[Previous] [Contents] [Next]

# Will That Be Custom or Standard Marshaling?

Custom marshaling, the fundamental marshaling mechanism of COM+, is generally the most difficult way to provide marshaling code for an interface. The raw power of this model is unparalleled, but most objects do not require the fine degree of control it offers. One typical situation in which you might use custom marshaling is when an object has one or more clients running on the same machine that want to access the object's state in memory. Suppose you're developing an image-processing component to which clients send bitmaps for a specific type of processing. If a client process is running on the same machine but in a different process from the component, it makes little sense to copy the bitmap from the client's address space to the component's address space. It is much more efficient to simply allocate shared memory between the two processes. Custom marshaling gives you full control over the marshaling process, and implementing marshaling using shared memory is one possibility. Obviously, this type of custom marshaling is limited to processes running on a single machine.

> **Network Data Representation**
>
> The NDR standard was originally developed by Apollo Corporation and was later adopted by the OSF as part of its Distributed Computing Environment (DCE). Windows uses NDR in its DCE-compatible RPC implementation and, therefore, in COM+. NDR provides a mapping of Interface Definition Language (IDL) data types to the streams sent over the wire. NDR supports a multi-canonical format, which means that certain aspects of NDR support alternative representations. Data properties such as byte order, character sets, and floating-point representation can assume one of several predefined forms. For example, NDR supports ASCII and EBCDIC character sets. This flexibility is accommodated by following a "reader makes right" policy. The proxy is allowed to write the data in the best way it sees fit from the available choices, and the stub (the reader) is expected to be able to read data in any of the NDR flavors. The proxy, however, must inform the stub of the particular variation of NDR transfer syntax being used. Note that the "reader makes right" strategy was chosen over the alternative approach, "writer makes right," whereby the proxy must determine the transfer syntax supported by the stub before writing the data.

Another common use of custom marshaling is for obtaining marshal-by-value semantics. Some objects, such as monikers, have an immutable state: their internal data never changes. For an object whose state never changes, it doesn't make sense to require the client to make remote method calls to retrieve data. For example, imagine an object that represents a rectangle. If the dimensions and coordinates of the rectangle never change after the object is created, the object is a good candidate for the marshal-by-value optimization. In such cases, the data that defines the object can be transmitted to the proxy using custom marshaling, and the data can be used to create an exact replica of the object. This technique allows the client to make in-process calls to obtain the same information that a remote call would; the client is none the wiser.[5]

When you use shared memory to implement custom marshaling, you must still consider what happens when the clients and components do not run on the same machine. In such cases, you can either build the support necessary to remote the interface across a network or delegate to the standard marshaler for this task. In fact, it is recommended that implementations of custom marshaling delegate to the standard marshaler for destination contexts they do not understand or for which they do not provide special functionality.

Before you marshal interface pointers, the system must know what type of marshaling is called for. To see how COM+ makes this determination, take a look at the following code, which shows the tentative first steps taken by a typical client:

```
// client.cpp

// Start your engines.
CoInitializeEx(NULL, COINIT_MULTITHREADED);

// Get a pointer to the object's class factory.
IClassFactory* pClassFactory;
CoGetClassObject(CLSID_InsideCOM, CLSCTX_LOCAL_SERVER, NULL,
    IID_IClassFactory, (void**)&pClassFactory);
```

*CoGetClassObject* instructs the Service Control Manager (SCM) to locate the executable component containing the *InsideCOM* coclass and launch it. Recall that *CoGetClassObject* is also used internally by *CoCreateInstance(Ex)*, the standard object creation function in COM+; we'll use the *CoGetClassObject* function for clarity.

On start-up, a typical executable component performs the following standard steps:

```
// component.cpp

// Start your engines.
CoInitializeEx(NULL, COINIT_MULTITHREADED);
```

```
// Instantiate the class factory object.
IClassFactory* pClassFactory = new CFactory();

// Register the object in the global object table.
DWORD dwRegister;
CoRegisterClassObject(CLSID_InsideCOM, pClassFactory,
    CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE, &dwRegister);
```

The *CFactory* class is instantiated using the C++ *new* operator, and the resulting *IClassFactory* interface pointer is passed to the *CoRegisterClassObject* function. *CoRegisterClassObject* must somehow take this *IClassFactory* pointer and marshal it to any client process (on any machine) that calls *CoCreateInstance(Ex)* or *CoGetClassObject*. *CoRegisterClassObject* in turn calls *CoMarshalInterface* to do the dirty work of marshaling the *IClassFactory* interface pointer.

