*Note: This file is also saved in Rich Text Format as APPB.RTF. We recommend that you use APPB.RTF if you have a word processor that can read Rich Text Format files.*

**A P P E N D I X B**

# The Details of Standard Marshaling

Chapter 6 briefly mentioned the architecture of COM's standard marshaling. Figure 6-3 on page 291 illustrates how a proxy and its facelets communicate with a stub and its stublets through an RPC Channel object, which itself communicates with a system RPC service. This appendix explores the overall architecture of standard marshaling. We'll look at the interfaces on these objects and their role in making Local/Remote Transparency work its magic, using a typical object creation sequence involving *IClassFactory::CreateInstance* as the focal point.

The information here applies to objects provided by local or remote servers and does not apply at all to in-process servers, for which no marshaling is necessary. As a convenience, the following text uses *remote* to refer to both local and remote objects and servers.

## Architectural Objects

In Figure 6-3 on page 291, you can see the five separate object types that make up COM's remoting architecture for each remote object:

- The RPC Channel, which implements *IRpcChannelBuffer* and performs the low-level RPC necessary to transmit information between processes and machines

- The proxy manager, which forms the shell of the overall proxy and controls which interfaces a client can access through *QueryInterface*

- Any number of facelets contained in the proxy, each of

which implements one specific interface exposed to the client as well as *IRpcProxyBuffer*, which is exposed only to the proxy manager

- The stub manager, which forms the shell of the overall stub and controls the remote object's lifetime

- Any number of stublets, one for each remote object interface that a client has requested (each stublet implements *IRpcStubBuffer* and maintains a single interface pointer to the remote object)

Let's look at these objects in more detail as well as the interfaces they implement. This will let us see how they fit into the overall architecture.

## New from CBS! It's the RPC Channel!

No, it's not one of those 500 new cable-TV channels that shows you nothing but hex dumps of RPC packets all day long.[1] (Now there's *true* nerd TV!) As you can discern from Figure 6-3, the RPC Channel is an object that implements the interface *IRpcChannelBuffer*:

```
interface IRpcChannelBuffer : IUnknown
   {
   HRESULT GetBuffer(RPCOLEMESSAGE *pMessage, REFIID riid);
   HRESULT SendReceive(RPCOLEMESSAGE *pMessage, ULONG *pStatus);
   HRESULT FreeBuffer(RPCOLEMESSAGE *pMessage);
   HRESULT GetDestCtx(DWORD *pdwDestContext, void **ppvDestContext);
   HRESULT IsConnected(void);
   };

typedef struct  tagRPCOLEMESSAGE
   {
   void         *reserved1;
   RPCOLEDATAREP dataRepresentation;   //An unsigned long
   void         *Buffer;
   ULONG         cbBuffer;             //Size of buffer to allocate
   ULONG         iMethod;              //Method being called
   void         *reserved2[5];
   ULONG         rpcFlags;
   } RPCOLEMESSAGE;
```

You can probably speculate about the sequence of calls that a facelet would make to this interface in order to generate a remote interface call. The facelet first obtains a marshaling packet from the RPC Channel using *IRpcChannelBuffer::GetBuffer*, in which the *riid* argument identifies the interface being called. The RPCOLEMESSAGE structure is also an in-parameter to *GetBuffer* that causes the facelet to initialize all fields except *Buffer* so that the channel can allocate the correct structures internally. Besides the self-explanatory *cbBuffer* and *iMethod* fields, *rpcFlags* indicates the type of call, such as synchronous or asynchronous, and *dataRepresentation* indicates specific information about the data structure, which includes character size (ANSI, Unicode, EBCDIC), floating-point format (IEEE, VAX, Cray, IBM), and big

endian vs. little endian. Data representation is obviously critical for interface remoting between different hardware architectures! Fortunately, COM is designed expressly with that in mind.

On return from *GetBuffer*, the facelet fills *Buffer* with the arguments to the member function. Once the buffer is filled, the facelet invokes *IRpcChannelBuffer::SendReceive* to send the function call across the wire, so to speak, to the corresponding stublet. (Although there's no wire, of course, in the strictly local case.) Every single consideration about how the interprocess or intermachine communication happens is encapsulated within *SendReceive*.

On the other side of the universe, in the remote process, the call shows up in the stublet that receives both the RPCOLEMESSAGE structure and the *IRpcChannelBuffer* pointer. The stublet reads arguments from the buffer and calls the remote object. When the remote object returns, the stublet changes the *cbBuffer* and *dataRepresentation* fields in the RPCOLEMESSAGE structure and calls *IRpcChannelBuffer::GetBuffer* to allocate the necessary space for return values and out-parameters. It then fills the buffer and returns from the call. The RPC Channel sends this new structure back to the facelet.

When *IRpcChannelBuffer::SendReceive* returns, the contents of the buffer into which the arguments were originally marshaled have been replaced by the return values and out-parameters from the remote object. The facelet unpacks these from the buffer, stores them in the proper places in memory, calls *IRpcChannelBuffer::FreeBuffer* to clean up, and returns to the client.

The other two member functions in *IRpcChannelBuffer* provide useful information for facelets. *GetDestCtx* returns the MSHCTX flags appropriate to the nature of the RPC connection. *IsConnected* indicates whether the connection to the remote object is still active—that is, whether a *SendReceive* call will even work. This can save a lot of time that would otherwise be spent waiting for the channel to time out before returning from *SendReceive. IsConnected* will always give a definite negative answer if the connection is dead, but a positive response is not so final: the server might die after the call returns, in which case the time-out will still occur. But subsequent calls to *IsConnected* will return the definite negative.

Keep in mind that regardless of what you might implement as

a custom interface, the RPC Channel is always implemented inside COM and is the core of standard marshaling. There is no COM API to create or access an RPC Channel itself—a facelet is explicitly given the RPC Channel's pointer when it's told to connect to a remote stublet. A stublet is always given a pointer to the RPC Channel whenever it's asked to invoke a member function in the remote object. Having access to an instantiation of this channel outside this context is simply not necessary.

## The Proxy Manager and Facelets

The proxy manager is an aggregation of any number of facelets. Whereas the proxy exposes an *IUnknown* along with *IMarshal* (its initialization interface), each facelet exposes one public interface to the client through aggregation with the proxy.

This collection of facelets in one proxy is entirely a matter of proxy implementation—the client doesn't care at all how that implementation is accomplished. The interfaces available through the proxy's *QueryInterface* are what matter to the client. The proxy's *QueryInterface* has to provide a pointer to whatever supported interface the client might request. Each pointer to each

different interface comes from an individual facelet for each interface. *QueryInterface* is, in fact, the part of a proxy that creates a new facelet when any given IID is requested the first time.[2] What the client then sees in the proxy is a single object with *IUnknown* and any other number of interfaces. The proxy implements *IUnknown* internally but obtains its other interfaces through aggregation on individual facelets.

