The following article (and its companion, "Understanding DCOM, Part II") are excerpt from the forthcoming book entitled "Understanding DCOM," (copyright © 1999 by Prentice Hall) due out later this year from Prentice Hall.

Understanding DCOM
Part I

by William Rubin & Marshall Brain

For many people, learning COM and DCOM is tough. You know that learning COM is the right thing to do - you hear constant hype and you know that many of Microsoft's products and programmer tools are based on COM, so it is obviously something that is important. But you also know that COM is really hard. You may have already tried to learn COM once, or maybe even several times. You may have slid through a couple of books, played with some wizards, etc... But it just doesn't make any sense. Everything seems extremely complicated and much harder than it needs to be. There's also the vocabulary: "marshalling", "apartment threads", "singleton objects" and so on. What is this?

The purpose of this set of tutorials is to help you to quickly understand what is going on in the world of DCOM so that you can create COM clients and servers easily. We do that starting at the beginning and laying things out for you simply and in the right order. By the time you finish these tutorials you will understand all of the basic concepts driving DCOM and you will be able to proceed quickly to learn the rest. You will be amazed at how easy DCOM can be once you get a good start.

ONE: The Basics of DCOM

Understanding how COM works can be intimidating at first. One reason for this intimidation is the fact that COM uses its own vocabulary. A second reason is that COM contains a number of new concepts. One of the easiest ways to master the vocabulary and concepts is to compare COM objects to normal C++ objects to identify the similarities and differences. You can also map unfamiliar concepts from COM into the standard C++ model that you already understand. This will give you a comfortable starting point, from which we'll look at COM's fundamental concepts. Once we have done this, the example presented in the following sections will be extremely easy to understand.

Classes and Objects

Imagine that you have created a simple class in C++ called xxx. It has several member functions, named MethodA, MethodB, and MethodC. Each member function accepts parameters and returns a result. The class declaration is shown here:

```
class xxx {
public:
    int MethodA(int a);
    int MethodB(float b);
    float MethodC(float c);
};
```

The class declaration itself describes the class. When you need to use the class, you must create an instance of the object. Instantiations are the actual objects; classes are just the definitions. Each object is created either as a variable (local or global) or it is created dynamically using the new statement. The new statement dynamically creates the variable on the heap and returns a pointer to it. When you call member functions, you do so by dereferencing the pointer. For example:

```
xxx *px;            // pointer to xxx class
px = new xxx;       // create object on heap
px->MethodA(1);     // call method
delete px;          // free object
```

It is important for you to understand and recognize that COM follows this same objected oriented model. COM has classes, member functions and instantiations just like C++ objects do. Although you never call new on a COM object, you must still create it in memory. You access COM objects with pointers, and you must de-allocate them when you are finished.

When we write COM code, we won't be using new and delete. Although we're going to use C++ as our language, we'll have a whole new syntax. COM is implemented by calls to the COM API, which provides functions that create and destroy COM objects. Here's an example COM program written in pseudo-COM code.

```
ixx *pi                      // pointer to xxx COM interface
CoCreateInstance(,,,,&pi)    // create interface
pi->MethodA();               // call method
pi->Release();               // free interface
```

In this example, we'll call class ixx an "interface". The variable pi is a pointer to the interface. The method CoCreateInstance creates an instance of type ixx. This interface pointer is used to make method calls. Release deletes the interface.

I've purposely omitted the parameters to CoCreateInstance. I did this so as not to obscure the basic simplicity of the program. CoCreateInstance takes a number of arguments, all of which need some more detailed coverage. For now, let's take a step back and look at the bigger issues with COM.

How COM Is Different

COM is not C++, and for good reason. COM objects are somewhat more complicated then their C++ brethren. Most of this complication is necessary because of network considerations. There are four basic factors dictating the design of COM:

- C++ objects always run in the same process space. COM objects can run across processes or across computers.

- COM methods can be called across a network.

- C++ method names must be unique in a given process space. COM object names must be unique throughout the world.

- COM servers may be written in a variety of different languages and on entirely different operating systems, while C++ objects are always written in C++.

Let's look at what these differences between COM and C++ mean to you as a programmer.

COM Can Run Across Processes

In COM, you as the programmer are allowed to create objects in other processes, and on any machine on the network. That does not mean that you will always do it (in many cases you won't). However, the possibility means that you can't create a COM object using the normal C++ new statement , and calling its methods with local procedure calls won't suffice.

To create a COM object, some executing entity (an EXE or a Service) will have to perform remote memory allocation and object creation. This is a very complex task. By remote, we mean in another process or on another process. This problem is solved by creating a concept called a COM server. This other entity will have to maintain tight communication with the client.

COM Methods Can Be Called Across a Network

If you have access to a machine on the network, and if a COM server for the object you want to use has been installed on that machine, then you can create the COM object on that computer. Of course, you must the proper privileges, and everything has to be set-up correctly on the other computer.

Since your COM object will not necessarily be on the local machine, you need a good way to "point to" it, even though its memory is somewhere else. Technically, there is no way to do this. In practice, it can be simulated by introducing a whole new level of objects. One of the ways COM does this is with a concept called a proxy/stub. We'll discuss proxy/stubs in some detail later.

Another important issue is passing data between the COM client and it's COM server. When data is passed between processes, threads, or over a network, it is called "Marshalling". Again, the proxy/stub takes care of the marshalling for you. COM can also marshal data for certain types of interface using Type Libraries and the Automation marshaller. The Automation marshaller does not need to be specifically built for each COM server.

COM Objects Must Be Unique Throughout the World

The whole world? Come on! This may seem like an exaggeration at first, but consider the Internet to be a worldwide network. Even if you're working on a single computer, COM must handle the possibility. Uniqueness is the issue. In C++ all classes are handled unequivocally by the compiler. The compiler can see the class definition for every class used in a program and match up all references to it to make sure they conform to the class exactly. The compiler can also guarantee that there is only one class of a given name. In COM there must be a good way to get a similarly unequivocal match. COM must guarantee that there will only be one object of a given name even though the number of objects available on a worldwide network is huge. This problem is solved by creating a concept called a GUID.

COM is Language Indpendent

COM servers may be written with a different language and an entirely different operating system. COM objects have the capability of being remotely accessible. That means they may be in a different thread, process, or even on a different computer. The other computer may even be running under a different operating system. There needs to be a good way to transmit parameters over the network to objects on other machines. This problem is solved by creating a new way to carefully specify the interface between the client and server. There is also a new compiler called MIDL (Microsoft® Interface Definition Language). This compiler makes it possible to generically specify the interface between the server and client. MIDL defines COM objects, interfaces, methods and parameters.

COM Vocabulary

One of the problems we're going to have is keeping track of two sets of terminology. You're probably already familiar with C++ and some Object Oriented terminology. This table provides a rough equivalency between COM and conventional terminology.

Concept	Conventional (C++/OOP)	COM
Client	A program that request services from a server.	A program that calls COM methods.
Server	A program that "serves" other programs.	A program that makes COM objects available to a COM client.
Interface	None.	A pointer to a group of functions that are called through COM.
Class	A data type. Defines a group of methods and data that are used together.	The definition of an object that implements one or more COM interfaces. Also, "coclass".
Object	An instance of a class.	The instance of a coclass.
Marshalling	None.	Moving data between client and server.

You'll notice the concepts of Interface and Marshalling don't translate well into the C++ model. The closest thing to an interface in C++ is the export definitions of a DLL. DLL's do many of the same things as COM when dealing with a tightly coupled (in-process) COM server. Marshalling in C++ is almost entirely manual. If you're trying to copy data between processes and computers, you'll have to write the code using some sort of inter-process communication. You have several choices, including sockets, the clipboard, and mailslots.

The Interface

Thus far, we've been using the word "interface" pretty loosely. My dictionary (1947 American College Dictionary) defines an interface as follows:

"Interface, n. a surface regarded as the common boundary of two bodies or surfaces"

That's actually a useful general description. In COM "interface" has a very specific meaning. COM interfaces are a completely new concept, not available in C++. The concept of an interface is initially hard to understand for many people. An interface is a ghostlike entity that never has a concrete existence. It's sort of like an abstract class - but not exactly.

At its simplest, an interface is nothing but a named collection of functions. In C++, a class (using this terminology) is allowed only one interface. The member functions of that interface are all the public member functions of the class. In other words, the interface is the publicly visible part of the class. In C++ there is almost no distinction between an interface and a class. Here's an example C++ class:

```
class yyy {
public:
    int DoThis();
private:
    void Helper1();
    int count;
    int x,y,z;
};
```

When someone tries to use this class, they only have access to the public members. (For the moment we're ignoring protected members and inheritance.) They can't call Helper1, or use any of the private variables. To the consumer of this class, the definition looks like this:

```
class yyy {
    int DoThis();
};
```

This public subset of the class is the 'interface' to the outside world. Essentially the interface hides the guts of the class from the consumer.

This C++ analogy only goes so far. A COM interface is not a C++ class. COM interfaces and classes have their own special set of rules and conventions.

COM allows a coclass (COM class) to have multiple interfaces, each interface having its own name and its own collection of functions. The reason for this feature is to allow for more complex and functional objects. This is another concept that is alien to C++. (Perhaps multiple interfaces could be envisioned as a union of two class definitions - something that isn't allowed in C++.)

Interfaces Isolate the Client From the Server

One of the cardinal rules of COM is that you can only access a COM object through an interface. The client program is completely isolated from the server's implementation through interfaces. This is an extremely important point.

The client program knows nothing about the COM object or C++ class that implements the COM object. All it can see is the interface. The interface is like window into the COM object. The interface designer allows the client to see only those parts of the object that he or she wishes to expose. The figure above illustrates how all client access to the COM object is funneled through the interface.
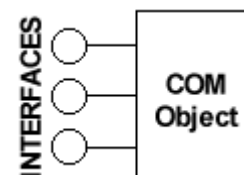
The notation used here, a small circle connected by a stick, is the conventional way to draw a COM interface. There are many important rules associated with interfaces. While critical for understanding the details how COM works, we can leave them until later. For now, we'll concentrate on the broad concepts of interfaces.

Imagining a COM Interface

Here's another way to visualize an interface. In this section we'll present a COM interface without any of the C++ baggage. We'll try to look at an interface in its abstract form. Imagine a "car" object. All "car" objects that you are familiar with in the real world have a "driving" interface that allows you to direct the car left and right and also to speed the car up and slow it down. The member functions for the driving interface might be "left", "right", "faster", "slower", "forward" and "reverse". Many cars also happen to have a "radio" interface as well, if they have a radio installed. The functions for the radio interface might be "on", "off", "louder", "softer", "next station" and "previous station".

Driving	Radio
Left()	On()
Right()	Off()
Slower()	Louder()
Faster()	Softer()
Forward()	NextStation()
Reverse()	PrevStation()

There are many kinds of cars, but not all of them have radios. Therefore, they do not implement the radio interface, although they do support the driving interface. In all cars that do have radios the capabilities of the radio are the same. A person driving a car without a radio can still drive, but cannot hear music. In a car that does have a radio, the radio interface is available.

COM supports this same sort of model for COM classes. A COM object can support a collection of interfaces, each of which has a name. For COM objects that you create yourself, you will often define and use just a single COM interface. But many existing COM objects support multiple COM interfaces depending on the features they support.

Another important distinction is that the driving interface is not the car. The driving interface doesn't tell you anything about the brakes, or the wheels, or the engine of the car. You don't drive the engine for example, you use the faster and slower methods (accelerator and brakes) of the driving interface. You don't really care how the slower (brake) method is implemented, as long as the car slows down. Whether the car has hydraulic or air brakes isn't important.

Imagine a Component

When you're building a COM object, you are very concerned about how the interface works. The user of the interface however, shouldn't be concerned about its implementation. Like the brakes on a car, the user cares only that the interface works, not about the details behind the interface.

This isolation of interface and implementation is crucial for COM. By isolating the interface from it's implementation, we can build components. Components can be replaced and re-used. This both simplifies and multiplies the usefulness of the object.

What's in a name?

One important fact to recognize is that a named COM interface is unique. That is, a programmer is allowed to make an assumption in COM that if he accesses an interface of a specific name, the member functions and parameters of that interface will be exactly the same in all COM objects that implement the interface. So, following our example, the interfaces named "driving" and "radio" will have exactly the same member function signature in any COM object that implements them. If you want to change the member functions of an interface in any way, you have to create a new interface with a new name.

