1Interface Remoting

In COM, clients communicate with objects solely through the use of vtable-based interface instances. The state of the object is manipulated by invoking functions on those interfaces. For each interface method, the object provides an implementation that does the appropriate manipulation of the object internals.

Interface remoting provides the infrastructure and mechanisms to allow a method invocation to return an interface pointer to an object that is in a different process, perhaps even on a different machine. The infrastructure that performs the remoting of interfaces is transparent to both the client and the object server. Neither the client or object server is necessarily aware that the other party is in fact in a different process.

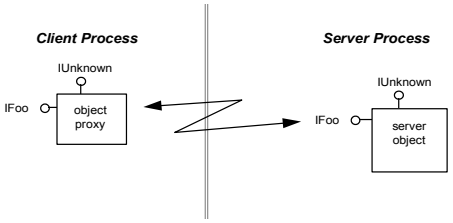
This chapter first explains how interface remoting works giving mention to the interfaces and COM API functions involved. The specifications for the interfaces and the API functions themselves are given later in this chapter. There is also a brief discussion about concurrency management at the end of the chapter that involves an interface called IMessageFilter.

1.1How Interface Remoting Works

The crux of the problem to be addressed in interface remoting can be stated as follows:

> "Given an already existing remoted-interface connection between a client process and a server process, how can a method invocation through that connection return a new interface pointer so as to create a second remoted-interface connection between the two processes?"

We state the problem in this way so as to avoid for the moment the issue of how an initial connection is made between the client and the server process; we will return to that later.



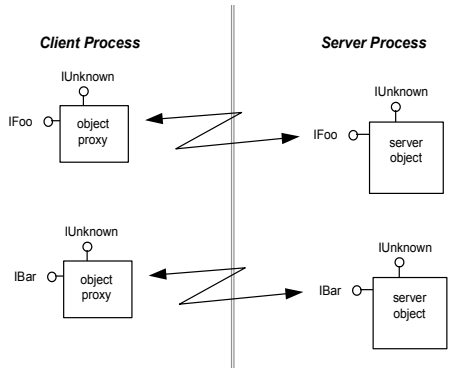
Let's look at an example. Suppose we have an object in a server process which supports an interface IFoo, and that interface of the object (and IUnknown) has sometime in the past been remoted to a client process through some means not here specified. In the client process, there is an object proxy which supports the exact same interfaces as does the original server object, but whose *implementations* of methods in those interfaces are special, in that they forward calls they receive on to calls on the real method implementations back in the server object. We say that the method implementations in the object proxy *marshal* the data, which is then conveyed to the server process, where it is *unmarshaled.* That is, "marshaling" refers to the packaging up of method arguments for transmission to a remote process; "unmarshaling" refers to the unpackaging of this data at the receiving end. Notice that in a given call, the method arguments are marshaled and unmarshaled in one direction, while the return values are marshaled and unmarshaled in the other direction.

For concreteness, let us suppose that the IFoo interface is defined as follows:

```
interface IFoo : IUnknown {
    IBar *      ReturnABar();
    };
```

If the in the client process pFoo->ReturnABar() is invoked, then the object proxy will forward this call on to the IFoo::ReturnABar() method in the server object, which will do whatever this method is supposed to do in order to come up with some appropriate IBar*. The server object is then required to return this IBar* back to the client process. The act of doing this will end up creating a second connection between the two processes:

Copyright © 1995 Microsoft Corporation
All Rights Reserved



It is the procedure by which this second connection is established which is the subject of our discussion here. This process involves two steps:

1. On the server side, the IBar* is packaged or marshaled into a data packet.
2. The data packet is conveyed by some means to the client process, where the data it contains is unmarshaled to create the new object proxy.

The term "marshaling" is a general one that is applied in the industry to the packaging of any particular data type, not just interface pointers, into a data packet for transmission through an RPC infrastructure. Each different data type has different rules for how it is to marshaled: integers are to be stored in a certain way, strings are to be stored in a certain way, etc.[1] Likewise, marshaled interface pointers are to be stored in a certain way; the Component Object Model function CoMarshalInterface() contains the knowledge of how this is to be done (note that we will in this document not mention further any kind of marshaling other than marshaling of interface pointers; that subject is well-explored in existing RPC systems).

The process begins with the code doing the marshaling of the returned IBar* interface. This code has in hand a pointer to an interface that it knows in fact to be an IBar* and that it wishes to marshal. To do so it calls CoMarshalInterface(). The first step in CoMarshalInterface() involves finding out whether the object of which this is an interface in fact supports *custom object marshaling* (often simply referred to as "custom marshaling"). Custom object marshaling is a mechanism that permits an object to be in control of creation of remote object proxies to itself. In certain situations, custom object marshaling can be used to create a more efficient object proxy than would otherwise be the case.[2] Use of custom marshaling is completely optional on the object's part; if the object chooses not to support custom marshaling, then *standard interface marshaling* is used to marshal the IBar*. Standard interface marshaling uses a system-provided object proxy implementation in the client process. This standard implementation is a generic piece of code, in that it can be used as the object proxy for any interface on any object. However, the act of marshaling (and unmarshaling) method arguments and return values is inherently interface-specific, since it is highly sensitive to the semantics and data types used in the particular methods in question. To accommodate this, the standard implementation dynamically loads in interface-specific pieces of code as needed in order to do the parameter marshaling.

We shall discuss in great detail in a moment how standard interface marshaling works. First, however, we shall review custom object marshaling, as this provides a solid framework in which standard marshaling can be better understood.

1.2Architecture of Custom Object Marshaling

Imagine that we are presently in a piece of code whose job it is to marshal an interface pointer that it has in hand. For clarity, in what follows we'll refer to this piece of code as the "original marshaling stub." The general case is that the original marshaling stub does not *statically*[3] know the particular interface identifier (IID) to which the pointer conforms; the IID may be passed to this code as a second parameter. This is a common paradigm in the Component Object Model. Extant examples of this paradigm include:

[1] In fact, there exist several standard sets of rules, each promoted by a different organization. Two common such sets of rules are known as "Network Data Representation" (NDR) and "External Data Representation" (XDR) chiefly promoted respectively by the Open Software Foundation and Sun Microsystems. ASN.1 is another standard for the same sort of technology.

[2] Notice here that we're only discussing the marshaling of pointers to interfaces, and that the term "custom object marshaling" applies only to the marshaling of this data type. In general in a given remote procedure call the many other kinds of data which appear as function parameters also needs to be marshaled: strings, integers, structures, etc. We shall not concern ourselves here with such other data types, but instead concentrate our discussion on marshaling interface pointers.

[3] i.e.: at compile time of the original marshaling stub

Copyright © 1995 Microsoft Corporation
All Rights Reserved

```
IUnknown::QueryInterface(REFIID riid, void** ppvObject);
IOleItemContainer::GetObject(..., REFIID riid, void** ppvObject);
IClassFactory::CreateInstance(..., REFIID riid, void** ppvNewlyCreatedObject);
```

Let us assume the slightly less general case where the marshaling stub in fact does know a little bit about the IID: that the interface in fact derives from IUnknown. This is a requirement for remoting: it is not possible to remote interfaces which are not derived from IUnknown.

To find out whether the object to which it has an interface supports custom marshaling, the original marshaling stub simply does a QueryInterface() for the interface IMarshal. That is, an object signifies that it wishes to do custom marshaling simply by implementing the IMarshal interface. IMarshal is defined as follows:

```
[
   local,
   object,
   uuid(00000003-0000-0000-C000-000000000046)
]
interface IMarshal : IUnknown {
      HRESULT GetUnmarshalClass ( [in] REFIID riid, [in, unique] void *pv,
            [in] DWORD dwDestContext, [in, unique] void *pvDestContext,
            [in] DWORD mshlflags, [out] CLSID *pCid);
      HRESULT GetMarshalSizeMax ([in] REFIID riid, [in, unique] void *pv,
            [in] DWORD dwDestContext, [in, unique] void *pvDestContext,
             [in] DWORD mshlflags, [out] DWORD *pSize);
      HRESULT MarshalInterface ([in, unique] IStream *pStm, [in] REFIID riid, [in, unique] void *pv,
            [in] DWORD dwDestContext, [in, unique] void *pvDestContext, [in] DWORD mshlflags);
      HRESULT UnmarshalInterface ( [in, unique] IStream *pStm, [in] REFIID riid, [out] void **ppv);
      HRESULT ReleaseMarshalData ( [in, unique] IStream *pStm);
      HRESULT DisconnectObject ( [in] DWORD dwReserved);
}
```

The idea is that if the object says "Yes, I do want to do custom marshaling" that the original marshaling stub will use this interface in order to carry out the task. The sequence of steps that carry this out is:

1. Using GetUnmarshalClass, the original marshaling stub asks the object which kind of (i.e.: which class of) proxy object it would like to have created on its behalf in the client process.
2. (optional on the part of the marshaling stub) Using GetMarshalSizeMax, the stub asks the object how big of a marshaling packet it will need. When asked, the object *will* return an upper bound on the amount of space it will need.[4]
3. The marshaling stub allocates a marshaling packet of appropriate size, then creates an IStream* which points into the buffer. Unless in the previous step the marshaling stub asked the object for an upper bound on the space needed, the IStream* must be able to grow its underlying buffer dynamically as IStream::Write calls are made.
4. The original marshaling stub asks the object to marshal its data using MarshalInterface.

