CORBA Objects in Python

Jason Tackaberry
(tack@linux.com)

April, 2000
Algoma University College
Supervised by George Townsend

Table of Contents

1. Introduction

1.1. Trends in Distributed Objects

Historically, large applications would typically be written as one, monolithic program. In the 80s, a shift in programming practices toward modular design occurred. Using modular techniques, the application is divided into a series of self−contained modules that are linked together to form the complete application. This practice was widely regarded as a step forward in a more sound design and increased maintainability.

Beginning in the early 90s, this approach was extended to what is known as component−based design. In a component model, a common framework is provided for each component. This framework defines a standard set of operations used to facilitate interoperability between components. Once a framework is available, independent software vendors can design components that are guaranteed to function properly together. Application design then becomes simply a matter of gluing together components.

Many different component technologies are available today. Perhaps the most popular implementation of this approach is Microsoft's *Component Object Model* (COM). COM defines a binary standard for the definition of component interfaces [1], such as what data types may be passed to and returned from operations. Another emerging component technology is Sun's *Enterprise Java Beans* (EJB). The EJB specification defines a component model up to the GUI level. A JavaBean, for instance, may be a spreadsheet table widget. This is in contrast to Microsoft's COM, which does not define GUI interfaces, but rather serves as the foundation to ActiveX, which provides the control embedding functionality. Because EJB is a Java technology, it skates around issues that COM and other component technologies must address, such as binary and data−type compatibility [1].

Arguably the most promising component technology is the Object Management Group's (OMG) *Common Object Request Broker Architecture*, or CORBA. CORBA specifies a language−neutral and platform−agnostic distributed object architecture. Whereas COM is more an implementation, CORBA is strictly a specification. OMG does not provide a CORBA implementation (known as an ORB); it is left up to vendors of CORBA−compliant ORBs to define their own binary interfaces [1].

CORBA, however, is more akin to COM than to EJB. CORBA does not define a container framework for components. Instead it provides a specification language that application vendors use to define their own interfaces to components. Technologies may be built on CORBA to provide component embedding functionality, such as Apple and IBM's OpenDoc [2], KDE's KOM/OpenParts [3], or GNOME's Bonobo [4].

1.2. CORBA Availability in High Level Languages

As computing power increases, there is less need for programming in low−level languages (such as assembly or C), which has traditionally been the case for efficiency reasons. Today, high level languages are quickly gaining popularity because of their high degree of readability, maintainability, and the trend toward rapid application development (RAD).

While CORBA was once seen as a framework used only by enterprise−level applications, today there are many robust and open source CORBA ORBs available. These free implementations have given CORBA a much wider exposure in the open source world. Today, free desktop environments such as KDE[1] and GNOME use CORBA as a foundation for distributed objects and an embedded component architecture. To ensure rapid growth, developers of these projects recognize the need to make it possible for applications to be

1 KDE uses both a shared library approach (KParts) as well as a CORBA−based approach (OpenParts).

developed in high−level languages such as Perl and Python.

Because high−level languages tend to serve as a glue between smaller components, there is a requirement for a CORBA implementation. Many ORBs exist for languages such as Perl and Python, including language bindings for ORBs written in other languages (omniORBpy, for example), or complete ORB implementations in the language (such as Fnorb).

1.3. Marrying CORBA and Python

Python, an object−oriented language which spawned in 1990, has caught the eye of many developers and rivals Perl in popularity. Because Python is an object−oriented, loosely typed language, it is ideal for seamless CORBA bindings. Python’s dynamic nature also makes it possible to process interface definitions at run−time, as opposed to the conventional pre−execution−time.

The Python Language Mapping Specification [5] suggests Python bindings for a CORBA implementation. These mappings are recommended to ensure a common ground for all CORBA objects (either server−side or client−side) written in Python.

2. Background

2.1. CORBA

2.1.1. Introduction to CORBA

CORBA (the *Common Object Request Broker Architecture*) was developed by the Object Management Group (OMG) to distribute applications across client−server networks. It allows applications to communicate with each other no matter where they are located and who has designed them.

The idea behind CORBA is a software intermediary called an *Object Request Broker* (ORB) that handles access requests on data sets. The ORB is the middleware that establishes a client−server relationship between objects. Using an ORB, a client can transparently invoke a method on a server object which may be located on the same machine or across a network [6]. The ORB intercepts the call, locates an object capable of implementing the request, invokes its method with the appropriate parameters, and returns the results to the caller. The location of the object, the operating system, or its programming language are completely transparent to the requesting client.

The goal of CORBA is to make modular and distributed programming easier by making CORBA−based applications highly portable. In designing typical client/server applications, developers must make decisions about protocol, which depends on the implementation language, network transport, and a dozen other factors [7]. With an ORB, however, the protocol is defined through a single, generic definition language called IDL (Interface Definition Language). In this way, ORBs provide flexibility as they let programmers choose the most appropriate operating system, execution environment, and programming language to use for each component in a system.

2.1.2. About the OMG

The *Object Management Group* (OMG) was founded as a consortia in 1989 to promote the adoption of standards for managing distributed objects. The original members of the OMG were primarily end−users, although a small number of vendors were members. Their goal was to "adopt interface and protocol specifications" that allowed inter−operation of applications created using distributed objects [8].

Today the OMG consists of over 500 members, both vendors (including Sun Microsystems, Cisco, and even Microsoft) and end−users, and is growing quickly. While the OMG has a good reputation for the ability to quickly adopt specifications, they are not a standards organization [8]. Instead, they promote the adoption of standards by vendors in the industry. Furthermore, the OMG will not adopt any specification which does not exist as an implementation. Many ideas seem viable in theory, but implementation often exposes oversights. When a standard is promoted by the OMG, one can be sure that it actually does work.

2.1.3. Architecture Overview

The OMG defined a higher level specification called the *Object Management Architecture* (OMA) [6], of which the CORBA ORB is just one part. The OMA is a four−part architecture consisting of:

- The core component called an Object Request Broker (ORB).
- Additional services used by developers called CORBAServices.
- Common application frameworks called CORBAFacilities.
- Distributed applications themselves.

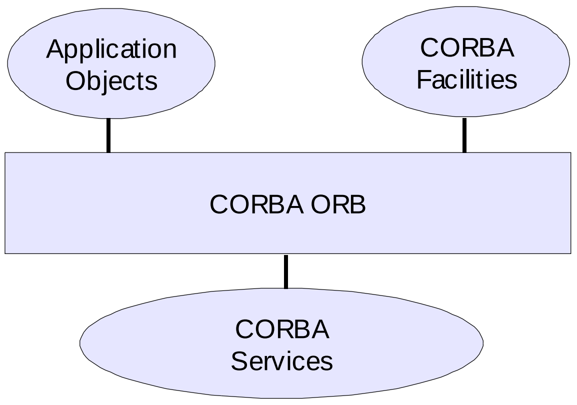

Figure 2.1 Object Management Architecture

This is depicted in figure 2.1.

CORBA can be considered an object−oriented version of Remote Procedure Calls (RPC). With RPC, a procedure is first defined, which consists of the name, the arguments which are passed to it, and the result which it returns. Using these definitions, both the client and server sides are then developed. For example, NFS is designed by wrapping the file system API with RPC calls. CORBA extends the concept of RPC by implementing traditional object−oriented concepts.

CORBA is object−oriented by necessity, not by choice. In CORBA, the three basic features of object−oriented programming are used. First, polymorphism among objects is allowed. A client may invoke methods of an object even if it knows only about the interface of its base class. Second, CORBA objects encapsulate data and methods. Each application knows nothing about the data it accesses; it merely makes a request through the ORB, and the object retrieves the data for the application. Finally, inheritance is provided. If one description of an object is designed to interface with an ORB, any object derived from that parent object will preserve its parent's interface. However, inheritance is restricted to interface inheritance only and provides no method overriding. Since the implementation of an object is intended to be transparent, implementation inheritance is not supported. On the other hand, nothing in CORBA prevents the developer of a set of service objects from using implementation inheritance, but this is not supported at the higher level CORBA specification.

All CORBA objects adhere to a classical object model. The classical model is one where all methods are contained within a class. This is in contrast to some object models (such as the generalized object model) where methods are not allocated to classes [9].