The Source of All Interfaces - IUnknown

Traditional explanations of COM start out with a thorough description of the IUnknown interface. IUnknown is the fundamental basis for all COM interfaces. Despite its importance, you don't need to know about IUnknown to understand the interface concept. The implementation of IUnknown is hidden by the higher level abstractions we'll be using to build our COM objects. Actually, paying too much attention to IUnknown can be confusing. Let's deal with it at a high level here so you understand the concepts.

IUnknown is like an abstract base class in C++. All COM interfaces must inherit from IUnknown. IUnknown handles the creation and management of the interface. The methods of IUnknown are used to create, reference count, and release a COM object. All COM interfaces implement these 3 methods and they are used internally by COM to manage interfaces. You will likely never call these 3 methods yourself.

A Typical COM Object

Now let's put all of these new concepts together and describe a typical COM object and a program that wants to access it. In the next section and the following chapters we will make this real by implementing the actual code for the object.

Imagine that you want to create the simplest possible COM object. This object will support a single interface, and that interface will contain a single function. The purpose of the function is also extremely simple - it beeps. When a programmer creates this COM object and calls the member function in the single interface the object supports, the machine on which the COM object exists will beep. Let's further imagine that you want to run this COM object on one machine, but call it from another over the network.

Here are the things you need to do to create this simple COM object:

• You need to create the COM object and give it a name. This object will be implemented inside a COM server that is aware of this object.

• You need to define the interface and give it a name.

• You need to define the function in the interface and give it a name.

• You'll need to install the COM server.

For this example, let's call the COM object Beeper, the interface IBeep and the function Beep. One problem you immediately run into in naming these objects is the fact that all machines in the COM universe

---

are allowed to support multiple COM servers, each containing one or more COM objects, with each COM object implementing one or more interfaces. These servers are created by a variety of programmers, and there is nothing to stop the programmers from choosing identical names. In the same way, COM objects are exposing one or more named interfaces, again created by multiple programmers who could randomly choose identical names. Something must be done to prevent name collision, or things could get very confusing. The concept of a GUID, or a Globally Unique IDentifier, solves the "how do we keep all of these names unique" problem.

How to be unique - the GUID

There are really only two definitive ways to ensure that a name is unique:

1. you register the names with some quasi-governmental organization.

2. you use a special algorithm that generates unique numbers that are guaranteed to be unique world-wide (no small task).

The first approach is how domain names are managed on the network. This approach has the problem that you must pay $50 to register a new name and it takes several weeks for registration to take effect.

The second approach is far cleaner for developers. If you can invent an algorithm that is guaranteed to create a unique name each time anyone on the planet calls it, the problem is solved. Actually, this problem was originally addressed by the Open Software Foundation (OSF). OSF came up with an algorithm that combines a network address, the time (in 100 nanosecond increments), and a counter. The result is a 128-bit number that is unique.

The number 2^{128} power is an extremely large number. You could identify each nanosecond since the beginning of the universe - and still have 39 bits left over. OSF called this the UUID, for Universally Unique Identifier. Microsoft uses this same algorithm for the COM naming standard. In COM Microsoft decided to re-christen it as a Globally Unique Identifier.

The convention for writing GUID's is in hexadecimal. Case isn't important. A typical GUID looks like this:

"50709330-F93A-11D0-BCE4-204C4F4F5020"

Since there is no standard 128-bit data type in C++, we use a structure. Although the GUID structure consists of four different fields, you'll probably never need to manipulate its members. The structure is always used in its entirety.

```
typedef struct _GUID
{
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
} GUID;
```

The common pronunciation of GUID is "gwid", so it sounds like 'squid'. Some people prefer the more awkward pronunciation of "goo-wid" (sounds like druid).

GUIDs are generated by a program called GUIDGEN. In GUIDGEN you push a button to generate a new GUID. You are guaranteed that each GUID you generate will be unique, no matter how many GUIDs you generate, and how many people on the planet generate them. This can work because of the following

assumption: all machines on the Internet have, by definition, a unique address. Therefore, your machine must be on the network in order for GUIDGEN to work to it's full potential. Actually, if you don't have a network address GUIDGEN will fake one, but you reduce the probability of uniqueness.

Both COM objects and COM interfaces have GUIDs to identify them. So the name "Beeper" that we choose for our object would actually be irrelevant. The object is named by its GUID. We would call the object's GUID the class ID for the object. We could then use a #define or a const to relate the name Beeper to the GUID so that we don't have 128-bit values floating throughout the code. In the same way the interface would have a GUID. Note that many different COM objects created by many different programmers might support the same IBeep interface, and *they would all use the same GUID to name it. If it is not the same GUID, then as far as COM is concerned it is a different interface. The GUID is the name.*

A COM Server

The COM server is the program that implements COM interfaces and classes. COM Servers come in three basic configurations.

- In-process, or DLL servers

- Stand-alone EXE servers

- Windows NT based services.

COM objects are the same regardless of the type of server. The COM interfaces and coclasses(***) don't care what type of server is being used. To the client program, the type of server is almost entirely transparent. Writing the actual server however, can be significantly different for each configuration:

- In-Process servers are implemented as Dynamic Link Libraries (DLL's). This means that the server is dynamically loaded into your process at run-time. The COM server becomes part of your application, and COM operations are performed within application threads. Traditionally this is how many COM objects have been implemented because performance is fantastic - there is minimal overhead for a COM function call but you get all of the design and reuse advantages of COM. COM automatically handles the loading and unloading of the DLL.

- An out-of-process server has a more clear-cut distinction between the client and server. This type of server runs as a separate executable (EXE) program, and therefore in a private process space. The starting and stopping of the EXE server is handled by the Windows Service Control Manager (SCM). Calls to COM interfaces are handled through inter-process communication mechanisms. The server can be running on the local computer, or on a remote computer. If the server is on a remote computer, we refer to this as "Distributed COM", or DCOM.

- Windows NT offers the concept of a service. A service is a program that is automatically managed by Windows NT, and is not associated with the desktop user. This means services can start automatically at boot time and can run even if nobody is logged on to Windows NT. Services offer an excellent way to run COM server applications.

- There is a fourth type of server, called a "surrogate". This is essentially a program that allows an in-process server to run remotely. Surrogates are useful when making a DLL-based COM server available over the network.

Interactions Between Client and Server

In COM, the client program drives everything. Servers are entirely passive, only responding to requests. This means COM servers behave in a synchronous manner toward individual method calls from the client.

- The client program starts the server.

- The client requests COM objects and interfaces.

- The client originates all method calls to the server.

- The client releases server interfaces, allowing the server to shut down.

This distinction is important. There are ways to simulate calls going from server to client, but they are difficult to implement and fairly complex (They are called callbacks). In general, the server does nothing without a client request.

Here is a typical interaction between a COM client and server:

Client Request	Server Response
Requests access to a specific COM interface, specifying the COM class and interface (by GUID)	• Starts the server (if required). If it is an In-Process server, the DLL will be loaded. Executable servers will be run by the SCM. • Creates the requested COM object. • Creates an interface to the COM object. • Increments the reference count of active interfaces. • Returns the interface to the client.
Calls a method of the interface.	Executes the method on a COM object.
Release the interface	• Decrements the interfaces reference count. • If the reference count is zero, it may delete the COM object. • If there are no more active connections, shut down the server. Some servers do not shut themselves down.

If you're going to understand COM, you must take a client-centric approach.

Summary

We've tried to look at COM from several points of view. C++ is the native language of COM, but it's important to see beyond the similarities. COM has many analogues in C++, but it has important differences. COM offers a whole new way of communicating between clients and servers.

The interface is one of the most important COM concepts. All COM interactions go through interfaces, and they shape that interaction. Because interfaces don't have a direct C++ counterpart, they are sometimes difficult for people to grasp. We've also introduced the concept of the GUID. GUID's are ubiquitous in COM, and offer a great way to identify entities on a large network.

COM servers are merely the vehicles for delivering COM components. Everything is focused on the delivery of COM components to a client application. In the following chapters, we'll create a simple client and server application to demonstrate these concepts.

TWO: Understanding the Simplest COM Client

The most straightforward way to begin understanding COM is to look at it from the perspective of a client application. Ultimately, the goal of COM programming is to make useful objects available to client applications. Once you understand the client, then understanding servers becomes significantly easier. Keeping clients and servers straight can be confusing; and COM tends to make the picture more complex when you are first learning the details. Therefore, let's start with the simplest definition: A COM client is a program that uses COM to call methods on a COM server. A straightforward example of this client/server relationship would be a User Interface application (the client) that calls methods on another application (the server). If the User Interface application calls those methods using COM, then the user Interface application is, by definition, a COM client.

We are belaboring this point for good reason - the distinction between COM servers and clients can get (and often is) much more complex. Many times, the application client will be a COM server, and the application's server will be a COM client. It's quite common for an application to be both a COM client and server. In this chapter, we will keep the distinction as simple as possible and deal with a pure COM client.

Four Steps to Client Connectivity

A client programmer takes four basic steps when using COM to communicate with a server. Of course, real-life clients do many more things, but when you peel back the complexity, you'll always find these four steps at the core. In this section we will present COM at it's lowest level - using simple C++ calls.

Here is a summary of the steps we are going to perform:

3. Initialize the COM subsystem and close it when finished.

4. Query COM for a specific interfaces on a server.

5. Execute methods on the interface.

6. Release the interface.

For the sake of this example, we will assume an extremely simple COM server. We'll assume the server has already been written and save its description for the next tutorial.

The server has one interface called IBeep. That interface has just one method, called Beep. Beep takes one parameter: a duration. The goal in this section is to write the simplest COM client possible to attach to the server and call the Beep method.

Following is the C++ code that implements these four steps. This is a real, working COM client application.

```
#include "..\BeepServer\BeepServer.h"

// GUIDS defined in the server
const IID IID_IBeepObj =
 {0x89547ECD,0x36F1,0x11D2,{0x85,0xDA,0xD7,0x43,0xB2,0x32,0x69,0x28}};
```

```
    const CLSID CLSID_BeepObj =
     {0x89547ECE,0x36F1,0x11D2,{0x85,0xDA,0xD7,0x43,0xB2,0x32,0x69,0x28}};

    int main(int argc, char* argv[])
    {
        HRESULT hr;                        // COM error code
        IBeepObj *IBeep;                   // pointer to interface

        hr = CoInitialize(0);              // initialize COM
        if (SUCCEEDED(hr))                 // macro to check for success
        {
            hr = CoCreateInstance(
                    CLSID_BeepObj,         // COM class id
                    NULL,                  // outer unknown
                    CLSCTX_INPROC_SERVER,  // server INFO
                    IID_IBeepObj,          // interface id
                    (void**)&IBeep );      // pointer to interface

            if (SUCCEEDED(hr))
            {
                hr = IBeep->Beep(800);     // call method
                hr = IBeep->Release();     // release interface
            }
        }
        CoUninitialize();                  // close COM
        return 0;
    }
```

The header "BeepServer.h" is created when we compile the server. BeepServer is the in-process COM server we are going to write in the next section. Several header files are generated automatically by developer studio when compiling the server. This particular header file defines the interface IBeepObj. Compilation of the server code also generates the GUIDs seen at the top of this program. We've just pasted them in here from the server project.

Let's look at each of the 4 steps in detail.

Initializing the COM subsystem:

This is the easy step. The COM method we need is CoInitialize().

```
    CoInitialize(0);
```

This function takes one parameter and that parameter is always a zero - a legacy from its origins in OLE. The CoInitialize function initializes the COM library. You need to call this function before you do anything else. When we get into more sophisticated applications, we will be using the extended version, CoInitializeEx.

Call CoUninitialize() when you're completely finished with COM. This function de-allocates the COM library. I often include these calls in the InitInstance() and ExitInstance() functions of my MFC applications.

Most COM functions return an error code called an HRESULT. This error value contains several fields which define the severity, facility, and type of error. We use the SUCCEEDED macro because there are several different success codes that COM can return. It's not safe to just check for the normal success code (S_OK). We will discuss HRESULT's later in some detail.

Query COM for a Specific Interface

What a COM client is looking for are useful functions that it can call to accomplish its goals. In COM you access a set of useful functions through an interface. An interface, in its simplest form, is nothing but a collection of one or more related functions. When we "get" an interface from a COM server, we're really getting a pointer to a set of functions.

You can obtain an interface pointer by using the CoCreateInstance() function. This is an extremely powerful function that interacts with the COM subsystem to do the following:

- Locate the server.

- Start, load, or connect to the server.

- Create a COM object on the server.

