DCOM Deployment Secrets

by Roy Nelson

The deployment of applications in our brave new interconnected world is relatively easy. Applications can be deployed and installed using standard installation tools on traditional media or more easily from a central internet-accessible location, normally via a web server. However, when it comes to DCOM clients, they can be deceptively easy to deploy but frustratingly difficult to get them to communicate with a DCOM server.

The problem lies somewhere between standard DCOM operation and the blind use of the code that the Delphi IDE generates to instantiate the DCOM server. The first sign of a long day ahead is seeing the dreaded `E_NOINTERFACE` error when a DCOM client tries to instantiate a DCOM server with a call to `CreateRemoteCOMObject`. However, when you register the DCOM server on the client machine, the DCOM client can miraculously be instantiated and call the DCOM server on a different machine! What is going on here?

Looking at the very simplified steps that the DCOM subsystem goes through when a DCOM client tries to call a DCOM server through a custom interface, it should become clear what needs to be done to make the system function as we would expect. In Delphi you would normally generate a dual interface Automation COM object as the DCOM server.

50 Ways To E_NOINTERFACE!

Here is a simplified sequence of events when instantiating a DCOM Automation server called from a COM client.

1. The client tries to instantiate the DCOM server with call to, for example, `CoCreateInstanceEx`. In the VCL this is implicitly called in `CreateRemoteCOMObject`. The COM server's Class ID and the machine name (where the COM server is expected to be residing) are specified. However, `CreateRemoteCOMObject` first requests an `IUnknown` interface, as all COM objects will be guaranteed to support it.

2. The `CoCreateInstanceEx` call is converted to one or more RPC calls to a server machine across the network.

3. On the server machine, the DCOM server is started by the operating system and the `IUnknown` interface pointer is packaged up and returned.

4. Back in the DCOM client a valid `IUnknown` interface pointer is returned by the DCOM subsystem.

5. Next the DCOM client queries the DCOM server by calling `QueryInterface` for the custom interface: normally the one you created in the type library editor.

6. Next, the client DCOM system will go through the following checks to see if the custom interface is known on the client machine, for example that it has been registered there. First, the operating system will check to see if the interface GUID (or, more correctly, the IID, or interface identifier) exists under the `HKEY_CLASSES_ROOT\Interface` key on the local client machine. If it does not exist, an `E_NOINTERFACE` error will be returned. Then, if an entry does exist for the interface, the operating system has to find out the layout of vtable for the interface and what data needs to be marshalled to and from the DCOM server, amongst other things. This is done by checking for the type library GUID in the `Typelib` key under `HKEY_CLASSES_ROOT\Interface`. If this check fails, an `E_NOINTERFACE` error will be returned. Next the operating system will check the `HKEY_CLASSES_ROOT\TypeLib` key for a GUID entry that corresponds to the GUID just found, if the check fails you will get an `E_NOINTERFACE` error returned. Finally, if the type library GUID was found, the operating system will check to see where the type library is located, under the `HKCR\TypeLib\{GUID}\1.0\0\win32` key: this contains the filename and path where the physical type library is located. The type library can be linked to an EXE, a DLL, as a resource, or just as a straight binary .tlb or .olb file.

7. Once the operating system has access to the type library it will call the server machine, where the operating system will go through something very similar to step 6. If all is well the COM server will return the valid interface pointer and start servicing calls from the client.

8. You might be wondering why the operating system on the client needs to have in-depth information about the layout of the COM server and interfaces. The reason is that the operating system provides a system generated proxy object (a local COM object that looks and acts like the remote object), which relays all calls to

Visual C++ Versus Delphi?

