CORBA Applications In GNOME

Elliot Lee
Red Hat Software

sopwith@redhat.com

The use of CORBA technology for inter-application communication and integration is explored in the context of the GNOME desktop project. The lessons learned in the ORBit CORBA implementation and GNOME CORBA integration libraries are explained, and hints on making effective use of CORBA in applications are given.

It is assumed that the audience of this paper is familiar with software development concepts and has heard of CORBA and the GNOME desktop project.

Introduction

Software systems have gone through various phases. In the early 1980's, the introduction of the personal computer necessitated a change in the design strategies of software. Instead of being run on relatively isolated timesharing or batch processing systems, software had to be able to cope with unskilled end users, and run on resource-limited hardware. On the software development front, Third generation programming languages came into wide use.

From the early 1980's through the early 1990's, the graphical user interface as a method of increasing usability was a prime focus. Object oriented programming was touted as the silver bullet of software engineering, and provided insights into decreasing software development and maintainance costs.

In the last few years, the widespread use of networking as part of computer systems means that software becomes much more valuable when it can talk to other software systems. The integration of the world wide web into many popular desktop environments is one example of this. On a programmatic, behind-the-scenes level, CORBA and DCOM have been created as protocols for inter-program communication.

The GNOME project was born out of a frustration with some of the KDE project's views on software design, implementation, and licensing. Initiated in August 1997, its goals are to produce an advanced desktop environment with reasonable resource requirements, total user configurability, and agreeable software licensing.

The ORBit project was initiated in February 1998 to fill the need for object request broker compliant with the CORBA 2.2 standard with minimal resource requirements and C language bindings.

CORBA

Microsoft's DCOM and the OMG (Object Management Group)'s CORBA are the two main object networking standards. Since CORBA has a history of being cross-platform and open, it was chosen for use in GNOME.

An important part of the CORBA is a standard syntax for defining the capabilities of an object. This is the OMG's Interface Definition Language. To start to build a simple hypothetical object, we would write the IDL that defines how the rest of the world can make use of this object:

```
interface Q {
void frobate();
};
```

An "interface" is what the world is able to see about a particular object, and roughly corresponds to a 'class' declaration in a C++ header file. A wide range of objects can be defined using OMG IDL syntax. For example, here is the IDL currently used by the GNOME help browser:

```
module help_browser {
  interface simple_browser {
//show URL in same browser
   void fetch_url(in string URL);
// make new browser and show URL
   simple_browser show_url(in string URL);
  };
};
```

This IDL shows some of the other commonly used features of CORBA interfaces. Related object interfaces may be grouped into "modules". Operations on objects can accept parameters and return values, including other objects.

An important part of software design is data representation. The CORBA data types allow representation of almost any data structure through a wide selection of basic data types, and a variety of ways to combine those basic data types into complex data structures.

Available CORBA data types:

Basic:

- Numeric (integer, floating and fixed point, and enumerated).
- Characters and strings, including wide characters.
- Object references - handles to CORBA objects.

Complex

- Structures and unions, which parallel C/C++ structs and unions.
- Arrays - runtime fixed-length, possibly multi-dimensional stores of multiple data elements.
- Sequences - runtime variable-length stores of multiple data elements.
- "any" values store values of any data type, along with the type information necessary to make use of that data type.
- The upcoming CORBA 2.3 specification will add "object by value" capabilities, which will allow expressing much more complex data relationships such as graphs.

CORBA objects can take advantage of a wide range of features. Interfaces can inherit operations from other interfaces, and exceptions can be raised if errors occur. In addition, the interfaces provided by the ORB and the CORBA services provide utility interfaces for applications along with additional “cool features”.

The ORB is the library used by an application to take care of the low-level CORBA implementation details, such as talking the IIOP protocol to objects over the network, and keeping track of object implementations that the application wishes to expose via CORBA.

The CORBA services cover functionality areas such as object trading, time synchronization, transactions, event channels, and security. Providing a detailed tour of the ORB and CORBA services interfaces is beyond the scope of this document, but the reader is encouraged to take advantage of the OMG’s web site for more information on these interfaces.