In designing CORBA objects, interface (i.e. specification) and implementation are clearly separated. An object interface specifies that object's behavior, such as the services it is willing to provide. This behavior of an object is therefore independent of its implementation. As long as the object behaves as it is specified by its interface, it is not necessary to know how it is implemented. This notion is is fundamental to CORBA.

To accomplish this, the CORBA architecture specifies an *Interface Definition Language* (IDL) [8]. IDL consistently describes object interfaces in a common manner. This way, clients need only to know an object's interface in order to make requests. Servers respond to requests made on those interfaces, and the actual detail of implementation is encapsulated inside the server behind its interface.

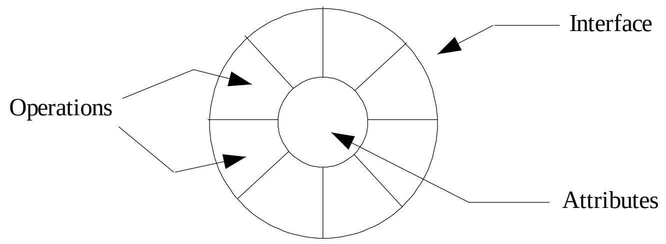

Figure 2.2 High level view of the Interface [6]

One of the goals cited by the OMA is location−transparent access to objects [9]. This means that a programmer can access an object without being required to know where on a

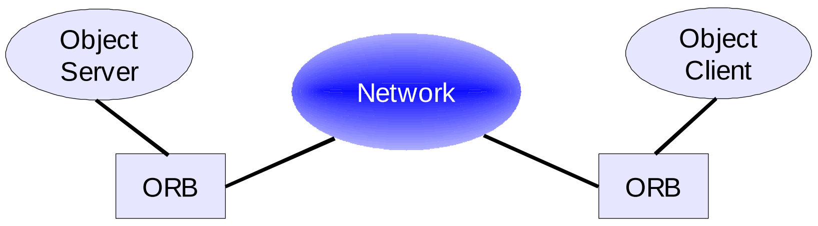

Figure 2.3 CORBA's location−transparent model

network that object exists. In accessing CORBA objects, the details of the object's location are hidden. Every object has an *Object Reference* (OR) identifier associated with it, which contains all necessary information to make requests of that object either locally or remotely. Thus, objects located across a network appear no different to the client than if they were local.

One side−effect that does make location somewhat apparent, however, is that asynchronous faults may occur so that the status of a request is unknown [6]. For example, if an request is made of an object across a network with broken connectivity, a negative response will never be received. The completion status for requests therefore allows completed, not completed, and maybe completed. The value of maybe reports the indeterminacy of success (or failure).

2.1.4. The IDL

CORBA objects are defined and mapped to a particular language (such as C++) through the IDL. This definition consists of methods and parameters that compose a complete object interface. However, the IDL is not another programming language. It is a language for expressing types, specifically interface types. It has no flow−control or iteration [9].

The IDL is a separate language within the CORBA specification with the same lexical rules as C++. New keywords are introduced to handle distributed−computing concepts [7].

The IDL describes the interfaces that client objects use when they want to reference an object implementation. Each IDL is defined completely for the object. The IDL also provides information necessary to develop clients that use an object's interface operations.

It is important to note that IDL is completely language independent. One of CORBA's strengths is that objects may be designed in any language or operating system. IDL specifications are conventionally passed through an IDL translator (typically called a compiler), which maps the interface to a specific language. IDL has been mapped successfully to many languages, including C, C++, Smalltalk, Ada95, Java, Perl, and Python.

CORBA IDL supports most data types in C++, as well as a few others (see table 2.1).

IDL	C++
boolean	bool
char	signed char
octet	8 bits
enum	enum
short	short
unsigned	unsigned
float	float
double	double
any	closest to void *
object	closest to class
string	class string
struct	struct (like C, rather than C++)
union	union (discriminated)
array	[ ]
sequence	(parameterized array)

Table 2.1 IDL types as compared to C++ [6]

Figure 2.4 shows an example of how the IDL might be used. The base interface *Person* defines the attributes and methods that might accompany a Person abstract data type. The interface *Student*, which subclasses *Person*, inherits all the attributes and methods from its parent, and defines two other attributes for a student number and grade point average.

```
interface Person {
   enum GenderType { male, female };
   readonly attribute GenderType gender;
   attribute string name;
   readonly attribute string sin;

   void set_gender(in GenderType type);
   void set_sin(in string sin_value);
};

interface Student : Person {
   readonly attribute string student_number, gpa;

   void set_student_number(in string sn_value);
   void set_gpa(in string gpa_value);
};
```

Figure 2.4 Sample IDL interface

The target code produced by the IDL compiler occurs in pairs: client−side and server−side mapping. The server−side mapping, called a skeleton, needs to be "fleshed out" with the actual implementation code for each method. The client−side mapping is referred to as a stub, and may be linked directly into the client application without modification. The client then calls the methods provided by the stub in order to request a service from the object.

Notice that each parameter in an operation (such as `set_gpa`) is prefixed with *in*. This specifies that the parameter is passed by value, and will be used by the remote end in some computation. Parameters may also be prefixed with *inout*, which means that the remote end will use the value and will also return a result through that parameter. Finally, a parameter may be indicated as *out*, which is used strictly for returning results.

2.1.5. The ORB

The ORB is the core of the CORBA architecture and provides a mechanism for transparently communicating client requests to target object implementations. The ORB simplifies distributed programming by decoupling the client from the details of the method invocations [7]. When a client invokes an object’s method, the ORB is responsible for locating the object implementation, transparently activating it (if necessary), delivering the request to the object, and returning any response to the caller.

The ORB itself is a logical entity, implemented either as a process or a set of libraries. The CORBA specification defines an abstract interface to the ORB so that clients are designed independent of the ORB. This means the ORB could be changed with little modification to the client.

To assist the ORB with delivering requests to the object, the OMA defines an entity called the object adapter (OA). The object adapter associates specific object implementations with the ORB, and is also responsible for activating the object if necessary to handle a request. There are four policies for object activation:

- **Per Method Server**: a new server for the object is spawned every time an object method is invoked. Each method call runs in its own server.
- **Shared Server**: servers can support multiple instances of the object. A single server may handle multiple objects simultaneously.
- **Unshared Server**: servers that support only one active object, but handle multiple method invocations as long as they are all on the same object.
- **Persistent Server**: the server is always active and do not require activation by the OA. These servers are assumed to be available as long as the system is operating.

Only in the Persistent Server policy is the object's implementation allowed to be active continuously. If this is not the case, then a CORBA system exception occurs. If a request is invoked under any other policy, the OA will explicitly activate the object and pass the request to it. The OA must have access to information about the object's location and operating environment for these other policies. The database containing this information is known as an Implementation Repository and is a standard component of the CORBA architecture [9]. When the OA needs to spawn an instance of the object server, it retrieves the location information from the implementation repository and executes it. Other information may also be included in this database, such as debugging, version, and administrative information.

2.1.6. Interoperability Issues

Initial versions of CORBA were criticized for not specifying ORB−interoperability. That is, it was possible for multiple vendors to develop ORB implementations to specification that were not compatible with one another. After extensive debate, this problem was finally resolved in version 2.0 of the CORBA specification (CORBA2).

CORBA2 defines the *General Inter−ORB Protocol* (GIOP) for a common interface, which includes specifications for the message type and format [9]. As mentioned in section 2.1.3, each object instance has an associated object reference (OR). However, this object reference identifier is implementation specific. The OR is not guaranteed to be compatible between ORBs. As part of the GIOP, a standard reference identifier is defined called the *Interoperable Object Reference* (IOR).

The GIOP is a generic protocol and must be extended to support a specific network transport protocol. A semantic layer, the IIOP (*Internet Inter−ORB Protocol*) is specified to map the GIOP onto TCP/IP. IIOP support is required by all ORBs for CORBA inter−ORB compliance.

