Using COM technologies on Unix platforms
Porting issues and strategies

Ashish Bansal
Software Engineer, Hughes Software Company
June 2000

COM/DCOM technologies make developing and distributing Windows components easy. But what can you do when the same components have to be developed on Unix platforms?

Contents:

COM components are easy to develop and help to speed up the application development process. But they are only available on Windows platforms. It would often be useful to port an application onto a Unix platform using COM. In this article, I will address some of the issues involved in porting COM to Unix, and suggest a few helpful strategies to approaching the process.

Options on Unix
The main function of COM is to support both the reuse and the distribution of components, although distribution is its main allure. Briefly, COM attempts to create a server object on request from a client. The client can then invoke the methods of the server object as if the methods belonged to the object. This article will address using COM only on local and remote out-of-process servers, although COM is capable of transcending these limits.

You can use the following approaches to provide functionality similar to COM, on Unix:

- Berkeley sockets can be used for communication between client and server. Their method and parameters can be packed into a packet and sent on the network.
- Remote procedure call (RPC), which is tightly integrated into Unix platforms, and provides a function similar to COM. However, it differs from COM because it is function call-oriented, and it has no concept of globally unique identifiers.

Sockets have the advantage of being available in all platforms because they form the core of network programming. However, developing with sockets is tedious because everything, particularly marshalling, must be implemented.

In this article, I use the RPC approach to porting, primarily for its simplicity. COM relies on RPC as its underlying mechanism for transport. This method produces implementations closer to the original COM implementations.

Analyzing a COM application
Let's analyze a COM application, and identify the various components that need to work before they are RPC-on-Unix compatible. (For a brief overview of programming with RPC, see RPC jumpstart). A COM component is specified in an *.idl* file. Listing 1 shows a sample of the type of file we will be working with.

Listing 1: test.idl file

```
import "oaidl.idl"; // for IUnknown and HRESULT
[
        uuid(D8C3D71D-36BA-11D4-88AC-0008C73BA039)
]
interface ITest : IUnknown
{
        HRESULT PrintNameOf(int number);
        HRESULT PrintSumOf(long a, long b,[out] long *sum, [out] long *tmp);
};
[
        uuid(D8C3D71E-36BA-11D4-88AC-0008C73BA039)
]
coclass Test
{
        [default] interface ITest;
};
```

In this example, Test is the class that implements the ITest interface. Client and server side stubs are generated on passing through the *midl* under the common header file. The Windows code for using this component is shown in Listing 2.

Listing 2: Using *ITest*

```
// assuming CoInitialize has been called
ITest *pTest = NULL;
HRESULT hr=CoCreateInstance(CLSID_Test, NULL, CLSCTX_ALL, IID_IFoo, (void **)pTest);
If(SUCCEEDED(hr))
{
        // do our work
}
:
:
:
```

The aim of this article is to show how to keep the above files unchanged on Unix while enabling COM. To do this, we must write an *.x* file for the *.idl* file that will form the input for the *rpcgen* program, the equivalent of *midl* on Windows. In Listing 2, COM function calls, which are a part of the Windows library, are called to create the required component. These also must be developed on Unix. In the next section we will address the issue of function calls and suggest some solutions to the problem.

Unix equivalents
Let's take this one step at a time. The first issue is the IDL file. RPC is a function call-oriented interface. In the IDL file, the interface is defined in terms of an interface name and a procedure name. RPC only accepts single arguments, so multiple arguments must be packed into a structure according to the eXternal Data Representation (XDR). We can solve the IDL problem using the following approach.

RPC jumpstart
The function calls in the RPC equivalent can be named in the format <*interfacename*>_<*methodname*>. By this method, *PrintNameOf()* of *Itest* becomes *ITest_PrintNameOf()*. The input and output data members can be packed into a structure. The output specification file for RPC for *test.idl* is shown in Listing 3.

Listing 3: *test.x* file

```
struct out_values {
        long sum;
        long tmp;
};

struct in_values {
        long a;
        long b;
};
program CLSID_ITest {
        version IID_ITest {
        long ITEST_PRINTNAMEOF(long number)=1;
        struct out_values ITEST_PRINTSUMOF(struct in_values)=2;
   } = 0xD8C3D71D;
} = 0x28C3D71E;
```

A note on RPC jumpstart
RPC procedures are defined in an .x file, which is similar to an .idl file in COM. This specification file is passed through rpcgen program, which produces the client and server side stubs. Here's a sample .x file:

```
/* sample.x file */

program SAMP_PROG { version SAMP_VERS { long BIN_AGE(void) =1;

/* 1 is proc number*/

string STR_NAME(long) =2; } = 1;

/* version number 32-bit*/

} = 0x20000001;

/* program number 32-bit*/
```

When this file is passed through rpcgen program, three files are produced: sample.h, sample_svc.c, and sample_clnt.c. On the client side, sample.h and sample_clnt.c have to be compiled with the rest of the program. On the server side, another file, which actually implements the functions BIN_AGE and STR_NAME, has to be written and compiled with sample.h and sample_svc.c. Function call names are changed after passing through rpcgen. For example, BIN_AGE(void) becomes bin_age_1(void). Version numbers are appended to the function names. Also, multiple arguments cannot be passed; they must be packed into a structure. A specification called XDR exists for this.