*CoMarshalInterface* is the fundamental COM+ interface pointer marshaling function. It starts by asking the object, "Hey, will that be custom or standard marshaling?" by calling *IUnknown::QueryInterface* for the *IMarshal* interface. *IMarshal* is the fundamental custom marshaling interface. If an object implements *IMarshal*, *CoMarshalInterface* knows that the object wants to use custom marshaling. If an object does not support *IMarshal*, as shown in the following code, *CoMarshalInterface* assumes that the object wants to use standard marshaling:

```
// component.cpp

HRESULT CFactory::QueryInterface(REFIID riid, void** ppv)
{
    if((riid == IID_IUnknown) || (riid == IID_IClassFactory))
        *ppv = (IClassFactory*)this;
    else
    {
        // IID_IMarshal?! No way!
        *ppv = NULL;          // No implementation of IMarshal,
        return E_NOINTERFACE; // so standard marshaling is used.
    }
    AddRef();
    return S_OK;
}
```

Since *IClassFactory* is a standard interface, Microsoft provides marshaling code as part of ole32.dll, the system-wide COM+ run time. You are able to override the built-in marshaling to provide custom marshaling code for *IClassFactory*, but the benefits of doing so are dubious. Thus, in the class object's *IUnknown::QueryInterface* implementation shown in the preceding code, any request for the *IMarshal* interface returns *E_NOINTERFACE*. Once *CoMarshalInterface* determines that standard marshaling is in order, it marshals the *IClassFactory* interface using the proxy/stub code provided by ole32.dll.

Returning to our examination of the client application, the next step normally executed is shown here in boldface:

```
// client.cpp

// Start your engines.
CoInitializeEx(NULL, COINIT_MULTITHREADED);

// Get a pointer to the object's class factory.
IClassFactory* pClassFactory;
CoGetClassObject(CLSID_InsideCOM, CLSCTX_LOCAL_SERVER, NULL,
    IID_IClassFactory, (void**)&pClassFactory);

// Instantiate the InsideCOM object
// and get a pointer to its IUnknown interface.
IUnknown* pUnknown;
pClassFactory->CreateInstance(NULL, IID_IUnknown,
    (void**)&pUnknown);
```

The client calls *IClassFactory::CreateInstance* to request that the *InsideCOM* coclass be instantiated and a pointer to *IUnknown* be returned. The *IUnknown* interface pointer about to be returned by the *IClassFactory::CreateInstance* method must be marshaled to the client, so the system-provided marshaling code for the *IClassFactory* interface calls *CoMarshalInterface* to marshal the *IUnknown* interface pointer. *CoMarshalInterface*[6] calls the *InsideCOM* object's *IUnknown::QueryInterface* method, requesting the *IMarshal* interface to determine whether the object wants to use standard or custom marshaling.

At this point, thoroughly exhausted by the narrative sequence above, you think, "What the heck, I'll take standard marshaling and call it a day." After all, *IUnknown* is a very standard interface, so surely you don't need to provide custom marshaling code for *IUnknown*. Custom marshaling, however, is done on a per-object basis, not on a per-interface basis like standard marshaling,

so it really is an all-or-nothing proposition. While the system will be more than happy to provide you with standard marshaling for *IUnknown*, the *InsideCOM* object also implements the *ISum* custom interface, which the system knows nothing about. So although the standard marshaler would properly handle the *IUnknown* interface, any attempt to get the *ISum* interface would fail. Thus, we are now at a crucial juncture. You must either respond affirmatively to the *CoMarshalInterface* query for *IMarshal*, opening your mind to the path of custom marshaling, or again respond with *E_NOINTERFACE*, indicating a desire to proceed with standard marshaling.

## Can You Say "Custom Marshaling"?

The *InsideCOM* object's implementation of the *IUnknown::QueryInterface* method is shown in the following code; support for the *IMarshal* interface is indicated in boldface:

```
HRESULT CInsideCOM::QueryInterface(REFIID riid, void** ppv)
{
    if(riid == IID_IUnknown)
        *ppv = (ISum*)this;

    // IMarshal? Absolutely!
    else if(riid == IID_IMarshal)
        *ppv = (IMarshal*)this;

    else if(riid == IID_ISum)
        *ppv = (ISum*)this;
    else
    {
        *ppv = NULL;
         return E_NOINTERFACE;
    }
    AddRef();
    return S_OK;
}
```