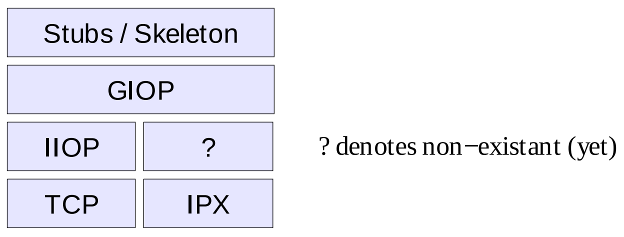

Figure 2.5 GIOP/IIOP layers

2.2. ORBit and the GNOME Project

The GNU Networking Object Model Environment (GNOME) is an open source initiative to provide a complete desktop environment, both from the user’s view, as well as the developer’s. GNOME aims to be a user−friendly desktop system that enables users to easily use and configure their computers.

GNOME’s lead developer, Miguel de Icaza, explains that GNOME was inspired by the component−based design of Microsoft’s Internet Explorer. From its inception, GNOME was built around the concept of a component object model. Because GNOME is a free, open source project built on standards, OMG’s CORBA standard was chosen as the foundation for GNOME components.

Initially GNOME used a free CORBA implementation called MICO [10]. MICO had several drawbacks for use with GNOME, notably that it was written in C++ (GNOME is written strictly in C) and that its performance did not meet GNOME’s requirements. In response, a lightweight ORB dubbed ORBit was developed in C specifically for the GNOME project.

While ORBit is currently only a partial implementation of the CORBA 2.2 specification, it is functional enough to be used extensively throughout GNOME. ORBit’s most notable characteristic is that it is fast. In fact, ORBit obliterates all other ORB implementations in benchmarks. Because it fast and lightweight, it is an ideal solution for the foundation of a component model (Bonobo, in particular).

However, because ORBit is written in C, its API is cumbersome. Designing CORBA objects using ORBit’s C API takes a significant amount of code, even compared to C++ ORBs. To rectify this, many projects have begun to bind the ORBit API to higher level languages, such as C++, Perl, Eiffel, and Python. This way, the developer reaps the benefits of the higher−level language as well as the superb speed of ORBit.

2.3. Python

2.3.1. Executive Summary

Python is an interpreted, object−oriented, high−level programming language with dynamic semantics. Its high−level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python’s simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Often, programmers fall in love with Python because of the increased productivity it provides. Since there is no compilation step, the edit−test−debug cycle is incredibly fast. Debugging Python programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program doesn’t catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on. The debugger is written in Python itself, testifying to Python’s introspective power. On the other hand, often the quickest way to debug a program is to add a few print statements to the source: the fast edit−test−debug cycle makes this simple approach very effective.[2]

2.3.2. A Brief Introduction to Python

Unlike languages such as Perl, Python has very little syntactic sugar. Also, instead of using braces or begin/end keywords to indicate grouping of statements, Python uses indentation. This has been perhaps one of its more controversial features, and while it does take some getting used to, it does seem to make the language much less intimidating to beginners and improves its readability. For example:

```
colors = (’yellow’, ’green’, ’blue’, ’red’, ’brown’, ’black’)
for i in colors:
   print i
```

Python’s greatest feature may be its easy−to−learn syntax, but it is by no means a toy language. There are classes for almost any system−level task one can imagine (including sockets, threads, pipes, and signals), as well as a myriad of high−level classes to regular expressions (both Perl and Emacs), many Internet protocols, XML, HTML, and multimedia.

In its core, Python supports most common data types such as integers, floats, and strings. In addition, there are tuples (like lists in Perl, only immutable), lists (like arrays in Perl), and dictionaries (hash tables). An example using these data structures:

```
# stuff is a dictionary
stuff = {}
# Store a tuple of colors into the stuff dictionary
stuff["colors"] = (’red’, ’green’, ’blue’)
# Store a dictionary of people and their ages into the
# stuff dictionary
stuff["ages"] = { ’Anne’: 37, ’John’: 25, ’Peter’: 28, ’Sara’: 20 }

# Create a list of people under 30
people = []
for person in stuff["ages"].keys():
```

2 The executive summary was written by Python lead developer Guido Van Rossum, available at http://www.python.org/doc/essays/blurb.html

```
        if stuff["ages"][person] < 30:
           people.append(person)
```

Python is an excellent language for developing GUI−based programs for Linux. There are Python modules for both GNOME (PyGtk) and KDE (PyKDE). PyGtk supports both a direct mapping to gtk+ (the toolkit used by GNOME) called gtkmodule, as well as an object oriented wrapper around gtkmodule. Anaconda, the installation tool used in RedHat Linux 6.1, is written mostly in Python using PyGtk.

Python has many similarities to Perl, but just enough differences to make it confusing at times for Perl programmers to learn Python. For instance, tuples, which resemble lists in Perl, are immutable. This means that the following is illegal in Python:

```
tuple = (1, 7, 3, 4)
tuple[2] = 2  # illegal!
```

Also, Python makes extensive use of exceptions, in some cases that may not make sense to the Perl programmer (such as accessing an unknown element of a dictionary or opening a file that doesn't exist). Using and handling exceptions properly in Python takes some getting used to, and may be the cause of some frustration to people who are not familiar with them.

In Python, the definition of an object differs somewhat from that of C++. Almost everything is an object in Python (represented internally as an instance of a *PyObject* type), even a class. What C++ calls objects, Python calls instances. (While in C++ instances and objects are synonymous, this is not the case with Python.) Exceptions can be almost any Python object, although they are commonly string objects or class objects. When a class object is used to raise an exception, the data passed for that exception is typically an instance of that class.

2.3.3. Extending Python from C

Like Perl, Python is extensible and embeddable. This means it is possible to create Python modules in C, usually for performance reasons, or to wrap an existing C library. The Python/C API is much more readable than perlguts/XS (Perl's C API for creating extensions), however. After a few revelations on how the API works, coding an extension module in Python is a delight. This is in contrast to Perl XS which leaves much to be desired in the readability department. However, what XS lacks for in simplicity it makes up for in documentation. Many areas of the Python/C API are not documented, which makes diving into the Python source a necessity. (Although this may not be such a bad thing.)

Although written in C, the Python/C API takes an object−based approach. All exported functions take the form Py*Object_Method*, where *Object* is any Python object (Object, String, Class, or Instance, for example), and *Method* is a particular operation to be performed on the object. The first parameter is generally a pointer to a PyObject data type. PyObject is the base "class" (the actual implementation is a structure, however because of the object−oriented approach it can be considered a class) from which all other Python objects are derived.

There are many methods associated with the PyObject type, and may be applied to any object derived from PyObject. For example, the following snippet creates an integer object, represents that object as a string object, converts the string object, and then prints that string object to stdout:

```
PyObject *the_answer = PyInt_FromLong(42);
PyObject *string = PyObject_Repr(the_answer);
printf("%s\n", PyString_AsString(string);
```

Note that because the Repr function is a method of PyObject, any Python object may be passed to it. In constrast, PyString_AsString expects a string object; passing anything else to it will cause an internal exception to be raised.

To create a module that can be imported from a Python program, one simply creates a shared library called *foo*module and a void function init*foo*() that calls Py_InitModule(), where *foo* is the name of the module.

It is often useful for a module to return a new data type to Python space. To accomplish this, one instantiates PyTypeObject and fills in the necessary attributes, such as the name of the type, and pointers to the get/set attribute functions. To return an object of this type to Python space, one calls PyObject_NEW passing the instance of the new type as a parameter and returns the new object. A simplified example of the above description might be:

```
struct PyTypeObject MyObjectType = {
   PyObject_HEAD_INIT(NULL)
   "MyObject",
   (getattrfunc)MyObject_getattr,
   (setattrfunc)MyObject_setattr
};

stuct MyObject {
   PyObject *dict; // attribute dictionary
};

PyObject *MyObject_getattr(MyObject *self, char *name)
{
   if (!strcmp(name, "answer"))
      return PyInt_FromLong(42);
   return PyObject_GetAttrString(self->dict, name);
}

int MyObject_setattr(MyObject *self, char *name, PyObject *value)
{
   if (!strcmp(name, "answer")) {
      PyErr_FromString(PyExc_TypeError, "There’s only 1 answer!");
      return −1;
   }
   return PyObject_SetAttrStr(self−>dict, name, value);
}


PyObject *MakeNewMyObject(PyObject *module, PyObject *args)
{
   return PyObject_NEW(MyObject, &MyObjectType);
}
```

From Python, we might do:

```
import mymodule

myobj = mymodule.MakeNewMyObject()
print myobj.answer
myobj.foo = 13
print myobj.foo
myobj.answer = 99
```

The above snippet will display 42, 13, and then will raise a TypeError exception claiming "There’s only 1 answer!"