- Return a pointer to an interface to the COM object.

There are two data types important to finding and accessing interfaces: CLSID and IID. Both of these types are Globally Unique ID's (GUID's). GUID's are used to uniquely identify all COM classes and interfaces.

In order to get a specific class and interface you need its GUID. There are many ways to get a GUID. Commonly we'll get the CLSID and IID from the header files in the server. In our example, we've defined the GUIDs with #define statements at the beginning of the source code. There are also facilities to look up GUIDs using the common name of the interface.

The function that gives us an interface pointer is CoCreateInstance.

```
hr = CoCreateInstance(
                CLSID_BeepObj,        // COM class id
                NULL,                 // outer unknown
                CLSCTX_INPROC_SERVER, // server INFO
                IID_IBeepObj,         // interface id
                (void**)&IBeep );     // pointer to interface
```

The first parameter is a GUID that uniquely specifies a COM class that the client wants to use. This GUID is the COM class identifier, or CLSID. Every COM class on the planet has its own unique CLSID. COM will use this ID to automatically locate a server that can create the requested COM object. Once the server is connected, it will create the object.

The second parameter is a pointer to what's called the 'outer unknown'. We're not using this parameter, so we pass in a NULL. The outer unknown will be important when we explore the concept known as "aggregation". Aggregation allows one interface to directly call another COM interface without the client knowing it's happening. Aggregation and containment are two methods used by interfaces to call other interfaces.

The third parameter defines the COM Class Context, or CLSCTX. This parameter controls the scope of the server. Depending on the value here, we control whether the server will be an In-Process Server, an EXE, or on a remote computer. The CLSCTX is a bit-mask, so you can combine several values. We're using CLSCTX_INPROC_SERVER - the server will run on our local computer and connect to the client as a DLL. We've chosen an In-Process server in this example because it is the easiest to implement.

Normally the client wouldn't care about how the server was implemented. In this case it would use the value CLSCTX_SERVER, which will use either a local or in-process server, whichever is available.

Next is the interface identifier, or IID. This is another GUID - this time identifying the interface we're requesting. The IID we request must be one supported by the COM class specified with the CLSID. Again, the value of the IID is usually provided either by a header file, or by looking it up using the interface name.

The last parameter is a pointer to an interface. CoCreateInstance() will create the requested class object and interface, and return a pointer to the interface. This parameter is the whole reason for the CoCreateInstance call. We can then use the interface pointer to call methods on the server.

Execute a Method on the Interface

CoCreateInstance() uses COM to create a pointer to the IBeep interface. We can pretend the interface is a pointer to a normal C++ class, but in reality it isn't. Actually, the interface pointer points to a structure called a VTABLE, which is a table of function addresses. We can use the -> operator to access the interface pointer.

Because our example uses an In-Process server, it will load into our process as a DLL. Regardless of the details of the interface object, the whole purpose of getting this interface was to call a method on the server.

```
hr = IBeep->Beep(800);
```

Beep() executes on the server - it will cause the computer to beep. There are a lot simpler ways to get a computer to beep. If we had a remote server, one which is running on another computer, that computer would beep.

Methods of an interface usually have parameters. These parameters must be of one of the types allowed by COM. There are many rules that control the parameters allowed for an interface. We will discuss these in detail in the section on MIDL, which is COM's interface definition tool.

Release the Interface

It's an axiom of C++ programming that everything that gets allocated should be de-allocated. Because we didn't create the interface with new, we can't remove it with delete. All COM interfaces have a method called Release() which disconnects the object and deletes it. Releasing an interface is important because it allows the server to clean up. If you create an interface with CoCreateInstance, you'll need to call Release().

Summary

In this chapter we've looked at a simple COM client. COM is a client driven system. Everything is oriented to making component objects easily available to the client. You should be impressed at the simplicity of the client program. The four steps defined here allow you to use a huge number of components, in a wide range of applications.

Some of the steps, such as CoInitialize() and CoUninitialize() are elementary. Some of the other steps don't make a lot of sense at first glance. It is important for you to understand, at a high level, all of the things that are going on in this code. The details will clarify themselves as we go through further examples.

Visual C++ Version 5 and 6 simplify the client program further by using 'smart pointers" and the #import directive. We've presented this example in a low level C++ format to better illustrate the concepts. We'll discuss smart pointers and imports in a later section.

In the next section, we'll build a simple in-process server to manage the IBeep interface. We'll get into the interesting details of interfaces and activation in later chapters.

THREE: Understanding a Simple DCOM Server

So far we've looked at how to use COM through a client application. To the client, the mechanics of COM programming are pretty simple. The client application asks the COM subsystem for a particular component, and it is magically delivered.

There's a lot of code required to make all this behind-the-scenes component management work. The actual implementation of the object requires a complex choreography of system components and standardized application modules. Even using MFC the task is complex. Most professional developers don't have the time to slog through this process. As soon as the COM standard was published, it was quickly clear that it wasn't practical for developers to write this code themselves.

When you look at the actual code required to implement COM, you realize that most of it is repetitive boilerplate. The traditional C++ approach to this type of complexity problem would be to create a COM class library. And in fact, the MFC OLE classes provide most of COMs features.

There are however, several reasons why MFC and OLE were not a good choice for COM components. With the introduction of ActiveX and Microsoft's Internet strategy, it was important for COM objects to be very compact and fast. ActiveX requires that COM objects be copied across the network fairly quickly. If you've worked much with MFC you'll know it is anything but compact (especially when statically linked). It just isn't practical to transmit huge MFC objects across a network.

Perhaps the biggest problem with the MFC/OLE approach to COM components is the complexity. OLE programming is difficult, and most programmers never get very far with it. The huge number of books about OLE is a testament to the fact that it is hard to use.

Because of the pain associated with OLE development, Microsoft created a new tool called ATL (Active Template Library). For COM programming, ATL is definitely the most practical tool to use at the present. In fact, using the ATL wizard makes writing COM servers quite easy if you don't have any interest in looking under the hood.

The examples here are built around ATL and the ATL Application Wizard. This chapter describes how to build an ATL based server and gives a summary of the code that the wizard generates.

One of the things that takes some getting used to about writing ATL servers is that they don't look like traditional programs. A COM server is really a collaboration between several disparate components:

- Your application
- The COM subsystem
- ATL template classes
- "IDL" code and MIDL Generated "C" headers and programs
- The system registry

It can be difficult to look at an ATL based COM application and see it as a unified whole. Even when you know what it's doing, there are still big chunks of the application that you can't see. Most of the real

server logic is hidden deep within the ATL header files. You won't find a single main() function that manages and controls the server. What you will find is a thin shell that makes calls to standard ATL objects.

In the following section we're going to put together all the pieces required to get the server running. First we will create the server using the ATL COM AppWizard. The second step will be to add a COM object and a Method. We'll write an In-Process server because it's one of the simpler COM servers to implement. An In-process server also avoids having to build a proxy and stub object.

Building a DLL Based (In-Process) COM Server

An In-Process server is a COM library that gets loaded into your program at run-time. In other words, it's a COM object in a Dynamic Link Library (DLL). A DLL isn't really a server in the traditional sense, because it loads directly into the client's address space. If you're familiar with DLLs, you already know a lot about how the COM object gets loaded and mapped into the calling program.

Normally a DLL is loaded when LoadLibrary() is called. In COM, you never explicitly call LoadLibrary(). Everything starts automatically when the client program calls CoCreateInstance(). One of the parameters to CoCreateInstance is the GUID of the COM class you want. When the server gets created at compile time, it registers all the COM objects it supports. When the client needs the object, COM locates the server DLL and automatically loads it. Once loaded, the DLL has a class factory to create the COM object.

CoCreateInstance() returns a pointer to the COM object, which is in turn used to call a method (in the example described here, the method is called Beep().) A nice feature of COM is that the DLL can be automatically unloaded when it's not needed. After the object is released and CoUninitialize() is called, FreeLibrary() will be called to unload the server DLL.

If you didn't follow all that, it's easier than it sounds. You don't have to know anything about DLL's to use COM. All you have to do is call CoCreateInstance(). One of the advatages of COM is that it hides these details so you don't have to worry about this type of issue.

There are advantages and disadvantages to In-process COM servers. If dynamic linking is an important part of your system design, you'll find that COM offers an excellent way to manage DLL's. Some experienced programmers write all their DLL's as In-process COM servers. COM handles all the chores involved with the loading, unloading, and exporting DLL functions and COM function calls have very little additional overhead.

Our main reason for selecting an In-process server is somewhat more prosaic: It makes the example simpler. We won't have to worry about starting remote servers (EXE or service) because our server is automatically loaded when needed. We also avoid building a proxy/stub DLL to do the marshalling.

Unfortunately, because the In-Process server is so tightly bound to our client, a number of the important "distributed" aspects of COM are not going to be exposed. A DLL server shares memory with it's client, whereas a distributed server would be much more removed from the client. The process of passing data between a distributed client and server is called marshaling. Marshaling imposes important limitations on COM's capabilities that we won't have to worry about with an in-proc server.

Creating the server using the ATL Wizard

We're going to create a very simple COM server in this example in order to eliminate clutter and help you to understand the fundamental principles behind COM very quickly. The server will only have one

method -- Beep(). All that this method will do is sound a single beep - not a very useful method. What we're really going to accomplish is to set up all the parts of a working server. Once the infrastructure is in place, adding methods to do something useful will be extremely straightforward.

The ATL AppWizard is an easy way to quickly generate a working COM server. The Wizard will allow us to select all the basic options, and will generate most of the code we need. Below is the step-by step process for creating the server. In this process we will call the server BeepServer. All COM servers must have at least one interface, and our interface will be called IBeepObj. You can name your COM interfaces almost anything you want, but you should always prefix them with an 'I' if you want to follow standard naming conventions.

NOTE: If you find the difference between a COM "Object" , "Class", and "Interface" confusing at this point, you're not alone. The terminology can be uncomfortable initially, especially for C++ programmers. The feelings of confusion will subside as you work through examples. The word "coclass" for COM class is used in most Microsoft documentation to distinguish a COM class from a normal C++ class.

Here are the steps for creating a new COM server with the ATL Wizard using Visual C++ version 6 (it looks nearly identical in version 5 as well):

1. First, create a new "ATL COM AppWizard" project. Select File/New from the main menu.

2. Select the "Projects" tab of the "New" dialog. Choose "ATL COM AppWizard" from the list of project types. Select the following options and press OK.

   • Project Name: BeepServer

   • Create New Workspace

   • Location: Your working directory.
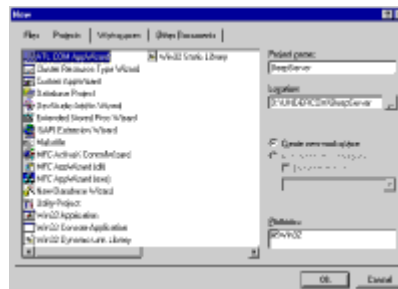
3. At the first AppWizard dialog we'll create a DLL based (In-process) server. Enter the following settings :

   • Dynamic Link Library

   • Don't allow merging proxy/stub code

   • Don't support MFC
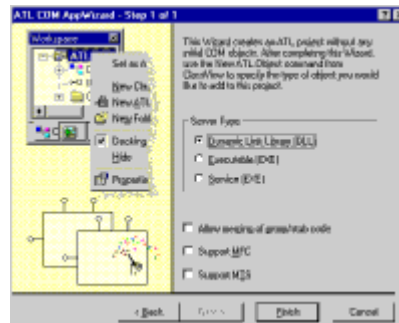
4. Press Finish.

The AppWizard creates a project with all the necessary files for a DLL-based COM server. Although this server will compile and run, it's just an empty shell. For it to be useful it will need a COM interface and the class to support the interface. We'll also have to write the methods in the interface.

Adding a COM Object and a Method

Now we'll proceed with the definition of the COM object, the interface, and the methods. This class is named BeepObj and has an interface called IBeepObj:

1. Look at the "Class View" tab. Initially it only has a single item in the list. Right click on "BeepServer Classes" item.