Once a pointer to *IMarshal* is returned by the *QueryInterface* method, the *CoMarshalInterface* function, called by the standard marshaling code for *IClassFactory::CreateInstance*, is entitled to call any of the six methods in the *IMarshal* interface. These methods are shown here in IDL notation:

```
interface IMarshal : IUnknown
{
    // Object implements this method.
    // Get the CLSID of the proxy object.
    HRESULT GetUnmarshalClass(
        [in] REFIID riid,
        [in, unique] void* pv,
        [in] DWORD dwDestContext,
        [in, unique] void* pvDestContext,
        [in] DWORD dwFlags,
        [out] CLSID *pClsid);

    // Object implements this method.
    // Get the maximum space needed to marshal the interface.
    HRESULT GetMarshalSizeMax(
        [in] REFIID riid,
        [in, unique] void* pv,
        [in] DWORD dwDestContext,
        [in, unique] void* pvDestContext,
        [in] DWORD dwFlags,
        [out] DWORD* pSize);

    // Object implements this method.
    // Marshal that interface and write it to the stream.
    HRESULT MarshalInterface(
        [in, unique] IStream* pStream,
        [in] REFIID riid,
        [in, unique] void* pv,
        [in] DWORD dwDestContext,
        [in, unique] void* pvDestContext,
        [in] DWORD dwFlags);
```

```
    // Object implements this method.
    // Tell the proxy that we're going to shut down.
    HRESULT DisconnectObject([in] DWORD dwReserved);

    // Object implements this method.
    // Release the marshaled data in the stream.
    HRESULT ReleaseMarshalData([in, unique] IStream* pStream);

    // Proxy implements this method.
    // Unmarshal that interface from the stream.
    HRESULT UnmarshalInterface(
        [in, unique] IStream *pStream,
        [in] REFIID riid,
        [out] void** ppv);
}
```

*IMarshal* is a peculiar interface because the first five of its six methods are called in the object and the remaining method, *UnmarshalInterface*, is called in the proxy. Although COM+ requires that any object returning an interface pointer from *QueryInterface* fully support that interface, in the case of *IMarshal* some of these methods are never called in the object or in the proxy. Nevertheless, you must provide dummy implementations of all six *IMarshal* methods in both the object and the proxy. For example, the object provides the following dummy implementation of the *IMarshal::UnmarshalInterface* method:

```
HRESULT CInsideCOM::UnmarshalInterface(IStream* pStream,
    REFIID riid, void** ppv)
{
    // This method should be called only in the proxy, not here.
    return E_UNEXPECTED;
}
```

After the object says, "Yes, I perform custom marshaling," the following steps are executed in the component:

1. *IMarshal::GetUnmarshalClass* is called to obtain the CLSID of the proxy object.
2. *IMarshal::GetMarshalSizeMax* is called to determine the size of the marshaling packet that the interface needs.
3. The system allocates the memory and then creates a stream object that wraps the memory buffer.
4. *IMarshal::MarshalInterface* is called to tell the object to marshal the interface pointer into the stream.

At this point, the buffer allocated in step 3 contains all the information necessary to create and initialize the proxy object within the client's address space. This buffer is communicated back to the client process, where the proxy object is created. Then *IMarshal::UnmarshalInterface* is called in the proxy to initialize the interface proxy. That's it—the whole purpose of the *IMarshal* interface is to give the object a chance to send one measly message back to the proxy!

Now that you have a high-level overview of the process, let's look at custom marshaling in detail.

## Pardon Me, What Is the CLSID of Your Proxy Object?

*CoMarshalInterface* first calls the *IMarshal::GetUnmarshalClass* method to request that the object provide the CLSID of the proxy object. In effect, *GetUnmarshalClass* asks the object, "What is the CLSID of your proxy object?" This value is returned via the CLSID pointer in the last parameter of *GetUnmarshalClass*. An implementation of *IMarshal::GetUnmarshalClass* might look like this:

```
CLSID CLSID_InsideCOMProxy =
    {0x10000004,0x0000,0x0000,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x01};

HRESULT CInsideCOM::GetUnmarshalClass(REFIID riid, void* pv,
    DWORD dwDestContext, void* pvDestContext, DWORD dwFlags,
    CLSID* pClsid)
{
    // We handle only the local marshaling case.
    if(dwDestContext == MSHCTX_DIFFERENTMACHINE)
    {
        IMarshal* pMarshal;
        // Create a standard marshaler (proxy manager).
        CoGetStandardMarshal(riid, (ISum*)pv, dwDestContext,
            pvDestContext, dwFlags, &pMarshal);

        // Load the interface proxy.
```

```
        HRESULT hr = pMarshal->GetUnmarshalClass(riid, pv,
            dwDestContext, pvDestContext, dwFlags, pClsid);
        pMarshal->Release();
        return hr;
    }
    *pClsid = CLSID_InsideCOMProxy;
    return S_OK;
}
```