2.3.4. Currently Available ORBs

Not including ORBit−Python (the name of the software designed as part of this project), there are only two currently available ORBs for Python that provide a useful feature set. The first is omniORBpy, a set of Python bindings for the C++ ORB omniORB, and Fnorb, a complete Python implementation (with a few modules written in C for performance).

While Fnorb is a commercial product and costs money, omniORB is not only freely available, but Open Source (licensed under the GNU General Public License).

3. Design and Implementation

3.1. Scope

The final goal of the project (called *ORBit−Python*) is to provide a complete wrapping of ORBit's API that is accessible from Python and a mechanism to process IDL files. The conventional approach to IDL processing is to use a compiler that generates stub and skeleton files that are included with the client and server respectively. However, because Python is a dynamic language, it is possible to process interfaces at run−time. This approach, while more complicated to implement than an IDL compiler, realizes many benefits over pre−execution compiling, including less coding overhead and the ability to discover new, arbitrary interfaces dynamically.

The scope of the project scales nicely. Implementation is divided into several stages, which are explained in later sections:

1. Create a framework for the CORBA module
2. Create a Python Object for the ORB
3. Create a Python wrapper object for a generic CORBA object
4. Create a Python Object for the POA
5. Implement the IDL processor
6. Implement marshalling/demarshalling
7. Implement client and server stubs
8. Flesh out ORB and POA objects

A minimal implementation would complete stages 1 through 4. During the implementation of the project, scope was continually reassessed against time constraints. Stages 5 through 7, while currently accounting for most of the project's code, were considered optional. In the absence of available time, ORBit’s native IDL compiler would be used and Python objects would be manually wrapped around the resulting stubs and skeletons. While not a preferred implementation, this would serve to demonstrate a proof of concept.

Stages 5 and 6 are also quite scalable. IDL type codes (see section 2.1.4) would be implemented incrementally from the simple, atomic types (char, boolean, short, long, etc.), to the more complicated types (struct, union, object reference, exception, etc.). The less used, exotic types (fixed and long double) were placed at the bottom of the priority list.

Once a fairly complete set of type codes have been implemented (that is, all type codes except the fixed and long double), the ORB and POA wrappers were to be fleshed out, completing the remainder of the functionality provided by ORBit’s API. This is the current stage of the project.

3.2. Design Overview

Initially, C++ was chosen as the implementation language. During the design, however, very few C++ features were exploited. Due to requests from the development community, and to increase ORBit−Python’s portability, the project was ported to C late in its development. An unanticipated 5−10% performance increase was also realized.

The current design consists of several modules (that is, logical modules, not Python modules), depicted in figure 3.1. Several terms are introduced in these brief descriptions that will be explained in the sections following:

- CORBA module: the entry point from Python space. This implements a thin wrapper function to load and process an IDL function (the actual processing code is in the IDL processing module), to initialize the ORB, as well as other CORBA

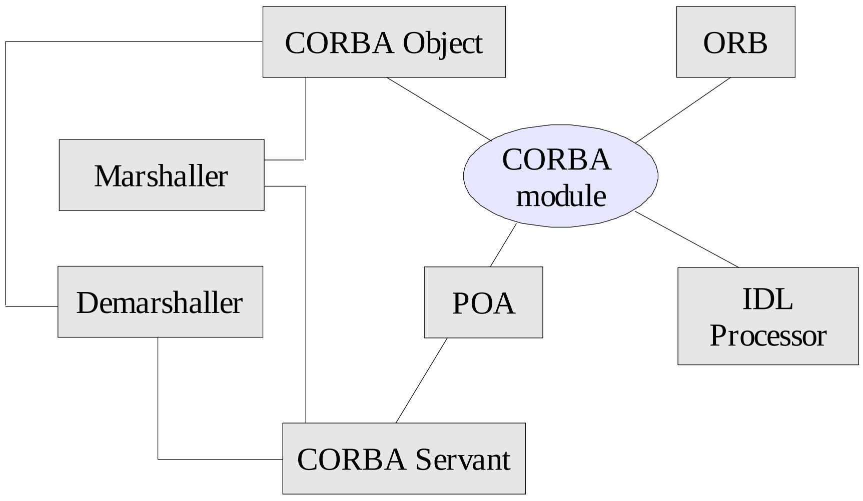

Figure 3.1 Architecture Overview

functionality such as CORBA::Any or CORBA::TypeCode

- CORBA ORB: implements the Python object representing the CORBA ORB. Currently only a subset of the ORB’s functions are implemented, including IOR to Object translation, and executing the main ORB loop (for the server side).
- CORBA Object: the Python Object representing any CORBA Object. New types for each CORBA object are generated at run−time, but this module implements generic functions for all CORBA object instances.
- IDL processor: responsible for reading and parsing IDL files and dynamically constructing the necessary Python objects.
- POA: implements the core functionality of the Portable Object Adapter, required to create object servers.
- CORBA Servant: a generic servant object
- Marshaller/Demarshaller: handles all marshalling and demarshalling of objects between Python space and the ORB.

3.3. Dynamic IDL

3.3.1. Benefits

Traditionally, interface descriptions written in IDL are parsed by an IDL compiler that generates stubs and skeletons to be used in the clients and servers. For statically typed languages such as C, this is a necessity. However, loosely typed languages such as Perl or Python don’t have to play by these rules. Because objects (and in the Python sense, a class is an object) can be created during execution, it becomes possible to discover CORBA interfaces dynamically.

This provides several benefits. First, the excess baggage of the IDL compiler−generated stubs and skeletons is done away with. Also, should the description of an interface change, recompiling and relinking is no longer required. Another advantage of dynamic interface discovery is that interfaces unknown at execution−time can be processed. Because of Python’s introspective capabilities, it then becomes possible to invoke methods of some arbitrary CORBA object at run−time. This method is not particularly expensive compared to IDL compiling, either. The processing happens only once, and once the necessary Python objects are created and accessible from Python space, there is no extra overhead incurred.

The only reason one might prefer to use the compiler is because currently the IDL file must be distributed with the client and server. (It’s not clear why one might not want to do this; perhaps security through obscurity is desired.) However, this issue is easily rectified by having the interface description embedded in the Python source and parsing it internally, rather than reading the description data from a file. (While trivial, this remains to be implemented; no one has requested this feature.)

Currently, the dynamic interface discovery approach is fairly unique. The only other project to implement this method is ORBit’s Perl bindings (CORBA::ORBit) written by

Owen Taylor [12].

3.3.2. libIDL

In order to load and parse IDL files, ORBit−Python relies on a library that is part of ORBit called libIDL. libIDL creates parse trees of the IDL file, and provides preprocessing functionality as well displaying detailed error and warning messages. While libIDL parses and tokenizes IDL files, because it is intended to be generic and reusable it does not process them further.

Processing the generated parse tree with libIDL is fairly simple. Once the IDL is parsed with `IDL_parse_filename`, the root node of the parse tree is returned. One then calls `IDL_tree_walk`, passing it two callback functions: one to be called when a new node in the tree is entered (`tree_pre_func`), and one that is called when that node is exited (`tree_post_func`).

The `tree_pre_func` function examines the type of node and takes the appropriate action. For instance, if the node is an interface, it will create the necessary internal data structures for the new interface. If the node is an attribute declaration, it will store the attribute’s metadata (such as its name and type) in the attribute list associated with its interface. The `tree_post_func` function constructs the Python object for the interface once it is finished processing all operations, attributes, and exceptions associated with that interface.