A facelet itself is a small object that implements only two interfaces: *IRpcProxyBuffer* and whatever interface it knows how to marshal. Figure 6-3 shows one facelet with *IAnimal* and one with *IKoala*; both have *IRpcProxyBuffer*. *IRpcProxyBuffer* is rather special because it also acts as the controlling *IUnknown* implementation for the facelet and is the interface that the proxy obtains when it first creates a new facelet. This bends the aggregation rules slightly (by which the outer object must ask for *IUnknown* when creating the inner object), but because *IRpcProxyBuffer* is never exposed outside the proxy-facelet relationship, and because this relationship is specifically defined, this minor variation is not a problem. Also, the proxy can ask *IRpcProxyBuffer::QueryInterface* for a pointer to the other interface on that facelet. This other interface's *IUnknown* functions, as you would expect, delegate to the proxy's *IUnknown*,

as defined by normal aggregation rules. Thus, the client sees the proper *IUnknown* behavior through any interface pointer it can get from the proxy.[3]

Marshaling a single interface is the life purpose of any given facelet implementation. It is how a facelet knows which member functions in its public interface require a new proxy entirely. The problem that remains is how the facelet becomes aware of the *IRpcChannelBuffer* pointer through which it can communicate with the remote stub. This is the purpose of the *IRpcProxyBuffer* interface, which contains only two specific member functions:

```
interface IRpcProxyBuffer : IUnknown
   {
   HRESULT Connect(IRpcChannelBuffer *pRpcChannel);
   HRESULT Disconnect(void);
   };
```

A new proxy maintains a pointer to its RPC Channel, which it received through its *IMarshal::Unmarshal* interface. When the proxy creates a new facelet in its *QueryInterface*, it obtains an *IRpcProxyBuffer* pointer in return. It then calls *Connect*, passing to the facelet the proxy's *IRpcChannelBuffer* pointer. Whenever a facelet subsequently receives a call from a client to one of its member functions, it uses this *IRpcChannelBuffer* pointer to marshal arguments and to make the remote call. Simple! Of

course, sooner or later the client will call *Release* often enough to destroy the remote object and tear down all the magic in between, which means destruction of the proxy. (The client's *Release* calls will decrement the proxy's reference count to 0 along with the remote object's.) During destruction, the proxy calls *IRpcProxyBuffer::Disconnect* to ensure that the facelet is finished with the RPC Channel.

## The Stub Manager and Stublets

Now that we understand a little more about how a client talks to a proxy and how a proxy talks to the RPC Channel, let's see how the RPC Channel talks to the stub to complete an interface call to a remote object.

The stub manager is a collection of stublets, although aggregation is not used, as it is with the proxy. The stub as a whole manages the individual stublets, telling them when to connect to an interface in the remote object and when to delete themselves. Because a stub is used only in standard marshaling, COM provides the implementation internally in all cases. There is no direct access to this code.

COM creates a stub within *CoMarshalInterface* for any object that is using standard marshaling. In creating the stub, COM hands it the object's *IUnknown* pointer. The stub holds this pointer as is illustrated in Figure 6-3 on page 291. Now it waits until the RPC Channel (created in response to client-side actions) informs it of a client's call to some interface. At this point, the most the client can do is call some *IUnknown* member function because the client has not yet requested any other interface pointer. What happens when the client does call *QueryInterface* is the interesting part. Any *QueryInterface* call from the client ends up in the RPC Channel on the server side, which then privately informs the stub of the request. "Privately" here means that there is no set interface on the stub itself through which the RPC Channel communicates—such implementation is entirely internal to COM, so this is likely some call to a C++ member function.

In any case, the stub receives the *QueryInterface* request. In response, it creates a stublet appropriate for the IID being requested, and the stublet implements the single interface *IRpcStubBuffer*:

```
interface IRpcStubBuffer : IUnknown
   {
```

```
HRESULT Connect(IUnknown *pUnkServer);
void    Disconnect(void);
HRESULT Invoke(RPCOLEMESSAGE *pMessage
           , IRpcChannelBuffer *pChannel);
BOOL    IsIIDSupported(REFIID riid);
ULONG   CountRefs(void);
};
```

After creating this object, the stub calls *Connect*, passing the remote object's *IUnknown*. The stublet then calls *QueryInterface* to check whether the object actually supports the interface in question. If that query fails, the stublet returns E_NOINTERFACE to the stub, which returns it to the RPC Channel and back across to the client.

If the query is successful, the stublet has an interface pointer of type IID, which it stores internally before returning NOERROR from *Connect*. A successful *QueryInterface* is returned to the proxy, which then creates a new facelet for the same interface, but this facelet is not effectively connected to the newly created stublet.

Eventually the client will make a call to a member function in this newly obtained interface. That call enters the facelet that marshals arguments in the RPC Channel and calls *IRpcChannelBuffer::SendReceive.* This call is picked up by the server-side RPC Channel, which internally informs the stub that a

call has occurred. Remember that the proxy passed to *IRpcChannelBuffer::GetBuffer* is the IID of the interface being called. This IID shows up in the channel's private call to the stub, so the stub knows which stublet needs to handle the call and returns that stublet's *IRpcStubBuffer* pointer to the channel. The channel then calls *IRpcStubBuffer::Invoke*, passing the RPCOLEMESSAGE structure and its own *IRpcChannelBuffer* pointer, which gives the stublet all the information it needs to generate the call into the real remote object.

The other member functions in *IRpcStubBuffer* are rather trivial compared to *Invoke*. *Disconnect* tells the stublet to release the interface pointer it holds to the remote object: the stub itself will instruct all stublets to release their holds when the client has released all of its references to the remote object. *IsIIDSupported* is usually a simple function that returns TRUE if the stublet handles the given IID; otherwise, it returns FALSE. This function must also, however, verify that the remote object itself supports the interface. Most often this has already happened through a call to *Connect*, but if not, the stublet can perform such a check here. Finally, *CountRefs* returns to the remote object the number of reference counts that the stublet holds.

# How Everything Comes into (and out of) Memory

Having learned a little about the objects involved in remoting and their functionality, we can put the pieces together to see when and how each piece is brought into memory and how the pieces are connected to one another. As a beginning point, consider the following client code:

```
HRESULT            hr;
IClassFactory     *pIClassFactory;
IProvideClassInfo *pIPCI;
ITypeInfo         &pITypeInfo;

hr=CoGetClassObject(CLSID_Local, CLXCTX_LOCAL_SERVER, NULL
   , IID_IClassFactory, (void **)&pIClassFactory);

if (FAILED(hr))
   <error handling>;

hr=pIClassFactory->CreateInstance(NULL, IID_IProvideClassInfo
   , (void **)&pIPCI);
pIClassFactory->Release();

if (FAILED(hr))
   <error handling>;

hr=pIPCI->GetClassInfo(&pITypeInfo);

if (SUCCEEDED(hr))
   pITypeInfo->Release();

pIPCI->Release();
```

The calls to *CoGetClassObject* and *IClassFactory* could be combined into *CoCreateInstance*, which we'd usually do in writing concise code, but here we want to see all calls explicitly. This is a

typical sequence involving one call to a fundamental COM API function and calls to interfaces:

```
CoGetClassObject
IClassFactory::CreateInstance
IClassFactory::Release
IProvideClassInfo::GetClassInfo
ITypeInfo::Release
IProvideClassInfo::Release
```

A total of three server objects are involved here: the class factory, the object with *IProvideClassInfo*, and the object with *ITypeInfo*. This gives us the opportunity to explore how COM creates the proxy and stub for the first object (the class factory), how the proxies and stubs for the other two objects come into being, and how all of it is removed from memory as well.

From the client's perspective, this process involves only a few simple function calls. But COM is doing a tremendous amount of work to make transparent the remoting of three interfaces on three different objects. This work happens in the following phases:

Phase 1 *CoGetClassObject* causes the local server to be launched. The server calls *CoRegisterClassObject*, making that object appear in COM's global class factory registration table. *CoRegisterClassObject* creates the stub for this class factory as well.

Phase 2 Still in *CoGetClassObject*, COM creates a proxy and an RPC Channel for the class factory. At this point, only that object's *IUnknown* interface is available to the client process.

Phase 3 *CoGetClassObject* requests an *IClassFactory* pointer from the class factory, causing the creation of a facelet in the proxy manager and a stublet in the stub manager. The resulting client-side pointer, which is implemented on a facelet, is returned to the client.

Phase 4 The client calls *IClassFactory::CreateInstance*, which creates a new object as well as a new proxy with a facelet for *IProvideClassInfo*, a new stub with a stublet for the same interface, and a new RPC Channel for the new object.

Phase 5 The client calls *IClassFactory::Release*, which destroys the proxy and RPC Channel for the class factory but not the stub.