2. Select "New ATL Object…". This step can also be done from the main menu. Select the "New ATL Object" on the Insert menu item.
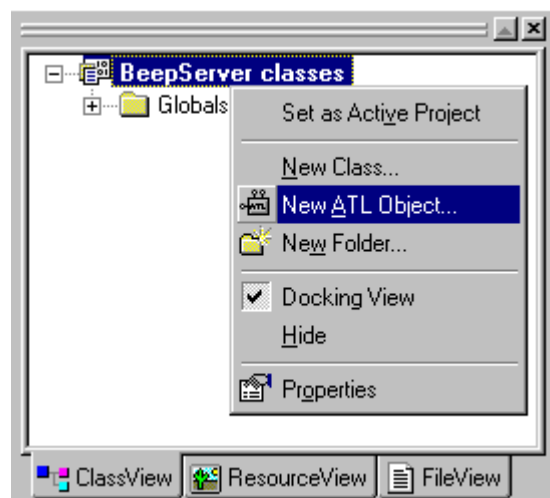
3. At the Object Wizard dialog select "Objects". Choose "Simple Object" and press Next.
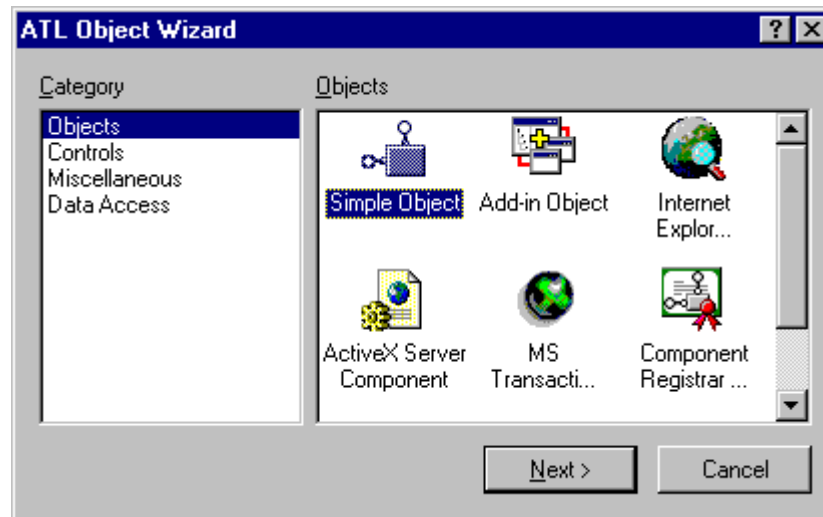
4. Choose the Names tab. Enter short name for the object: BeepObj. All the other selections are filled in automatically with standard names.
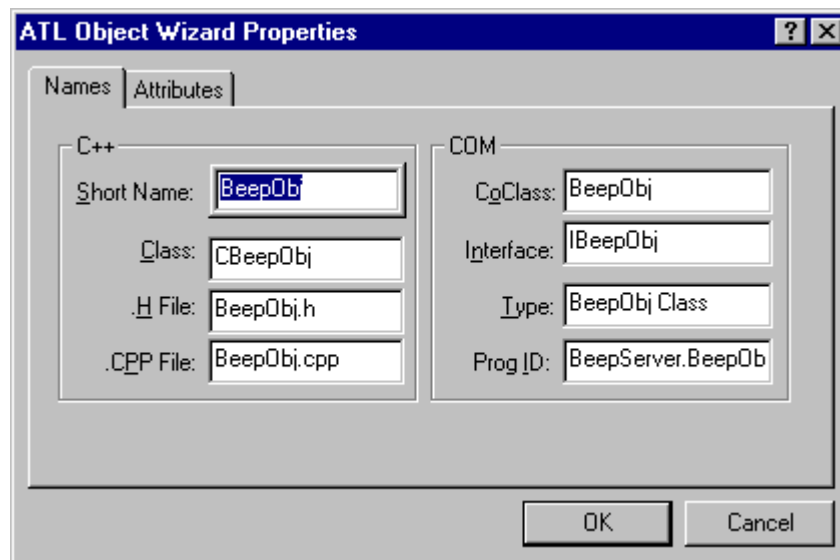
5. Press the "Attributes" tab and select: Apartment Threading, Custom Interface, No Aggregation. Actually, the aggregation isn't important for this server.

6. Press OK. This will create the Com Object.

## *Adding a Method to the Server.*

We have now created an empty COM object. As of yet, it's still a useless object because it doesn't do anything. We will create a simple method called Beep() which causes the system to beep once. Our COM method will call the Win32 API function ::Beep(), which does pretty much what you would expect.

1. Go to "Class View" tab. Select the IBeepObj interface. This interface is represented by a small icon that resembles a spoon.

2.
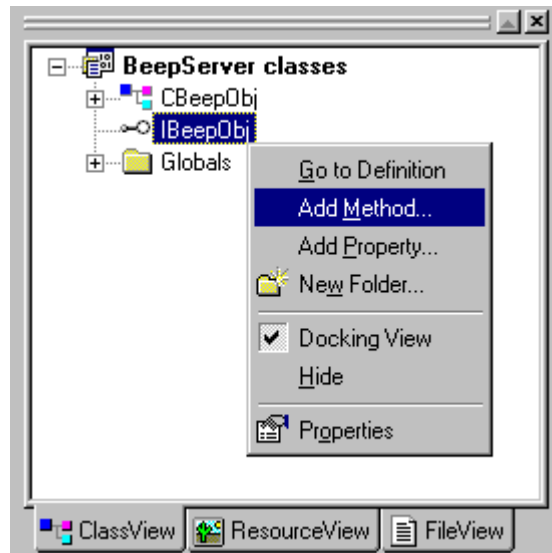
3. Right click the IBeepObj interface. Select "Add Method" from the menu.

4. At the "Add Method to Interface" dialog, enter the following and press OK. Add the method "Beep" and give it a single [in] parameter for the duration. This will be the length of the beep, in milliseconds.
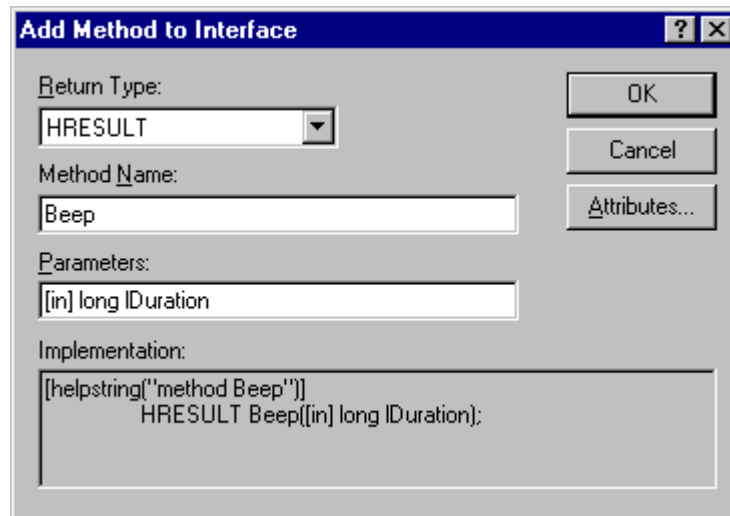
5. "Add Method" has created the MIDL definition of the method we defined. This definition is written in IDL, and describes the method to the MIDL compiler. If you want to see the IDL code, double click the "IBeepObj" interface at the "Class View" tab. This will open and display the file BeepServer.IDL. No changes are necessary to this file, but here's what our interface definition should look like.

```
interface IBeepObj : IUnknown
{
    [helpstring("method Beep")]
        HRESULT Beep([in] LONG duration);
};
```

The syntax of IDL is quite similar to C++. This line is the equivalent to a C++ function prototype. We will cover the syntax of IDL in the next issue.

6. Now we're going to write the C++ code for the method. The AppWizard has already written the empty shell of our C++ function, and has added it to the class definition in the header file (BeepServer.H).

    Open the source file BeepObj.CPP. Find the //TODO: line and add the call to the API Beep function. Modify the Beep() method as follows:

```
STDMETHODIMP CBeepObj::Beep(LONG duration)
{
    // TODO: Add your implementation code here
    ::Beep( 550, duration );
    return S_OK;
}
```

7. Save the files and build the project.

We now have a complete COM server. When the project finishes building, you should see the following messages:

```
------------------Configuration: BeepServer - Win32 Debug-------------------
Creating Type Library...
Microsoft (R) MIDL Compiler Version 5.01.0158
Copyright (c) Microsoft Corp 1991-1997. All rights reserved.
Processing D:\UnderCOM\BeepServer\BeepServer.idl
BeepServer.idl
Processing C:\Program Files\Microsoft Visual Studio\VC98\INCLUDE\oaidl.idl
oaidl.idl
.
.
Compiling resources...
Compiling...
StdAfx.cpp
Compiling...
BeepServer.cpp
BeepObj.cpp
Generating Code...
Linking...
    Creating library Debug/BeepServer.lib and object Debug/BeepServer.exp
Performing registration

BeepServer.dll - 0 error(s), 0 warning(s)
```

This means that the Developer Studio has completed the following steps:

•   Executed the MIDL compiler to generate code and type libraries

•   Compiled the source files

•   Linked the project to create BeepServer.DLL

•   Registered COM components

•   Registered the DLL with RegSvr32 so it will automatically load when needed.

Let's look at the project that we've created. While we've been clicking buttons, the AppWizard has been generating files. If you look at the "FileView" tab, the following files have been created:

Source File	Description
BeepServer.dsw	Project workspace
BeepServer.dsp	Project File
BeepServer.plg	Project log file. Contains detailed error information about project build.
BeepServer.cpp	DLL Main routines. Implementation of DLL Exports
BeepServer.h	MIDL generated file containing the definitions for the interfaces
BeepServer.def	Declares the standard DLL module parameters: DllCanUnload-Now, DllGetClassObject, DllUnregisterServer
BeepServer.idl	IDL source for BeepServer.dll. The IDL files define all the COM components.
BeepServer.rc	Resource file. The main resource here is IDR_BEEPDLLOBJ which defines the registry scripts used to load COM information into the registry.
Resource.h	Microsoft Developer Studio generated include file.
StdAfx.cpp	Source for precompiled header.
Stdafx.h	Standard header
BeepServer.tlb	Type Library generated by MIDL. This file is a binary description of COM interfaces and objects. The TypeLib is very useful as an alternative method of connecting a client.
BeepObj.cpp	Implementation of CBeepObj. This file contains all the actual C++ code for the methods in the COM BeepObj object.
BeepObj.h	Definition of BeepObj COM object.
BeepObj.rgs	Registry script used to register COM components in registry. Registration is automatic when the server project is built.
BeepServer_i.c	Contains the actual definitions of the IID's and CLSID's. This file is often included in cpp code.
	There are several other proxy/stub files that are generated by MIDL.

In just a few minutes, we have created a complete COM server application. Back in the days before wizards, writing a server would have taken hours. Of course the down-side of wizards is that we now have a

large block of code that we don't fully understand. In the next section we will look at the generated modules in detail, and then as a whole working application.

Summary

The server code was almost entirely generated by the ATL wizards. It provides a working implementation of the server. We examined a DLL based server, but the process is almost identical for all server types. This framework is an excellent way to quickly develop a server application because you don't have to know the myriad of details required to make it work.

Next issue, we will conclude this series with a look at In-Proc Servers and ATL code. We will also create a DCOM server.

Understanding DCOM

Part Two

by William Rubin & Marshall Brain

In the previous issue of the Developer's Journal, Part One of this series discussed the basics of
DCOM. We looked at how to create a simple DCOM server and a client for it. What you saw is
that the basic process is extremely simple - the ATL Wizard handles most of the details on the
server side, and 10 or so lines of code on the client side are all you need to activate the server.

In Part Two of this series, we will look at three realted topics. The first part, "Creating Your
Own COM Clients and Servers," takes what we learned last time and shows you exactly what
you need to do to incorporate a DCOM server in your own code. Then, in we take a quick tour
through the code generated by the ATL Wizard to demystify it a bit. In section three we dem-
onstrate the steps you must take to create a Distributed COM server (DCOM server) - a server
that can live elsewhere on the network and be activated over the network easily and transpar-
ently.

ONE: Creating your own COM Clients and Servers

Based on the previous trio of DCOM articles that appeared in Developer's Journal, you can see that it is
extremely easy to create COM clients and servers. In fact, you were probably stunned. Just a handful of lines
on both the client and server sides yields a complete COM application. You can now see why many develop-
ers use COM whenever they want to create a DLL - if only takes about 5 minutes to set up and in-proc COM
DLL and get it working. The purpose of this article is to discuss how you might create your own COM serv-
ers and use them in real applications you create. As you will recall, the client code previously presented was
a bit sparse. We will examine the standard steps you will take to create any server, and then look at the code
you need to embed in any client to activate the server properly.