You might well ask why you never see Microsoft VC++ developers having these problems... Well, they do, they just don't realise it! For a custom interface COM server, MSVC wizards and the MIDL compiler generate code for proxy and stub DLL pairs. These DLLs contain the code to make the raw RPC calls to marshal the data from the client to the server and back. This, of course, is extremely fast, as the operating system does not need to manufacture its own proxy and stub, they come neatly packaged in these DLLs. However, the same rules apply to these DLLs: they also need to be registered on the client machine. Microsoft has found that more people are actually *not* using the proxy and stub DLLs and have come up with a new solution. This is to 'pre-compile' the IDL to byte code (referred to as 'fast format strings' in MIDL.HLP). This dramatically shrinks the size of the proxy DLLs: with the Microsoft compiler you can merge the stub code into the COM server. Don Box has described these format strings in his COM column in the January 1999 MSJ, the catch is these strings are undocumented by Microsoft.

the remote COM server and does all the required marshalling of data between the server and client.

So now it should be clear why registering the COM server on the client all of a sudden allows the remote DCOM server to be called. So, as long as the server EXE (or DLL if you use a DLL surrogate) exists on the remote server, the server will be callable from the client. If the DCOM server is registered on the client machine and then removed after being registered, a call will fail because the type library (in the EXE) does not exist on the client machine. The custom interface's description will not exist on the client machine and an `E_NOINTERFACE` error will occur.

Just imagine having hundreds of client machines connecting to a server machine using dial-up networking: it is just not feasible to deploy both the COM client and server, and register the COM server on each of the client's machines. So, all we need is to have the interface queried for by the DCOM client to be known on the client machine.

We can now see why `CreateRemoteComObject` returned a valid `IUnknown` from the server object, and the query (as `IDemoAutoObj`, see Listing 1) for the requested custom interface failed: because the custom interface was not registered on the client.

Using a custom interface like this is called early binding, as the interface needs to known by the COM client at compile-time and by the operating system at runtime.

Solutions

Let's look at various solutions we could use to solve the deployment problem, using a simple Delphi generated automation object. We'll take the easiest solution first.

Because the DCOM server is marked as an Automation object, chances are that it has a dual interface. That is to say, a custom interface (`IDemoAutoObj`), and a dispinterface (`IDispatch` derived interface), or `IDemoAutoObjDisp`: see Listing 1.

If a COM object supports an `IDispatch` interface, it means that some of the COM object's callable methods can have a corresponding dispatch ID. This dispatch ID is used to call the corresponding method, which is done by calling the `IDispatch`'s `Invoke` method with the appropriate dispatch ID. This mechanism is called late or ID binding.

This second interface is normally also listed in the generated Pascal file (XXX_TLB.PAS) as the dispinterface. Because `IDispatch` is a known (or canned) interface (eg it is built into COM like `IUnknown`). The operating system can find the interface and it knows the vtable layout of the `IDispatch` interface without needing a type library, thus nothing needs to be registered on the client machine.

Although the type library editor (in XXX_TLB.PAS) generates the dispinterface, no corresponding dummy class is generated to provide easy access to the `IDispatch` interface. The easy access class for the custom interface will have the code in Listing 2 generated.

If we were to create our own dummy class, as in Listing 3, so as to call through the dispinterface, by querying for the `IDispatch` interface we would not need to have the type library registered on the client machine!

```
{ Dispatch interface for DemoAutoObj Object }
  IDemoAutoObj = interface(IDispatch)
    ['{F07F2941-4BF2-11D2-BBE0-0000C0B5D6A0}']
    function Get_AMessage: WideString; safecall;
    procedure Set_AMessage(const Value: WideString); safecall;
    property AMessage: WideString read Get_AMessage write Set_AMessage;
  end;
{ DispInterface declaration for Dual Interface IDemoAutoObj }
  IDemoAutoObjDisp = dispinterface
    ['{F07F2941-4BF2-11D2-BBE0-0000C0B5D6A0}']
    property AMessage: WideString dispid 1;
  end;
{ DemoAutoObjObject }
  CoDemoAutoObj = class
    class function Create: IDemoAutoObj;
    class function CreateRemote(const MachineName: string): IDemoAutoObj;
  end;
...
class function CoDemoAutoObj.Create: IDemoAutoObj;
begin
  Result := CreateComObject(Class_DemoAutoObj) as IDemoAutoObj;
end;
class function CoDemoAutoObj.CreateRemote(const MachineName: string):
  IDemoAutoObj;
begin
  Result := CreateRemoteComObject(MachineName, Class_DemoAutoObj) as
    IDemoAutoObj;
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
CoDemoAutoObj = class
  class function Create: IDemoAutoObj;
  class function CreateRemote(const MachineName: string): IDemoAutoObj;
end;
...
class function CoDemoAutoObj.Create: IDemoAutoObj;
begin
  Result := CreateComObject(Class_DemoAutoObj) as IDemoAutoObj;
end;
class function CoDemoAutoObj.CreateRemote(const MachineName: string):
  IDemoAutoObj;
begin
  Result :=
    CreateRemoteComObject(MachineName, Class_DemoAutoObj) as IDemoAutoObj;
end;
```

```
CoDispDemoAutoObj = class
  class function Create: IDispatch;
  class function CreateRemote(const MachineName: string): IDispatch;
end;
...
class function CoDispDemoAutoObj.Create: IDispatch;
begin
  Result := CreateComObject(Class_DemoAutoObj) as IDispatch;
end;
class function CoDispDemoAutoObj.CreateRemote(const MachineName: string):
  IDispatch;
begin
  Result := CreateRemoteComObject(MachineName, Class_DemoAutoObj) as IDispatch;
end;
```