Phase 6 The client calls *IProvideClassInfo::GetClassInfo* to obtain an *ITypeInfo* pointer. This creates a new object, which means the creation of a new proxy, stub, and RPC Channel.

Phase 7 The client calls *ITypeInfo::Release*, which destroys that object and its remoting support but nothing else.

Phase 8 The client calls *IProvideClassInfo::Release*, destroying the object in the server and also terminating the server because this is the only remaining server object and no locks exist. The server starts shutdown and calls *CoRevokeClassObject*. This destroys the class factory and its stub (which were not destroyed in Phase 5).

After all of this is complete, no server will be in memory, no proxies, no stubs, no RPC Channels—just as it should be. Nothing will be in memory that wasn't there before the client executed its code. Of course, each phase in itself has a complex series of operations, so let's look at each one in turn.

## Phase 1: Launching the Server and Registering the Class Factory

As we learned in Chapter 5, *CoGetClassObject* delegates responsibility for locating and launching a server for some CLSID to the SCM (Service Control Manager). *CoGetClassObject* also checks for the *TreatAs* key (by calling *CoGetTreatAsClass*) to determine the correct CLSID to give to the SCM. For whatever local server CLSID is used, the SCM launches that server and returns some sort of connection information. This information can be thought of as an RPC handle, but as we'll see, that's not

entirely accurate: there's much more to it than that, most of which happens in the server process. *CoGetClassObject*, once it has asked the SCM to locate a server, waits patiently for the server to start and for it to register its class factory. Let's leave the client process spinning in this little loop while we look at the server process.

When the server is launched to service a component, it sees **-***Embedding* on its command line. In response, it initializes COM and creates its class factory object. At this point, nothing else is in memory except the COM Library, the server EXE itself, and its newly created class factory, to which the server has, say, an *IUnknown* pointer. The problem that Local/Remote Transparency solves is the creation of the structures necessary to allow a remote client to call member functions in this *IUnknown* interface. Fortunately for the server, COM makes the process simple: the server needs only to pass the *IUnknown* pointer to its class factory to *CoRegisterClassObject* and wait for calls to happen.

So what does happen when the server passes a pointer to *CoRegisterClassObject*? The COM Library loaded into the server's process maintains a table of class factories registered in that process. Each entry in this table includes an identifier for the class

factory (an integer called an object identifier [OID], not a GUID) and a pointer to the stub for that object. The stub itself manages the object's real interface pointer passed to *CoRegisterClassObject*.

Obviously, a task-specific table of registered class factories doesn't do us much good, especially considering that there are other registration mechanisms as well, such as the running object table. For that reason, COM also maintains a single global *object table* in shared memory, which is accessible to all instances of the COM Library in all processes and is used for the registration of any objects whatsoever. In this table, COM stores a pointer to the object's stub manager along with a process identifier (task handle) that associates this information with a machine-unique OID.

We can see where these tables come into play through the following sequence of steps performed in *CoRegisterClassObject*:

1. Check whether a multiple-use class factory for the CLSID is already registered, in which case fail with CO_E_OBJISREG. Otherwise, call *AddRef* to safeguard the object. (The reference count is now 1

as far as COM is concerned.)

2. Call *CoMarshalInterface* to determine whether the object wants custom marshaling or whether standard marshaling should be used. In the latter case, *CoMarshalInterface* will create an instance of the generic stub and retrieve from it a generic marshaling packet.

3. Create an entry in the global object table storing the marshaling packet (regardless of the form of marshaling). If standard marshaling is used, also store the stub pointer and its task handle, assigning an OID to the entry.

4. If standard marshaling is used, connect the stub to the class factory by passing it the factory's *IUnknown* pointer as passed to *CoRegisterClassObject*. The stub holds this pointer and calls *AddRef.* (The reference is now 2.)

5. In the class factory table, store the OID, the marshaling packet, the server CLSID, and the proxy

CLSID (also the stub pointer if standard marshaling is used). Include as well the *dwUsage* and *dwContext* flags passed to *CoRegisterClassObject.* This act creates the registration key returned to the server.

6. Call *Release* to reverse the safeguard *AddRef* call in step 1. (The reference count is now 1; the *AddRef* was called in step 4.)

At the end of this process, we have a new entry in the class factory table that identifies the stub for the object and the flags and proxy CLSID necessary to manage it. In addition, an entry in the global object table identifies the stub and the server task with a machine-unique OID. All that we've built so far is shown in Figure B-1. *CoRegisterClassObject* is complete and returns to the server, which completes its initialization and enters its message loop. The class factory is now available to the client, still waiting inside *CoGetClassObject.*