COM+ holds onto this CLSID, which it sends to the client as part of the marshaling packet for use in launching the proxy object in the client's address space. Notice that much of the code in *GetUnmarshalClass* deals with the situation in which the client runs on a different machine. Since this custom marshaling example uses shared memory, we must delegate marshaling to the standard marshaler when the client is not running on the local machine. *CoGetStandardMarshal* returns a pointer to the standard marshaler's implementation of the *IMarshal* interface, on which we call the *IMarshal::GetUnmarshalClass* method to get the CLSID of the standard marshaler's proxy.

## How Big Did You Say Your Interface Is?

Next, *CoMarshalInterface* calls *IMarshal::GetMarshalSizeMax*, as shown in the following code. *GetMarshalSizeMax* asks the object, "What is the maximum space you need for marshaling your interface?" In this implementation, the object declares that it currently needs a maximum of 255 bytes. Notice that once again the code checks to see whether the client is running on the same machine. If it is not, we delegate the call to the standard marshaler.

```
HRESULT CInsideCOM::GetMarshalSizeMax(REFIID riid, void* pv,
    DWORD dwDestContext, void* pvDestContext, DWORD dwFlags,
    DWORD* pSize)
{
    // We handle only the local marshaling case.
    if(dwDestContext == MSHCTX_DIFFERENTMACHINE)
    {
        IMarshal* pMarshal;
        CoGetStandardMarshal(riid, (ISum*)pv, dwDestContext,
            pvDestContext, dwFlags, &pMarshal);
        HRESULT hr = pMarshal->GetMarshalSizeMax(riid, pv,
            dwDestContext, pvDestContext, dwFlags, pSize);
        pMarshal->Release();
        return hr;
    }

    // We need 255 bytes of storage to marshal the ISum
    // interface pointer.
    *pSize = 255;
    return S_OK;
}
```

To the value returned by the *GetMarshalSizeMax* method, *CoMarshalInterface* adds the space needed for the marshaling data header and for the proxy CLSID obtained in the call to *IMarshal::GetUnmarshalClass*. This technique yields the true maximum size in bytes required to marshal the interface, which is used to allocate a buffer large enough to hold the marshaled interface pointer. This buffer is then turned into a stream[7] by calling the *CreateStreamOnHGlobal* function. *CoMarshalInterface*, before it does anything else, writes the CLSID of the proxy object into the stream object by calling the *WriteClassStm* function.

Finally, *CoMarshalInterface* is ready to marshal the interface, so it calls *IMarshal::MarshalInterface* to say to the component, "Pack up that interface and let's get moving!" *MarshalInterface* marshals the requested interface pointer into the stream object provided as the first parameter. It normally does this by calling the *IStream::Write* method to write the marshaled interface pointer into the stream object.

> **What Is a Marshaled Interface Pointer?**
>
> Perhaps you're wondering exactly what a marshaled interface pointer consists of. A marshaled interface pointer can be whatever you want it to be. Once the *IMarshal* dance is over, you want the proxy to be able to communicate with the component to make method calls. That's right, *IMarshal* is designed to marshal interface pointers only. Once the interface pointer is available on the client side, all the work of packing function parameters, sending them to the component, and unpacking them is left to you! You can perform these tasks however you like. For cross-machine calls, you might use sockets or named pipes; for cross-process calls, you might use a file-mapping object for shared memory.
>
> Your implementation of *IMarshal::MarshalInterface* must write to the stream whatever data is needed to initialize the proxy on the client side. Such data might include the information needed to connect to the object,

such as a handle to a window, the name of a named pipe, or an Internet Protocol (IP) address and port number. When you implement custom marshaling to obtain marshal-by-value semantics, the marshaled interface pointer should contain the entire state of the object.