3.3.3. IDL Processing Details

All IDL types require an object representation accessible from Python. IDL modules and interfaces are represented as Python instances, while structures, unions, and exceptions are represented as classes. Top−level modules or interfaces are inserted in the top−level (__main__) Python namespace.[3] For example, consider the following IDL:

```
module Bank {
   struct Transaction {
      string date;
      short code;
   };
   interface Account {
      exception Overdrawn {
         double amount;
      };
      attribute string id;
      readonly attribute double balance;

      void withdraw(in Transaction t, in double amount)
                    raises (Overdrawn);
    };
};
```

Figure 3.2 depicts the object hierarchy accessible from Python for this interface description. Each box represents an individual Python object, with its type noted in parentheses. Note that Bank and Account are instances and not classes. This means that they cannot be instantiated from Python. In fact, instantiating them is not necessary; the proper way to obtain an instance of a CORBA object is through an object factory or the Portable Object Adapter. Both these methods are described in detail later.

Also note that attributes and operations are not in the object hierarchy. The objects shown in this hierarchy are not CORBA objects. Rather, they are interfaces to construct the necessary parameters for the operations of CORBA objects. Consider the withdraw operation of the Account interface. The client needs some way to construct the Transaction structure for the first parameter of the withdraw operation. For example:

```
t = Bank.Transaction(data = "03/23/2000", code = 1)
```

The object *t* is a class instance with the two attributes, data and code, defined. Assuming we have a CORBA object *o* that is of type Bank::Account, we can then do:

```
o.withdraw(t, 42.50)
```

3 This is considered a design flaw; IDL objects should be inserted into the Python namespace from which the IDL file was loaded. This issue remains to be resolved.

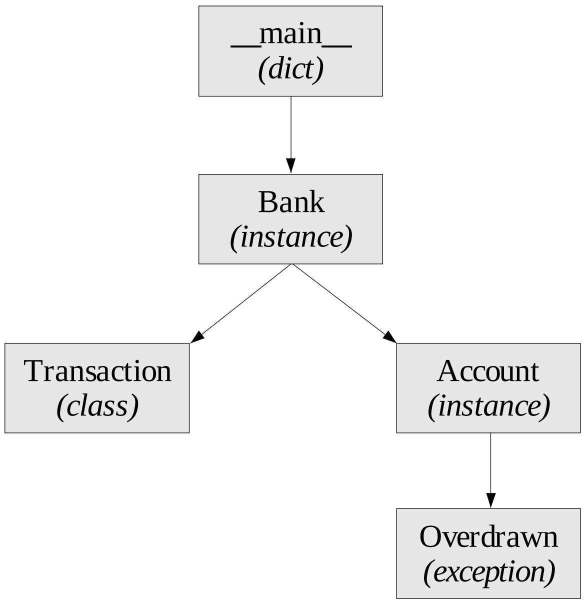

Figure 3.2 Example Python object hierarchy

The notation Bank::Account indicates that this object is a CORBA object of that type (in fact, the CORBA type code for this object is IDL:Bank/Account:1.0; this notation is discussed later). This type, however, is not related to the object hierarchy described above. Again, the object hierarchy is strictly a means to construct structures, unions, and exceptions required for interface operations, as well as accessibility to constant definitions and enumeration types.

Internally, these objects are stored in a hash table by their type codes. A CORBA type code is a notation that defines a specific data type, whether it is a built−in, atomic type (such as a boolean), or a complex data type defined by an external interface (such as the Bank::Transaction structure). Below is an incomplete table comparing some CORBA types with their type codes:

Type	Type Code
boolean	IDL:CORBA/Boolean:1.0
char	IDL:CORBA/Char:1.0
any	IDL:CORBA/Any:1.0
Bank::Account	IDL:Bank/Account:1.0
Bank::Transaction	IDL:Bank/Transaction:1.0

Table 3.1 An incomplete list of IDL types and their type codes

An internal hash table called *object_glue* is created upon initialization. Each entry in the hash table maps a CORBA object to a Python object representing it, and is hashed on the type code string. For example, a lookup of *IDL:Bank/Transaction:1.0* will return the Python class object representing the structure. Looking up *IDL:Bank/Account/Overdrawn:1.0* will return the Python exception object mapped to the CORBA exception Bank::Account:: Overdrawn. Interfaces are a special exception: a lookup of *IDL:/Bank/Account:1.0* returns a structure containing a description of the interface (its operations, attributes, exceptions, and a list of parent interfaces), as well as a Python type object for the interface. This type is used with `PyObject_NEW` (see section 2.3.3) to create a new Python object of that type.

3.4. Implementation Details

3.4.1. Marshalling and Demarshalling

Marshalling is the process in which the parameters passed to a CORBA operation are converted from the native language type (in this case, Python objects) to the CORBA type and placed in the local ORB's transmission buffer to be sent to the remote ORB. Once the remote ORB receives the data on the wire, it *demarshals* the parameters by converting the CORBA type. This process is depicted in figure 3.3.

The process shown in figure 3.3 applies only to operations that do not return any results back to the caller. For example, consider the following IDL specification for an

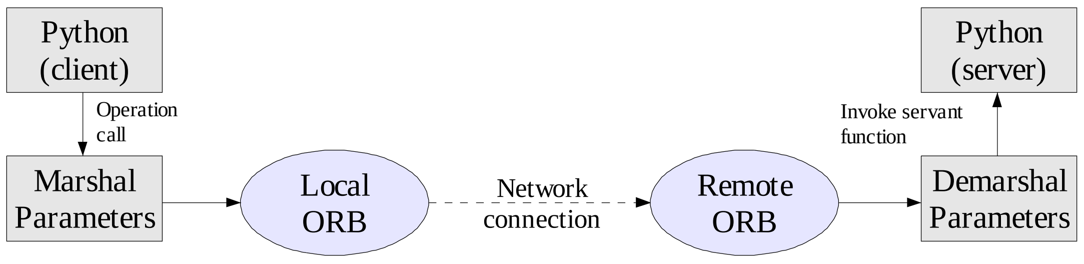

Figure 3.3 The marshalling and demarshalling process

operation:

```
short foo(in string a, in float b);
```

Suppose the client invoked this operation from Python as:

```
result = object.foo("hello world!", 12.34)
```

First, the client sets up a communication buffer with the local ORB for this operation. Then it marshals the two parameters, *a* and *b*, and transmits the buffer to the remote ORB. The remote ORB demarshals the parameters and invokes the server implementation of this operation. Since this operation returns a value, the server end marshals the return results and transmits the data back to the client side. The client finally demarshals the return value and the operation is completed.

For operations that return multiple results (that is, through *out* or *inout* parameters), a tuple is returned to the Python caller containing the results ordered in which they appeared in the operation description. So, given the IDL

```
short foo(in string a, inout short b, out string c);
```

Calling this operation from Python will return a tuple (<short>, <short>, <string>).

3.4.2. Type Mappings

Table 3.2 shows the mappings ORBit−Python uses between CORBA types and Python types. This table mostly follows the Python Language Mapping Specification [5]:

CORBA Type	Python Type
octet	PyInt
short	PyInt
long	PyLong
unsigned short	PyInt
unsigned long	PyLong
long long	PyLong
unsigned long long	PyLong
float	PyFloat
double	PyDouble
long double	Not yet implemented
boolean	PyInt
char	PyString (length 1)
string	PyString
wide char/string	Not yet implemented
fixed	Not yet implemented
sequence	PyList or PyTuple
enum	PyInt

Table 3.2 Python mappings for CORBA types

When marshalling sequences or lists, either lists or tuples are accepted. However, one should not assume that the values of sequence or arrays types are mutable [5]. For this reason, demarshalling either of these types returns a tuple.

For efficiency reasons, sequences and arrays of characters and octets are mapped to the string type. This functionality has not yet been implemented.