**Kraig Brockschmidt, *Inside OLE, 2nd edition.* Appendix B, EG2, dC**

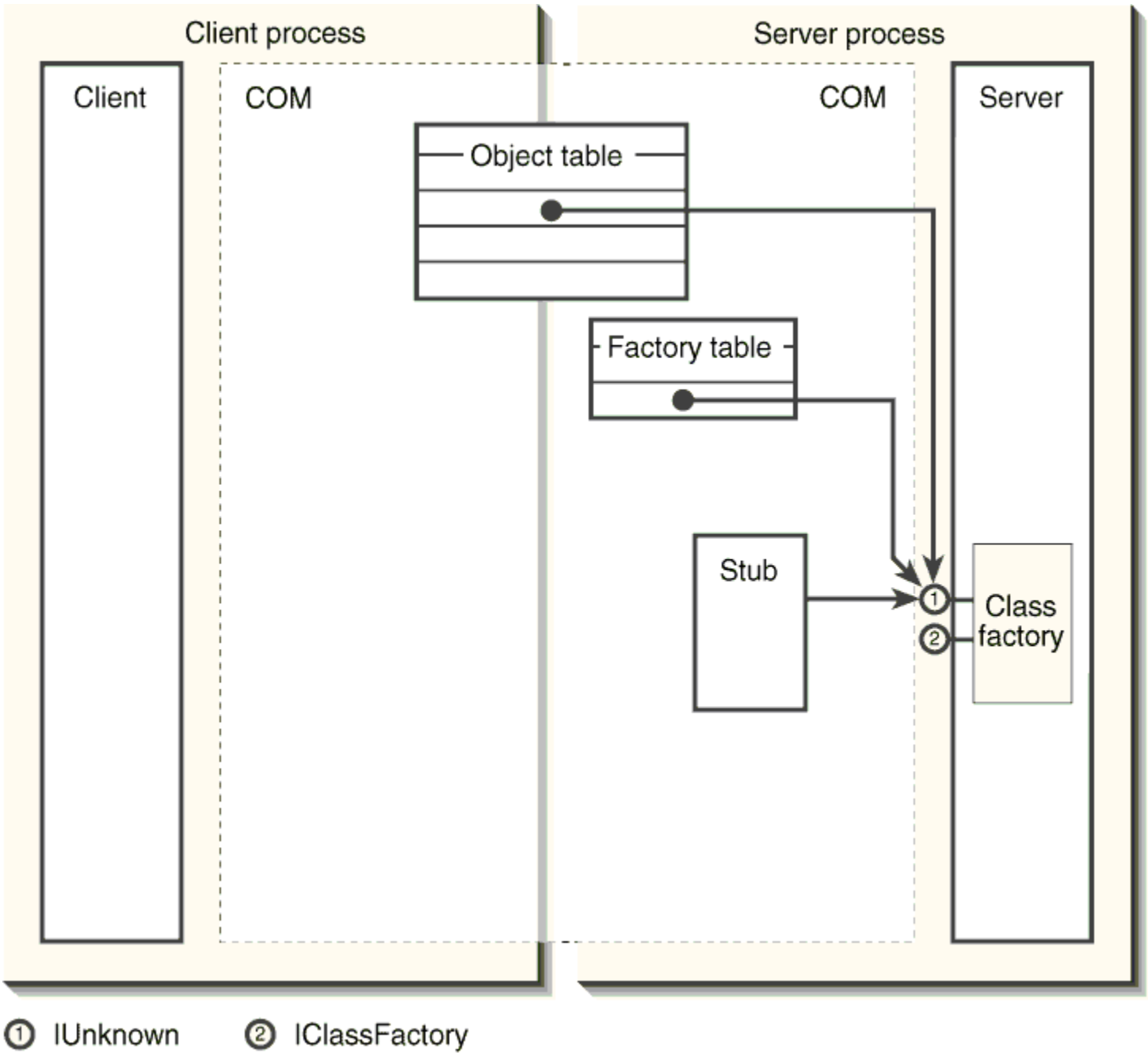

**Figure B-1.**
*The server-side results of* CoRegisterClassObject.

## Phase 2: Creation of the First Proxy and RPC Channel

While all the business of Phase 1 is going on, *CoGetClassObject*, and thus the client as a whole (at least that one thread), waits patiently for the new class factory to appear in the global object table. It is entirely possible, however, that the server is already running and that its class factory is already registered when the client calls *CoGetClassObject*, in which case the client doesn't have to wait. What really happens in this function is that it first checks whether the class factory is already registered, and if not, it waits until any new registration occurs in the global object table, checks again, and then continues to wait until a time-out occurs (5–30 minutes or so).

Because this example started from scratch, the first check for a class factory failed, and *CoGetClassObject* is simply waiting until a new registration happens. When that event occurs, *CoGetClassObject* checks whether that new registration is the CLSID it wants. This basically involves walking through the global

object table to find an OID for a class factory matching the CLSID. (There can be multiple single-use factories in the table, mind you.)

*CoGetClassObject* now has the OID for the remote object, and with that OID, the function can retrieve the proxy CLSID and marshaling packet necessary to connect the proxy to the stub (or to the object if custom marshaling is used). The process, which occurs in *CoUnmarshalInterface*, is as follows:

1. Create a proxy object through *CoCreateInstance(CLSCTX_INPROC_HANDLER | CLSCTX_INPROC_SERVER)* using the CLSID obtained from the stub. (Because CLSCTX_INPROC_* is used, there is no chance of winding up back in this process again with a different remote class factory.) This call creates either a custom proxy or the standard generic proxy using *CLSID_StdMarshal (00000017-0000-0000-C000-000000000046)*.

2. *CoCreateInstance* calls *CoGetClassObject*, which, for CLSIDs internal to COM (such as *CLSID_StdMarshal*), creates the object using its own

internal means. Otherwise, *CoGetClassObject* and *CoCreateInstance* proceed to instantiate the proxy exactly as they would for any other in-process object, as we saw in Chapter 5.

3. Connect the new proxy to the remote object by passing it the marshaling packet that points to *IMarshal::UnmarshalInterface*. When standard marshaling is involved, the packet includes the necessary *IRpcChannelBuffer* pointer, which the proxy then holds (calling *AddRef*) until destroyed (when it calls *Release*).

After this process, *CoGetClassObject* (effectively client code because we're outside the proxy itself) now has in hand an *IUnknown* pointer to the new proxy object. By this time, we've created, in both processes, all that is shown in Figure B-2 below. *CoGetClassObject* can call an *IUnknown* function through this pointer, and the proxy will marshal that call through the RPC Channel to the stub. The stub unmarshals the call and sends it to the object. Or does it? This is true for *QueryInterface*, as we'll see below, but for *AddRef* and *Release*, the generic proxy does not forward every call to the stub. The simple reason is that a single

reference count, which the stub has already by virtue of its existence, will keep the remote object alive and the server running. Therefore, any number of additional calls to *AddRef* and matching calls to *Release* really don't accomplish anything—only the last call to *Release* matters. The proxy doesn't bother to forward every *AddRef* and *Release* call, and this is, as we discovered in Chapter 5, the reason why the reference count returned from *AddRef* and *Release* for a local or remote object is some large and meaningless number unless the return value is 0. COM's generic proxy simply implements it that way.

**Kraig Brockschmidt, *Inside OLE, 2nd edition.* Appendix B, EG2, dC**

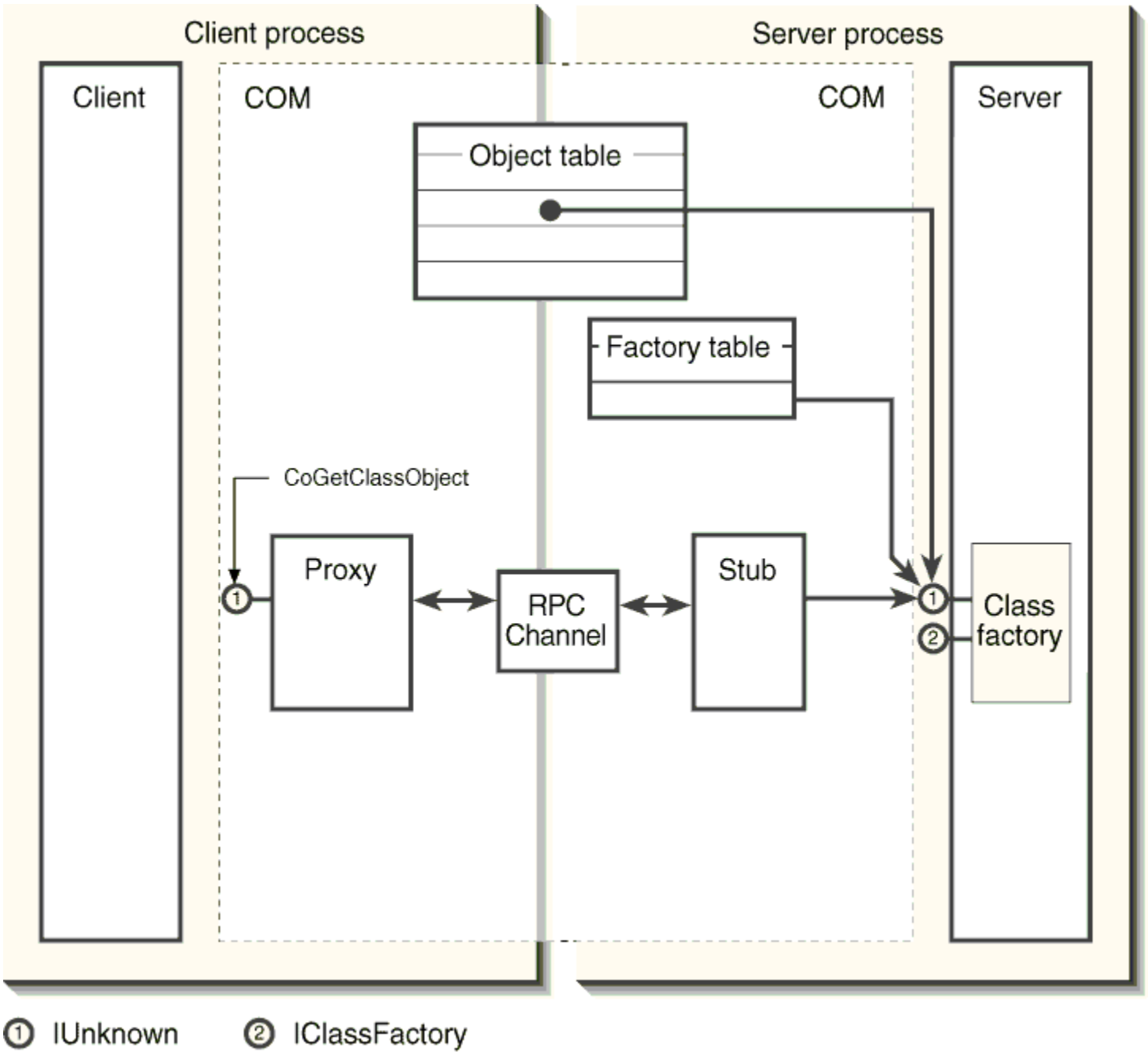

**Figure B-2.**
*Objects in memory by the time* CoGetClassObject *obtains an* IUnknown *pointer.*

## Phase 3: Creation of an Interface Proxy and Stub for *IClassFactory*

*CoGetClassObject* is almost ready to return a pointer to the client. In this example, the client originally asked for *IClassFactory*, but *CoGetClassObject* has only an *IUnknown* pointer, implemented in the proxy. All it needs to do is to call *QueryInterface*; the proxy has to get an *IClassFactory* pointer. This involves quite a bit of new processing in the proxy and the stub because at the moment the only open communication path is through *IUnknown*. To handle this, the proxy must create a new *IClassFactory* facelet and hand it the *IRpcChannelBuffer* pointer through which that facelet can make its calls. At the same time, a new *IClassFactory* stublet in the remote stub must be created, with the stublet maintaining a pointer to the remote object's actual interface.