The overall result of all the CORBA features is a standardized mechanism of abstraction. In CORBA implementations, this allows benefits of network transparency and programming language transparency to be realized. Computing resources can be more effectively used, and language-independent object reuse becomes practical. As an generalized inter-program communications mechanism, the value of CORBA is unparalleled.

ORBit

In the early stages of the GNOME project, the MICO ORB was used. While some find it, it caused many problems because of its large resource footprint, long compile times, and C++ specificity. This significantly retarded CORBA integration in GNOME. Therefore, the ORBit project was born to create a thin, fast CORBA implementation with C language bindings. In its present state ORBit stably fulfills all the needs of GNOME.

ORBit has two main advantages. The first is its use of the C language. This is both salvation for fans of the C language, and good news for proponents of other languages. C has the advantage of being the lowest common denominator of almost all other language implementations. Efforts are underway to produce C++ bindings, Python, Eiffel, Ada, and Perl bindings.

A second advantage is the resource friendliness of ORBit. In preliminary performance tests, ORBit came out ahead of `ILU`, `mico`, `omniORB`, and `TAO`, other well-known free ORB's. The test used to measure performance was ten thousand invocations of the "echoString" operation, part of the ORBit test suite. The IDL for this operation is:

```
interface Echo {
  Echo echoString(in string astring, out long anum);
};
```

Invoking the operation sends the object implementation a string, and the object implementation returns an object reference (to itself) and a random number in return.

This is not by any means a comprehensive measure of performance one will receive on a particular application, but it does test the overhead of the ORB in general, as well as the speed of object reference creation, basic marshaling/de-marshaling, and network traffic scheduling. Table 1 shows the performance of these ten thousand operation invocations (as timed by the 'time' command, average of best 3 of 6 runs), with the client program running on the same system as the server. These took place on a K6-2 350 running Linux 2.2.2ac1.

Table 1. Overhead and Speed

ORB	Time
Mico 2.2.3	22.48 seconds
ILU 2.0alpha13	9.17 seconds
TAO 0.2.42	8.81 seconds
omniORB 2.7.0	5.53 seconds
ORBit 0.4.2	2.93 seconds

ORBit supports collocated servers, aka in-process servers. This means that if the object server and the object "client" are located in the same address space, and the object server meets a few basic conditions, ORBit automatically skips sending the data over the network, and runs the object server's implementation function directly, causing performance to increase dramatically.

ORBit also uses less memory than the other ORBs. The *top* status display, for the "echo-server" process used in the above performance test, indicated the sizes for each ORB represented in Table 2

Table 2. Memory Usage

ORB	Size(K)	Unshared(K)
ILU	1032	144
ORBit	844	196

ORB	Size(K)	Unshared(K)
omniORB	1124	272
TAO	2876	460
mico	2948	584

In addition to these two main advantages, ORBit also has been tailored to make it easy to use with GUI toolkits such as GTK+. It has hooks to allow custom security—authentication. Other expositions of the ORB internals and extensions allow libraries and applications to leverage ORBit to the fullest advantage. Sample implementations of the event and name services are included in ORBit to make it easy to build more complex applications.

Looking ahead in ORBit development, one can for-see a rewrite of the performance-critical modules, compliance with the upcoming CORBA 2.3 standard, more CORBA services, and documentation. The development tree is available to everyone via anonymous CVS, an ORBit mailing list facilitates interesting discussions, and contributions are welcomed. More information on all of these is available at the ORBit web page.

The GNOME CORBA Framework

Even though the CORBA specification, and the ORBit implementation thereof, provide much functionality for GNOME, there are still pieces of a complete CORBA framework that are missing from those components. One of the biggest gaps in the CORBA specification is the lack of a standard way to start a server program for a particular object.

The implementation repository has sometimes been touted as the solution to this problem. The problem with the impl repo is that none of its interfaces are specified by the CORBA spec, meaning that one would have to be tied to a specific CORBA implementation in order to access the impl repo. When used via persistent object references, one must also have an implementation repository daemon running at all times on a well-known port. The impl repo also does not typically allow specifying server activation options, nor does it allow activating a server from a shared library.