We will discuss the methods of this interface in detail later in this chapter.

At this point, the contents of the memory buffer pointed to by the IStream* together with the class tag returned in step (1) comprises all the information necessary in order to be able to create the proxy object in the client process. It is the nature of remoting and marshaling that "original marshaling stubs" such as we have been discussing know how to communicate with the client process; recall that we are assuming that an initial connection between the two processes had already been established. The marshaling stub now communicates to the client process, by whatever means is appropriate, the class tag and the contents of the memory that contains the marshaled interface pointer. In the client process, the proxy object is created as an instance of the indicated class using the standard COM instance creation paradigm. IMarshal is used as the initialization interface; the initialization method is IMarshal::UnmarshalInterface(). The unmarshaling process looks something like the following:

```
void ExampleUnmarshal(CLSID& clsidProxyObject, IStream* pstm, IID& iidOriginallyMarshalled, void** ppvReturn)
{
      IClassFactory* pcf;
      IMarshal* pmsh;
      CoGetClassObject(clsidProxyObject, CLSCTX_INPROC_HANDLER,  NULL, IID_IClassFactory, (void**)&pcf);
      pcf->CreateInstance(NULL, IID_IMarshal, (void**)pmsh);
```

[4] That is, it is explicitly legal for the caller of GetMarshalSizeMax() to allocate a fixed size marshaling buffer containing no more than the indicated upper bound number of bytes.

 Copyright © 1995 Microsoft Corporation All Rights Reserved

```
        pmsh->UnmarshalInterface(pstm, iidOriginallyMarshalled, ppvReturn);
        pmsh->ReleaseMarshalData(pstm)
        pmsh->Release();
        pcf->Release();
    }
```

There are several important reasons why an object may choose to do custom marshaling.

- It permits the server implementation, transparently to the client, to be in complete control of the nature of the invocations that actually transition across the network. In designing component architectures, one often runs into a design tension between the interface which for simplicity and elegance one wishes to exhibit to client programmers and the interface that is necessary to achieve efficient invocations across the network. The former, for example, might naturally wish to operate in terms of small-grained simple queries and responses, whereas the latter might wish to batch requests for efficient retrieval. The client and the network interfaces are in design tension; custom marshaling is the crucial hook that allows us to have our cake and eat it too by giving the server implementor the ability to tune the network interface without affecting the interface seen by its client.
  When the object does custom marshaling, the client loses any "COM provided" communication to the original object. If the proxy wants to "keep in touch", it has to connect through some other means (RPC, Named pipe…) to the original object. Custom Object Marshaling can not be done on a per interface basic, because object identity is lost! Custom Object Marshaling is a sophisticated way for an object to pass a copy of an existing instance of itself into another execution context.
- Some objects are of the nature that once they have been created, they are immutable: their internal state does not subsequently change. Many monikers are an example of such objects. These sorts of objects can be efficiently remoted by making independent copies of themselves in client processes. Custom marshaling is the mechanism by which they can do that, yet have no other party be the wiser for it.
- Objects which already are proxy objects can use custom marshaling to avoid creating proxies to proxies; new proxies are instead short-circuited back to the original server. This is both an important efficiency and an important robustness consideration.
- Object implementations whose whole state is kept in shared memory can often be remoted to other process on the same machine by creating an object in the client that talks directly to the shared memory rather than back to the original object. This can be a significant performance improvement, since access to the remoted object does not result in context switches. The present Microsoft Compound File implementation is an example of objects using this kind of custom marshaling.

1.3Architecture of Standard Interface / Object Marshaling

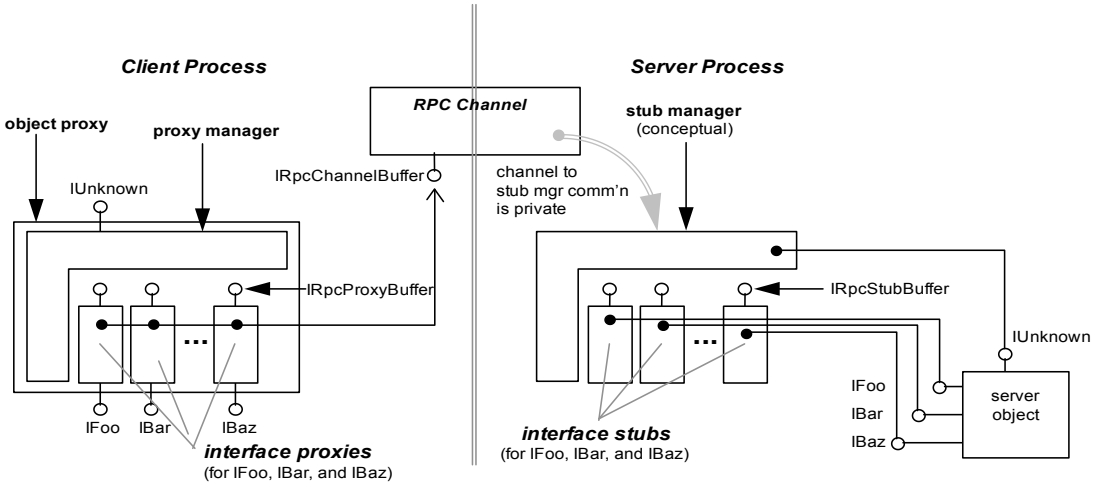
If the object being marshaled[5] chooses not to implement custom object marshaling, a “default” or “standard” object marshaling technique is used. An important part of this standard marshaling technique involves locating and loading the interface-specific pieces of code that are responsible for marshaling and unmarshaling remote calls to instances of that interface. We call these interface-specific pieces of code used in standard marshaling and unmarshaling “interface proxies” and “interface stubs” respectively.[6] (It is important not to confuse *interface* proxies with the *object* proxy, which relates to the *whole* representative in the client process, rather than just one interface on that representative. We apologize for the subtleties of the terminology.)

The following figure gives an slightly simplified view of how the standard client- and server-side structures cooperate.

[5] Astute readers will notice an abuse of terminology here: what is really being marshaled in hand is one particular interface on the object, not the *whole* object, though in fact in the remote process access to the whole process is indeed obtained: new interfaces on the object will be marshaled later as needed. We trust that this will not lead to too much confusion.

[6] Other RPC systems sometimes instead call these “client side stubs” and “server side stubs.” Sometimes *we* mix things up a bit and refer to “proxy interfaces” and “stub interfaces” instead of “interface proxies” and “interface stubs.”

Copyright © 1995 Microsoft Corporation
All Rights Reserved



Simplified conceptual view of client - server remoting structures

When an interface of type IFoo needs to be remoted, a system registry is consulted under a key derived from IID_IFoo to locate a class id that implements the interface proxy and interface stub for the given interface. Both the interface proxies and the interface stubs for a given interface must be implemented by the same class. Most often, this class is automatically generated by a tool whose input is a description of the function signatures and semantics of the interface, written in some "interface description language," often known as "IDL." However, while highly recommended and encouraged for accuracy's sake, the use of such a tool is by no means required; interface proxies and stubs are merely Component Object Model components which are used by the RPC infrastructure, and as such, can be written in any manner desired so long as the correct external contracts are upheld. *From a logical perspective, it is ultimately the programmer who is the designer of a new interface who is responsible for ensuring that all interface proxies and stubs that ever exist agree on the representation of their marshaled data.* The programmer has the freedom to achieve this by whatever means he sees fit, but with that freedom comes the responsibility for ensuring the compatibility.