The implementations of the facelet and stublet for any particular interface are now provided through a proxy/stub server, as mentioned in Chapter 6. The server is an in-process server

registered as follows (*ProxyStubClsid* is used on 16-bit systems):

```
\
   Interface
       {<IID>} = <name of interface>
           NumMethods = <total number of interface members>
           BaseInterface = <{IID} of base interface>
           ProxyStubClsid32 = <{CLSID} of a server for the marshaler>
```

These entries map an IID to a CLSID whose server implements the specific marshalers for this interface. The *BaseInterface* entry frees these marshalers from having to implement marshalers for every member function when other marshalers already exist for those members of the base interface. COM will use the marshaler for the base interface for any of its members. In any case, the class factory in the server identified with *ProxyStubClsid32* implements the interface *IPSFactory* through which the proxy or stub can create either a facelet or a stublet:

```
interface IPSFactoryBuffer : IUnknown
   {
   HRESULT CreateProxy(IUnknown *pUnkOuter, REFIID riid
       , IRpcProxyBuffer *ppProxy, void **ppv);
   HRESULT CreateStub(REFIID riid, IUnknown *pUnkServer
       , IRpcStubBuffer **ppStub);
   }
```

With this interface, you can see that a Proxy/Stub Factory, or simply, PSFactory, can create both an interface proxy and an interface stub for a single interface. When a proxy or a stub needs

a new facelet or stublet, it goes to the registry, looks up the *ProxyStubClsid32* for the IID in question, and then calls *CoGetClassObject(IID_IPSFactoryBuffer, CLSCTX_INPROC_HANDLER | CLSCTX_INPROC_SERVER)*. When an *IPSFactoryBuffer* pointer is returned, the proxy calls *CreateProxy* and the stub calls *CreateStub* in that interface. Both these member functions take an IID argument that identifies the interface in question. This argument serves the same purpose as the CLSID passed to *DllGetClassObject.* In this latter case, the CLSID allows an in-process server to handle multiple CLSIDs. In the PSFactory case, the IID allows that factory to create a different object for each different interface, as it really must anyway.

The *CreateProxy* function actually returns two interface pointers. (Both have *AddRef* called through them, of course.) One is a pointer to the facelet's *IRpcProxyBuffer*, and the other is a pointer to the interface that the proxy exposes to the client. This second interface must be of the type matching the *riid* argument. In a sense, *CreateProxy* has a built-in *QueryInterface* because the proxy always needs both pointers at the same time. As we've seen, the proxy calls *IRpcProxyBuffer::Connect* shortly after this to make the facelet aware of the RPC Channel to the stub.

On the other hand, *CreateStub* only returns the *IRpcStubBuffer* pointer to the new stublet—there's no other interface to worry about. As we've also seen, the stub calls *IRpcStubBuffer::Connect* shortly after this, passing the remote object's *IUnknown*. In *Connect*, the stublet calls *QueryInterface* through that pointer to obtain the one it holds on to for later handling of *IRpcStubBuffer::Invoke*.

The remaining argument to both *CreateProxy* and *CreateStub* is an *IUnknown* pointer, but keep in mind that *CreateProxy* takes a *pUnkOuter*, whereas *CreateStub* takes a *pUnkServer*. The pointer given to *CreateProxy* is the outer proxy's *IUnknown,* the controlling unknown for the whole proxy. The facelet must delegate all *IUnknown* calls made to its public—and only its public—interface. This allows the proxy to control the interfaces available to the client. The facelet does not delegate any *IUnknown* calls to its *IRpcProxyBuffer* interface because that acts as the controlling *IUnknown* for the facelet.

The *pUnkServer* passed to *CreateStub* is entirely different and is the same pointer that can be passed to *IRpcStubBuffer::Connect* later on; in either case, it is the remote object's *IUnknown*. This argument to *CreateStub* can be NULL,

meaning that the stub must call the stublet's *Connect* before ever calling *IRpcStubBuffer::Invoke*. If the stub passes the object's *IUnknown* to *CreateStub*, the stublet calls its own *Connect* internally so that the stub can call *Invoke* without calling *Connect* itself.

Now that we understand how facelets and stublets come into memory, spelling out the sequence of operations that makes it happen is fairly simple. It begins with a call from *CoGetClassObject* into the proxy's *QueryInterface* with *IID_IClassFactory*:

1. The proxy's *QueryInterface* checks whether the IID is *IID_IUnknown* and, if so, returns its own pointer.

2. Otherwise, *QueryInterface* checks whether a facelet for the IID is already present in this proxy. If so, it retrieves that facelet's public interface, calls *AddRef* through it, and returns that pointer.

3. If the facelet is not present, the proxy first verifies that the remote object itself supports the IID

being requested. The proxy marshals the necessary arguments into the RPC Channel and calls *IRpcChannelBuffer::SendReceive*, which is picked up in the server process and sent to the stub.

4. The stub unmarshals the arguments and calls the remote object's *QueryInterface*. If that function returns E_NOINTERFACE, the same error code is propagated all the way back to the client, eventually becoming the value returned from *CoGetClassObject.*

5. If *QueryInterface* succeeds, the stub instantiates a stublet for the IID using the PSFactory entries in the registry as described above. It calls *IPSFactoryBuffer::CreateStub* followed by *IRpcStubBuffer::Connect*, handing the stublet that object's *IUnknown.*

6. The stublet calls *QueryInterface* to obtain and save a pointer to the appropriate interface on the object and returns to the stub. With the stublet fully created and initialized, the stub returns from the

RPC Channel's call.

7. The proxy's call to *IRpcChannelBuffer::SendReceive* returns successfully, so the proxy knows that a remote stublet has been created. It then creates a corresponding facelet inside itself using aggregation through *IPSFactoryBuffer::CreateProxy*.

8. The proxy calls *IRpcProxyBuffer::Connect*, passing the *IRpcChannelBuffer* pointer that COM passed the proxy during its own connection. The facelet stores this pointer and considers itself connected.

9. The proxy now knows that both stublet and facelet exist, and it has in hand the correct interface pointer, in this case an *IClassFactory* pointer, which it returns to *CoGetClassObject*.

We're now back inside *CoGetClassObject*, just after its call to *QueryInterface(IID_IClassFactory)*. If all was successful, *CoGetClassObject* now returns the *IClassFactory* pointer to the

client. Everything we need to make local or remote calls through the interface is now in memory, as shown in Figure B-3.