Enter GOAD. The GNOME Object Activation Directory stores information on all the programs on a system that can act as an object server. The information stored in this directory includes the server type (shared library, executable, or "relay"), a listing of the CORBA interfaces supported by the server, a unique "server_id" that identifies the particular implementation, a human-readable description, and the filename of the executable or shared library.

The “executable” activation method of GOAD executes a program and returns an object reference to the object server that the program provides. Using the GNOME plug-in specification, the “shared library” activation method loads the object server from a shared library directly into the address space of the current program. This takes advantage of ORBit’s in-process operation invocation abilities to make lightweight CORBA objects practical.

The “relay” activation method adds extensibility to the GOAD directory. It acquires an object reference to the object server by contacting a CORBA “relay” server which can do further processing on the activation request. This may become useful, for example, when large numbers of GOAD objects need to be load-balanced across multiple computer systems.

In addition to the above methods, GOAD also can check if a server for the object is already running. GOAD uses the CORBA name service as a central registry of running object servers. The information necessary to contact the name service is stored as a property on the root window of the X display of the GNOME session, contributing to GNOME's goal of network transparency. The approach of using existing object servers whenever possible allows multiple clients to share the same server, reducing resource usage.

When activating an object server, the client also has the ability to specify how it would prefer that the object be activated and other activation details. The result is a system of object server activation that allows maximum flexibility for object clients, object servers, and system administrators alike.

In addition to GOAD, the GNOME library that stores CORBA-related functionality (libgnorba) also has the behind-the-scenes routines for handling CORBA request authentication and ORB integration with the GTK main loop. Request authentication is handled on a per-display basis by passing a secret cookie along with every CORBA operation invocation. This is all completely transparent to the GNOME application, allowing future use of new security mechanisms such as SSL or the CORBA security service.

Besides GOAD and other libgnorba functionality, the GNOME CORBA framework also defines a number of standard interfaces that programs commonly implement, including generic factory and reference-counted object interfaces. These are implemented by many of the CORBA servers in the various GNOME applications, such as the panel, file manager, and spreadsheet.

CORBA For You

When introduced to CORBA and the advantages that it can bring to programs, two different reactions are common. The first common reaction is to dismiss it altogether as a buzzword or a bloat-producing phenomenon. This objection can be valid. CORBA can be misused to produced bloated software. CORBA definitely is not the silver bullet that will reduce software development and maintainance time to zero. The problem with this view is that it invalidates all the benefits of CORBA—its network transparency, programming language independence, and standardization.

On the other hand are those who take up the CORBA cause as if it were the end to all means. Some software projects have taken to exposing the C++ string classes via CORBA. This view is equally dangerous, because it is bound to make unnecessary demands on resources, and stands to add a fair amount of bloat and slowdown to the application for no good reason.

How, then, should a developer approach CORBA use in the small to medium size programs that are common in the free software world, what are the mistakes that CORBA utilizers sometimes make, and how can CORBA be used effectively in applications?

Possible Steps to Determining an Application CORBA Interface

1. Determine Usefulness to Others.

   Your nethack client may be the most wonderful program on the face of the earth, but unless it implements functionality that many other applications can make use of, then you probably do not need to "CORBAize" it.

Coordination with other applications is a common reason for creating a CORBA interface to a program. The GNOME panel exports its ability to hold applets via the `GNOME::Panel` CORBA interface, and similarly, each applet allows the panel to talk to it via the `GNOME::Applet` interface.

Another common reason for creating a CORBA interface to a program is to allow control from scripts. Nethack players may very well want to be able to write perl scripts that

2. Write it out.

   Once you understand what interface you want your application to offer to the world, codify its data types and operations into an IDL file.

3. Genericize.

   Look at the set of operations and data types just defined, and see if any of them are specific to your implementation. If they are, try to rework the interface so that it could comfortably be implemented by other programs of the same genre. Also try to split functionality

   Take this example of a possible interface to a text editor:

```
interface MyEditor {
  typedef long FileID;
  FileID open_file(in string filename);
  void close_file(in FileID fid);
  void save_file(in FileID fid);
};
```

   This interface has no editor-specific functionality; it could just as well be called the "FileManipulator" interface and be used by a spreadsheet.

4. Normalize.

   Often people will create an interface that contains implicit objects. In the above "MyEditor" example, the "FileID" data type is acting as an object handle, even though it is just an integer.