In the following code, *MarshalInterface* creates a file-mapping object to share memory between the proxy and the component in order to pass function parameters. Then it creates two event objects that are later used for synchronizing method calls. The names of these Win32 kernel objects are written into the marshaling stream. In other words, a marshaled interface pointer for *ISum* consists of the string "*FileMap,StubEvent,ProxyEvent*".

```
HRESULT CInsideCOM::MarshalInterface(IStream* pStream,
    REFIID riid, void* pv, DWORD dwDestContext,
    void* pvDestContext, DWORD dwFlags)
{
    // We handle only the local marshaling case.
    if(dwDestContext == MSHCTX_DIFFERENTMACHINE)
    {
        IMarshal* pMarshal;
        CoGetStandardMarshal(riid, (ISum*)pv, dwDestContext,
            pvDestContext, dwFlags, &pMarshal);
        HRESULT hr = pMarshal->MarshalInterface(pStream,
            riid, pv, dwDestContext, pvDestContext, dwFlags);
        pMarshal->Release();
        return hr;
    }

    ULONG num_written;
    char* szFileMapName = "FileMap";
    char* szStubEventName = "StubEvent";
    char* szProxyEventName = "ProxyEvent";
    char buffer_to_write[255];

    // Don't let your object fly away.
    AddRef();

    hFileMap = CreateFileMapping((HANDLE)0xFFFFFFFF, NULL,
        PAGE_READWRITE, 0, 255, szFileMapName);
    hStubEvent = CreateEvent(NULL, FALSE, FALSE,
        szStubEventName);
    hProxyEvent = CreateEvent(NULL, FALSE, FALSE,
        szProxyEventName);

    strcpy(buffer_to_write, szFileMapName);
    strcat(buffer_to_write, ",");
    strcat(buffer_to_write, szStubEventName);
    strcat(buffer_to_write, ",");
    strcat(buffer_to_write, szProxyEventName);

    return pStream->Write(buffer_to_write,
        strlen(buffer_to_write)+1, &num_written);
}
```

As with the *IMarshal::GetUnmarshalClass* and *IMarshal:: GetMarshalSizeMax* methods shown earlier, the *MarshalInterface* method delegates to the standard marshaler if the destination context does not indicate that the client is running on the local machine. The extra *AddRef* thrown into the preceding *MarshalInterface* code is necessary. When you use custom marshaling, no stub is available to hold a reference count on the object itself. Without this *AddRef*, the *InsideCOM* object would simply exit before the client has had a chance to call it. In the next section, you'll see that the proxy eventually calls *Release*, which is forwarded to the component to undo this *AddRef*.

Remember that *CoMarshalInterface* carefully orchestrates all of these steps. To review what we've covered so far, look at the following pseudo-code for *CoMarshalInterface*, which shows how interface pointers are marshaled:

```
// AddRef the incoming pointer to verify that it is safe.
pUnknown->AddRef();

// Do you support custom marshaling?
IMarshal* pMarshal;
HRESULT hr = pUnknown->QueryInterface(IID_IMarshal,
    (void**)&pMarshal);
```

```
// I guess not, so we'll use standard marshaling.
if(hr == E_NOINTERFACE)
    CoGetStandardMarshal(riid, pUnknown, dwDestContext,
        pvDestContext, dwFlags, &pMarshal);

// OK, what's the CLSID of your proxy?
CLSID clsid;
pMarshal->GetUnmarshalClass(riid, pUnknown, dwDestContext,
    pvDestContext, dwFlags, &clsid);

// How much space do you need?
ULONG packetSize;
pMarshal->GetMarshalSizeMax(riid, pUnknown, dwDestContext,
    pvDestContext, dwFlags, &packetSize);

// Allocate that memory.
HGLOBAL pMem = GlobalAlloc(GHND, packetSize + sizeof(CLSID));

// Turn it into a stream object.
IStream* pStream;
CreateStreamOnHGlobal(pMem, FALSE, &pStream);

// Write the CLSID of the proxy into the stream.
WriteClassStm(pStream, clsid);

// Marshal that interface into the stream.
pMarshal->MarshalInterface(pStream, riid, pUnknown,
    dwDestContext, pvDestContext, dwFlags);

// Release everything in sight.
pStream->Release();
pMarshal->Release();
pUnknown->Release();
```

At this point, the SCM is ready to send the stream object back to the client process. Since the SCM was responsible for launching the component to begin with, it knows exactly what sort of a barrier lies between the client and the component. This barrier is described by one of the marshaling contexts in the *MSHCTX* enumeration, shown in the table below. The mechanism you use to transmit data between the client and the component depends on the marshaling context. For example, if the client is communicating with the component process on the same machine (*MSHCTX_LOCAL*), you can use a file- mapping object to share memory. If the two processes run on different computers (*MSHCTX_DIFFERENTMACHINE*), you can use a network protocol such as named pipes, sockets, RPC, NetBIOS, or mailslots.