**Kraig Brockschmidt, *Inside OLE, 2nd edition.* Appendix B, EG2, dC**

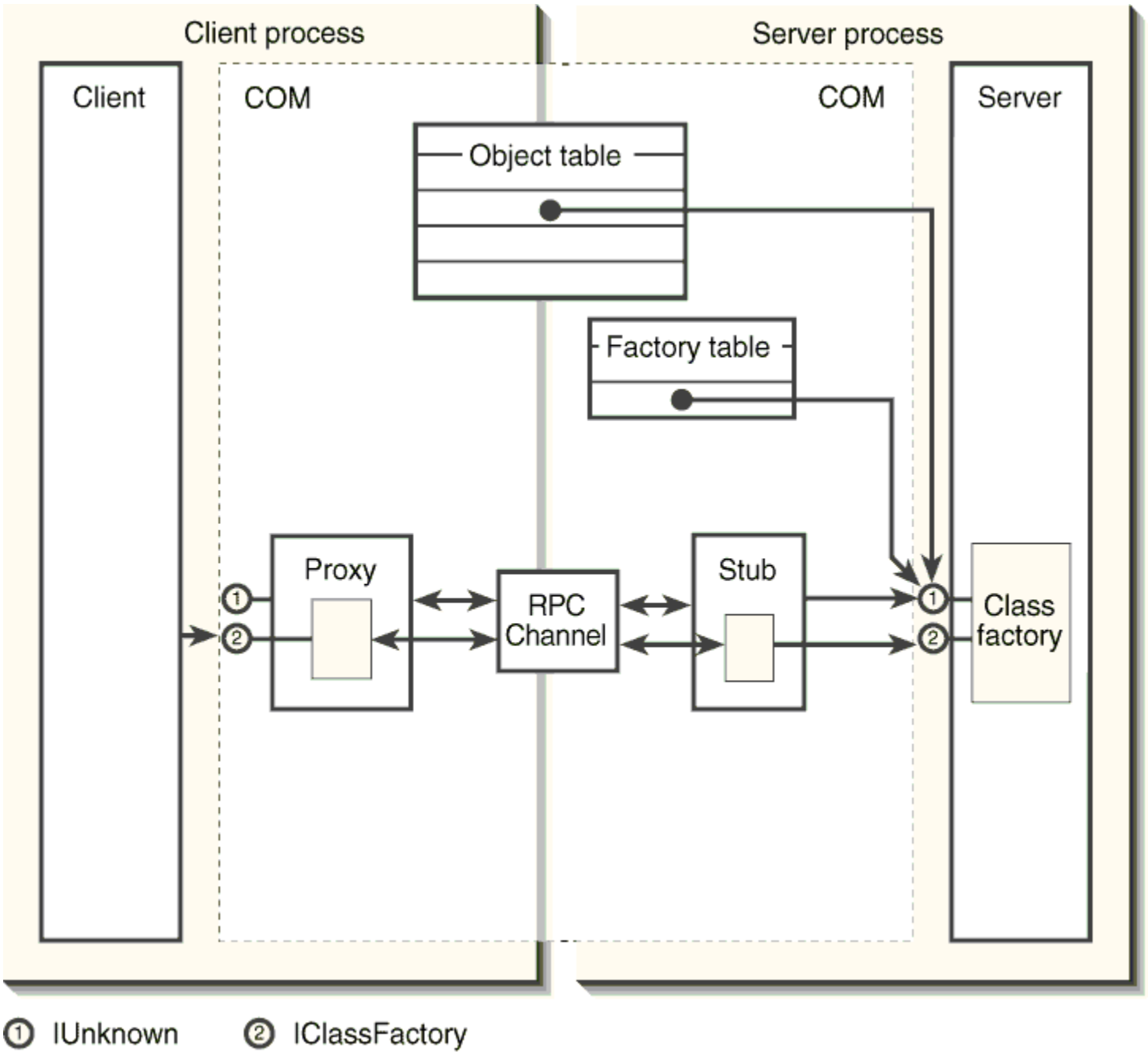

**Figure B-3.**
*Objects in memory by the time* CoGetClassObject *returns an* IClassFactory *pointer to the client, ready for making cross-process or cross-machine calls.*

## Phase 3½: Intermission

Wow! We've really covered a lot of the Local/Remote Transparency architecture in the previous pages. It's worth it to take a short break to realize just how far we've come. Starting from scratch, with nothing in memory but the client's code and its instance of the COM Library, we've seen how a simple call to *CoGetClassObject* spawns a flurry of activity inside COM: launching the server, registering the class factory in object tables, instantiating proxies and stubs and facelets and stublets, and connecting everything together through the RPC Channel. That's a lot of work! What we now have after all this work is a client with an *IClassFactory* pointer, and any call through that pointer winds up as a call to an object that exists in another process or on another machine. This, my friends, is *way hot* technology.

Well, let's see now. I need to go to the post office and mail a

package. I'll be back shortly. Let me pop in a CD for you while you're waiting.

> Take a walk outside myself
> In some exotic land
> Greet a passing stranger
> Feel the strength in his hand
> Feel the world expand
>
> *Hand Over Fist,* from the "Presto" CD by Rush (the band)

OK, I'm back. Sorry I took a little longer than I expected, but there was a long line at the post office.[4] Anyway, let's explore what happens as we execute the rest of the client code.

## Phase 4: Creation of a New Object and Its Remoting Support

Now that the client has an *IClassFactory* pointer, it calls *CreateInstance*, asking for an *IProvideClassInfo* pointer in return. This call goes directly to the proxy's *IClassFactory* facelet because that's the object implementing the interface. Well, "implement" is the wrong word—this facelet actually marshals this interface, relying on the remote object for the complete implementation. Once again, this is why a proxy is often called a handler.

Inside its implementation of *CreateInstance*, the facelet takes whatever arguments are required in the remote object and marshals them in an RPC Channel buffer. The only argument that really needs marshaling is the *riid*, which contains *IProvideClassInfo*; the *pUnkOuter* is unimportant because aggregation is in-process only, and the pointer stored in *ppv* will be that or some new facelet in the client process. So *riid* is all that the facelet stuffs into the RPC Channel before calling *IRpcChannelBuffer::SendReceive.*

Let me point out how this example of an interface such as *IClassFactory* illustrates why standard marshaling needs interface-specific facelets and stublets—only small pieces of code that understand an interface can know which arguments to marshal and which ones to manipulate solely on the client side. Marshaling is different for every method of every interface; the facelets and stublets encapsulate that intelligence.

To continue, the facelet's *SendReceive* call is picked up by the RPC Channel in the server process and sent to the stub connected to that channel. The RPC Channel privately tells the stub which interface is being called so that the stub passes the

call to the correct stublet through *IRpcStubBuffer::Invoke.* The *IClassFactory* stublet is told to invoke *CreateInstance*, in which it knows that the RPC Channel buffer holds the IID to request. It unmarshals that single argument, allocates a variable to hold the new pointer (probably a temporary stack variable), and calls the real class factory's *CreateInstance* with NULL, the IID, and the pointer to the pointer variable.

If *CreateInstance* returns successfully, the *IClassFactory* stublet now holds a brand-new interface pointer—*IProvideClassInfo* in our example here—attached to a brand-new object that is unrelated to the class factory that already exists. The stublet, by virtue of understanding what *CreateInstance* does, knows that it must marshal this pointer to the new object. At this point, the stublet is in the same situation as *CoRegisterClassObject*: it has a pointer to a server-process object, which it now makes available to the client process by calling *CoMarshalInterface*, which then creates the new stub as needed.

Now the original *IClassFactory* stublet is satisfied that it has built the necessary server-side structures, so it returns from *SendReceive*, storing NOERROR as the HRESULT as well as the new object's OID. Because the stublet cannot marshal the new

object's interface pointer to the facelet, it must marshal the OID to identify the new object. After we're back in the client process, the facelet unmarshals the HRESULT; if that code indicates failure, the facelet returns failure to the client. Otherwise, it takes the new OID and repeats the process of creating a new proxy for the remote object through *CoUnmarshalInterface*. The facelet now has the proxy's *IUnknown* pointer in hand and can call *QueryInterface* with *IID_IProvideClassInfo* to have the proxy create a new facelet for this new interface and connect that facelet to the RPC Channel. Again, this uses the same mechanisms we've already explored.