Server Side

The ATL Wizard makes COM server creation extremely easy. The first step to creating any COM server
is coming up with one or more pieces of functionality that you want to separate from the main body of an
application's code. You probably want to separate the functionality in order to make it reusable across multi-
ple applications. But you may also want to do it because it allows a team of programmers to divide easily
into separate working groups, or because it makes code development or maintenance easier. Whatever the
reason, defining the functionality is the first step.

One thing that makes defining the boundary easy is the fact that a COM server can act almost identi-
cally to a normal C++ class. Like a class, you instantiate an instance of the COM class and then start calling
its methods. The sytax of COM instantiation and method calling is slightly different from the syntax in C++,

but the ideas are identical. If a COM server has only one interface, then it is, for all practical purposes, a class.

Once you have decided on the functionality and the methods that will be used to access it, you are ready to build your server. As we saw last month, there are 4 basic steps you must take to create a server:

1. Use the ATL Wizard to create the shell for your COM server. You choose whether you want the server to be a DLL, an EXE or a server.

2. Create a new COM object inside the server shell. You will choose the threading model. This creates the interface into which you can install your methods.

3. Add the methods to your object and declare their parameters.

4. Write the code for your methods.

Each of these tasks has been described in detail in the previous article entitled "Understanding a Simple COM Server."

After Part One of "Understanding DCOM" appeared, one question was frequently asked concerned threading models. Specifically, what is the difference between apartment-threaded and free-threaded COM objects? The easiest way to understand the difference is to think of apartment-threaded COM objects as single-threaded, while free-threaded COM objects as multi-threaded.

In apartment threading, method calls from multiple clients are serialized in the COM object on the server. That is, each individual method call completes its execution before the next method call can begin. Apartment-threaded COM objects are therefore inherently thread safe. Free threaded COM objects can have multiple method calls executing in the COM object at the same time. Each method call from each client runs on a different thread. In a free-threaded COM object you therefore have to pay attention to multi-threading issues such as synchronization.

Initially you will want to use apartment threading because it makes your life easier, but over time the move to free threading may have more advantages.

Client Side

The client presented Part One of this series has the benefits of clarity and compactness. However, it contains little error-checking code and that makes it nearly insufficient in a real application. Let's review that code, however, because it is so simple and it shows the exact steps that you must take to create a successful client:

```
void main()
{
        HRESULT hr;                     // COM error code
        IBeepDllObj *IBeep;             // pointer to interface

        hr = CoInitialize(0);        // initialize COM
        if (SUCCEEDED(hr))              // macro to check for success
        {
            hr = CoCreateInstance(
                clsid,                  // COM class id
                NULL,                   // outer unknown
                CLSCTX_INPROC_SERVER,   // server INFO
                iid,                    // interface id
                (void**)&IBeep );       // pointer to interface

            if (SUCCEEDED(hr))
            {
                hr = IBeep->Beep(800);  // call method
                hr = IBeep->Release();  // release interface
```

```
        }
        CoUninitialize();                 // close COM
}
```

The call to CoInitialize and CoCreateInstance initializes COM and gets a pointer to the necessary inter-face. Then you can call methods on the interface. When you are done calling methods you release the inter-face and call CoUninitialize to finish with COM. That's all there is to it.

That would be all there is to it, that is, if things always worked as planned. There are a number of things that can go wrong when a COM client tries to start a COM server. Some of the more common include:

- The client could not start COM

- The client could not locate the requested server

- The client could locate the requested server but it did not start properly

- The client could not find the requested interface

- The client could not find the requested function

- The client could find the requested function but it failed when called

- The client could not clean up properly

In order to track these potenital problems, you have to check things every step of the way by looking at HRESULT values. The above code does the checking, but it is difficult to tell what has gone wrong because the code is completely silent if an error occurs. The following function remedies that situation:

```
// This function displays detailed information contained in an HRESULT.
BOOL ShowStatus(HRESULT hr

{
    // construct a _com_error using the HRESULT
    _com_error e(hr);

    // The hr as a decimal number
    cout << "hr as decimal: " << hr << endl;

    // show the 1st 16 bits (SCODE)
    cout << "SCODE: " << HRESULT_CODE( hr ) << endl;

    // Show facility code as a decimal number
    cout << "Facility: " << HRESULT_FACILITY( hr ) << endl;

    // Show the severity bit
    cout << "Severity: " << HRESULT_SEVERITY( hr ) << endl;

    // Use the _com_error object to format a message string. This is
    // Much easier then using ::FormatMessage
    cout << "Message string: " << e.ErrorMessage() << endl;
    return TRUE;}return TRUE;
}
```

This function dismantles an HRESULT and prints all of its components, including the extremely useful English ErrorMessage value. You can call it any time with this function call:

```
// display HRESULT on screen
ShowStatus( hr );
```

To fully explore the different error modes of a simple COM program, the client in the following demon-stration code uses an MFC dialog application to let you control a number of possible errors and see the effect they have on the HRESULT. When the client runs it will look like this:
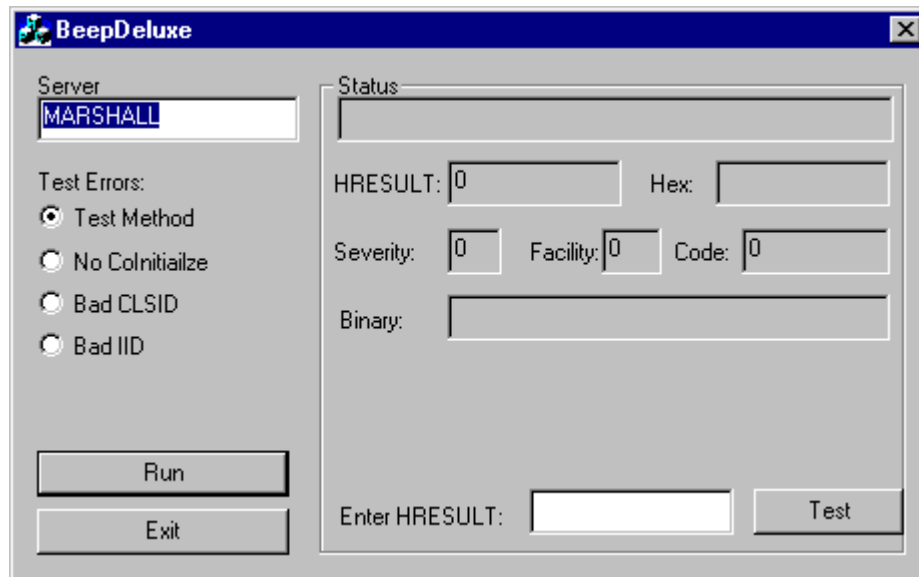
You can see that the radio buttons on the left hand side let you experiment with a lack of a CoInitialize funtion, a bad class ID and a bad interface ID. If you click the Run button the area on the right will show the effect of the different errors on the HRESULT returned by different functions in the client.

[To download code, go to the article at http://journal.iftech.com/.]

NOTE: if you have trouble compiling or linking this code, it may be because the configuration is incorrect. For some reason VC++ v6 will sometimes default to a very odd Unicode Release build instead of the expected Win32 Debug build. Use the Active Configuration... option in the Build menu to check the configuration and to set it to Win32 Debug if it is incorrect.

When you explore the client code in this example, you will find that it is a somewhat more robust version of the standard client code we used above. It also allows remote connections through DCOM. For example, it sets default security using the CoInitializeSecurity function to introduce you to that function, and it also makes use of the CoCreateInstanceEx function so that remote servers on other machines can be called. Walk through the code, look up those two functions in the documentation and you will be amazed to find how easy it is to understand now that you know something about COM!

TWO: Understanding ATL-generated Code.

The source code for our server DLL was generated by ATL. For many people it is perfectly acceptable to never look at the code ATL created. For others, "not knowing" the details of this code is unacceptable. This tutorial gives you a quick tour of the code produced by ATL The code for the server DLL that is now sitting on your hard drive really resides in three different types of files.

- First, there are the traditional C++ source and header files. Initially, all of this code is generated by the ATL wizards.

- The Beep method was added by the "Add Method" dialog, which modified the *MIDL interface definition.* The MIDL source code is in an IDL file - in this example it's BeepServer.IDL. The MIDL compiler will use this file to create several output files. These files will take care of much of the grunt work of implementing the server. As we add methods to the COM object, we'll be editing the MIDL file.

- The third group of source files are automatically generated MIDL output files created by the MIDL compiler. These files are source code files, but because they are automatically generated by the MIDL compiler from MIDL source code, these files are never modified directly either by wizards or by developers. You might call them "second generation files" - the wizard created an IDL file and the MIDL compiler created source code files from that IDL file. The files created by the MIDL include:

  - BeepServer.RGS - Registration script for the server.

  - BeepServer.h - This file contains definitions for the COM components.

  - BeepServer_i.c - GUID structures for the COM components.

  - Proxy/Stub files - This includes "C" source code, DLL definitions, and makefile (.mk) for the Proxy and Stub.

The ATL wizard also creates an application "resource". If you look in the project resources, you'll find it under "REGISTRY". This resource contains the registration script defined in BeepServer.RGS. The name of the resource is IDR_BEEPOBJ.

We look at all of these different components in the sections below.

The Main C++ Module

When we ran the ATL COM Appwizard, we chose to create a DLL-based server and we chose not to use MFC. The first selection screen of the wizard determined the overall configuration of the server.

The AppWizard created a standard DLL module. This type of standard DLL does not have a WinMain application loop, but it does have a DllMain function used for the initialization of the DLL when it gets loaded:

```
CComModule _Module;

BEGIN_OBJECT_MAP(ObjectMap)
    OBJECT_ENTRY(CLSID_BeepObj, CBeepObj)
END_OBJECT_MAP()
```

```
/////////////////////////////////////////////////////////////////////////
// DLL Entry Point

extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID /*lpReserved*/)

{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        _Module.Init(ObjectMap, hInstance);
        DisableThreadLibraryCalls(hInstance);
    }
    else if (dwReason == DLL_PROCESS_DETACH)
        _Module.Term();
    return TRUE;    // ok
}
```

All the DllMain function really does is check if a client is attaching to the DLL and does some initialization. At first glance, there's no obvious indication that this is a COM application at all.

The COM portion of our new server is encapsulated in the ATL class CComModule. CComModule is the ATL server base class. It contains all the COM logic for registering and running servers, and for starting and maintaining COM objects. CComModule is defined in the header file "altbase.h". This code declares a global CComModule object in the following line:

```
CComModule _Module;
```

This single object contains much of the COM server functionality for our application. Its creation and initialization at the start of program execution sets a chain of events in motion.

ATL requires that your server always name it's global CComModule object "_Module". It's possible to override CComModule with your own class, but you aren't allowed to change the name.

If we had chosen an executable-based server, or even a DLL with MFC, this code would be significantly different. There would still be a CComModule based global object, but the entry point of the program would have been WinMain(). Choosing a MFC-based DLL would have created a CWinApp-based main object.

Object Maps

TheCComModule is connected to our COM object (BeepObj) by the object map seen in the previous section. An object map defines an array of all the COM objects the server controls. The object map is defined in code using the OBJECT_MAP macros. Here is our DLL's object map:

```
BEGIN_OBJECT_MAP(ObjectMap)
    OBJECT_ENTRY(CLSID_BeepObj, CBeepObj)
END_OBJECT_MAP()
```

The OBJECT_ENTRY macro associates the CLSID of the object with a C++ class. It's common for a server to contain more than one COM object – in that case there will be an OBJECT_ENTRY for each one.

Export File

Our In-Process DLL, like most DLLs, has an export file. The export file will be used by the client to connect to the exported functions in our DLL. These definitions are in the file BeepServer.def:

```
; BeepServer.def : Declares the module parameters.

LIBRARY        "BeepServer.DLL"

EXPORTS
    DllCanUnloadNow        @1 PRIVATE
    DllGetClassObject      @2 PRIVATE
    DllRegisterServer      @3 PRIVATE
    DllUnregisterServer    @4 PRIVATE
```

It is important to note what is **not** exported. There are no custom methods, and there is no export for the "Beep" method. These are the only exports you should see in a COM DLL.

Looking into the BeepServer.CPP file, we see that the implementation of these four functions is handled by the COM application class. Here's the code for DllRegisterServer:

```
// DllRegisterServer - Adds entries to the system registry
STDAPI DllRegisterServer(void)
{
    // registers object, typelib and all interfaces in typelib
    return _Module.RegisterServer(TRUE);
}
```

In this case, the DLL just calls ATL's CComModule::RegisterServer() method. CComModule implements the server registration in a way that is compatible with In-Process, Local, and Remote COM servers. The other four exported DLL functions are equally spartan. The actual implementation is hidden in the ATL templates.