In the figure, the "stub manager" is "conceptual" in the sense that while it useful in this documentation to have a term to refer to the pieces of code and state on in the server-side RPC infrastructure which service the remoting of a given object, there is no direct requirement that the code and state take any particular well-specified form.[7] In contrast, on the client side, there is an identifiable piece of state and associated behavior which appears to the client code to be the one, whole object. The term "proxy manager" is used to refer to the COM Library provided code that manages the client object identity, etc., and which dynamically loads in interface proxies as needed (per QueryInterface calls). The proxy manager implementation is intimate with the client-side RPC channel implementation, and the server-side RPC channel implementation is intimate with the stub manager implementation.

Interface proxies are created by the client-side COM Library infrastructure using a code sequence resembling the following:

```
clsid = LookUpInRegistry(key derived from iid)
CoGetClassObject(clsid, CLSCTX_SERVER, NULL, IID_IPSFactoryBuffer, &pPSFactory));
pPSFactory->CreateProxy(pUnkOuter, riid, &pProxy, &piid);
```

Interface stubs are created by the server-side RPC infrastructure using a code sequence resembling:

```
clsid = LookUpInRegistry(key derived from iid)
CoGetClassObject(clsid, CLSCTX_SERVER, NULL, IID_IPSFactoryBuffer, &pPSFactory));
pPSFactory->CreateStub(iid, pUnkServer, &pStub);
```

In particular, notice that the class object is talked-to with IPSFactoryBuffer interface rather than the more common IClassFactory.

The interfaces mentioned here are as follows:

```
interface IPSFactoryBuffer : IUnknown {
    HRESULT      CreateProxy(pUnkOuter, iid, ppProxy, ppv);
```

[7] There are, however, implied requirements for the existence of some piece of code / state that manages the *entire set of* external remoting connections for a given object. See CoLockObjectExternal(), for example.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

```
        HRESULT     CreateStub(iid, pUnkServer, ppStub);
        };

    interface IRpcChannelBuffer : IUnknown {
        HRESULT     GetBuffer(pMessage, riid);
        HRESULT     SendReceive(pMessage, pStatus);
        HRESULT     FreeBuffer(pMessage);
        HRESULT     GetDestCtx(pdwDestCtx, ppvDestCtx);
        HRESULT     IsConnected();
        };

    interface IRpcProxyBuffer : IUnknown {
        HRESULT     Connect(pRpcChannelBuffer);
        void        Disconnect();
        };

    interface IRpcStubBuffer : IUnknown {
        HRESULT         Connect(pUnkServer);
        void            Disconnect();
        HRESULT         Invoke(pMessage, pChannel);
        IRPCStubBuffer* IsIIDSupported(iid);
        ULONG           CountRefs();
        HRESULT         DebugServerQueryInterface(ppv);
        void            DebugServerRelease(pv);
        };
```

Suppose an interface proxy receives a method invocation on one of it's interfaces (such as IFoo, IBar, or IBaz in the above figure). The interface proxy's implementation of this method first obtains a marshaling packet from its RPC channel using IRpcChannelBuffer::GetBuffer(). The process of marshaling the arguments will copy data into the buffer. When marshaling is complete, the interface proxy invokes IRpcChannelBuffer::SendReceive() to send the method invocation across the "wire" to the corresponding interface stub. When IRpcChannelBuffer::SendReceive() returns, the contents of buffer into which the arguments were marshaled will have been replaced by the return values marshaled from the interface stub. The interface proxy unmarshals the return values, invokes IRpcChannelBuffer::FreeBuffer() to free the buffer, then returns the return values to the original caller of the method.

It is the implementation of IRpcChannelBuffer::SendReceive() that actually sends the request over to the server process. It is only the channel who knows or cares how to identify the server process and object within that process to which the request should be sent; this encapsulation allows the architecture we are describing here to function for a variety of different kinds of channels: intra-machine channels, inter-machine channels (i.e.: across the network), etc. The channel implementation knows how to forward the request onto the appropriate stub manager object in the appropriate process. From the perspective of this specification, the channel and the stub manager are intimate with each other (and intimate with the proxy manager, for that matter). Through this intimacy, eventually the appropriate interface stub receives an IRpcStubBuffer::Invoke() call. The stub unmarshals the arguments from the provided buffer, invokes the indicated method on the server object, and marshals the return values back into a new buffer, allocated by a call to IRpcChannelBuffer::GetBuffer(). The stub manager and the channel then cooperate to ferry the return data packet back to the interface proxy, who is still in the middle of IRpcChannelBuffer::SendReceive(). IRpcChannelBuffer::SendReceive() returns to the proxy, and we proceed as just described above.

When created, interface proxies are always aggregated into the larger object proxy: at interface-proxy-creation time, the proxy is given the IUnknown* to which it should delegate its QueryInterface(), etc., calls, as per the usual aggregation rules. When connected, the interface proxy is also given (with IRpcProxyBuffer::-Connect()) a pointer to an IRpcChannelBuffer interface instance. It is through this pointer that the interface proxy actually sends calls to the server process. Interface proxies bring a small twist to the normal everyday aggregation scenario. In aggregation, each interface supported by an aggregateable object is classified as either "external" or "internal." External interfaces are the norm. They are the ones whose instances are exposed directly to the clients of the aggregate as whole. It is *always* the case that a QueryInterface() that requests an external interface of an aggregated object should be delegated by the object to its controlling unknown (ditto for AddRef() and Release()). Internal interfaces, on the other hand, are never exposed to outside clients. Instead, they are solely for the use of the controlling unknown in manipulating the aggregated object. QueryInterface() for internal interfaces should *never* be delegated to the controlling unknown (ditto again). In the common uses of aggregation, the IUnknown interface on the

Copyright © 1995 Microsoft Corporation
All Rights Reserved

object is the only internal interface. The twist that interface proxies bring is that IRpcProxyBuffer is *also* an internal interface.

Interface stubs, by contrast with interface proxies, are not aggregated, since there is no need that they appear to some external client to be part of a larger whole. When connected, an interface stub is given (with IRpcStubBuffer::Connect()) a pointer to the server object to which they should forward invocations that they receive.

A given interface proxy instance can if it chooses to do so service more than one interface. For example, in the above figure, one interface proxy could have chosen to service *both* IFoo and IBar. To accomplish this, in addition to installing itself under the appropriate registry entries, the proxy should support QueryInterface()ing from one supported interface (and from IUnknown and IRpcProxyBuffer) to the other interfaces, as usual. When the Proxy Manager in a given object proxy finds that it needs the interface proxy for some new interface that it doesn't already have, before it goes out to the registry to load in the appropriate code using the code sequence described above, it first does a QueryInterface() for the new interface id (IID) on all of its *existing* interface proxies. If one of them supports the interface, then it is used rather than loading a new interface proxy.

Interface stub instances, too, can service more than one interface on a server object. However, the extent to which they can do so is quite restricted: a given interface stub instance may support one or more interfaces only if that set of interfaces has in fact a strict single-inheritance relationship. In short, a given interface stub needs to know how to interpret a given method number that it is asked to invoke without at that same time also being told the interface id (IID) in which that method belongs; the stub must already *know* the relevant IID. The IID which an interface stub is initially created to service is passed as parameter to IPSFactoryBuffer::CreateStub(). After creation, the interface stub may from time to time be asked using IRpcStubBuffer::IsIIDSupported() if it in fact would also like be used to service another IID. If the stub also supports the second IID, then it should return the appropriate IRpcStubBuffer* for that IID; otherwise, the stub buffer should return NULL. This permits the stub manager in certain cases to optimize the loading of interface stubs.

Both proxies and stubs will at various times have need to allocate or free memory. Interface proxies, for example, will need to allocate memory in which to return out parameters to their caller. In this respect interface proxies and interface stubs are just normal Component Object Model components, in that they should use the standard task allocator; see CoGetMalloc(). See also the earlier discussion regarding specific rules for passing in, out, and in out pointers.

On Microsoft Windows platforms, the "key derived from IID" under which the registry is consulted to learn the proxy/stub class is as follows:

```
Interfaces
    {IID}
        ProxyStubClsid32 = {CLSID}
```

Here {CLSID} is a shorthand for any class id; the actual value of the unique id is put between the {}'s; e.g. {DEADBEEF-DEAD-BEEF-C000-000000000046}; all digits are upper case hex and there can be no spaces. This string format for a unique id (without the {}'s) is the same as the OSF DCE™ standard and is the result of the StringFromCLSID routine. {IID} is a shorthand for an interface id; this is similar to {CLSID}; StringFromIID can be used to produce this string.

1.4Architecture of Handler Marshaling

Handler marshaling is a third variation on marshaling, one closely related to standard marshaling. Colloquially, one can think of it as a middle ground between raw standard marshaling and full custom marshaling.