The *IClassFactory* facelet's calls to *QueryInterface* result in a new client-side interface pointer (to the *IProvideClassInfo* facelet in the new proxy) that it can now, finally, return from the client's original call to *IClassFactory::CreateInstance.* The necessary remoting architecture now exists in memory for both the class factory and the object it created, as illustrated in Figure B-4. You can also picture the intermediate stages that were shown in Figures 6-3 and 6-4 on pages 291 and 307, respectively; here it's the same process for another object.

**Kraig Brockschmidt, *Inside OLE, 2nd edition.* Appendix B, EG2, dC**

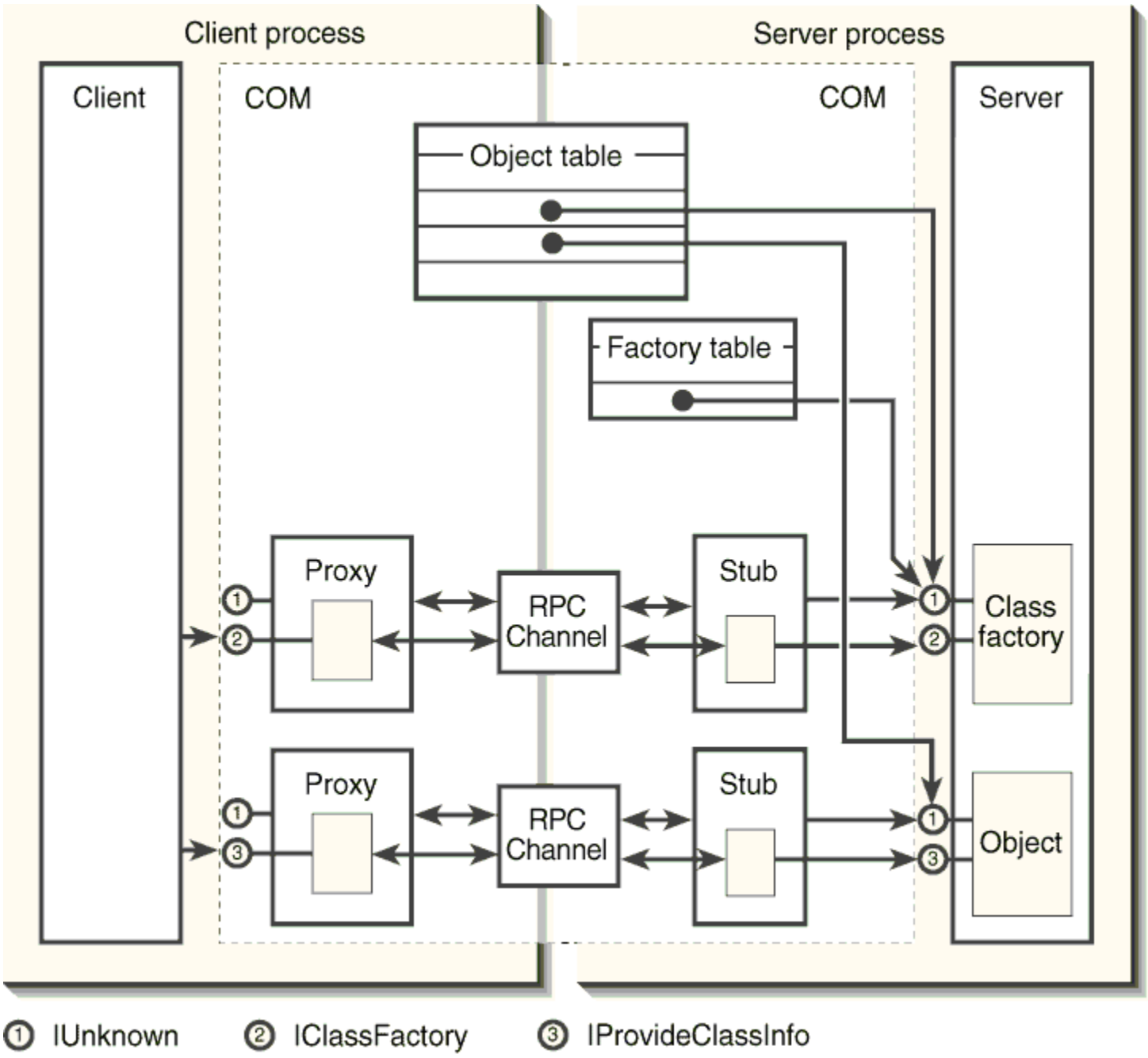

**Figure B-4.**
*Objects in memory after* IClassFactory::CreateInstance *returns with a new* IProvideClassInfo *pointer.*

## Phase 5: Releasing the Class Factory and Destroying Its Proxy

Now that the client has obtained the *IProvideClassInfo* pointer it wanted from *IClassFactory::CreateInstance*, it is finished with the class factory and calls *IClassFactory::Release*. Because the facelet was created as an inner object in the proxy's aggregation, this call is delegated to the proxy's *IUnknown::Release* implementation. The proxy maintains the total number of reference counts that the client believes it has on the remote object. If this count is nonzero after the decrement in *Release*, the proxy returns some meaningless nonzero number (not the actual internal count).

In our example, the client's *IClassFactory* pointer is the only reference, so the proxy's *Release* decrements this count to 0. The proxy thus knows that it has to start its self-destruction process, which proceeds as follows:

1. The proxy iterates over all of its contained facelets, calling each *IRpcProxyBuffer::Disconnect*

(which calls *IRpcChannelBuffer::Release*) followed by *IRpcProxyBuffer::Release*, causing the facelet to delete itself.

2. The proxy sends an *IUnknown::Release* over the RPC Channel to the stub, which removes the only reference between the two processes.

3. The stub picks up this *IUnknown::Release* call and decrements an internal count of the number of client connections to this stub.

4. If the last connection is being removed, the stub iterates over all of its contained stublets, calling each *IRpcStubBuffer::Disconnect* (which calls the real object's *Release*) followed by *IRpcStubBuffer::Release*, which deletes the stublet.

5. If the stub maintains its own pointer to the object's *IUnknown*—in other words, a strong connection—both the object and the stub remain alive. This is true for a registered class factory, which remains registered and available to other

clients that might call *CoGetClassObject* to connect to this already running class factory. If the stub does not maintain such a pointer—a weak connection—the object may well be dead, and the stub deletes itself because it no longer has any references to that object, alive or dead. Some of the issues surrounding cases in which the stub disappears are described in “Strong and Weak Connections” in Chapter 6.

6. The stub returns from *IUnknown::Release*, and we’re back in the proxy, which now knows that its responsibility is taken care of. The proxy calls *IRpcChannelBuffer::Release* to remove the last reference to the channel.

7. The RPC Channel closes the RPC connection between the two processes and deletes itself.

8. The proxy finally deletes itself and is removed from memory.

At the end of all this, nothing is left in the client process but the

client, regardless of whether that object is still running. The class factory's proxy is gone, along with the RPC Channel and all facelets, as it should be. However, the class factory and its stub still exist in the server process, as shown in Figure B-5.

