar. However, once the objects are created, they behave identically. We can write the following code which will search for a particular string, and it will work with both `fasta_iterator` objects!

```
import string

SEARCH_STRING = "CAGAATG"

print "Searching for %s..." % SEARCH_STRING

while 1:
    seq_record = fasta_iterator.next()

    if seq_record is None:
        break

    my_seq = seq_record.seq
    location = string.find(string.upper(my_seq.data), SEARCH_STRING)
    if location != -1:
        print 'Id:', seq_record.description
```

This is a big advantage if you have standard Biopython code that you want to rewrite to use through BioCorba. The amount of code you have to change will be minimal – you should only have to change how you load your objects. Additionally, the raw CORBA interfaces can often be very different from how you would do things in python, so the BioCorba Bio interfaces should be easier to deal with if you feel most comfortable thinking in python (who doesn't!).

The Bio interface is also designed to be maximally interoperable with Biopython. This means that it will start to return Biopython objects as soon as non-corba objects start being returned by function calls. For example, performing a slice on a Biocorba sequence will give you a Biopython sequence object, just as performing a slice on a Biopython sequence object gives you as Biopython sequence object:

```
>>> print my_seq
<BioCorba.Bio.Seq.Seq instance at 0x1028e034>
>>> print my_seq._aseq_obj.type()
DNA
>>> seq_slice = my_seq[1:30]
>>> print seq_slice
Seq('ACTCATAAGCGGGTTTTATATAATTCATT', DNAAlphabet())
```

So, as they are fond of saying in Perl-land: There is more than one way to do it. This is designed to make things easier to learn, and hopefully not more confusing. If you would like to learn one set of coding standards for both Biopython and Biocorba, then the 'Bio' interface is for you. If you want to be close to CORBA to have as much power and control as possible, then the Biocorba only interface is for you. Either way, the client interfaces try to remove the burden of using CORBA, and allow you to concentrate on doing interesting biological research.

Hopefully this introduction gave you a good idea of how to access Biocorba objects using the python clients. Feedback on the client interfaces is especially welcome, since this will help to make them more intuitive to use.

6.5 Creating a python Biocorba server

Similar to the python Biocorba clients, the servers are designed to be easy to setup and run without extensive CORBA knowledge. As you will see, most of the work is in knowing which Biocorba interface to use in serving out a particular biopython object.

So, the first step in creating a server is to initialize a local Biopython object that implements the functionality of the server. This Biopython object should match up with the Biocorba interface that you want to support. Table 3 illustrates which Biopython objects match up with which Biocorba interfaces:

Biocorba interface	Biopython objects required
BioEnv	No object needed (just an interface to get other objects)
AnonymousSeq	Bio.Seq ⇒ Seq
PrimarySeq	Bio.SeqRecord ⇒ SeqRecord
Seq	Bio.SeqRecord ⇒ SeqRecord
SeqFeature	Bio.SeqFeature ⇒ SeqFeature
PrimarySeqIterator	Any Biopython Iterator that returns Bio.Seq ⇒ Seq objects
PrimarySeqDB	Any Biopython Dictionary that returns Bio.Seq ⇒ Seq objects
SeqDB	Any Biopython Dictionary that returns Bio.SeqRecord ⇒ SeqRecord objects
PrimarySeqVector	Python list of Bio.Seq ⇒ Seq objects
SeqVector	Python list of Bio.SeqRecord ⇒ SeqRecord objects
SeqFeatureVector	Python list of Bio.SeqFeature ⇒ SeqFeature objects

Table 3: Relationship between Biocorba interfaces and the Biopython objects that provide the functionality