In handler marshaling, the object specifies that it would like to have some amount of client-side state; this is designated by the class returned by IStdMarshalInfo::GetClassForHandler. However, this handler class rather than fully taking over the remoting to the object instead aggregates in the default handler, which carries out the remoting in the standard manner as described above.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

1.5Standards for Marshaled Data Packets

In the architecture described here, nothing has yet to be said about representation or format standards for the data that gets placed in marshaling packets. There is a good reason for this. In the Component Object Model architecture, the only two parties that have to agree on what goes into a marshaling packet are the code that marshals the data into the packet and the code that unmarshals it out again: the interface proxies and the interface stubs. So long as we are dealing only with intra-machine procedure calls (i.e.: non-network), then we can reasonably assume that pairs of interface proxies and stubs are always installed together on the machine. In this situation, we have no need to specify a packet format standard; the packet format can safely be a private matter between the two piece of code.

However, once a network is involved, relying on the simultaneous installation of corresponding interface proxies and stubs (on different machines) is no longer a reasonable thing to do. Thus, when the a method invocation is in fact remoted over a network, it is strongly recommended that the data marshaled into the packet to conform to a published standard (NDR), though, as pointed out above, it is technically the interface-designer's responsibility to achieve this correspondence by whatever means he sees fit.

1.6Creating an Initial Connection Between Processes

Earlier we said we would later discuss how an initial remoting connection is established between two processes. It is now time to have that discussion.

The real truth of the matter is that the initial connection is established by some means outside of the architecture that we have been discussing here. The minimal that is required is some primitive communication channel between the two processes. As such, we cannot hope to discuss all the possibilities. But we will point out some common ones.

One common approach is that initial connections are established just like other connections: an interface pointer is marshaled in the server process, the marshaled data packet is ferried the client process, and it is unmarshaled. The only twist is that the ferrying is done by some means *other* than the RPC mechanism which we've been describing. There are many ways this could be accomplished. The most important, by far is one where the marshaled data is passed as an out-parameter from an invocation on a well-known endpoint to a Service Control Manager.

1.7Marshaling Interface and Function Descriptions

Having discussed on a high level how various remoting related interfaces work together, we now present each of them in detail.

1.7.1IPSFactoryBuffer Interface

IPSFactoryBuffer is the interface through which proxies and stubs are created. It is used to create proxies and stubs that support IRpcProxyBuffer and IRpcStubBuffer respectively. Each proxy / stub DLL must support IPSFactory interface on the class object accessible through its DllGetClassObject() entry point. As was described above, the registry is consulted under a key derived from the IID to be remoted in order to learn the proxy/stub class that handles the remoting of the indicated interface. The class object for this class is retrieved, asking for this interface. A proxy or a stub is then instantiated as appropriate.

```
interface IPSFactoryBuffer : IUnknown {
    HRESULT     CreateProxy(pUnkOuter, iid, ppProxy, ppv);
    HRESULT     CreateStub(iid, pUnkServer, ppStub);
    };
```

IPSFactoryBuffer::CreateProxy

HRESULT IPSFactoryBuffer::CreateProxy(pUnkOuter, iid, ppProxy, ppv)

Create a new interface proxy object. This function returns both an IRpcProxy instance and an instance of the interface which the proxy is being created to service in the first place. The newly created proxy is initially in the unconnected state.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

Argument	Type	Description
pUnkOuter	IUnknown *	the controlling unknown of the aggregate in which the proxy is being created.
iid	REFIID	the interface id which the proxy is being created to service, and of which an instance should be returned through ppv.
ppProxy	IRpcProxyBuffer**	on exit, contains the new IRpcProxyBuffer instance.
ppv	void **	on exit, contains an interface pointer of type indicated by iid.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_NOINTERFACE, E_UNEXPECTED, no others.

IPSFactoryBuffer::CreateStub

HRESULT IPSFactoryBuffer::CreateStub(iid, pUnkServer, ppStub)

Create a new interface stub object. The stub is created in the connected state on the object indicated by pUnkServer.

If pUnkServer is non-NULL, then before this function returns the stub must verify (by using QueryInterface()) that the server object in fact supports the interface indicated by iid. If it does not, then this function should fail with the error E_NOINTERFACE.

Argument	Type	Description
iid	REFIID	the interface that the stub is being created to service
pUnkServer	IUnknown*	the server object that is being remoted. The stub should delegate incoming calls (see IRpcStubBuffer::Invoke()) to the appropriate interface on this object. pUnkServer may legally be NULL, in which case the caller is responsible for later calling IRpcStubBuffer::Connect() before using IRpcStubBuffer::Invoke().
ppStub	IRpcStubBuffer**	the place at which the newly create stub is to be returned.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_NOINTERFACE, E_UNEXPECTED, no others.

1.7.2IRpcChannelBuffer interface

IRpcChannelBuffer is the interface through which interface proxies send calls through to the corresponding interface stub. This interface is implemented by the RPC infrastructure. The infrastructure provides an instance of this interface to interface proxies in IRpcProxyBuffer::Connect(). The interface proxies hold on to this instance and use it each time they receive an incoming call.

```
interface IRpcChannelBuffer : IUnknown {
    HRESULT    GetBuffer(pMessage, riid);
    HRESULT    SendReceive(pMessage, pStatus);
    HRESULT    FreeBuffer(pMessage);
    HRESULT    GetDestCtx(pdwDestCtx, ppvDestCtx);
    HRESULT    IsConnected();
    };
```

RPCOLEMESSAGE and related structures

Common to several of the methods in IRpcChannelBuffer is a data structure of type RPCOLEMESSAGE. This structure is defined as is show below. The structure is to be packed so that there are no holes in its memory layout.

```
typedef struct RPCOLEMESSAGE {
    void *                reserved1;
    RPCOLEDATAREP         dataRepresentation;   // in NDR transfer syntax: info about endianness, etc.
    void  *               pvBuffer;             // memory buffer used for marshalling
    ULONG                 cbBuffer;             // size of the marshalling buffer
    ULONG                 iMethod;              // the method number being invoked
    void *                reserved2[5];
    ULONG                 rpcFlags;
    } on the ultimate destination machine MESSAGE;[8]
```

[8] The layout of this structure is as odd as it is for historical reasons. Apologies are extended to those whose design aesthetics are offended.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

The most significant member of this structure is pvBuffer. It is through the memory buffer to which pvBuffer points that marshaled method arguments are transferred. cbBuffer is used to indicate the size of the buffer. iMethod is indicates a particular method number within the interface being invoked. The IID of that interface is identified through other means: on the client side as a parameter to GetBuffer(), and on the server side as part of the internal state of each interface stub.

At all times all reserved values in this structure are to be initialized to zero by non-RPC-infrastructure parties (i.e.: parties other than the channel / RPC runtime implementor) who allocate RPCOLEMESSAGE structures. However, the RPC channel (more generally, the RPC runtime infrastructure) is free to modify these reserved fields. Therefore, once initialized, the reserved fields must be ignored by the initializing code; they cannot be relied on to remain as zero. Further, there are very carefully specified rules as to what values in these structures may or may not be modified at various times and by which parties. In almost all cases, aside from actually reading and writing data from the marshaling buffer, which is done by proxies and stubs, only the channel may change these fields. See the individual method descriptions for details.

Readers familiar with the connection-oriented DCE protocol may notice that the "transfer syntax" used for marshaling the arguments, the particular set of rules and conventions according to which data is marshaled, is not explicitly called out. Architecturally speaking, it is only the interface proxy for a given interface and its corresponding interface stub that cares at all about what set of marshaling rules is in fact used. However, in the general case these interface proxies and stubs may be installed on different machines with a network in the middle, be written by different development organizations on different operating systems, etc. Accordingly, in cases where the author of an interface proxy for a given IID cannot guarantee that all copies of the corresponding interface stub are in fact always revised and updated in synchrony with his interface proxy, a well-defined convention should be used for the transfer syntax. Indeed, formal transfer syntax standards exist for this purpose. The one most commonly used is known as "Network Data Representation" (NDR), originally developed by Apollo Corporation and subsequently enhanced and adopted by the Open Software Foundation as part of their Distributed Computing Environment (DCE). The Windows NT operating system also uses NDR in its RPC implementation. Unless very good reasons exist to do otherwise, programmers are encouraged to use the NDR transfer syntax.