➤ *Listing 3*

```
procedure TForm1.Button1Click(Sender: TObject);
var  ADemoAutoObj : IDemoAutoObjDisp;
begin
  //Note this is a HARD cast. If we were to use the "as" operator a
  //QI will result and the interface would not be found again!
  ADemoAutoObj := IDemoAutoObjDisp(CoDispDemoAutoObj.CreateRemote('Obelix'));
  Caption := ADemoAutoObj.AMessage;
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
procedure TForm1.Button1Click(Sender: TObject);
var ADemoAutoObjVar : Variant;
begin
  ADemoAutoObjVar := CoDispDemoAutoObj.CreateRemote('Obelix');
  Caption := ADemoAutoObjVar.AMessage;
end;
```

```
CoDispDemoAutoObj = class
  class function Create: IDemoAutoObjDisp;
  class function CreateRemote(const MachineName: string): IDemoAutoObjDisp;
end;
...
class function CoDispDemoAutoObj.Create: IDemoAutoObjDisp;
begin
  Result := IDemoAutoObjDisp(CreateComObject(Class_DemoAutoObj) as IDispatch);
end;
class function CoDispDemoAutoObj.CreateRemote(const MachineName: string):
  IDemoAutoObjDisp;
begin
  Result := IDemoAutoObjDisp(CreateRemoteComObject(MachineName,
    Class_DemoAutoObj) as IDispatch);
end;
```

➤ *Above: Listing 6*

➤ *Below: Listing 6a*

```
procedure TForm1.Button1Click(Sender: TObject);
var ADemoAutoObj : IDemoAutoObjDisp;
begin
  ADemoAutoObj := CoDispDemoAutoObj.CreateRemote('Obelix');
  Caption := ADemoAutoObj.AMessage;
end;
```

```
uses ComObj,ActiveX;
procedure TForm1.FormCreate(Sender: TObject);
var
  //ITypeLib requires the ActiveX unit to be in the uses clause
  pTypeLib : ITypeLib;
begin
  //OleCheck(...) requires the COMObj unit to be in the uses clause
  //LoadTypeLibEx(...) requires the ActiveX unit to be in the uses clause
  OleCheck(LoadTypeLibEx('Demoauto.tlb',REGKIND_REGISTER, pTypeLib));
end;
```

➤ *Listing 7*

The DCOM server object can now be called in two ways. The first is by hard casting (the `as` operator will force OS to search for `IDemoAutoObjDisp`, which will fail) the returned `IDispatch` pointer to an `IDemoAutoObjDisp` interface pointer (Listing 4). The second is by assigning an `IDispatch` pointer to a variable of type `Variant` (Listing 5). This is possible because variants can be used to call through an `IDispatch` interface. Using the `Variant` technique is slightly slower than the `IDispatch` technique employed in the first option, because the compiler has to work some magic to make it possible (ie, more code is generated).