6.5.1 Setting up an Iterator Server

So to take an example, let's look at implementing a PrimarySeqIterator server to make the famous Arabidopsis drought sequence data available. Looking at the table above, we need to first initialize a Fasta Iterator object to iterate over the Fasta file. We can do this with the following Biopython code:

```
from Bio import Fasta
from Bio.Alphabet import IUPAC

import os
FASTA_FILE = os.path.join(os.getcwd(), "a_drought.fasta")
```

```
fasta_handle = open(FASTA_FILE, 'r')

seq_parser = Fasta.SequenceParser(alphabet = IUPAC.unambiguous_dna)
fasta_iterator = Fasta.Iterator(fasta_handle, seq_parser)
```

Now that we've got the local object, the next step is to make this into a Biocorba object so that it can be available through the Biocorba interface. To do this, we first need to import the appropriate server object to create. All of the server interfaces are implemented in the package `BioCorba.Server.Seqcore`.

Peeking into the Server directory, you can see that the modules are divided based on the type of information you want to make available. All of the basic objects, such as PrimarySeq and SeqFeature, are in modules prefixed with Corba (ie. `CorbaPrimary.py`, `CorbaSeqFeature.py` ...). The derived objects, such as Iterators, Vectors and Databases, are located in their own modules. This setup hopefully makes it easy to find the type of object you want to serve out.

So we will need to import the appropriate interface that we want to implement. For our example, we are implementing a PrimarySeqIterator, so we need to import `CorbaPrimarySeqIterator` from the Iterator module:

```
from BioCorba.Server.Seqcore.Iterator import CorbaPrimarySeqIterator
```

Once we get an understanding of what we need to import, it is time to initialize the server and make it available. For each Biocorba server object, the following convenience functions are available to make it easier to set up the server:

1. `get_string_ior()` – Retrieve the string representation of the server's IOR, so that it can be made available to a client.

2. `string_ior_to_file(file)` – Write the string representation of the IOR to the specified file. This is useful if you are running BioCorba processes on the same machine.

3. `run()` – This will start the server running to accept requests, once you have everything all set up. This call will block and keep the server process running, so if you want to work in the same shell as the server, you need to run it in the background.

These functions provide what you need to get a server up and running, and to make it available to clients.

Now, let's set up our Biocorba server. This can be done with the following line of code:

```
pseq_server = CorbaPrimarySeqIterator(fasta_iterator)
```

Now that the server is ready, we need to make a reference to it available so clients can find it. To do this, we'll write the string version of the IOR to the file `/tmp/my_server.ior`:

```
pseq_server.string_ior_to_file("/tmp/my_server.ior")
```

The final thing we need to do is to start the server running. During the run, the server will just sit and wait for client requests to come in, and handle them as they come. So you just created your very own full fledged server. To get it going, you just need the following line of code:

```
pseq_server.run()
```

Now we are running and set! You can now create clients in any language to get the information from your server!

6.5.2 Setting up a Database server

An iterator server is nice, but it is rather limited in its functionality. For instance, once a client iterates through the entire list of items, there is no way to reset the server to start again. To get more advanced functionality like this, you should use the Database family of interfaces.

As mentioned in Table 3, Database objects require Biopython Dictionary objects. Many biopython parsers provide this Dictionary class, so any format which Biopython handles and provides a Dictionary class for can be used to create a server. Additionally, you can of course create your own classes which fulfill the same interface. The Dictionary object you pass to create a sequence database must define the following functions:

- `__getitem__` – Allow retrieval of SeqRecord objects by their names.
- `keys()` – Return a listing of all keys which can be used to retrieve objects.

So any object which returns SeqRecord objects and implements these two functions can be used to create a Database.

Now we'll show how to make a SeqDB that serves out a GenBank file, using the strandard Biopython and biopython-corba libraries.

First, we need to create an index file for the our GenBank file, so that individual entries in the file can be retrieved. We do this using the function `index_file` defined in the GenBank module:

```
from Bio import GenBank

import os
gb_file = os.path.join(os.getcwd(), "a_drought.gb")
index_file = os.path.join(os.getcwd(), "a_drought.idx")

GenBank.index_file(gb_file, index_file)
```

By default, this indexes the file based on the accession number, You can pass in, as a third argument, a function that will take a GenBank Record and return the item to index by.