**Kraig Brockschmidt, *Inside OLE, 2nd edition.* Appendix B, EG2, dC**

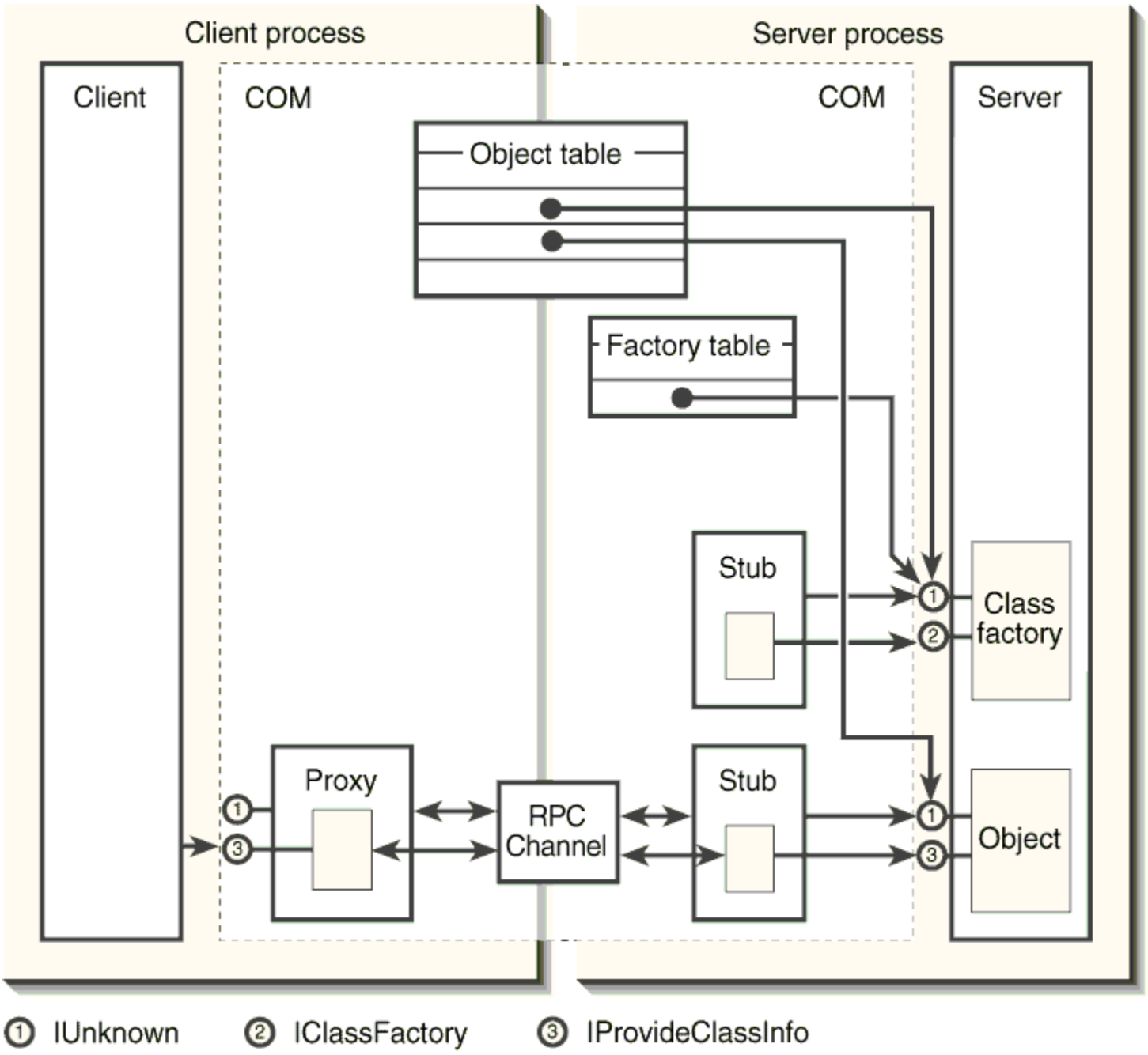

**Figure B-5.**
*Objects in memory after a client calls* IClassFactory::Release*. The class factory remains in memory because the server must call* CoRevokeClassObject *to destroy it.*

## Phase 6: Obtaining a Pointer to Yet Another Object

After releasing the class factory, the client still has a pointer to the second object's *IProvideClassInfo* interface, so a proxy for that object is still in memory. The client now calls *IProvideClassInfo::GetClassInfo*, which returns an *ITypeInfo* pointer to yet another separate object. Just as the *IClassFactory* facelet and stublet recognize that *CreateInstance* creates a new and separate object, so do the *IProvideClassInfo* facelet and stublet recognize that *GetClassInfo* does exactly the same thing.

In fact, as far as the creation of a new proxy and stub for the new object is concerned, *CreateInstance* and *GetClassInfo* involve the same operations that we've already seen. This is true generally: any interface function that creates a new object and returns an interface pointer to that new object necessitates the creation of a new proxy and stub in client and server processes,

including a facelet in the proxy and a stublet in the stub. Same problem, same solution. A function such as *CoGetClassObject* is special only in that it is the first time any of this happens; afterward, the same process occurs from within interface calls themselves.

## Phase 7: Releasing an Object and Removing Its Complete Remoting Support

Having obtained the *ITypeInfo* pointer it was looking for all along, the client does whatever it needs to with that interface and eventually calls its *Release.* The proxy called knows that this is the client's only reference to the remote object, so it performs the same steps as described earlier in Phase 5. In this situation, however, the remote object itself is destroyed because the stub's only reference to that object is through its contained *ITypeInfo* stublet. Thus, the object deletes itself, the stub deletes itself, the proxy deletes itself and all its stublets, and the RPC Channel disappears, leaving no trace of the object, stub, or proxy anywhere in memory. Cleanup is complete.

The server, however, is still running because it still has at

least one active object, the one with *IProvideClassInfo*, so its object count is still 1. The memory snapshot is the same as that shown in Figure B-5. The remoting objects for *ITypeInfo* came into memory and went right back out again.

## Phase 8: Terminating the Server and Revoking the Class Factory

The final act in this show is the client's call to *IProvideClassInfo::Release.* As with any other object, this last call to *Release* destroys the proxy and the facelets on the client side along with the object, the stub, and the stublets on the server side. This leaves only the few server-side objects in memory, exactly as is shown in Figure 6-3 on page 291. However, because this was the last object being maintained in the server, its termination conditions are met, and the server begins its shutdown process.

As we saw in Chapter 5, part of this process is the server's call to *CoRevokeClassObject*. This function performs the following steps:

1. Removes the class factory from the class factory

table.

2. Tells the stub to disconnect from the class factory, which calls that class factory's *Release.* This will be the final *Release* for that object, which now deletes itself.

3. Removes the object from the global object table, making it completely unavailable.

4. Deletes the stub object.

*CoRevokeClassObject* then returns to the server, which completes its shutdown by calling *CoUninitialize* before exiting its message loop and *WinMain*, thus unloading itself completely and terminating the entire process (which unloads the COM Library in that process as well).

If we now looked in memory, we would see no server process, no server or COM Library in such a process, no stubs, no stublets, no proxies, no facelets, and no entries in the global object table. In fact, there would be nothing, absolutely nothing that was not there before the client made its first call to *CoGetClassObject*,

which is exactly as it should be.

Q.E.D. (Aren't you glad *you* didn't have to implement all of this yourself?)

---

[1] This CBS is the COM Broadcasting Service, which, well, can't be real or else a few trademark lawyers from the real CBS might come a knockin' and a litigatin'!

[2] A proxy can either maintain any previously instantiated facelets or allow facelets to delete themselves when their internal reference counts go to 0. In this case, the proxy's *QueryInterface* may need to re-create them later. This is a proxy implementation decision and does not affect client or facelet implementation.

[3] You can prove to yourself that a client cannot access *IRpcProxyBuffer* by modifying Chapter 5's ObjectUser to query one of the EKoala servers for IID_IRpcProxyBuffer. The call will always come back E_NOINTERFACE.

[4] Really, I mailed a package to a new baby cousin. I did go to the post office today (1/17/95), and I did put in this particular CD, and I did play this particular song. Whaddya mean you didn't hear it?