| ***MSHCTX* Enumeration** | **Value** | **Description** |
|---|---|---|
| *MSHCTX_LOCAL* | 0 | The object is running in another process of the current machine. |
| *MSHCTX_NOSHAREDMEM* | 1 | The object does not have shared memory access; unused. |
| *MSHCTX_DIFFERENTMACHINE* | 2 | The object is running on a different machine. |
| *MSHCTX_INPROC* | 3 | The object is running in another apartment of the current process. |
| *MSHCTX_CROSSCTX* | 4 | The object is running in another context of the current apartment. |

At the end of the marshaling sequence, we're left with a marshaled interface pointer that can be sent directly to the client or be stored in a table waiting for a client to request it. The flag passed as the last parameter to *CoMarshalInterface* indicates whether the marshaled data is to be transmitted back to the client process—the normal case—or written to the class table, from which multiple clients can retrieve it. This flag can be one of the values from the *MSHLFLAGS* enumeration constants listed in the following table.

| ***MSHLFLAGS* Enumeration** | **Description** |
|---|---|
| *MSHLFLAGS_NORMAL* | Indicates that unmarshaling is occurring immediately. |
| *MSHLFLAGS_TABLESTRONG* | Unmarshaling is not occurring immediately. Keeps object alive; must be explicitly released. |
| *MSHLFLAGS_TABLEWEAK* | Unmarshaling is not occurring immediately. Doesn't keep object alive; still must be released. |

| | |
|---|---|
| *MSHLFLAGS_NOPING* | Turns off garbage collection by not pinging objects to determine whether they are still alive.* Can be combined with any of the other *MSHLFLAGS* enumeration constants. |

*For more about garbage collection, see Chapter 19.

The *MSHLFLAGS_NORMAL* flag indicates that the unmarshaling is occurring immediately, such as when the component returns an interface pointer in response to a client's call to *IUnknown::QueryInterface*. *MSHLFLAGS_ TABLESTRONG* and *MSHLFLAGS_TABLEWEAK*, however, indicate that the unmarshaling isn't happening at the present time. These flags indicate that the marshaled interface pointer is to be stored in a global table that is accessible to all processes via COM+. When a client process wants to connect, the marshaled interface pointer is already there, ready and waiting. For example, table marshaling is used by the *CoRegisterClassObject* function, which is called by executable components. Internally, *CoRegisterClassObject* calls *CoMarshalInterface* with the *MSHLFLAGS_TABLESTRONG* flag. This stores a marshaled interface pointer to the class object in the class table, which clients can request using *CoGetClassObject*. Only when a client calls *CoGetClassObject* is the marshaled interface pointer sent to the client for unmarshaling.

If *MSHLFLAGS_TABLESTRONG* is specified, the *IUnknown::AddRef* method is automatically called to ensure that the object remains in memory. In other words, the presence of the marshaled interface pointer in the class table counts as a strong reference to the interface being marshaled, meaning that it is sufficient to keep the object alive. This is the case with *CoRegisterClassObject*. When the *CoRevokeClassObject* function is later called to remove the marshaled interface pointer from the class table, *CoRevokeClassObject* calls the *CoReleaseMarshalData* function to free the stream object containing the marshaled interface pointer. *CoReleaseMarshalData* is a relatively simple helper function that instantiates the proxy and invokes the *IMarshal::ReleaseMarshalData* method.

*MSHLFLAGS_TABLEWEAK* indicates that the presence of the marshaled interface pointer in the class table acts as a weak reference to the interface being marshaled, which means that it is not sufficient to keep the object alive because *AddRef* is not called. You typically use *MSHLFLAGS_TABLEWEAK* when you register an object in the Running Object Table (ROT). The presence of this flag prevents the object's entry in the ROT from keeping the object alive in the absence of any other connections.[8]

## Unmarshaling the Interface Pointer

The *IMarshal* interface provides a mechanism by which the component can marshal its interface pointer into a stream object that the client can unmarshal. Just as *CoMarshalInterface* directs the marshaling of an interface pointer, *CoUnmarshalInterface* directs the unmarshaling of an interface pointer. Look at the following function declaration for *CoUnmarshalInterface*; you should be able to tell that *QueryInterface* will be involved somewhere along the way:

```
STDAPI CoUnmarshalInterface(IStream* pStream, REFIID riid,
    void** ppv);
```

*CoUnmarshalInterface* examines the stream object containing the marshaled interface pointer to find the CLSID of the proxy object, which is required to unmarshal the remainder of the stream object. The CLSID is read out of the stream by the *ReadClassStm* function. The SCM then instantiates the proxy object specified by the CLSID by calling *CoCreateInstance* and requesting the *IMarshal* interface. A proxy used for custom marshaling an object must implement the *IMarshal* interface.[9]

Figure 14-3 shows how the SCM loads the proxy in the client's address space and that the proxy and the object communicate directly via a private protocol of their own choosing.