Most of the code described above is DLL-specific code. You will only get this configuration if you choose to create a DLL-based server. None of the code in the main module is COM specific. The main module is entirely devoted to the infrastructure required to deliver COM objects in a DLL, and this code will be significantly different depending on the type of server. The actual code *inside* the server is much more uniform. The implementation of a coclass and interface is identical regardless of the type of server (DLL, EXE, server) you create. You can take a coclass from a DLL server and implement it in an EXE-based server with few changes.

The COM Object — "CBeepObj"

A COM server has to implement at least one COM object. We are using a single object named "CBeepObj". One of the most interesting things about this object is that the code was entirely generated by ATL wizards. It is quite remarkable how compact this object definition turns out to be. The class definition is found in BeepObj.h:

```
// BeepObj.h : Declaration of the CBeepObj
#include "resource.h"       // main symbols

/////////////////////////////////////////////////////
// CBeepObj
class ATL_NO_VTABLE CBeepObj :
    public CComObjectRootEx,
    public CComCoClass<CBeepObj, &CLSID_BeepObj>,
    public IBeepObj
{
public:
    CBeepObj()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_BEEPOBJ)

BEGIN_COM_MAP(CBeepObj)
    COM_INTERFACE_ENTRY(IBeepObj)
```

```
END_COM_MAP()

// IBeepObj
public:
    STDMETHOD(Beep)(/*[in]*/ long lDuration);
};
```

This simple header file defines a tremendous amount of functionality, as described in the following sections.

Object Inheritance

Probably the first thing you noticed about this code is the multiple inheritance. Our COM object has three base classes. These base classes are template classes which implement the standard COM functionality for our object. Each of these classes defines a specific COM behavior.

CComObjectRootEx< > (and CComObjectRoot< >) are the root ATL object class. These classes handle all the reference counting and management of the COM class. This includes the implementation of the three required IUnknown interfaces, QueryInterface(), AddRef(), and Release(). When our CBeepObj object is created by the server, this base class will keep track of it throughout its lifetime.

The template for CComObjectRootEx specifies the argument CComSingleThreadModel. Single threading means that the COM object won't have to handle access by multiple threads. During the setup of this object we specified "Apartment threading". Apartment threading uses a windows message loop to synchronize access to the COM object. This approach is the easiest because it eliminates many threading issues.

CComCoClass< > defines the Class factories that create ATL COM objects. Class factories are special COM classes that are used to create COM objects. The CComCoClass uses a default type of class factory and allows aggregation.

IBeepObj is the interface this server implements. An interface is defined as a C++ struct (recall that structs in C++ act like a class but can have only public members). If you dig into the automatically generated file BeepServer.h, you'll find that MIDL has created a definition of our interface.

```
interface DECLSPEC_UUID(
"36ECA947-5DC5-11D1-BD6F-204C4F4F5020")
    IBeepObj : public IUnknown
    {
    public:
        virtual /* [helpstring] */ HRESULT
STDMETHODCALLTYPE Beep(
            /* [in] */ long lDuration) = 0;
};
```

The DECLSPEC_UUID macro lets the compiler associate a GUID with the interface name. Note that our single method "Beep" is defined as a pure virtual function. When the CBeepObj is defined, it will have to provide an implementation of that function.

One peculiar thing about this Class definition is that it has the ATL_NO_VTABLE attribute. This macro is an optimization which allows for faster object initialization.

The Class Definition

Our object uses a default constructor. You can add special initialization here if required, but there are some limitations. One consequence of using the ATL_NO_VTABLE, is that you aren't allowed to call any

virtual methods in the constructor. A better place for complex initialization would be in the FinalConstruct method (which is inherited from CComObjectRootEx.) If you want to use FinalConstruct, override ATL's default by declaring it in the class definition. It will be called automatically by the ATL framework. (Final-Construct is often used to create aggregated objects.)

The DECLARE_REGISTRY_RESOURCEID() macro is used to register the COM object in the system registry. The parameter to this macro, IDR_BEEPOBJ, points to a resource in the project. This is a special kind of resource that loads the MIDL ".rgs" file.

BEGIN_COM_MAP is a macro that defines an array of COM interfaces that the CComObjectRoot< > class will manage. This class has one interface, IBeepObj. IBeepObj is our custom interface. It's common for COM objects to implement more than one interface. All supported interfaces would show up here, as well as in the class inheritance at the top of the class definition.

The Method

At last, we get to the methods. As an application programmer, our main interest will be in this section of the code. Our single Beep() method is defined in the line:

```
STDMETHOD(Beep)(/*[in]*/ LONG duration);
```

STDMETHOD is an OLE macro that translates to the following:

```
typedef LONG HRESULT;
#define STDMETHODCALLTYPE        __stdcall
#define STDMETHOD(method)        virtual HRESULT STDMETHODCALLTYPE method
```

We could have written the definition in a more familliar C++ style as follows:

```
virtual long _stdcall Beep(long lDuration);
```

We'll find the code for this method in the BeepObj.cpp module. Because this COM object has only one method, the COM object's source code is pretty sparse. All the COM logic of the object was defined in the ATL template classes. We're left with just the actual application code. When you are writing real applications, most of your attention will be focused on this module.

```
STDMETHODIMP Beep(long lDuration)
{
    ::Beep( 660, lDuration );
    return S_OK;
}
```

Again, the function definition translates into a standard function call.

```
long _stdcall CBeepObj::Beep( long lDuration )
```

If you neglect it, the method will be calling itself. The API beep routine takes two parameters: the frequency of the beep, and it's duration in milliseconds. If you're working with Windows 95, these two parameters are ignored and you get the default beep. The scope operator "::" is important, but it's easily forgotten.

The **_stdcall** tag tells the compiler that the object uses standard windows calling conventions. By default C and C++ use the __cdecl calling convention. These directives tell the compiler which order it will use for placing parameters on, and removing them from the stack. Win32 COM uses the _stdcall atribute. Other operating systems may not use the same calling conventions. Notice that our Beep() method returns a status of S_OK. This doesn't mean that the caller will always get a successful return status. Remember that

calls to COM methods aren't like standard C++ function calls. There is an entire COM layer between the calling program (client) and the COM server.

It's entirely possible that the CBeepObj::Beep() method would return S_OK, but the connection would be lost in the middle of a COM call. Although the function would return S_OK, the calling client would get some sort of RPC error indicating the failure. Even the function result has to be sent through COM back to the client!

In this example the COM server is running as an In-Process server. Being a DLL, the linkage is so tight, there's very little chance of transmission error. In future examples, where our COM server is running on a remote computer, things will be very different. Network errors are all-too-common, and you need to design your applications to handle them.

Server Registration

The COM subsystem uses the Windows registry to locate and start all COM objects. Each COM server is responsible for self-registering, or writing it's entries into the registry. Thankfully, this task has been mostly automated by ATL, MIDL and the ALT wizard. One of the files created by MIDL is a registry script. This script contains all the definitions required for the successful operation of our server. Here is the generated script:

```
HKCR
{
    BeepObj.BeepObj.1 = s 'BeepObj Class'
    {
        CLSID = s '{861BFE30-56B9-11D1-BD65-204C4F4F5020}'
    }
    BeepObj.BeepObj = s 'BeepObj Class'
    {
        CurVer = s 'BeepObj.BeepObj.1'
    }
    NoRemove CLSID
    {
        ForceRemove {861BFE30-56B9-11D1-BD65-204C4F4F5020}
= s 'BeepObj Class'
        {
            ProgID = s 'BeepObj.BeepObj.1'
            VersionIndependentProgID = s 'BeepObj.BeepObj'
            ForceRemove 'Programmable'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Apartment'
            }
        }
    }
}
```

Registry Scripts

You may be familiar with .REG scripts for the registry. RGS scripts are similar, but use a complete different syntax and are only used by ATL for object registration. The syntax allows for simple variable substitution, as in the %MODLUE% variable. These scripts are invoked by the ATL Registry Component (Registrar). This was done with a macro in the object header:

```
DECLARE_REGISTRY_RESOURCEID(IDR_BEEPOBJ)
```

Basically, this script is used to load registry settings when the server calls CComModule::RegisterServer(), and to remove them when CComModule::UnregisterServer() is called. All COM registry keys are located in HKEY_CLASSES_ROOT. Here are the registry keys being set:

- BeepObj.BeepObj.1 - Current version of the class

- BeepObj.BeepObj - Identifies the COM object by name

- CLSID - The unique class identifier for the Object

There are several sub-keys under the CLISD:

- ProgID - The programmatic identifies.

- VersionIndependentProgID - Associates a ProgID with a CLSID.

- InprocServer32 - Defines a server type (as a DLL). This will vary depending on whether this is an In-Process, Local, or Remote server.

- ThreadingModel - The COM threading model of the object.

- TypeLib - The GUID of the server's type library.

This series on DCOM by William Rubin and Marshall Brain will continue in future issues of **Developer's Journal** *with an article entitled, "Creating a DCOM Server."*

Distributed COM

By William Rubin and Marshall Brain

Introduction

DCOM stands for "Distributed" COM. In the previous two articles, we have demonstrated COM clients and servers running on the same computer. In this article, we'll discuss how to extend our range into the area of DCOM and distributed computing.

Most COM programmers only use local "in-process" servers, which run as DLLs. A DLL loads into the process space of the client program, and is therefore quite reliable and efficient. We are going to be using an EXE-based server. This means the server and client run as separate programs. This makes sense, when you consider that the two programs are running on different computers. It does, however, introduce a new level of difficulty.

The good news is that converting from COM to DCOM is easy. The bad news is that there are a lot of things that can go wrong in the process of connecting the client and server. In this article, the emphasis is on helping you avoid the problems.

The Difference Between COM and DCOM

Most of the differences between COM and DCOM applications are hidden from the developer. The client and server program can be written identically, regardless of where the programs are running. This concept is known as *Local/Remote Transparency*. This makes your job as a programmer much more consistent.

Of course, there are some real differences in how distributed and local COM work internally. Local communications can be accomplished many ways, including simple Windows messages. Connecting to a remote computer requires a whole new layer of objects and network traffic. Despite these rather large differences, there are only a few things in your program you'll need to change.

Like all COM communication, everything starts when the client requests an interface from a server. In DCOM, the client calls CoCreateInstanceEx(), passing in a description of the server computer and requesting a class identifier (CLSID) and Interface. This request is handled by the Service Control Manager (SCM), which is a part of Windows. The SCM is responsible for the creation and activation of the COM object on the server computer. In the case of DCOM, the SCM will attempt to launch the server on the remote computer.
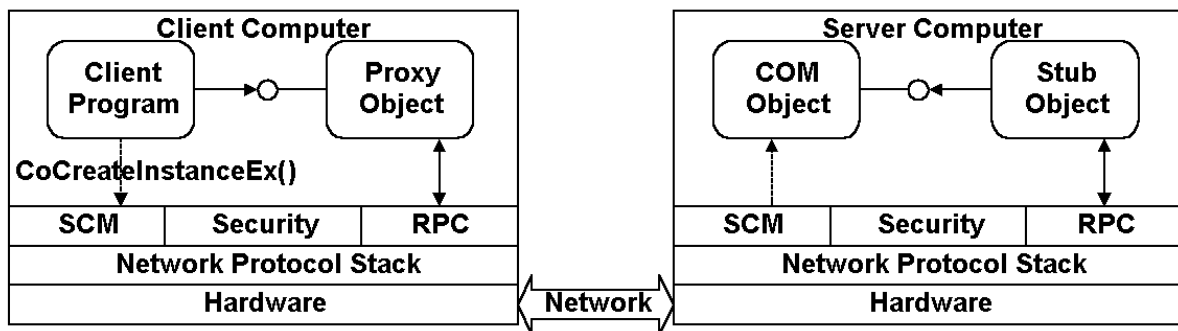
Figure 1. DCOM System Relationships

Once the remote COM server has been created, all calls will be marshaled through the proxy and stub objects. The proxy and stub communicate using RPCs (Remote Procedure Calls), which handle all the network interaction. On the server side, the stub object takes care of marshaling. On the client, the proxy does the work.