As with most things that are easy, there is a downside. When late binding is used, the method calls are generally slower than when using the custom interface.

However, we do not have to use variants. By moving the hard cast into the class method call, the easy access class's code is as shown in Listing 6. The calling code will now look much cleaner (Listing 6a).

If you want all the speed you can get then the only solution is to use the custom interface.

So, to be able to use the custom interface from a client, the IID must exist in the `HKEY_CLASSES_ROOT\Interface` key, the type library GUID must be in the `HKEY_CLASSES_ROOT\Interface\Typelib` key, the type library GUID needs to be in the `HKEY_CLASSES_ROOT\Typelib` key and finally `HKEY_CLASSES_ROOT\Typelib` needs to contain the physical location of a file containing the binary type library.

LoadTypeLibEx

Fortunately, the operating system provides a very useful function for registering type libraries: `LoadTypeLibEx`. This can register plain binary type libraries or those linked into EXEs and DLLs. Again I would like to start with the simplest scenario.

Option 1: deploy the type library (.tlb) as a separate file with the DCOM client application. All that needs to be done is to call `LoadTypeLibEx` with the location of the .tlb file before the DCOM server's interface pointer is obtained. In the example code, Listing 7, it is done in the `OnCreate` handler of the client application's main form.

This is quite an efficient way of doing the deployment. Normally .tlb files are small (a few Kb). However, we can add logic to first check if the interface has previously been registered, to avoid registering it again. The code to call the DCOM server in the DCOM client now looks more as it should, see Listing 8.

Option 2: the .tlb file can also be linked to a resource-only DLL as a resource and registered by using `LoadTypeLibEx`. The type library is linked in as a resource using the `{$R}` directive. The code for the empty DLL is very simple, see Listing 9. This will generate a DLL of approximately 20Kb in size. The type library can be registered again using the same code as in *Option 1* (see Listing 10). A drawback with this technique is that the operating system only allows one type library per DLL or EXE; so, if your DCOM client makes use of more that one DCOM server, you would have to deploy a resource DLL for each DCOM server.

Option 3: link the .tlb file to the DCOM client itself. All that is then needed is to call `LoadTypeLibEx`, passing it the name of client EXE itself! However, if you have your own COM objects in the DCOM client they might have their own

type library linked into the EXE, thus you will not be allowed to link in the DCOM server's .tlb file. The code is simple, see Listing 11.

Registering For Web Use

Probably the most contrived way of getting a DCOM server .tlb file to be registered is to link it to a DLL but to make the DLL look as if it is an in-process COM server (that is, export all the functions for registering and unregistering). Most of the code used in this example is from the VCL and shows a different way of registering a type library using the operating system function `RegisterTypeLib`. The main benefit of this technique is that the DLL can be embedded in an HMTL page, as a DLL or in a CAB file. This makes it downloadable and installable using a web browser such as Internet Explorer 4.x.

The code is in two units that you just need to link to an empty DLL project containing a type library resource. After you have compiled and linked the DLL, the type library contained within will be registered when the `DLL RegisterServer` function is called and unregistered when `DLLUnRegister Server` is called. These functions are used by Internet Explorer, most install packages, and the built-in installation support in Windows.

The registration unit looks like Listing 12 and contains the code for all the exported functions. The second unit contains the code used to register and unregister the type library: see Listing 13. All the type definitions and imported functions needed by this unit have been declared in the unit, to minimise the size of the DLL. The main DLL code merely serves to link in the above units (Listing 14).

Unregistering

The code to unregister the type library is slightly more complex, mainly due to the fact that it caters for older versions of COM and OLE. The type library is first loaded using `LoadTypeLib` which returns an `ITypeLib` interface. This is then used to register the type library, or to get the attributes of the type library, using these to unregister the type library. One important thing is that a binary .tlb file can also be passed to `UpdateRegistry Lib` in the Listing 14 code, instead of a DLL or EXE filename.