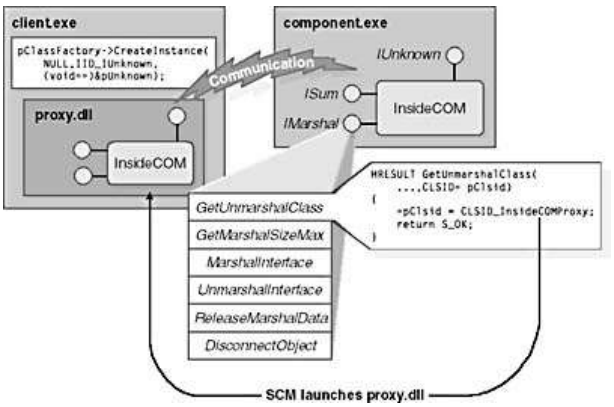


**Figure 14-3.** *An example of custom marshaling showing the proxy and the object communicating.*

Here's a quick recap. The client calls *CoCreateInstance* to instantiate the *InsideCOM* component. Then *CoMarshalInterface* calls *IUnknown::QueryInterface* to determine whether the object supports the *IMarshal* interface and hence custom marshaling. Since this is the case, the *ISum* interface pointer is marshaled into a stream object and returned to the proxy manager.

The *CoUnmarshalInterface* function instantiates the proxy and manages the unmarshaling of the stream object. Once a pointer to the *IMarshal* interface of the proxy object has been retrieved, *CoUnmarshalInterface* calls *IMarshal::UnmarshalInterface*. This method, implemented by the proxy, reads data from the marshaling stream using *IStream::Read*, and uses that information to establish a connection with the object. Again, you can use any communication mechanism (named pipes, sockets, and so on).

From this communication, the *UnmarshalInterface* method returns the unmarshaled pointer to the requested interface. In reality, the unmarshaled pointer is simply a pointer to the proxy's implementation of the requested interface that is now loaded in the client's address space. *CoUnmarshalInterface* also calls *IMarshal::ReleaseMarshalData* to free whatever data might be stored in the marshaling packet; this means that you need not call the *CoReleaseMarshalData* function when unmarshaling an interface pointer.

Here is the proxy's implementation of *IMarshal::UnmarshalInterface*, which reads the data from the stream previously marshaled by the object in *IMarshal::MarshalInterface*:

```
HRESULT CInsideCOM::UnmarshalInterface(IStream* pStream,
    REFIID riid, void** ppv)
{
    unsigned long num_read;
    char buffer_to_read[255];
    char* pszFileMapName;
    char* pszStubEventName;
    char* pszProxyEventName;

    pStream->Read((void*)buffer_to_read, 255, &num_read);

    pszFileMapName = strtok(buffer_to_read, ",");
    pszStubEventName = strtok(NULL, ",");
    pszProxyEventName = strtok(NULL, ",");

    hFileMap = OpenFileMapping(FILE_MAP_WRITE, FALSE,
        pszFileMapName);
    pMem = MapViewOfFile(hFileMap, FILE_MAP_WRITE, 0, 0, 0);

    hStubEvent = OpenEvent(EVENT_MODIFY_STATE, FALSE,
        pszStubEventName);
    hProxyEvent = OpenEvent(EVENT_MODIFY_STATE|SYNCHRONIZE,
        FALSE, pszProxyEventName);

    return QueryInterface(riid, ppv);
}
```

Eventually, when the component exits, *CoDisconnectObject* is called, which in turn calls the *IMarshal::DisconnectObject* method in the object. Note that clients do not call *CoDisconnectObject*; they should use *IUnknown::Release* for that purpose. *CoDisconnectObject* is a helper function that disconnects all remote client connections that maintain interface pointers to the specified object. First, *CoDisconnectObject* queries the object for its *IMarshal* interface pointer; if the object does not support custom marshaling, *CoGetStandardMarshal* is called to obtain a pointer to the standard marshaler. *CoDisconnectObject* then uses the *IMarshal* pointer to call the *IMarshal::DisconnectObject* method. It is the job of *DisconnectObject* to notify the proxy that the object itself is exiting and that it should return the error *CO_E_OBJECTNOTCONNECTED* to client requests.