The transmittal of data across the network is taken care of by RPC. Actually, DCOM uses an extended type of RPC called an Object RPC, or ORPC. RPCs can run on a number of protocols, including TCP/IP, UDP, NetBEUI, NetBIOS, and named pipes. The standard RPC protocol is UDP (User Datagram Protocol). UDP is a connectionless protocol, which seems like a bad fit for a connection-oriented system like DCOM. This isn't a problem however; DCOM automatically takes care of connections.

As you can see, distributed COM is accomplished through a complex interaction of different hardware, operating system, and software components. You should take two things from this realization: 1) COM does a lot of work behind the scenes, and 2) there are a lot of things that can go wrong.

At the time of writing, only the TCP/IP protocol is available for DCOM data transfer on Windows 95/98 systems. This can be an annoying limitation, requiring you to install TCP/IP on all Windows 95 systems, even when other network protocols are available.

The Server Doesn't Change (much)

Any server that runs as a program (EXE) will work across a network. In general, you don't have to make any changes to a server to get it to work for DCOM. You may, however, want to add some security to your server, which will involve some effort. I've ignored security in this tutorial to keep things simple, but you can learn more in our book, entitled *Understanding DCOM* http://devcentral.iftech.com/Books/dcom/.

If you're using an in-process server, you will need to make some changes. An in-process server is a DLL, which can't load across a network. A DLL loads into the client program's address space, which will not work for remote connections. There is a work-around called a *surrogate*, which wraps the DLL in an executable program; however, it is usually more appropriate to change the

server over to an EXE. Generally, the easiest way to convert a DLL to an EXE is to recreate the server using the ATL Wizard and transfer the code from the DLL to the EXE.

For our example I have provided source code for a simple EXE-based DCOM server, named **RemoteServer.exe** (available as a download at Web version of this article). If you look at the code you will see it is out-of-the-box, wizard-generated code. However, I have added two methods – one to get the server name and another to get the server's system time.

After compiling the example, we'll simply copy the client EXE to the client computer. Note that you will also need a proxy/stub DLL because this is a custom interface. You will need to register the proxy/stub DLL on both the client and the server computers. If you are using an automation server with a type library, you will need to copy the typelib and register it on the client computer.

You Can Also Set Up a Remote Server Using OLEVIEW and DCOMCNFG

Actually, you can make a server program run remotely just by changing registry settings. There are two Microsoft tools to do this: 1) OLEVIEW, and 2) DCOMCNFG. Both tools set the registry so that DCOM tries to find the server on a remote computer. You can type in the remote computer name under the Activation tab of OLEVIEW, and it will be started on that computer.

This is a rather inelegant and inflexible solution, but it does work with older non-DCOM applications. I'm going to concentrate on setting up the remote activation using programmatic changes to the client. This approach will prove to be more flexible, once you get the hang of it.

The Client Source Code

Following is a listing of the client program. We will describe all the important parts of this code in the following sections. The definition of the ShowStatus() method was omitted -- all it does is display the HRESULT if there is an error.

```cpp
// RemoteClient.cpp : Defines the entry point for the console application.

#include "stdafx.h"     // added _WIN32_DCOM
#include <iostream.h>   // get "cout"
#include <comdef.h>     // get _com_error
#include <time.h>       // get time_t

// extract definitions from server project
#include "..\RemoteServer\RemoteServer.h"
#include "..\RemoteServer\RemoteServer_i.c"

// forward reference for status display method
void ShowStatus( HRESULT hr );

int main(int argc, char* argv[])
{
    HRESULT hr;     // COM error code
    IGetInfo *pI;   // pointer to interface

    // Get the server name from user
```

```cpp
char name[32];
cout << "Enter Server Name:" << endl;
gets( name );
_bstr_t Server = name;

// remote server info
COSERVERINFO cs;
// Init structures to zero
memset(&cs, 0, sizeof(cs));
// Allocate the server name in the COSERVERINFO  struct
cs.pwszName = Server;

// structure for CoCreateInstanceEx
MULTI_QI qi[1];
memset(qi, 0, sizeof(qi));

// initialize COM
hr = CoInitialize(0);
ShowStatus( hr );

// macro to check for success
if (SUCCEEDED(hr))
{
    // set a low level of security
    hr = CoInitializeSecurity(NULL, -1, NULL, NULL,
            RPC_C_AUTHN_LEVEL_NONE,
            RPC_C_IMP_LEVEL_IMPERSONATE,
            NULL,
            EOAC_NONE,
            NULL);

    // init security
    ShowStatus( hr );
}

    if (SUCCEEDED(hr))
{
    // Fill the qi with a valid interface
    qi[0].pIID = &IID_IGetInfo;

    // get the interface pointer
    hr = CoCreateInstanceEx(
            CLSID_GetInfo,  // clsid
            NULL,           // outer unknown
            CLSCTX_SERVER,  // server context
            &cs,            // server info
            1,              // size of  qi
            qi );           // MULTI_QI array
    ShowStatus( hr );
}

if (SUCCEEDED(hr))
{
    BSTR bsName; // Basic style string
```

```
        // Extract the interface from the MULTI_QI strucure
        pI = (IGetInfo*)qi[0].pItf;

        // Call a method on the remote server
        hr = pI->GetComputerName( &bsName );
        ShowStatus( hr );

        // Convert name to a printable string
        _bstr_t bst( bsName );
        cout << "Server Name :" << bst << endl;

        // get time from remote computer
        time_t tt;
        hr = pI->GetTimeT(&tt );
        ShowStatus( hr );

        // display time_t as a string
        cout << "Server Time :" << ctime( &tt ) << endl;

        // Release the interface
        pI->Release();
    }

    // Close COM
    CoUninitialize();

    // Prompt user to continue
    cout << "Press ENTER to continue" << endl;
    getchar();

    return 0;
}
```

Adding DCOM to the Simple Client

To activate our DCOM server we take the basic COM client shell and add some additional methods to make it DCOM-ready. One obvious change is that we specify the name of the server computer. Here are the other things we must add to the client:

• A way to specify the server computer name. This will load into the COSERVERINFO structure.

• Call `CoCreateInstanceEx()` instead of `CoCreateInstance()`. This also involves some different parameters and a structure called MULTI_QI.

• The first thing you do in any COM program is call CoInitialize. We'll use the default-threading model, which is *apartment threading*.

```
    // initialize COM
    hr = CoInitialize(0);
```

Specifying the Server with COSERVERINFO

When making a remote DCOM connection you must specify the server computer. The name of the computer can be a standard UNC computer name or a TCP/IP address. We'll ask the user to enter the computer name. Our example is a console application, so we'll use a simple input stream (include <iostream.h>) to get user input and display messages.

```
// Get the server name from user
char name[32];
cout << "Enter Server Name:" << endl;
gets( name );
```

The server name will be loaded into a COSERVERINFO structure. This structure requires a pointer to a wide-character string for the server name. We'll use _bstr_t copy constructor to convert the string. _bstr_t is a useful class when working with BSTR's and wide characters. Note that the COSERVERINFO structure was initialized to zero with the memset() function.

```
// remote server info
COSERVERINFO cs;
// Init structures to zero
memset(&cs, 0, sizeof(cs));
// Allocate the server name in the COSERVERINFO struct
// use _bstr_t copy constructor
cs.pwszName = _bstr_t(name);
```

Specifying the Interface with MULTI_QI

Normally, we get an interface pointer by calling CoCreateInstance. For DCOM we need to use the extend version, CoCreateInstanceEx. This extended function works perfectly well for local COM servers as well. CoCreateInstanceEx has several important differences. First, it lets you specify the server name; and second, it allows you to get more than one interface in a single call.

We've already set up our COSERVERINFO structure. We'll pass it into CoCreateInstanceEx to specify the server. (If you leave this parameter NULL, you'll get the local computer).

Unlike its predecessor, CoCreateInstanceEx returns more than one interface at a time. It does this by passing in an array of MULTI_QI structures. Each element of the array specifies a single interface. CoCreateInstanceEx will fill in the data requested.

```
// structure for CoCreateInstanceEx
MULTI_QI qi[2];
// Array of structures
// set to zero
memset(qi, 0, sizeof(qi));

// Fill the qi with a valid interface
qi[0].pIID = &IID_IGetInfo;
qi[1].pIID = &IID_ISomeOtherInterface;
```

```
        // get the interface pointer
        hr = CoCreateInstanceEx(
                CLSID_GetInfo,   // clsid
                NULL,            // outer unknown
                CLSCTX_SERVER,   // server context
                &cs,             // server info
                2,               // size of qi
                qi );            // MULTI_QI array
```

The MULTI_QI structure holds three pieces of information: a) a pointer to the IID, b) the returned interface pointer, and c) an HRESULT. Here's the structure, as defined in OBJIDL.IDL:

```
        typedef struct tagMULTI_QI
        {
            // pass this one in
            const IID  *pIID;
            // get these out (must set NULL before calling)
            IUnknown   *pItf;
            HRESULT     hr;
        } MULTI_QI;
```

The `qi` variable is an array of MULTI_QI structures. We start by setting the entire array to zero. This is done with the call to `memset()`. Next we fill the `pIID` element with a pointer to the interface GUID (IID) of the interface we want to extract. In our example here, we're requesting a pointer to `IGetInfo` and `ISomeOtherInterface` (`ISomeOtherInterface` is a fictional interface.). If we only need a single interface, we'd make the array size 1 and skip the second element. The 5th parameter to `CoCreateInstanceEx` defines the size of the array.

Calling `CoCreateInstanceEx` will populate the MULTI_QI array. Like most COM API functions, `CoCreateInstanceEx` returns an HRESULT. This function typically returns any of three HRESULTS.

- S_OK. All interfaces were returned.

- CO_S_NOTALLINTERFACES. At least one of the interfaces was returned, but some others failed.

- E_NOINTERFACE. None of the interfaces could be returned.

To determine which interface failed, you can check the HRESULT in the `qi` array.

```
    if (SUCCEEDED(hr))
        if (SUCCEEDED(qi[0].hr)) {
            // pItf pointer is OK.
```

CoCreateInstanceEx Will Fail if there are Network Problems.

Once you work out the initial bugs, your call to `CoCreateInstanceEx` will either succeed, or it will fail with an error like RPC_S_SERVER_UNAVAILABLE. In my code, I usually only

check the return status of `CoCreateInstanceEx` and abort the program if it isn't S_OK. This is because I can't continue without all the requested interfaces. Note that you still have to call `Release` on all the interfaces that were successfully returned.

The subject of error codes is extremely important for DCOM over the network. The lowly HRESULT contains more information then you might suspect and this information is especially useful in tracking down network errors. I'll cover errors in detail later in this article.

Once you've established that the interfaces were returned, you can use the interface pointers. In this case, I'm copying the pointer I requested into a `IGetInfo` interface pointer for use in the program.

```
// pointer to interface
IGetInfo *pI;
if (SUCCEEDED(hr))
{
    // Basic style string
    BSTR bsName;

    // Extract the interface from the MULTI_QI structure
    pI = (IGetInfo*)qi[0].pItf;

    // Call a method on the remote server
    hr = pI->GetComputerName( &bsName );

    pI->Release();
...
```

The rest of the code is just normal COM client code. There's nothing special about DCOM clients once you've connected to the server. There is one big difference that we'll cover later: errors. You can expect a lot of problems when getting your client and server to work together over a network for the first time. Many of those problems are related to server and proxy/stub registration.

Registering the Server and Proxy/Stub

If you're working on a single machine, registration for DCOM is identical to standard COM. The server program will typically register itself when you run it with the -REGSERVER switch. Standard Wizard-generated servers will have this code built into them. When the EXE is run with the -REGSERVER switch, it registers itself in the system registry and exits.

```
C\> remoteServer -regserver
```

In these examples we're using custom interfaces. This means that the proxy/stub DLL is required on the client machine. The proxy/stub is the component that will send all information between the client and server over the network. To use a proxy/stub DLL, you need to register it.

```
C:/> REGSVR32 remoteserverps.dll
```

This registers the proxy/stub DLL on the client so DCOM can automatically activate it. If you're using an IDispatch (or dual) based automation client, you won't have a proxy/stub DLL. In this

case, you'll use a type library to register.

Copying the Client to Another Computer

You've probably built your client and server on the same computer. You probably also tested it on the same computer. Now it's time to copy the client program to another computer and test it remotely. Copy both the client EXE and the proxy/stub DLL generated by the server. You don't need to register the server on the client machine. This is because we've specified the server computer name in the COSERVERINFO structure. The server does have to be registered on the server computer, or nothing will work.