3.4.3. Walking Through the Code

Perhaps the clearest way to describe the implementation is by stepping through a short example. Consider the IDL from section 3.3.3, called *Bank.idl*:

```
module Bank {
   struct Transaction {
      string date;
      short code;
   };
   interface Account {
      exception Overdrawn {
         double amount;
      };
      attribute string id;
      readonly attribute double balance;

      void withdraw(in Transaction t, in double amount)
                    raises (Overdrawn);
   };
```

Now let’s examine a possible implementation of the server object for the above IDL, written using ORBit−Python. This is a complete listing of the code, not just a code fragment. Each line is prefixed with a line number as it will be referred to later:

```
01:  import CORBA
02:  import sys
03:
04:  class Account:
05:     def __init__(self):
06:        self.id = "12345" # set a default id for this account
07:        self.balance = 500.0 # account opened with $500 balance
08:
09:     def withdraw(self, t, v):
10:        if self.balance - v < 0:
11:           exc = Bank.Account.Overdrawn(amount = v - self.balance)
12:           raise Bank.Account.Overdrawn, exc
13:        else:
14:           self.balance = self.balance - v
15:           print "Transaction date:", t.date, "code:", t.code
16:           print "New balance:", self.balance
17:
18:  CORBA.load_idl("Bank.idl")
19:  orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
20:  poa = orb.resolve_initial_references("RootPOA")
21:
22:  servant = POA.Bank.Account(Account())
23:  poa.activate_object(servant)
24:  ref = poa.servant_to_reference(servant)
25:  print orb.object_to_string(ref)
26:
27:  poa.the_POAManager.activate()
28:  orb.run()
```

Lines 01−02 import the CORBA and system modules. The CORBA module is the only Python module generated by ORBit−Python. All other modules are not related. Lines 04−16 constitute the actual implementation of the Account interface. The name of this class is arbitrary, but for readability and clarity, the convention is to use *InterfaceName* or *InterfaceName_Impl*. Lines 05−07 implement the constructor for this class, and simply initialize the attributes that will be glued to the interface. (Note that this class is not at this point glued to a CORBA object.) Lines 09−16 represent the implementation of the withdraw operation. First the balance is checked and it is verified that the account has enough money for the withdrawal. If not, Bank::Account::Overdrawn exception is raised. On line 11, the exception data is constructed.

Observe how the exception is created, as described in section 3.3.3. Internally, Python represents exceptions created by `PyErr_NewException` as classes. Line 11 then creates an instance of this class. Then, in line 12, the exception is raised, with the data (the instance of the exception) passed. When control is returned to ORBit−Python, it will check for this exception and pass it to remote ORB. Lines 13 through 16 are executed during a successful transaction. The transaction details are displayed, and the account balance is updated. Line 16 concludes the implementation of the Account interface.

Lines 18−26 perform the setup and initialization of the ORB; any CORBA server will follow the same pattern of code. On line 18, the Bank.idl file is read and CORBA types are glued to their Python objects as described in section 3.3.3. Line 19 initializes the ORB and returns an CORBA.ORB instance. Then, line 20 fetches the Portable Object Adapter.

The *Portable Object Adapter*, or POA, is the primary means of making implementation objects (such as an instance of Account) available to the ORB for servicing requests. The POA supersedes the *Basic Object Adapter* (BOA) from early CORBA versions. The BOA was under−specified and required vendors to implement their own, proprietary solutions to make the BOA useful. The POA was introduced to solve the shortcomings of the BOA, and includes functionality for large−scale systems, as well as location transparency.

Line 22 creates the servant for the Account object. A CORBA servant is programming language interface representing the CORBA server. While an instance of the Account class has no binding to any CORBA object (it is by itself merely a Python instance object), the servant represents the glue between the Python instance object and the CORBA server. Then, in line 23, we register the servant with the POA and activate the object so it can begin servicing requests.

The next two lines, 24−25, merely print the IOR to stdout, so that we can manually pass it to the server. Obviously for complete applications this is not acceptable. For these situations, one can register the IOR with the CORBA NamingService. The NamingService is a CORBAService that maps names (type codes, usually) to the instance's IOR.

Finally, in lines 27−28, we activate the POA manager and enter ORBit's main loop. The CORBA server is now ready to receive requests.

The code below shows how the client side might be implemented:

```
01:  import CORBA
02:  import sys
03:
04:  CORBA.load_idl("Bank.idl")
05:  orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
06:  ior = sys.stdin.readline()[:−1]
07:  acct = orb.string_to_object(ior)
08:
09:  acct.id = "00112233"
10:  t = Bank.Transaction(data = "03/23/2000", code = 1)
12:
13:  try:
14:     acct.withdraw(t, 12.34)
15:  except Bank.Account.Overdrawn, exc:
16:     print "Failed, would overdraw by", exc.amount
17:
18:  print "Current balance:", acct.balance
```

Lines 01−05 load the IDL and initialize the ORB in the same manner as the server code above. In line 06, the IOR is read from stdin. (The [:−1] trims the trailing newline from the resulting string.) We assume the user will paste the IOR output from the server into the client. Again, the preferred method is to use the NamingServer, but for the purposes of an example this will suffice. The real magic happens in line 07: the stringified IOR is passed to

CORBA::ORB::string_to_object, and Bank.Account Python object is returned. This object represents the servant object on the server side. The interaction with the servant through the client object is completely transparent; with only lines 09−18 to inspect, there is no way to tell that the *acct* object is actually a CORBA object that resides in a different process, on a different system, or is possibly written in a different programming language. This is the beauty of CORBA.

Line 9 sets the *id* attribute of the Account object. Keep in mind that the location of this attribute is on the server side. In fact, the value is not stored on the client side at any point. Also note that there is no explicit code in the server to handle the assignment of this attribute. This is handled transparently by ORBit−Python.

The method in which the attributes are stored and retrieved in ORBit−Python goes against the guidelines set in the Python Language Mapping Specification [5]. The mapping specification recommends that accessor pairs be implemented, one for setting and one for retrieving the value of the attribute. So, while in ORBit−Python one does:

```
acct.id = "00112233"
print acct.id
```

The specification suggests:

```
acct._set_id("00112233")
print acct._get_id()
```

There are several reasons why ORBit−Python goes against the specification on this issue. Firstly, the accessor pair method was recommended for efficiency reasons. However, efficiency is only an issue in this case when the ORB implementation uses a conventional IDL compiler approach. With an IDL compiler, Python code would be generated for the stubs and skeletons. In order to implement equivalent functionality to ORBit−Python, the Python code for the stub and skeleton would have to implement the Python methods *__setattr__* and *__getattr__*.

The __*setattr*__ and __*getattr*__ methods are private methods of a class and are invoked when an attribute (attributes in Python consist of both methods and data) is referenced. For instance:

```
class foo:
   def __getattr__(self, name):
      if name == "foo":
         return self.bar

   def __setattr__(self, name, value):
      if name == "foo":
         self.bar = value

f = foo()
f.foo = 4
print f.foo
```

So while the attribute *foo* is set by the caller, the actual attribute set internally in the instance is *bar*. Overriding the default setattr/getattr pairs of a class incurs additional overhead, however. Thus, a stub or skeleton that implements this method will also suffer from this overhead.

ORBit−Python processes IDL files dynamically, however, and all CORBA objects are separate Python types. A Python type has associated with it C functions for setting and retrieving attributes. Therefore, the logic that would normally be implemented in the Python __*setattr*__ method is handled in the C function. Ultimately, ORBit−Python gains this functionality for free because of its inherent design.

Nevertheless, in order to comply with the standards and to remain portable with Python bindings for other ORBs, eventually the _set_*attr*/_get_*attr* accessor pair will be implemented in addition to the current technique. This modification is trivial but has a low priority as it has not been requested.

4. Testing and Evaluation

4.1. Peer Review

4.1.1. Background

In the late 1990s, Eric S. Raymond coined the term Open Source. For software to comply with the OSI (Open Source Initiative) guidelines, it must meet the following requirements [13]:

- source code freely available
- free to distribute
- free to modify and redistribute

The Linux operating system has been built under this model and it is widely regarded as the most stable and bug−free platform because of its development model. In particular, it follows Linus' Law:

"Given enough eyeballs, all bugs are shallow."

This introduces the notion of peer review. With peer review, other developers with varying skill levels and backgrounds can examine the code, submit modifications, make suggestions, and report bugs. With a model in which the source code is unavailable, bug reports tend to be vague and difficult to track.

Eric Raymond also explains *plausible promise* [14]. For Open Source software (OSS) to be successful, it needn't be particularly good, stable, or well−documented. The only true requirement for a successful OSS project is that it must convince other developers that it has the potential to evolve into a useful and appealing product.

GNOME, the free desktop environment for which ORBit was designed, is also an Open Source project. As with Linux, GNOME's development follows a *bazaar* model.

With the bazaar model, developers from all across the globe contribute to the project in any way possible, from core development to bug reporting. While Open Source software need not follow a bazaar model or vice versa, it typically does.