The following pseudo-code summarizes the steps taken in the client process by *CoUnmarshalInterface* to ensure that unmarshaling is done correctly:

```
// Turn the marshaled interface pointer into a stream object.
IStream* pStream;
CreateStreamOnHGlobal(hMem, FALSE, &pStream);

 // Get the CLSID of the proxy object out of the stream.
CLSID clsid;
ReadClassStm(pStream, &clsid);

// Instantiate that proxy object.
IMarshal* pMarshal;
CoCreateInstance(clsid, 0, CLSCTX_INPROC_SERVER, IID_IMarshal,
    (void**)&pMarshal);

// Clone the stream.
IStream* pStreamClone;
pStream->Clone(&pStreamClone);

// Unmarshal your interface.
pMarshal->UnmarshalInterface(pStream, riid, ppv);

// Release any data in the stream.
pMarshal->ReleaseMarshalData(pStreamClone);
```

```
// Release anything left.
pStream->Release();
pStreamClone->Release();
pMarshal->Release();

// Free the marshaled memory packet.
GlobalFree(hMem);
```

When the client calls the *ISum::Sum* method, it is actually talking to the in-process proxy object. One of the major reasons for performing custom marshaling is to be able to intelligently reduce the number of cross-process or cross-machine calls. For example, rather than making a call to the component every time the client calls *AddRef* or *Release*, the proxy simply keeps a reference count locally. Only when the last *Release* call is made and the reference count returns to 0 does the proxy actually forward the *Release* call to the object. Interestingly, the remoting infrastructure of COM+ performs this type of *IUnknown* optimization automatically for objects that use standard marshaling.[10]

Here is the proxy's implementation of the *Release* method:

```
ULONG CInsideCOM::Release()
{
    // Regular Release; don't bother calling the object.
    if(--m_cRef != 0)
        return m_cRef;

    // Notify the object that this is the last Release.
    short method_id = 1; // ISum::Release
    memcpy(pMem, &method_id, sizeof(short));
    SetEvent(hStubEvent);
    delete this;
    return 0;
}
```

When the reference count returns to 0, this function copies the value of the *method_id* variable into the file-mapping memory shared by the proxy and the component. It then calls *SetEvent* to notify the object that it is requesting service. The object is waiting for this event in the following code:

```
void TalkToProxy()
{
    while(hStubEvent == 0)
        Sleep(0);

    void* pMem = MapViewOfFile(hFileMap, FILE_MAP_WRITE,
        0, 0, 0);
    short method_id = 0;

    while(true)
    {
        WaitForSingleObject(hStubEvent, INFINITE);
        memcpy(&method_id, pMem, sizeof(short));
        switch(method_id) // What method did the proxy call?

        {
        case 1:    // IUnknown::Release
            CoDisconnectObject(reinterpret_cast<IUnknown*>(
                g_pInsideCOM), 0);
            g_pInsideCOM->Release();
            return;
        case 2:    // ISum::Sum
            SumTransmit s;
            memcpy(&s, (short*)pMem+1, sizeof(SumTransmit));
            g_pInsideCOM->Sum(s.x, s.y, &s.sum);
            memcpy(pMem, &s, sizeof(s));
            SetEvent(hProxyEvent);
        }
    }
}
```

After the proxy sets the stub event, the object retrieves the first byte, which indicates what method is being called. In this simple implementation, the object expects to receive only one of two calls from the proxy: *Release* or *Sum*. The object then deciphers the call and forwards it to the actual implementation. The *Sum* method, for example, requires that the object unpack the parameters from the SumTransmit structure and then repack the return value. The SumTransmit structure, defined below, stores the parameters and the return value of the *Sum* method:

```
struct SumTransmit
{
    int x;
    int y;
    int sum;
};
```

As shown in the following proxy code, we wait until the object has finished adding the values before we retrieve the result for the client. We use a second event object (*hProxyEvent*) to determine when the component has finished processing.

```
HRESULT CInsideCOM::Sum(int x, int y, int* sum)
{
    SumTransmit s;
    s.x = x;
    s.y = y;
    short method_id = 2; // ISum::Sum

    memcpy(pMem, &method_id, sizeof(short));
    memcpy((short*)pMem+1, &s, sizeof(SumTransmit));

    SetEvent(hStubEvent);
    WaitForSingleObject(hProxyEvent, INFINITE);

    memcpy(&s, pMem, sizeof(s));

    *sum = s.sum;
    return S_OK;
}
```

The custom marshaling mechanism in this example depends on shared memory being available between the proxy and the object; this is possible only if both the client and the object are running on the same machine. To create a custom marshaling sample that works across machines, we would have to use a network-capable mechanism.