Considering the possible ways in which we can solve the problem, by far the easiest way is to use the `IDispatch` technique. Very little code needs to be changed, even though the call might be 'slow', it might be unnoticeable against the

➤ *Above: Listing 8*

```
procedure TForm1.Button2Click(Sender: TObject);
var ADemoAutoObj :IDemoAutoObj;
begin
  ADemoAutoObj := CoDemoAutoObj.CreateRemote('Obelix');
  Caption := ADemoAutoObj.AMessage;
end;
```

➤ *Below: Listing 9*

```
//Filename: EmptyDLL.DPR
library EmptyDLL;
uses Windows; // we only need to link in the bare minimum,
              //this gives us the smallest possible DLL
//**IMPORTANT**  Linking the  type library in.
{$R DemoAuto.tlb}
begin
end.
```

```
uses ComObj,ActiveX;
procedure TForm1.FormCreate(Sender: TObject);
var pTypeLib : ITypeLib;
begin
  //OleCheck(...) requires COMObj to be in the uses clause
  //LoadTypeLibEx(...) requires ActiveX to be in the uses clause
  //uses ComObj,ActiveX;
  OleCheck(LoadTypeLibEx('EmptyDLL.DLL',REGKIND_REGISTER, pTypeLib));
end;
```

➤ *Above: Listing 10*

➤ *Below: Listing 11*

```
//Link the typelibrary resource to the EXE. Make sure you EXE does not
//contain a COM server that also requires a type library
{$R DemoAuto.tlb}
procedure TForm1.FormCreate(Sender: TObject);
var pTypeLib : ITypeLib;
begin
  //OleCheck(...) requires COMObj to be in the uses clause
  //LoadTypeLibEx(...) requires ActiveX to be in the uses clause
  //uses ComObj,ActiveX;
  //Note how the client EXE itself is loaded to register the type library
  OleCheck(LoadTypeLibEx(StringToOLEStr(Application.ExeName),
    REGKIND_REGISTER, pTypeLib))
end;
```

➤ *Listing 12*

```
unit registerunit;
interface
uses
  ActiveX;
function DllRegisterServer: HResult;
function DllUnregisterServer: HResult;
function DllGetClassObject(const CLSID, IID: TGUID;
  var Obj): HResult; stdcall;
function DllCanUnloadNow: HResult; stdcall;
//Export the functions for registration
exports
   DllRegisterServer,DllUnregisterServer,
   DllGetClassObject, DllCanUnloadNow;
implementation
uses
  Registraunit;
function DllGetClassObject(const CLSID, IID: TGUID; var
  Obj): HResult;
const
  CLASS_E_CLASSNOTAVAILABLE = $80040111;
begin
  Pointer(Obj) := nil;
  Result := CLASS_E_CLASSNOTAVAILABLE;
end;
function DllCanUnloadNow: HResult;
begin
  Result := S_FALSE;
end;
function DllRegisterServer: HResult;
begin
  Result := UpdateRegistryLib(true, ModuleFileName);
end;
function DllUnregisterServer: HResult;
begin
  Result  := UpdateRegistryLib(false ,ModuleFileName);
end;
end.
```

```
unit Registraunit;
interface
uses ActiveX;
const
  E_FAIL = $80004005;
  S_FALSE = $00000001;
  S_OK    = $00000000;
type
  HINST = Integer;
  DWORD = Integer;
  THandle = Integer;
  HMODULE = HINST;
  LPCSTR = PAnsiChar;
  FARPROC = Pointer;
function UpdateRegistryLib(Register: Boolean; TypelibName:
  string): HResult;
function ModuleFileName: string;
//These functions need to be here because of a bug in the
//D3 compiler/fixed in D4
function GetModuleFileName(hModule: HINST; lpFilename:
  PChar; nSize: DWORD): DWORD; stdcall;
function GetModuleHandle(lpModuleName: PChar): HMODULE;
  stdcall;
function GetProcAddress(hModule: HMODULE; lpProcName:
  LPCSTR): FARPROC; stdcall;
implementation
const
  kernel32  = 'kernel32.dll';
function GetModuleFileName; external kernel32
  name 'GetModuleFileNameA';
function GetModuleHandle; external kernel32
  name 'GetModuleHandleA';
function GetProcAddress; external kernel32
  name 'GetProcAddress';
function ModuleFileName: string;
var Buffer: array[0..261] of Char;
begin
  SetString(Result, Buffer, GetModuleFileName(HInstance,
    Buffer, SizeOf(Buffer)));
end;
```

```
function UnregisterTypeLibrary(TypeLib: ITypeLib):HRESULT;
type
  TUnregisterProc = function(const GUID: TGUID; VerMajor,
    VerMinor: Word; LCID: TLCID; SysKind: TSysKind):
    HResult stdcall;
var
  Handle         : Integer;
  UnregisterProc : TUnregisterProc;
  LibAttr        : PTLibAttr;
begin
  Handle := GetModuleHandle('OLEAUT32.DLL');
  Result := E_FAIL;
  if (Handle <> 0) and (Handle > 32) then begin
    @UnregisterProc :=
      GetProcAddress(Handle, 'UnRegisterTypeLib');
    if @UnregisterProc <> nil then begin
      Result := TypeLib.GetLibAttr(LibAttr);
      if Result = S_OK then begin
       with LibAttr^ do
         Result := UnregisterProc(guid, wMajorVerNum,
           wMinorVerNum, lcid, syskind);
       TypeLib.ReleaseTLibAttr(LibAttr);
      end;
    end;
  end;
end;
function UpdateRegistryLib(Register: Boolean; TypelibName :
  string): HResult;
var TypeLib: ITypeLib;
begin
  Result  :=
    LoadTypeLib(StringToOLEStr(TypelibName), Typelib);
  if (TypeLib <> nil) and (Result = S_OK) then
    if Register then
      Result := RegisterTypeLib(TypeLib, StringToOLEStr(
        TypelibName), StringToOLEStr(TypelibName))
    else
      Result := UnregisterTypeLibrary(TypeLib);
end;
end.
```