When NDR transfer syntax is used (and whether it *is* in use or not is implicitly known by the proxy or stub), the member dataRepresentation provides further information about the rules by which data in the buffer is marshaled. NDR is a "multi-canonical" standard, meaning that rather than adopting one standard for things like byte-order, character set, etc., multiple standards (a fixed set of them) are accommodated. Specifically, this is accommodated by a "reader make right" policy: the writer / marshaler of the data is free to write the data in any of the supported variations and the reader / unmarshaler is expected to be able to read any of them. The particular data type in use is conveyed in an RPCOLEDATAREP structure, which is defined as follows. Note that this structure, too, is packed; the size of the entire structure is exactly four bytes. The actual layout of the structure in all cases always corresponds to the data representation value as defined in the DCE standard; the particular structure shown here is equivalent to that layout in Microsoft's and other common compilers.

```
typedef RPCOLEDATAREP {
    UINT        uCharacterRep   : 4;    // least signficant nibble of first byte
    UINT        uByteOrder      : 4;    // most signficant nibble of first byte
    BYTE        uFloatRep;
    BYTE        uReserved;
    BYTE        uReserved2;
    } RPCOLEDATAREP;
```

The values which may legally be found in these fields are as shown in Table 1. Further information on the interpretation of this field can be found in the NDR Transfer Syntax standards documentation.

Field Name	Meaning of Field	Value in field	Interpretation
uCharacterRep	determines interpretation of single-byte-character valued and single-byte-string valued entities	0	ASCII
		1	EBCDIC
uByteOrder	integer and floating point byte order	0	Big-endian (Motorola)
		1	Little-endian (Intel)
uFloatRep	representation of floating point numbers	0	IEEE

Copyright © 1995 Microsoft Corporation
All Rights Reserved

1	VAX
2	Cray
3	IBM

Table 2. Interpretation of dataPresentation

IRpcChannelBuffer::GetBuffer

HRESULT IRpcChannelBuffer::GetBuffer(pMessage, iid)

This method returns a buffer into which data can be marshaled for subsequent transmission over the wire. It is used both by interface proxies and by interface stubs, the former to marshal the incoming arguments for transmission to the server, and the latter to marshal the return values back to the client.

Upon receipt of an incoming call from the client of the proxy object, interface proxies use GetBuffer() to get a buffer into which they can marshaling the incoming arguments. A new buffer must be obtained for every call operation; old buffers cannot be reused by the interface proxy. The proxy needs to ask for and correctly manage a new buffer even if he himself does not have arguments to marshal (i.e.: a void argument list).[9] Having marshaled the arguments, the interface proxy then calls SendReceive() to actually invoke the operation. Upon return from SendReceive(), the buffer no longer contains the marshaled arguments but instead contains the marshaled return values (and out parameter values). The interface proxy unmarshals these values, calls FreeBuffer() to free the buffer, then returns to its calling client.

On the server side (in interface stubs), the sequence is somewhat different. The server side will not be explored further here; see instead the description of IRpcStubBuffer::Invoke() for details.

On the client side, the RPCOLEMESSAGE structure argument to GetBuffer() has been allocated and initialized by the caller (or by some other party on the caller's behalf). Interface proxies are to initialize the members of this structure as follows.

Member Name	Value to initalize to
reserved members	as always, reserved values must be initialized to zero / NULL.
pvBuffer	must be NULL.
cbBuffer	the size in bytes that the channel should allocate for the buffer; that is, the maximum size in bytes needed to marshal the arguments. The interface proxy will have determined this information by considering the function signature and the particular argument values passed in. It is explicitly legal to have this value be zero, indicating that that the caller does not himself require a memory buffer.
iMethod	the zero-based method number in the interface iid which is being invoked
dataRepresentation	if NDR transfer syntax is being used, then this indicates the byte order, etc., by which the caller will marshal data into the returned buffer.
rpcFlags	◆ *Exact values to be listed here.*

If the GetBuffer() function is successful, then upon function exit pvBuffer will have been changed by the channel to point to a memory buffer of (at least) cbBuffer bytes in size into which the method arguments can now be marshaled (if cbBuffer was zero, pvBuffer may or may not be NULL). The reserved fields in the RPCOLEMESSAGE structure may or may not have been changed by the channel. However, neither the cbBuffer nor iMethod fields of RPCOLEMESSAGE will have been changed; the channel treats these as read-only.[10] Furthermore, until such time as the now-allocated memory buffer is subsequently freed (see SendReceive() and FreeBuffer()), no party other than the channel may modify any of the data accessible from pMessage with the lone exceptions of the data pointed to by pvBuffer and the member cbBuffer, which may be modified only in limited ways; see below.

[9] This permits the channel to behind-the-scenes add additional space into the buffer. Such a capability is needed, for example, in order to support remote debugging.

[10] The fact that cbBuffer is unchanged can be of particular use to interface stubs. See IRpcStubBuffer::Invoke().

Copyright © 1995 Microsoft Corporation
All Rights Reserved

The arguments to GetBuffer() are as follows:

Argument	Type	Description
pMessage	RPCOLEMESSAGE *	a message structure initialized as discussed above.
iid	REFIID	the interface identifier of the interface being invoked.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_UNEXPECTED

IRpcChannelBuffer::SendReceive

HRESULT IRpcChannelBuffer::SendReceive(pMessage, pStatus)

Cause an invocation to be sent across to the server process. The caller will have first obtained access to a transmission packet in which to marshal the arguments by calling IRpcChannelBuffer::GetBuffer(). The same pMessage structure passed as an argument into that function is passed here to the channel a second time.

In the intervening time period, the method arguments will have been marshaled into the buffer pointed to by pMessage->pvBuffer. However, the pvBuffer pointer parameter must on entry to SendReceive() be exactly as it was when returned from GetBuffer(). That is, it must point to the start of the memory buffer. The caller should in addition set pMessage->cbBuffer to the number of bytes actually written into the buffer (zero is explicitly a legal value). No other values accessible from pMessage may be different than they were on exit from GetBuffer().

Upon *successful* exit from SendReceive(), the incoming buffer pointed to by pvBuffer will have been freed by the channel. In its place will be found a buffer containing the marshaled return values / out parameters from the interface stub: pMessage->pvBuffer points to the new buffer, and pMessage->cbBuffer indicates the size thereof. If there are no such return values, then pMessage->cbBuffer is set to zero, while pMessage->pvBuffer may or may not be NULL.

On *error* exit from SendReceive(),[11] the incoming buffer pointed to by pvBuffer may or may not have been freed. If it has been freed, then on error exit pMessage->pvBuffer is set to NULL and pMessage->cbBuffer is set to zero. If in contrast, pMessage->pvBuffer is on error exit not NULL, then that pointer, the data to which it points, and the value pMessage->cbBuffer will contain exactly as they did on entry; that is, the marshaled arguments will not have been touched. Thus, on error exit from SendReceive(), in no case are any marshaled *return values* passed back; if a marshaling buffer is in fact returned, then it contains the marshaled *arguments* as they were on entry.

The exact cases on error exit when the incoming buffer has or has not been freed needs careful attention. There are three cases:

1) The channel implementation knows with certainty either that all of the incoming data was successfully unmarshaled or that if any errors occurred during unmarshaling that the interface stub correctly cleaned up. In practical terms, this condition is equivalent to the stub manager having actually called IRpcStubBuffer::Invoke() on the appropriate interface stub.

   In this case, on exit from SendReceive() the incoming arguments will *always* have been freed.

2) The channel implementation knows with certainty the situation in case 1) has *not* occurred.

   In this case, on exit from SendReceive(), the incoming arguments will *never* have been freed.

3) The channel implementation does not know with certainty that either of the above two cases has occurred.

   In this case, on exit from SendReceive(), the incoming arguments will *always* have been freed. This is a possible resource leakage (due to, for example, CoReleaseMarshalData() calls that never get made), but it safely avoids freeing resources that should not be freed.

If pMessage->pvBuffer is returned as non-NULL, then the caller is responsible for subsequently freeing it; see FreeBuffer(). A returned non-NULL pMessage->pvBuffer may in general legally be (and will commonly be, the success case) different than the (non-NULL) value on entry; i.e.: the buffer may be legally be reallocated. Further, between the return from SendReceive() and the subsequent freeing call no data accessible from pMessage may be modified, with the possible exception of the data actually in the memory buffer.

[11] That is, if SendReceive() returns an error. Note that this does NOT indicate an error returned from the function invocation on the server object, for in that case SendReceive() returns success; rather, it indicates an error that occurred somewhere in the RPC transmission.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

Upon successful exit from SendReceive(), the pMessage->dataRepresentation field will have been modified to contain whatever was returned by the interface stub in field of the same name value on exit to IRpcStubBuffer::Invoke(). This is particularly important when NDR transfer syntax is used, as dataRepresentation indicates critical things (such as byte order) which apply to the marshaled return / out values. Upon error exit from SendReceive(), pMessage->dataRepresentation is undefined.