Now that we've got the file ready, we need to create our GenBank Dictionary:

```
seqfeat_parser = GenBank.FeatureParser()
gb_dict = GenBank.Dictionary(index_file, seqfeat_parser)
```

Once we have the dictionary, we are ready to set up the SeqDB. We create it by passing the Dictionary we just created:

```
from BioCorba.Server.Seqcore.Database import CorbaSeqDB

db_server = CorbaSeqDB(gb_dict)
```

Now we've got the server primed and ready to go. As before, we now write the IOR of the server to a file so a client can get it, and start the server running:

```
db_server.string_ior_to_file("/tmp/my_server.ior")
db_server.run()
```

And voila, we've created a Database server! This is a quick and convenient way to make data in a file available over a network.

6.5.3 Setting up a super BioEnv server

In the last sections we set up a simple server that makes available an object of one type. This is great for many cases, but Biocorba also provides a more advanced interface which can be used to provide references to several different kinds of Biocorba objects. This interface is the BioEnv interface, and this section describes how to work with this powerful interface.

Basically, BioEnv provides facilities for two things:

1. Creating sequence and iterator objects from files in the filesystem. This requires that the client and server are on the same machine.

2. Holding multiple SeqDB objects which can be retrieved by name and version.

When you create a BioEnv server, the ability to create objects from files is provided "for free." That is, you don't have to do anything special to get it to work – all of the coding for this is integrated into the BioEnv object itself.

To make SeqDBs available to the client, you'll need to register Dictionary-like objects with the BioEnv server. For more information about the format of these Dictionary-like items see Section 6.5.2 for more information on the SeqDB objects. Each dictionary object should be associated with a unique name and a version, since the BioEnv interface allows the databases to be retrieved by a name and version. Once you've got your Dictionary objects, you can register them in one of two ways:

1. They can be passed into the initialization function when creating a BioEnv server. In this case, the Dictionary-like objects should be passed in as a python dictionary. The keys of the dictionary should be tuples consisting of `(database_name, database_version)`, and the values are the sequence dictionaries themselves.

2. Using the function `add_seq_db`. In this case, after you create the SeqDB and BioEnv server, you register the Dictionary with the server using a function call like:

   ```
   bioenv_server.add_seq_db(db_name, db_version, the_seq_dict)
   ```

Either way you choose to do it, the BioEnv server will now make these sequence databases available to any clients.

The following code shows how you would create a GenBank sequence dictionary, register it with a BioEnv server, and then make this server available:

```
# create the Dictionary
from Bio import GenBank

gb_file = 'my_file.gb'
index_file = 'my_file.idx'

GenBank.index_file(gb_file, index_file)

seqfeat_parser = GenBank.FeatureParser()
gb_dict = GenBank.Dictionary(index_file, seqfeat_parser)

# create a python dictionary to reference the GenBank Dictionary
my_dbs = {('gb_database', 1) : gb_dict}

# now create the server and start it up
from BioCorba.Server.Seqcore import CorbaBioEnv

bio_env_server = CorbaBioEnv.CorbaBioEnv(my_dbs)
bio_env_server.string_ior_to_file('my_file.ior')
bio_env_server.run()
```

Now that you understand BioEnv servers, you're ready to create real-life servers which can make lots of data available to clients. The BioCorba interfaces have been used to make the entire EMBL database available through BioCorba, so there is lots of power available to you.

This concludes your tour through the wonderful world of python and Biocorba. I hope you had a fun time and feel inspired to start utilizing all that Biocorba has to offer. Suggestions to make this documentation more helpful are greatly appreciated.

7 Advanced usage topics

7.1 Interoperability with other life science idls

7.1.1 Biomolecular Sequence Analysis (BSA)

7.1.2 CORBA stuff at EBI

7.2 Memory management

7.3 Security issues

8 Where to go from here

There are lots of exciting things you could do if you are really pumped about Biocorba.