➤ *Above: Listing 13*

➤ *Below: Listing 14*

```
//FileName: RegisterDLL.DPR
library RegisterDLL;
uses
  registerunit in 'registerunit.pas',
  Registraunit in 'Registraunit.pas';
//**IMPORTANT** Add any type library file in here to be linked in.
{$R Demoauto.tlb}
begin
end.
```

Out Of Process Problems

When developing custom COM objects derived from `IUnknown`, you might have found that you cannot place the COM object in an EXE (out of process) server, or calling the COM object on different threads. Whenever you try to access the COM object you get an *'interface not supported'* error. Well, the main issue is that the data cannot be marshalled because, firstly, you do not have your own proxy and stub DLLs, or secondly, you have implemented your own custom marshalling to marshal data. Thankfully, there is a way out: yes, you guessed it, type libraries. When you create an automation object the marshalling of data and the interface is handled by the Universal marshaller or Automation marshaller. Normally an Automation object is derived from `IDispatch`, however, a COM object does not need to be derived from `IDispatch` to get the benefit of the Universal marshaller. All that is needed is that the COM object is marked as supporting Automation, that a type library is available to describe the interface, and the data used by the interface. In the type library editor setting the OLE automation flag on for the COM object's interface does this. There are caveats: normal COM rules apply, the method uses `stdcall`, so no more `safecall`, and all the methods have to return an `HRESULT`, so `out` parameters have to be used to return values to a caller.

time it takes for the network request to be completed. It is easy to adapt the code to create a small generic installer, which reads and installs type libraries. A sample wizard is included on the disk, which will register and unregister type libraries contained in EXEs, DLLs or normal .tlb files. There is also a modified version of `CreateRemoteCOMObject`, which queries for `IDispatch` instead of `IUnknown`.

Conclusion

Hopefully this article will help you avoid some of the more puzzling and painful experiences of DCOM. The concepts are simple, but not well documented. One word of warning: as soon as you get your client talking to the server other issues (especially security) will start to arise, but these will have to wait for another time.

Roy Nelson (aka Mzwanele, email rnelson@borland.com) is a Technical Consultant at Inprise UK and the founder of beerware: if any beerware code snippets help you, payment is a beer when you meet me in a pub somewhere.