Argument	Type	Description
pMessage	RPCOLEMESSAGE *	message structure containing info to transmit to server.
pStatus	ULONG *	may legally be NULL. If non-NULL, then if either 1) an RPC-infrastructure-detected server-object fault (e.g.: a server object bug caused an exception which was caught by the RPC infrastructure) or 2) an RPC communications failure occurs, then at this location a status code is written which describes what happened. In the two error cases, the errors E_RPCFAULT and E_RPCSTATUS are (respectively) returned (and are always returned when these errors occur, irrespective of the NULL-ness of pStatus).
return value	HRESULT	S_OK, E_RPCFAULT, E_RPCSTATUS

IRpcChannelBuffer::FreeBuffer

HRESULT IRpcChannelBuffer::FreeBuffer(pMessage)

Free a memory buffer in pMessage->pvBuffer that was previously allocated by the channel.

At various times the RPC channel allocates a memory buffer and returns control of same to a calling client. Both GetBuffer() and SendReceive() do so, for example. FreeBuffer() is the means by which said calling client informs the channel that it is done with the buffer.

On function entry, the buffer which is to be freed is pMessage->pvBuffer, which explicitly may or may not be NULL. If pMessage->pvBuffer is non-NULL, then FreeBuffer() frees the buffer, NULLs the pointer, and returns NOERROR; if pMessage->pvBuffer is NULL, then FreeBuffer() simply returns NOERROR (i.e.: passing NULL is *not* an error). Thus, on function exit, pMessage->pvBuffer is always NULL. Notice that pMessage->cbBuffer is never looked at or changed.

There are strict rules as to what data accessible from pMessage may have been modified in the intervening time between the time the buffer was allocated and the call to FreeBuffer(). In short, very little modification is permitted; see above and below for precise details.

Argument	Type	Description
pMessage	RPCOLEMESSAGE *	pointer to structure containing pointer to buffer to free.
return value	HRESULT	S_OK, E_UNEXPECTED

IRpcChannelBuffer::GetDestCtx

HRESULT IRpcChannelBuffer::GetDestCtx(pdwDestCtx, ppvDestCtx)

Return the destination context for this RPC channel. The destination context here is as specified in the description of the IMarshal interface.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

Argument	Type	Description
pdwDestCtx	DWORD *	the place at which the destination context is to be returned.
ppvDestCtx	void **	May be NULL. If non-NULL, then this is the place at which auxiliary information associated with certain destination contexts will be returned. Interface proxies may not hold on to this returned pointer in their internal state; rather, they must assume that a subsequent call to IRpcChannel::Call() may in fact invalidate a previously returned destination context.[12]
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_UNEXPECTED, but no others.

IRpcChannelBuffer::IsConnected

HRESULT IRpcChannelBuffer::IsConnected()

Answers as to whether the RPC channel is still connected to the other side. A negative reply is definitive: the connection to server end has definitely been terminated. A positive reply is tentative: the server end may or may not be still up. Interface proxies can if they wish use this method as an optimization by which they can quickly return an error condition.

Argument	Type	Description
return value	HRESULT	S_OK, S_FALSE. No error values may be returned.

1.7.3IRpcProxyBuffer Interface

IRpcProxyBuffer interface is the interface by which the client-side infrastructure (i.e. the proxy manager) talks to the interface proxy instances that it manages. When created, proxies are aggregated into some larger object as per the normal creation process (where pUnkOuter in IPSFactoryBuffer::CreateProxy() is non-NULL). The controlling unknown will then QueryInterface() to the interface that it wishes to expose from the interface proxy.

```
interface IRpcProxyBuffer : IUnknown {
    virtual    HRESULT    Connect(pRpcChannelBuffer) = 0;
    virtual    void       Disconnect() = 0;
    };
```

IRpcProxyBuffer::Connect

HRESULT IRpcProxyBuffer::Connect(pRpcChannelBuffer)

Connect the interface proxy to the indicated RPC channel. The proxy should hold on to the channel, AddRef()ing it as per the usual rules. If the proxy is currently connected, then this call fails (with E_UNEXPECTED); call Disconnect() first if in doubt.

Argument	Type	Description
pRpcChannelBuffer	IRpcChannelBuffer*	the RPC channel that the interface proxy is to use to effect invocations to the server object. May not be NULL.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_NOINTERFACE, E_UNEXPECTED

IRpcProxyBuffer::Disconnect

void IRpcProxyBuffer::Disconnect()

Informs the proxy that it should disconnect itself from any RPC channel that it may currently be holding on to. This will involve Release()ing the IRpcChannel pointer to counteract the AddRef() done in IRpcProxy::Connect().

Notice that this function does not return a value.

[12] It is possible that in the future a less restrictive rule as to the duration in which the interface proxy may hold on to ppvDestCtxt may be established, such as (perhaps) guaranteeing that the pointer is valid for the lifetime of the interface proxy itself. However, as it stands today, the rule, as stated here, is in fact the law.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

1.7.4IRpcStubBuffer interface

IRpcStubBuffer is the interface used on the server side by the RPC runtime infrastructure (herein referred to loosely as the "channel") to communicate with interface stubs that it dynamically loads into a server process.

```
interface IRpcStubBuffer : IUnknown {
    virtual   HRESULT          Connect(pUnkServer) = 0;
    virtual   void             Disconnect() = 0;
    virtual   HRESULT          Invoke(pMessage, pChannel) = 0;
    virtual   IRpcStubBuffer*  IsIIDSupported(iid) = 0;
    virtual   ULONG            CountRefs() = 0;
    virtual   HRESULT          DebugServerQueryInterface(ppv) = 0;
    virtual   void             DebugServerRelease(pv) = 0;
    };
```

IRpcStubBuffer::Connect

HRESULT IRpcStubBuffer::Connect(pUnkServer)

Informs the interface stub of server object to which it is now to be connected, and to which it should forward all subsequent Invoke() operations. The stub will have to QueryInterface() on pUnkServer to obtain access to appropriate interfaces. The stub will of course follow the normal AddRef() rules when it stores pointers to the server object in its internal state.

If the stub is currently connected, then this call fails with E_UNEXPECTED.

Argument	Type	Description
pUnkServer	IUnknown *	the new server object to which this stub is now to be connected.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_NOINTERFACE, E_UNEXPECTED

IRpcStubBuffer::Disconnect

void IRpcStubBuffer::Disconnect()

Informs the stub that it should disconnect itself from any server object that it may currently be holding on to. Notice that this function does not return a value.

IRpcStubBuffer::Invoke

HRESULT IRpcStubBuffer::Invoke(pMessage, pChannel)

Invoke the pMessage->iMethod'th method in the server object interface instance to which this interface stub is currently connected. The RPC runtime infrastructure (the "channel") calls this method on the appropriate interface stub upon receipt of an incoming request from some remote client. See the discussion on page 7 regarding how interface stubs implicitly know the IID which they are servicing.

On entry, the members of pMessage are set as follows:

Member Name	Value on entry to Invoke()
reserved members	indeterminate. These members are neither to be read nor to be changed by the stub.
pvBuffer	points to a buffer which contains the marshaled incoming arguments. In the case that there are no such arguments (i.e.: cbBuffer == 0), pvBuffer may be NULL, but will not necessarily be so.
cbBuffer	the size in bytes of the memory buffer to which pvBuffer points. If pvBuffer is NULL, then cbBuffer will be zero (but the converse is not necessarily true, as was mentioned in pvBuffer).
iMethod	the zero-based method number in the interface which is being invoked
dataRepresentation	if NDR transfer syntax is being used, then this indicates the byte order, etc., according to which the data in pvBuffer has been marshaled.
rpcFlags	indeterminate. Neither to be read nor to be changed by the stub.

The stub is to do the following:

 Copyright © 1995 Microsoft Corporation All Rights Reserved

- unmarshal the incoming arguments,
- invoke the designated operation in the server object,
- ask the channel to allocate a new buffer for the return values and out values,
- marshal the return values and out values into the buffer, then
- return successfully (i.e.: NOERROR) from Invoke().

Errors may of course occur at various places in this process.[13] Such errors will cause the stub to return an error from Invoke() rather than NOERROR. In cases where such an error code is returned, it is the stub's responsibility to have cleaned up any data and other resources allocated by the unmarshaling and marshaling processes or returned as out values from the server object. However, the stub is *not* responsible for invoking FreeBuffer() to free the actual marshaling buffer (i.e.: it is illegal for the stub to do so); rather, on error return from Invoke() the caller of Invoke() will ignore pvBuffer, and will also free it if non-NULL. Having made that general statement as to the exit conditions of Invoke(), let us examine its operation in greater detail.