4.1.2. ORBit−Python Released

On March 10, 2000, ORBit−Python was released to the open source development community. Notices were placed on several development−related web sites, and a site for the project was created (http://projects.sault.org/orbit−python). A to−do list was packaged with the project that identified areas that required improvements or implementation.

By the next day, I had received several emails praising ORBit−Python. One developer even volunteered to complete one of the items on the to−do list. Since then, I have received several bug reports, feature requests, patch submissions, and encouraging feedback.

Due to the nature of the bug reports, the first version of ORBit−Python (0.1.0) seemed to be reasonably stable. The current version as of this writing (0.1.3) has fixed several bugs, implemented many requested features, and is much more stable and portable than the initial version. (See the ChangeLog in Appendix A for details.) These improvements would never have been possible without peer review.

4.2. Objectives Assessment

The goals set during the study phase of the project were fairly ambitious. The vision of the project was the completion of useful Python bindings for ORBit (that is, handle all commonly used type codes) using dynamic IDL.

In the initial stages of the project, it was not known if time would permit to complete the dynamic IDL processing functionality. A contingency plan was devised in this event.

After some revelations on CORBA internals and a great deal of help from the development community, significant progress was made in completing these goals.

The end result exceeded the initial vision by a long shot. The project is currently being used by several developers in the GNOME community, and it has a clear path for evolution.

4.3. Testing

In order to test each feature of ORBit−Python, a test suite was developed (listed in appendix B). The test suite unveiled several subtle bugs that were fixed before the initial release. Fortunately, testing did not uncover any serious design flaws.

The goal of the test suite was to verify that each of the supported type codes were being handled properly and that the API worked as expected. And because there is currently no developer documentation, the test suite also served as a brief tutorial on how to use ORBit−Python.

4.4. Performance

Initial benchmarks[4] that measured ORBit−Python's raw performance were quite promising. Table 4.1 compares the performance of ORBit−Python with native ORBit, omniORB [11], Fnorb (an complete Python implementation) [15], and Java IDL. The following IDL was used for the benchmark:

```
module Counter {
         interface Count {
            attribute long sum;
```

4 Benchmarks submitted by Jon Kåre Hellan; system configuration and version details listed in Appendix C.

```
            long increment();
        };
    };
```

Client	Server	Local calls/s	Remote calls/s
ORBit−C	ORBit−C	4150	730
ORBit−C	ORBit−Python	2360	670
ORBit−Python	ORBit−C	2630	555
ORBit−Python	ORBit−Python	**1960**	**515**
ORBit−C	omniORBpy	480	375
omniORBpy	ORBit−C	1450	515
omniORBpy	omniORBpy	450	315
ORBit−C	Fnorb	78	77
Fnorb	ORBit−C	55	18
Fnorb	Fnorb	38	20
ORBit−C	Java IDL	450	380
Java IDL	ORBit−C	810	500
Java IDL	Java IDL	320	230

Table 4.1 Benchmarks comparing various ORBs

The conclusions are fairly encouraging. For local calls, ORBit−Python is over 50 times faster than Fnorb, 6 times faster than Java IDL, and 4 times faster than omniORB's Python bindings. For remote calls, ORBit−Python leads again, being 25 times faster than Fnorb, 2 times faster than Java IDL, and 1.6 times faster than omniORBpy.

Also, these benchmarks were performed on ORBit−Python 0.1.0. Since then, thanks to the port to C and some minor performance tweaks, the current version is approximately 5−10% faster.

4.5. Future Plans

There are several items on the to−do list that have yet to be completed:

- Go through the TODOs and FIXMEs in the source

- Fixed type
- Long Double type
- Wide types (char and string)
- Handle internal errors more gracefully (raise appropriate exceptions)
- Better TypeCode support
- Interface Repository support
- Add classes/objects to import caller’s namespace instead of __main__
- Fix structs and exceptions so that they don’t require keywords
- More thorough testing
- Documentation

These items will be completed in the order in which they are requested by the development community, otherwise in the order in which they are listed. (Some parallelism can be applied.)

Another fairly important step is to audit the current code for memory leaks and performance issues. While ORBit−Python is currently only 50% slower than ORBit−C (which is still quite fast, given that Python is an interpreted language), profiling and optimization should raise ORBit−Python’s performance by another 20−30%.

5. Conclusions

The current state of the ORBit−Python has exceeded the goals set during the initial phases of the project. The software is already quite useful for most projects in its current state. It has also been received extremely well by the development community. Given the potential uses of ORBit−Python and its direction, it is possible that it may be packaged with the GNOME project and released with many Linux distributions.

During the project's development, several important observations were noted. First, the importance of a comprehensive test suite is not to be underestimated. The test suite for ORBit−Python (see Appendix B) revealed a number of small bugs in the implementation, which, in the end, were show−stoppers. Because of the inherent complex web of interconnections in software, it is easy for modifications or additions in one part of the code to affect other areas in unanticipated ways. A well−written test suite will ensure that all areas of the software work properly.

Interacting with the development community, receiving and responding to bug reports, suggestions, criticisms, and other feedback was also an interesting learning experience. Developers were using ORBit−Python in ways that were not anticipated, and so bugs were uncovered that even a well−written test suite could not have discovered. Also, the opportunity to discuss issues with other extremely knowledgeable people involved with CORBA and Python was very useful, and at times perhaps even a bit intimidating.

Python's design and clean syntax pairs nicely with CORBA. Because of the rising popularity in both Python and CORBA, and because of the unprecedented speed offered by ORBit−Python, I expect the software to be useful to many developers.

6. References

[1] *ADT Magazine*, January 2000. **Component Strategies and Architectures**; http://www.adtmag.com/pub/jan2000/com_strat.htm

[2] *Apple Computers, Inc.* **OpenDoc**; http://opendoc.apple.com

[3] *The K Desktop Environment (KDE).* **KOM/OpenParts**; http://www.kde.org

[4] *GNU Network Object Model Environment (GNOME).* **Bonobo**; http://developer.gnome.org/arch/component/

[5] *Martin Von Loewis.* **The Python Language Mapping Specification**; http://www.omg.org/cgi−bin/doc?ptc/00−11−12

[6] *A. Pope, 1998.* **The CORBA Reference Guide**; http://www.qds.com/people/apope/Corba/

[7] *Z. Yang, 1997.* **CORBA: A Platform for Distributed Computing**; http://www.infosys.tuwien.ac.at/Research/Corba/archive/intro/OSR.ps.gz

[8] *The Object Management Group*; http://www.omg.org

[9] *K. Keahey, 1995.* **A Brief Tutorial on CORBA**; http://www.cs.indiana.edu/hyplan/kksiazek/tuto.html

[10] *MICO Development Group.* **MICO;** http://www.mico.org

[11] *AT&T Laboratories Cabridge.* **OmniORB;** http://www.uk.research.att.com/omniORB/

[12] *Owen Taylor, 2000.* **CORBA::ORBit;** http://people.redhat.com/otaylor/corba/orbit.html

[13] *The Open Source Initiative.* **The Open Source Definition;** http://www.opensource.org/osd.html

[14] *Eric S. Raymond.* **The Cathedral and The Bazaar;** http://www.tuxedo.org/~esr/writings/cathedral−bazaar/