On the client machine, create a new directory that will hold the client code and copy the client code into the directory. Then register the proxy/stub. You might type something like this into the MS-DOS <mark>prompt</mark> on the client machine:

```
C:> COPY \\Raoul\UnderCOM\RempteClient\Debug\Remoteclient.EXE
C:> COPY \\Raoul\UnderCOM\RemoteServer\RemoteserverPS.DLL
C:> REGSVR32 RemoteserverPS.DLL
```

Many people forget the proxy/stub DLL registration step on the third line. Without it, nothing will work.

Security is a Big Headache

Security is a really important issue, especially for network applications. Eventually, you will have to ensure that your DCOM application is secure. Once you understand the concepts and get the server working, then you can add security to your application. My advice is to get things working before you introduce any security.

Security varies between Win95/98 environments and Windows NT. Windows 95/98 offers some limited security features. You can go wild with security on Windows NT.

You can manipulate security settings for both the client and sever in your program. This is done with the `CoInitializeSecurity` API call. You can use this call to either 1) Add security, or 2) turn off security. You call this method immediately after calling `CoInitialize`.

```
// turn off security - overrides defaults
hr = CoInitializeSecurity(NULL, -1, NULL, NULL,
        RPC_C_AUTHN_LEVEL_NONE,
        RPC_C_IMP_LEVEL_IMPERSONATE,
        NULL,
        EOAC_NONE,
        NULL);
```

`CoInitializeSecurity` takes quite a few parameters. All these parameters are important, and represent significant security concepts. You'll find extensive (and challenging) material in the help files on CoInitializeSecurity and RPC security. (See the "Security in COM" article in

MSDN).

It is important to note that you can also specify security information for a server using DCOMCNFG and OLEVIEW. These utilities save security information in the registry. You can also specify security for the client program by explicitly calling `CoInitializeSecurity` (thus overriding the default registry settings.)

An interesting aspect of `CoInitializeSecurity` is that it is called from both the client and server. Many of the parameters are specific to the client and others to the server.

Errors, Problems, and More Errors: Debugging

Your client and server programs might work the first time you activate them, but probably they won't. Herein lies the big problem with DCOM. If you're trying to make things work in a real world network situation, you're going to hit some snags.

Remote connection problems usually come in four flavors:

1. The client and server didn't work to start with. Debug locally before you try it over the network.

2. Connection problems. If you can't get a good network connection between the client and server computer, DCOM will never work. Net configuration can be complex.

3. Launching problems. COM will not start the COM server program on the server computer. This is usually a security or registration problem.

4. You can't connect to the COM object on the server. This is usually a security or registration problem.

The following section discusses some of the problems you may encounter when working across a network. I've tried to outline some of the approaches I've found useful in diagnosing and fixing problems. Here are some suggestions about how to debug network problems with DCOM.

Step 1: Get it working locally

The first step in debugging is to get the client and server working locally. Install both components on the server machine, and keep at it until you can successfully communicate. If a component won't work locally, it won't work across a network. You probably developed and tested the application on a single computer, but if the server is being installed on a different computer, test both client and server on that system too.

By getting the system to work locally, you've eliminated most of the common programming and registration errors. There are still a few things, like security and remote activation, that you can only test across the network. Specify your local computer in the COSERVERINFO structure, which will exercise much of the network-related code.

Step 2: Be sure you can connect

Before you even try to install your program, first test and debug the network configuration between the client and server machines. Start by checking the network neighborhood to ensure that you can browse the remote computer. This is not always possible, but a failure to browse doesn't necessarily mean that DCOM won't work. In most cases, however, browsing is good starting place for checking connections. Check the connection in both directions.

Perhaps the most useful tool is PING. PING sends a series of network packets to a specific machine and waits for a response. Most installations support PING.

```
C:\> PING www.iftech.com

Pinging www.iftech.com 216.27.33.21 with 32 bytes of data:

Reply from 216.27.33.21: bytes=32 time=217ms TTL=120
Reply from 216.27.33.21: bytes=32 time=210ms TTL=120
Reply from 216.27.33.21: bytes=32 time=197ms TTL=120
Reply from 216.27.33.21: bytes=32 time=209ms TTL=120


Ping statistics for 216.27.33.21:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 197ms, Maximum =  217ms, Average =  208ms
C:\>
```

PING does a number of interesting things for you. First, it resolves the name of the remote computer. If you're using TCP/IP, the name of the remote computer will be converted to a TCP/IP address. In the above example, PING converts the name "www.iftech.com" into the TCP/IP address 216.27.33.21.

You should try PING from both directions. If you are using callbacks or connection points, you must have COM working in both directions. Callbacks and connection points can be very difficult to debug unless you know the network connection is working.

Step 3: Try it out. Don't forget to register the proxy/stub DLL or type library

Once you've finished the first two steps, it's worth trying the remote connection. If it works, consider yourself blessed.

The most common reasons for an otherwise OK connection to fail are registration problems. Be sure you registered the COM server on the server computer. Be sure you registered the proxy/stub DLL on both the client and server computer. If you're using an automation or dual interface, be sure the type library is registered on both systems. (To register a proxy/stub DLL, use the REGSVR32 command.)

Step 4: Try to get around name resolution problems

Name resolution can be a vexing problem in remote connections. Most people want to work with names like "\\RAOUL" and "\\SERVER" rather then TCP/IP addresses. The process of turning that readable name into a network address is called Name Resolution, and it can be very complicated on some system configurations. A common work-around is to refer to the server by its TCP/IP address. This will eliminate many name resolution problems – which are outside the scope of this discussion. You can easily put a TCP/IP address instead of a standard computer name into the COSERVERINFO structure.

You can also glean interesting information from the TRACERT (trace route) utility. If you have weird network configurations, they may show up here.

```
C:\> TRACERT www.iftech.com

Tracing route to www.iftech.com 216.27.33.21
over a maximum of 30 hops:

  1    184 ms    169 ms    182 ms  ct1.intercenter.net [207.211.129.2]
  2    182 ms    189 ms    188 ms  ts-gw1.intercenter.net [207.211.129.1]
  3    195 ms    192 ms    161 ms  ilan-gw1.intercenter.net [207.211.128.1]
  4    220 ms    178 ms    206 ms  206.152.70.33
  5    188 ms    207 ms    216 ms  207.211.122.2
  6    196 ms    189 ms    205 ms  216.27.1.71
  7     *         *         *      Request timed out.
  8    201 ms    221 ms    197 ms  rampart.iftech.com [216.27.12.142]
  9    210 ms    205 ms    192 ms  www.iftech.com [216.27.33.21]

Trace complete.
C:\>
```

As you can see, the route your DCOM packets are taking to their destination may be surprising!

Be especially aware of gateways, routers, proxies, and firewalls, as they will often block your connection. Firewalls in particular must be scrutinized, because firewalls will often block DCOM packets. Check with the network administrator.

Step 5: Can't launch the server?

By launching, we mean that COM automatically starts the server on request. This should always work on a local COM connection, but it can be problematic with network connections. If you're working with Windows 95/98, COM won't automatically launch a server. This limitation is necessary because of the lack of security on these operating systems. Because Windows NT provides full-blown security, it can automatically launch servers.

The work-around for this problem with Windows 95/98 clients is simple. Manually start the server before the client connects. Once the server is launched, it will connect normally.

There is one big 'gotcha'. The server will shutdown as soon as the last client calls `Release()`. The COM server's reference count will be zero, and the server will shut itself down. This means the client will fail the next time it tries to connect. You'll get RPC_S_SERVER_UNAVAILABLE or E_ACCESSDENIED from your call to `CoCreateInstanceEx`.

My favorite solution is to run a simple "watchdog" program on the server computer. This simple application always maintains a COM connection to the server. Because this program runs from the server computer, it launches the server automatically when it starts. As long as the watchdog program is running, the server will stay alive. If desired, you can start the watchdog program from the Startup group to bring the server up when the machine boots.

Step 6: Try DCOMCNFG and OLEVIEW

You've probably already been using these tools. DCOMCONF and OLEVIEW are two Microsoft utilities that show you registration information about your COM servers. Both of these products show you registry information in a more structured format. Both tools have disadvantages. OLEVIEW shows you a lot more information than you may want to see. The latest version of this utility has many good features, and you should get the latest download at Microsoft's site: http://www.microsoft.com/com/resource/oleview.asp. DCOMCNFG is a fairly crude tool. It shows a lot of information, but many people have trouble using it effectively. If you aren't already using one of these tools, you should start. There are also some good third-party COM utilities you can preview on the Internet.

Step 7: Re-Register the server and proxy/stub (or Type Library)

Sometimes re-registering the server and proxy/stub DLL can do wonders. If you're using Automation (or dual) interfaces, re-register the type library.

Step 8: Ask a network administrator

Sometimes the best line of defense will be a competent network administrator. If you've got one, you're lucky. Unfortunately, competent network administrators are a) rare, b) very busy, and c) averse to working with pesky programmers. An administrator may be able to help you resolve tough security and name resolution problems.

Typical DCOM Errors and Some Hints About What They Mean

The following error codes are typical HRESULTs you may get back from your client program. I've included some suggestions about what they might mean. I've included the symbolic name, as well as the translated message text. This is by no means a comprehensive list, but it's a great starting point for the most common errors.

CO_E_BAD_SERVER_NAME - A remote activation was necessary but the server name

provided was invalid.

> This is one of the few self-explanatory error messages. Note that it doesn't mean you entered the wrong server name. It means you entered an invalid server name. Check that the name is in the proper network format -- check for invalid characters.

> Unresolved or non-existent servers show up with a different error message: RPC_S_SERVER_UNAVAILABLE.

CO_E_SERVER_EXEC_FAILURE – Server execution failed.

> Check the COM security FAQ for more information. http://support.microsoft.com/support/kb/articles/Q158/5/08.asp (This site requires registration.)

E_ACCESSDENIED – General access denied error.

> This is an error from the security subsystem. The server system rejected a connection. These problems can be very difficult to diagnose. This error is most likely when using DCOM for remote connections.

> Check for activation problems, especially on Windows 95/98.

> Check security settings with DCOMCNFG.

> Reinstall the server and the Proxy/Stub DLL.

> Be sure you have File/Print sharing enabled.

E_FAIL - Unspecified error.

> This is often an application-specific error caused by a method returning E_FAIL.

E_NOINTERFACE – No such interface supported.

> You asked a server for an interface it doesn't support. This means your CLSID is probably OK, but the IID is not. This call is returned by `QueryInterface` (or through `CoCreateInstance`) when it doesn't recognize an interface. It may also be a proxy/stub problem.

> This may be a registration problem. Try re-registering the server and proxy/stub.

E_OUTOFMEMORY- Ran out of memory.

> This message may be unrelated to the actual error. See the security FAQ for some other possibilities. http://support.microsoft.com/support/kb/articles/Q158/5/08.asp (This site requires registration.)

ERROR_INVALID_PARAMETER– The parameter is incorrect..

You have a problem in one of the parameters to your function call. This is commonly seen in functions such as `CoCreateInstance`, `CoCreateInstanceEx`, `CoInitializeSecurity`, etc.

ERROR_SUCCESS – The operation completed successfully.

The same message as S_OK and NO_ERROR.

REGDB_E_CLASSNOTREG – Class Not Registered.

A registration or CLSID problem. Check your GUIDs.

RPC_S_SERVER_UNAVAILABLE – RPC Server is unavailable.

This problem is very common when working with remote servers. This is a generic remote connection error. RPC is the protocol used to implement DCOM. This can be a system setup problem or a security problem. You're about to learn a lot about networking!

You may be trying to connect to an incorrect or disconnected computer. Check the server name.

Be sure DCOM and RPC are enabled on the computer. Use DCOMCNFG or OLEVIEW (in "System Configuration" under the File menu).

Re-register the server and proxy/stub.

Check server activation problems, especially on Window 95/98.

Conclusion

When we made the big move from local to remote servers, the programming differences were small. The implementation was, however, a lot more difficult. Setting up a remote connection takes a lot more time and effort. However, what we have gained is remarkable. Running your server on a remote machine adds a whole new dimension to applications.

I believe there is some good news in store for COM programmers. Some of the changes involved in the COM+ effort *may* make DCOM programming easier. Don't expect much out of COM+ on the first few releases. Most DCOM issues and problems will remain.

In this series of articles you have seen how you can use COM to easily create an object server and access it from a client application. You have also seen how to implement a remote object server that runs over the network using DCOM. With the knowledge gained in these articles, you should be able to: a) use COM whenever you would normally create a DLL, and b) Create object-based servers to component-ize your code. Have fun!