A translation layer that will convert the method invocation to function calls now has to be written for both the client and the server sides. The code for the client side is shown in Listing 4.

Listing 4: Client side translation layer

```
typedef bool HRESULT; // for simplifying SUCCEEDED macro
Class IUnknown
{
        // declaration of Iunknown
}
class ITest : public IUnknown // this is an abstract base class
{
        // declaration of ITest
};
class Test : public ITest
{
public:
        Test() {
        if (ref == 0) cl = clnt_create("hostname", CLSID_ITEST, IID_ITEST, "udp");
                }
/* in place of hostname, localhost or the name of the machine where the
 * server is running is to be specified. Also, in place of udp, tcp could
 * be used as the transport protocol*/
        ~Test(){
                if (cl !=NULL) clnt_destroy(cl);
                cl = NULL;
                };
long AddRef() {
                ref++;
                return ref;
        };
        long Release() {
                ref--;
                if(ref == 0) delete this;
                return ref;
        };
        HRESULT PrintNameOf(long number)
        {
        return ( (itest_printnameof_3636713245(number) == 1) ? true : false);
                        // see sidebar
        };

        HRESULT PrintSumOf(long a, long b, long *sum, long *tmp) {
                struct in_values inval;
```

```
                struct out_values outval; // see Listing 3
                inval.a = a; inval.b = b;
                if( (outval = itest_printsumof_3636713245 (inval)) == NULL)
                                                //  see sidebar
                {
                        // some error func call was unsuccessful
                        return false;
                }
                else
                {
                        *sum = outval.sum;
                        *tmp = outval.tmp;
                        return true;
                }
        };
private:
        CLIENT *cl;
        static long   ref;
};
long Test::ref == 0;
```

The server side then has to be written to construct the proper instance of the object and to invoke the appropriate method. Use a static class or any other suitable mechanism.

We now have several files: the translation files, *rpcgen*-generated files, and source files with the implementation of the interface. All that remains now is to see how the components are created via COM function calls.

Let's take the basic COM function calls `CoInitialize()` and `CoUninitialize()` that initialize the COM library in Windows. This kind of function does not have to be performed in Unix, making it a *null function* (see Listing 5).

Listing 5: COM initialization and deinitialization functions

```
CoInitialize()
{
        return true;
}
CoUninitialize()
{
        return true;
}
```

Listing 6 shows `CoCreateInstance()`, which creates the component on the server side.

Listing 6: Creating an object

```
CoCreateInstance(clsid, /*don't care*, /* dont care*/, iid, void** ppv)
{
        switch (clsid)
        {
                case CLSID_ITest:
                        Test *tst = new Test();
                        Tst->AddRef();
/* Release is the responsibility of user */
                        *ppv = (ITest*) tst;
                        break;
                case /* similar cases for other classes in the system*/
                default:
                        /* do some error reporting as clsid was not found*/
        }
};
```

This treatment remains almost the same for every COM application. Consider the COM component's reference counting mechanism, which must be implemented on the client side. The code for the reference counting will be the same as the code in Listing 4 for the translation layer that converted the method invocation to function calls. The client is destroyed through an RPC call as soon as it has been detected that the instances of a particular interface have reached zero.

Putting it all together

Now that we have all the individual components in place, let's see how they fit together. Here's a list of steps to compile the application. (The inputs are the IDL specification file and the implementation of the interface. Note that the implementation of the interface must also be Unix compatible.)

1. Write the relate .x file from the IDL file.
2. Write the client and server side translation layers.
3. Pass the *.x* file through *rpcgen* to generate client and server side stubs (see RPC jumpstart).
4. Compile the main program, the client side translation layer, and the client side stubs generated by *rpcgen* to build the client.
5. Compile the server side translation layer, interface implementation, and server side stubs by *rpcgen* to build the server. These stubs have a default main.
6. Voila! COM on Unix is ready!

We have now taken a basic, introductory look at COM. Although this analysis was not in-depth, it offers a comprehensive strategic approach. You can now easily develop other advanced features of COM by enhancing these methods.

Resources

- Read W.R. Stevens' reference book, UNIX Network Programming (Prentice Hall, 1997); ISBN: 013490012X
- Visit MSDN's Web site for the informative and lighthearted Dr. GUI series: Dr. GUI on Components, COM, and ATL, MSDN, July 1998
- Use Don Box's Essential COM to learn more about COM technologies (Addison-Wesley, 1997); ISBN: 0201634465

About the author
Ashish Bansal has a bachelor's degree in Electronics and Communications Engineering from the Institute of Technology, Banaras Hindu University, Varanasi, India. He is currently working as a software engineer with Hughes Software Systems. He can be reached at abansal@ieee.org.