[15] *Fnorb Development Group;* **Fnorb.** http://www.fnorb.org

A. Appendix A: ORBit−Python ChangeLog

```
2000−03−19  Jason Tackaberry <tack@linux.com>

   * Fixed a bug that fudged union discriminators.

2000−03−18  Jason Tackaberry <tack@linux.com>

   * Ported the whole mess to C.  Aside from compiling 20% faster, it should
     hopefully build properly on more platforms.  I’ve also (through some
     crude measurements) noticed a 5−10% speed increase.
   * idl.c, except.c, types.c: fixed unions, structs, and exceptions so that
     attributes set on construction are set for the instance object, not for
     the class object.
   * Released 0.1.3

2000−03−16  Jon Kåre Hellan <hellan@acm.org>

   * idl.cc: pass IDL_parse_filename IDLF_CODEFRAGS which is required for
     Gnumeric.idl

2000−03−16  Jason Tackaberry <tack@linux.com>

   * CORBA_ORB.cc: CORBA_ORB_PyObject__new properly accepts either lists or
     tuples for parameter 1.
   * CORBA_ORB.cc: resolve_initial_references handles RootPOA properly now
   * CORBA_ORB.cc, CORBAmodule.cc: made ORB_init more conformant
   * exceptions raised by check_corba_ex() now work properly

2000−03−15  Jason Tackaberry <tack@linux.com>

   * CORBA_ORB.cc: implemented CORBA::ORB::resolve_initial_references
     (still need to handle RootPOA and POACurrent)
   * idl.cc, CORBAmodule.cc: load_idl() now takes optional preprocessor
     parameters to pass to IDL_parse_filename
   * demarshal.cc: can demarshal generic Objects now.
   * idl.cc, CORBAmodule.cc: now handles libIDL errors (such as missing file)
     gracefully by raising an exception.

2000−03−14  Jason Tackaberry <tack@linux.com>

   * CORBAmodule.h, idl.cc, marshal.cc, demarshal.cc: started adding debugging
     output.  Set __DEBUG_LEVEL__ in CORBAmodule.h to enable (a 0 to 9 value).
   * CORBAmodule.cc: CORBA_Object_to_PyObject returns Py_None if the passed
     object is NULL rather than bailing out with an error.

2000−03−14  Jason Tackaberry <tack@linux.com>

   * CORBA_ORB.cc: fixed a stupid bug that caused a segfault with CVS ORBit,
	and as a result created one more FIXME item. :)
	* Released 0.1.2

2000−03−13  Jason Tackaberry <tack@linux.com>

	* configure.in: forced precedence to g++ until I can fix the code to
	  compile properly with Sun’s Workshop.  (Okay, it still doesn’t work
	  properly.  I’m working on that.  Hack the Makefile to link in libgcc
	  in the meantime.)

2000−03−13  Phil Dawes <philipd@users.sourceforge.net>

	* idl.cc: fixed a buglet that stops it from compiling with gcc−2.95
	* Added automake/autoconf support
	* Added a ’make check’ test target to the src directory
	* Added an autogen.sh to generate the configure script and run it

2000−03−12  Jason Tackaberry <tack@linux.com>

	* PortableServermodule.[cc,h]: renamed to PortableServer.[cc,h] and the
	  module is initialized from CORBAmodule.cc.  This will eliminate the
	  shared library dependency problem between the two modules.
	* Fixed a cosmetic bug in test−client
	* Packaged this as version 0.1.1


2000−03−11  Jason Tackaberry <tack@linux.com>

	* initial version (0.1.0) released
```

B. Appendix B: The Test Suite

B.1. The Interface Definition

```
// Yes, this is an absurd and stupid example.  I tried. :)

module Fruit {
   enum Color { orange, red, yellow, green };

   struct Properties {
      string name;
      Color color;
   };

   interface Instance {
      readonly attribute Properties fruit;
      readonly attribute short left;

      exception AllEaten {};
      exception BigBite { short too_much_by; };
      void bite(in short size) raises (BigBite, AllEaten);

      oneway void throw_out();
   };

   typedef sequence<Properties> fruits;

   interface Factory {
      attribute fruits fruit_list;

      exception AlreadyExists {};
      void add_fruit(in Properties f) raises (AlreadyExists);
      Instance get_instance(in Properties fruit);

      short get_random_fruit(out Properties fruit);

      // Okay, I ran out of fruit ideas for this stuff. :)
      const double pi = 3.141592653;
      union TestUnion switch (Color) {
         case orange: string a;
         case red: short b;
         case yellow: float c;
         case green: boolean d;
      };

      TestUnion test_union(in Color color);
      any test_any();
   };
};
```

B.2. The Server

```
import CORBA

class Instance:
   def __init__(self, fruit):
      self.fruit = fruit
      self.left = 100

   def bite(self, size):
      if self.left - size < 0:
         exdata = Fruit.Instance.BigBite(too_much_by = size - self.left)
         raise Fruit.Instance.BigBite, exdata
         return

      self.left = self.left - size
      print "Eating %d%%, %d%% left" % (size, self.left)
      if self.left == 0:
         raise Fruit.Instance.AllEaten

   def throw_out(self):
      print "Client threw me (%s) in garbage!" % self.fruit.name
      poa.deactivate_object(self._servant)


class Factory:
   def __init__(self):
      self.fruit_list = []

   def add_fruit(self, f):
      for fruit in self.fruit_list:
         if fruit.name == f.name:
            raise Fruit.Factory.AlreadyExists
            return
      self.fruit_list.append(f)

   def get_instance(self, fruit):
      new_instance = POA.Fruit.Instance(Instance(fruit))
      poa.activate_object(new_instance)
      return poa.servant_to_reference(new_instance)

   def get_random_fruit(self):
      import random
      index = random.randint(0, len(self.fruit_list) - 1)
      return index, self.fruit_list[index]

   def test_union(self, color):
      if color == Fruit.orange:
         return Fruit.Factory.TestUnion(color, "foobar")
      elif color == Fruit.red:
         return Fruit.Factory.TestUnion(color, 42)
      elif color == Fruit.yellow:
         return Fruit.Factory.TestUnion(color, 2.71828)
      elif color == Fruit.green:
         return Fruit.Factory.TestUnion(color, CORBA.TRUE)

   def test_any(self):
      import random
      pick = random.randint(0, 2)
      if pick == 0:
         return CORBA.Any(CORBA.TypeCode("IDL:CORBA/String:1.0"), "abc123")
      elif pick == 1:
         return CORBA.Any(CORBA.TypeCode("IDL:CORBA/Short:1.0"), 42)
      elif pick == 2:
         p = Fruit.Properties(name = "pineapple", color = Fruit.yellow)
         return CORBA.Any(CORBA.TypeCode(p), p)
```

```
CORBA.load_idl("test-suite.idl")
CORBA.load_idl("/usr/share/idl/name-service.idl")
orb = CORBA.ORB_init((), CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")

servant = POA.Fruit.Factory(Factory())
poa.activate_object(servant)
ref = poa.servant_to_reference(servant)
open("./test-server.ior", "w").write(orb.object_to_string(ref))

poa.the_POAManager.activate()
orb.run()
```

B.3. The Client

```
import CORBA
import sys

CORBA.load_idl("test−suite.idl")
orb = CORBA.ORB_init((), CORBA.ORB_ID)
ior = open("./test−server.ior").readline()
o = orb.string_to_object(ior)
print o.__repo_id
try:
   o.add_fruit(Fruit.Properties(name = "orange", color = Fruit.orange))
   o.add_fruit(Fruit.Properties(name = "banana", color = Fruit.yellow))
   o.add_fruit(Fruit.Properties(name = "apple", color = Fruit.red))
   o.add_fruit(Fruit.Properties(name = "lime", color = Fruit.green))
except: pass

for fruit in o.fruit_list:
   print "%s is %s" % (fruit.name, Fruit.Color[fruit.color])

i = o.get_instance(o.fruit_list[0])
print "I’ve an instance of an", i.fruit.name
i.bite(50)
try: i.bite(60)
except Fruit.Instance.BigBite, exd:
   try: i.bite(60 − exd.too_much_by)
   except Fruit.Instance.AllEaten:
      print "I’m done eating my %s; throwing it out" % i.fruit.name
      i.throw_out()

print "Pi is", Fruit.Factory.pi
r = o.get_random_fruit()
print "Random fruit of the day: %s (at index %d)" % (r[1].name, r[0])

union = o.test_union(Fruit.orange)
print "Testing union: discriminate:", union.d, "−− value:", union.v
print "Testing any:", o.test_any()
```

C. Benchmark Information

System Configuration	**Client**	**Server**
Local test	AMD K6/233 64MB, Linux 2.2.14, glibc 2.1.3	AMD K6/233 64MB, Linux 2.2.14, glibc 2.1.3
Remote test	P5/133, 45MB, Linux 2.2.14, glibc 2.1.3	AMD K6/233 64MB, Linux 2.2.14, glibc 2.1.3

Product	**Version**
OmniORBpy	2.8.0
ORBit	0.5.1
Fnorb	1.0.1
ORBit−Python	0.1.0
Java IDL	Blackdown Pre−release 2.0 with JIT and native threads