If the stub cannot deal with the indicated dataRepresentation, it is to return RPC_E_SERVER_INVALIDDATA-REP. If it understands the data representation, the stub is to then unmarshal the arguments from the buffer provided in pMessage->pvBuffer, the size of which is passed in pMessage->cbBuffer. If the argument data cannot be completely unmarshaled, the server is to free any partially unmarshaled data, then return RPC_E_SERVER_CANTUNMARSHALDATA from Invoke().

If the data is successfully completely unmarshaled, then the interface stub is to invoke the designated method in the designated interface on the server object. Notice that the incoming pvBuffer memory buffer is at this time still valid, and that therefore the stub may if it wishes and if appropriate for the argument and data representations in question pass to the server object pointers which point directly into this buffer. The memory allocation and data copying that is thus avoided can at times be a significant performance optimization.

Once the invocation of the server object returns, the stub is to marshal the return value and out parameters returned from the server back to the client. It does so irrespective of whether the server object invocation returned an error or success code; that is, the stub marshals back to the client whatever the server object returned.[14] The stub gets a reply buffer into which to do this marshaling by calling pChannel->GetBuffer(), passing in the pMessage structure that it received in Invoke(). Before calling GetBuffer(), the stub is to set the cbBuffer member to the size that it requires for the to-be-allocated reply buffer. Zero is explicitly a legal value for cbBuffer, and the stub must *always* call GetBuffer() (more precisely, to be clear about the error case: the stub must always call GetBuffer() if the server object method has actually been invoked)[15] to allocate a reply buffer, even if the stub itself does not require one (such as would be the case if for a void-returning function with no out parameters). The stub must also set dataRepresentation as appropriate for the standard by which it intends to marshal the returning values (or would marshal them if there were some).[16] Aside from cbBuffer, dataRepresentation and possibly the contents of the bytes inside the memory buffer, on entry to GetBuffer() no other data accessible from pMessage may be different than they were on entry to Invoke().

Before it allocates a reply buffer, the call to GetBuffer() has the side effect of freeing the memory buffer to which pvBuffer presently points. Thus, the act by the interface stub of allocating a reply buffer for the return values necessarily terminates access by the stub to the incoming marshaled arguments.

If GetBuffer() successfully allocates a reply buffer (see GetBuffer() for a description of how the stub determines this), then the stub is to marshal the return value and returned out parameters into the buffer according to the rules of the transfer syntax. Once this is complete, the stub is to set the cbBuffer member to the number of bytes it actually marshaled (if it marshaled nothing, then it must explicitly set this to zero (but see also GetBuffer())), and then return NOERROR from Invoke().

---

[13] Be careful with the terminology here: we are not talking at all about what values are returned from the invocation of the server object, but rather only about errors that occur in the unmarshaling and marshaling process itself.

[14] However, debugging versions of the stub may if they wish to at this time check that certain details of the contract of the interface have been upheld. A common example of this is checking that on error return from the server allocated out-values are explicitly NULLed, a policy which is common to many interfaces. This is simply in the interest of improving the debug capabilities. It is illegal, however, to do such things in non-debug versions of stubs; they must always simply marshal back whatever the server returned.

[15] This policy exists in order to enable behind-the-scenes things such as debugging support to function in all cases.

[16] Presently, this is only significant if NDR transfer syntax is in use. In NDR, it is explicitly the case that the return values may be marshaled using a different data representation than was used for the incoming arguments.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

If an error occurs during the unmarshaling of the incoming arguments or the marshaling of the return values, then the interface stub is responsible for correctly freeing any resources consumed by the marshaled data. See in particular CoReleaseMarshalData(). See also the discussion of this topic in IRpcChannelBuffer::SendRecieve().

Argument	Type	Description
pMessage	RPCOLEMESSAGE *	channel-allocated message structure.
pChannel	IRpcChannelBuffer *	the channel to use for buffer management, etc.
return value	HRESULT	S_OK, RPC_E_SERVER_INVALIDDATAREP, RPC_E_SERVER_-CANTUNMARSHALDATA, RPC_E_SERVER_CANTMARSHALDATA

IRpcStubBuffer::IsIIDSupported

IRpcStubBuffer* IRpcStubBuffer::IsIIDSupported(iid)

Answer whether this stub is designed to handle the unmarshaling of the indicated interface.

If the stub buffer supports the specified IID, then it should return an appropriate IRpcStubBuffer* for that interface. Otherwise, the it should return NULL.

When presented with the need to remote a new IID on a given object, the RPC runtime typically calls this function on all the presently-connected interface stubs in an attempt to locate one that can handle the marshaling for the request before it goes to the trouble of creating a new stub.

As in IPSFactoryBuffer::CreateStub(), if this stub is presently connected to a server object, then not only must this function verify that the *stub* can handle the requested interface id, but it must also verify (using QueryInterface()) that the connected server object in fact supports the indicated interface (depending on the IID and previous interface servicing requests, it may have already done so).

A common special case is the following: interface stubs which are designed to only support one interface id (as most are designed to do) can simply check if iid designates the one interface that they handle. If not, return false. Otherwise, then if connected check that the server object supports the interface. Otherwise return true.

Argument	Type	Description
iid	REFIID	the interface that the caller wishes to know if the stub can handle. iid is never to be IID_IUnknown.
return value	IRpcStubBuffer*	see above.

IRpcStubBuffer::CountRefs

ULONG IRpcStub::CountRefs()

Return the total number of references that this stub interface instance has on the server object.

Argument	Type	Description
return value	ULONG	the number of such references.

IRpcStubBuffer::DebugServerQueryInterface

HRESULT IRpcStubBuffer::DebugServerQueryInterface(ppv)

This function exists in order to facilitate the support of debuggers which wish to provide transparency when single-stepping, etc., across remote invocations on objects. As such, the semantics of this function are a little strange in order to avoid the unnecessarily disturbing the state of the actual server object.

If the stub is not presently connected then set *ppv to NULL (per the usual error-case convention) and return E_UNEXPECTED. If connected but this stub does not support the indicated interface (in the sense expressed in IsIIDSupported()), then (set *ppv to NULL and) return E_NOINTERFACE instead.

Otherwise, return the interface pointer on the connected server object which would be used by an immediate subsequent invocation of Invoke() on this interface stub (see the discussion on page 7 regarding how interface stubs implicitly know the IID which they are servicing). DebugServerQueryInterface() is analogous to invoking QueryInterface() on the server itself with the important difference that the caller will later

 Copyright © 1995 Microsoft Corporation All Rights Reserved

call DebugServerRelease() to indicate that he is done with the pointer instead of releasing the returned pointer himself. It is required that DebugServerRelease() be called before the interface stub itself is destroyed or, in fact, before it is disconnected.

In the vast majority of interface stub implementations, DebugServerQueryInterface() can therefore be implemented simply by returning an internal state variable inside the interface stub itself without doing an AddRef() on the server or otherwise running any code in the actual server object. In such implementations, DebugServerRelease() will be a completely empty no-op. The other rational implementation is one where DebugServerQueryInterface() does a QueryInterface() on the server object and DebugServerRelease() does a corresponding Release(), but as this actually runs server code, the former implementation is highly preferred if at all achievable.

Argument	Type	Description
ppv	void**	the place at which the interface pointer is to be returned.
return value	HRESULT	S_OK, E_NOINTERFACE, E_UNEXPECTED

IRpcStubBuffer::DebugServerRelease

void IRpcStubBuffer::DebugServerRelease(pv)

Indicate that an interface pointer returned previously from DebugServerQueryInterface() is no longer needed by the caller. In most implementations, DebugServerRelease() is a completely empty no-op; see the description of DebugServerQueryInterface() for details.