   Given this information, we could rework the "MyEditor" interface to come up with the following IDL:

```
interface ManipulatedFile {
  void close();
  void save();
};
interface FileManipulator {
  ManipulatedFile
   open_file(in string filename);
  ManipulatedFile new_file();
};
```

   This process may be viewed as similar to the process of normalizing relational databases.

5. Optimize.

   Have others review your design. Use your knowledge of how CORBA is implemented, your good judgment as a software developer, and the information below to improve on your original design. Look at the existing GNOME CORBA interfaces, and see if using one of them would be more advantageous.

Common CORBA Mistakes

Confusing CORBA with program-local object systems

A useful analogy here would be "Store strings inside string objects provided by the programming languages. Consider allowing access to catalogues of guitar strings via CORBA objects." For single-language objects that do not hold data that specifically needs to be accessed from other programs, CORBA object servers hold no advantage over traditional object implementations in shared libraries. For data that of and by itself can be useful to other programs, CORBA object servers are a good way of making this availability happen.

Confusion implementing with sub-classing

The common method of providing a new implementation of a given programming language object is to subclass that object and write new routines to override the inherited operation implementations. CORBA inheritance is for interfaces only.

Sub-classing an object just to denote a new implementation makes for a confusing class hierarchy. It also introduces the possible problem of handling implementation-specific extensions, which counteract the whole idea of having a generic interface.

Providing a new implementation of an existing interface does not require inheritance on the CORBA level. The flip side of this, of course, is that implementation inheritance in CORBA is not standardized, normally requiring the object implementor to re-implement all the operations on an object.

Confusing CORBA operation invocations with function calls

They look like normal function calls. They act like normal function calls. Beware! They often are much slower than normal function calls. If you find your application doing a tight loop with a CORBA operation on data elements, instead try modeling your data structures so that one operation invocation can process all the data. Remember, it can take as long to perform thousand loop iterations of a local function call as it can to perform a single operation invocation on a remote object.

Utilizing CORBA Effectively

Keep interface exposition at a high level

Not only does exposing low-level interfaces cause increased dependence upon the internal organization of a software system, but it also means you have to put more code into exporting your interfaces, introducing the risk for more bugs and increasing bloat.

Take advantage of the "object by value" concept

The basic premise of the "object by value" concept is to transport the data to the program where it is needed and perform data manipulations there. For example, instead of a spreadsheet client making repeated calls on a remote spreadsheet object to have it change the colors of cells, the object-by-value approach would be for the client to request all relevant spreadsheet data in an agreed-upon format (such as XML), manipulate that data, and send it back to the spreadsheet object server. The CORBA 2.3 specification includes a built-in "value" type to make using OBV much easier.

Use 'oneway' operations if you can

This is one of the few tips that can greatly increase the speed of some CORBA operation invocations without any real cost. For CORBA operations that do not return any values and do not raise any exceptions, specifying the ’oneway’ keyword in the IDL declaration will mean the ORB will not have to await a reply from the remote end before returning control to the calling program. Given the high latency of network links (relative to that of a intra-process function call), ’oneway’ operations can increase speed dramatically in a good ORB.

Mix other IPC mechanisms as appropriate

CORBA is not the ultimate IPC mechanism. For programs that need to share large amounts of data efficiently, shared memory systems (system-local or network-wide) can be much more efficient. Real-time synchronization can often better be accomplished via semaphores and message queues. A good example is the GNU Image Manipulation Program, which controls plug-ins via pipes, but shares the actual image data via shared memory. (In the future the GIMP may switch to using CORBA for plug-in control, but image data will still be exported via shared memory).

Conclusion

Use what works

CORBA is not yet the ultimate solution to software development problems. It *is* a valuable tool for allowing programs to share data. By becoming aware of the capabilities of CORBA technology and the ways it can most effectively be used, the software developer can be ready to make use of the most appropriate tool for the task.

More Information

CORBA (http://omg.org)

ORBit (http://www.labs.redhat.com/orbit/)

GNOME (http://www.gnome.org)

GNOME CORBA Whitepaper (http://gnome.ingsoc.io.nu/)