1.8Marshaling - Related API Functions

The following functions are related to the process of remoting interface pointers and to marshaling in general.

```
HRESULT     CoMarshalInterface(pstm, riid, pUnk, dwDestContext, pvDestContext, mshlflags);
HRESULT     CoUnmarshalInterface(pstm, iid, ppv);
HRESULT     CoDisconnectObject(pUnkInterface, dwReserved);
HRESULT     CoReleaseMarshalData(pstm);
HRESULT     CoGetStandardMarshal(iid, pUnkObject, dwDestContext, pvDestContext, mshlflags, ppmarshal);

typedef enum tagMSHLFLAGS {
    MSHLFLAGS_NORMAL        = 0,
    MSHLFLAGS_TABLESTRONG   = 1,
    MSHLFLAGS_TABLEWEAK     = 2,
    } MSHLFLAGS;
```

CoMarshalInterface

HRESULT CoMarshalInterface(pstm, riid, pUnk, dwDestContext, pvDestContext, mshlflags)

Marshal the interface riid on the object on which pUnk is an IUnknown* into the given stream in such a way as it can be reconstituted in the destination using CoUnmarshalInterface().[17] This the root level function by which an interface pointer can be marshaled into a stream. It carries out the test for custom marshaling, using it if present, and carries out standard marshaling if not. This function is normally only called by code in interface proxies or interface stubs that wish to marshal an interface pointer parameter, though it will sometimes also be called by objects which support custom marshaling.

riid indicates the interface on the object which is to be marshaled. It is specifically *not* the case that pUnk need actually be of interface riid; this function will QueryInterface from pUnk to determine the actual interface pointer to be marshaled.

dwDestContext is a bit field which identifies the execution context relative to the current context in which the unmarshaling will be done. Different marshaling might be done, for example, depending on whether the unmarshal happens on the same workstation vs. on a different workstation on the network; an object could choose to do custom marshaling in one case but not the other. The legal values for dwDestContext are taken from the enumeration MSHCTX, which presently contains the following values.

[17] That is, the mechanism for unmarshaling a marshaled interface pointer is the *same* irrespective of whether the marshaling was done using custom or standard marshaling.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

```
typedef enum tagMSHCTX {
    MSHCTX_NOSHAREDMEM          = 1,
    MSHCTX_DIFFERENTMACHINE     = 2,
    MSHCTX_SAMEPROCESS          = 4,
    } MSHCTX;
```

These flags have the following meanings.

Value	Description
MSHCTX_NOSHAREDMEM	The unmarshaling context does not have shared memory access with the marshaling context.
MSHCTX_DIFFERENTMACHINE	If this flag is set, then it cannot be assumed that this marshaling is being carried out to the same machine as that on which the marshaling is being done. The unmarshaling context is (very probably) on a computer with a different set of installed applications / components than the marshaling context (i.e.: is on a different computer). This is significant in that the marshaling cannot in this case assume that it knows whether a certain piece of application code is installed remotely.
MSHCTX_SAMEPROCESS	The interface is being marshaled to another apartment within the same process in which it is being unmarshaled.

In the future, more MSHCTX flags may be defined; recall that this is a bit field.

pvDestContext is a parameter that optionally supplies additional information about the destination of the marshaling. If non-NULL, then it is a pointer to a structure of the following form.

```
typedef struct MSHCTXDATA {
    ULONG               cbStruct;
    IRpcChannelBuffer*  pChannel;
    } MSHCTXDATA;
```

The members in this structure have the following meanings:

Value	Description
cbStruct	The size of the MSHCTXDATA structure in bytes.
pChannel	The channel object involved in the marshaling process.

pvDestContext may legally be NULL, in which case such data is not provided.

mslflags indicates the purpose for which the marshal is taking place, as was discussed in an earlier part of this document. Values for this parameter are taken from the enumeration MSHLFLAGS, and have the following interpretation.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

Value	Description
MSHLFLAGS_NORMAL	The marshaling is occurring because of the normal case of passing an interface from one process to another. The marshaled-data-packet that results from the call will be transported to the other process, where it will be unmarshaled (see CoUnmarshalInterface). With this flag, the marshaled data packet will be unmarshaled either one or zero times. CoReleaseMarshalData is always (eventually) called to free the data packet.
MSHLFLAGS_TABLESTRONG	The marshaling is occurring because the data-packet is to be stored in a globally-accessible table from which it is to be unmarshaled zero, one, or more times. Further, the presence of the data-packet in the table is to count as a reference on the marshaled interface. When removed from the table, it is the responsibility of the table implementor to call CoReleaseMarshalData on the data-packet.
MSHLFLAGS_TABLEWEAK	The marshaling is occurring because the data-packet is to be stored in a globally-accessible table from which it is to be unmarshaled zero, one, or more times. However, the presence of the data-packet in the table is *not* to count as a reference on the marshaled interface. Destruction of the data-packet is as in the MSHLFLAGS_TABLESTRONG case.

A consequence of this design is that the marshaled data packet will want to store the value of mshlflags in the marshaled data so as to be able to do the right thing at unmarshal time.

Argument	Type	Description
pstm	IStream *	the stream onto which the object should be marshaled. The stream passed to this function must be dynamically growable.
riid	REFIID	the interface that we wish to marshal.
pUnk	IUnknown *	the object on which we wish to marshal the interface riid.
dwDestContext	DWORD	the destination context in which the unmarshaling will occur.
pvDestContext	void*	as described above.
mshlflags	DWORD	the reason that the marshaling is taking place.
return value	HRESULT	S_OK, STG_E_MEDIUMFULL, E_NOINTERFACE, E_FAIL

CoUnmarshalInterface

HRESULT CoUnmarshalInterface(pstm, iid, ppv)

Unmarshal from the given stream an object previously marshaled with CoMarshalInterface.

Argument	Type	Description
pstm	IStream *	the stream from which the object should be unmarshaled.
iid	REFIID	the interface with which we wish to talk to the reconstituted object.
ppv	void **	the place in which we should return the interface pointer.
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE

CoDisconnectObject

HRESULT CoDisconnectObject(pUnkInterface, dwReserved)

This function severs any extant Remote Procedure Call connections that are being maintained on behalf of all the interface pointers on this object. This is a very rude operation, and is not to be used in the normal course of processing; clients of interfaces should use IUnknown::Release() instead. In effect, this function is a privileged operation, which should only be invoked by the process in which the object actually is managed.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

The primary purpose of this operation is to give an application process certain and definite control over remoting connections to other processes that may have been made from objects managed by the process. If the application process wishes to exit, then we do not want it to be the case that the extant reference counts from clients of the application's objects in fact keeps the process alive. When the application process wishes to exit, it should inform the extant clients of its objects[18] that the objects are going away. Having so informed its clients, the process can then call this function for each of the object that it manages, even without waiting for a confirmation from each client. Having thus released resources maintained by the remoting connections, the application process can exit safely and cleanly. In effect, CoDisconnectObject() causes a controlled crash of the remoting connections to the object. It is also (one of) the triggers by which a client's subsequent IRpcChannel::IsConnected() call may return false.

For illustration, contrast this with the situation with Microsoft's elderly Dynamic Data Exchange (DDE) desktop application integration protocol. If it has extant DDE connections, an application is required to send a DDE Terminate message before exiting, and it is *also* responsible for waiting around for an acknowledgment from each client before it can actually exit. Thus, if the client process has crashed, the application process will wait around forever. Because of this, with DDE there simply is no way for an application process to reliably and robustly terminate itself. Using CoDisconnectObject(), we avoid this sort of situation.

Argument	Type	Description
punkInterface	IUnknown *	the object that we wish to disconnect. May be any interface on the object which is polymorphic with IUnknown*, not necessarily the exact interface returned by QueryInterface(IID_IUnknown...).
dwReserved	DWORD	reserved for future use; must be zero.
return value	HRESULT	S_OK, E_FAIL

CoReleaseMarshalData

HRESULT CoReleaseMarshalData(pstm)

This helper function destroys a previously marshaled data packet. This function must always be called in order to destroy data packets. Examples of when this occurs include:

1. an internal error during an RPC invocation prevented the UnmarshalInterface() operation from being attempted.
2. a marshaled-data-packet was removed from a global table.
3. following a successful, normal, unmarshal call.

This function works as should be expected: the class id is obtained from the stream; an instance is created; IMarshal is obtained from that instance; then IMarshal::ReleaseMarshalData() is invoked.

Note for clarity: CoReleaseMarshalData() is not to be called following a normal, successful CoUnmarshalInterface(), as the latter function does this automatically for MSHLFLAGS_NORMAL. However, clients that use IMarshal interface directly, rather than simply going through the functions CoMarshal/UnmarshalInterface(), etc., must of course themselves always call IMarshal::ReleaseMarshalData() after calling IMarshal::UnmarshalInterface().

Argument	Type	Description
pstm	IStream*	a pointer to a stream that contains the data packet which is to be destroyed.
return value	HRESULT	S_OK, E_FAIL

CoGetStandardMarshal

HRESULT CoGetStandardMarshal(iid, pUnkObject, dwDestContext, pvDestContext, mshlflags, ppmarshal)

Return an IMarshal instance that knows how to do the standard marshaling and unmarshaling in order to create a proxy in the indicated destination context. Custom marshaling implementations should delegate to the marshaler here returned for destination contexts that they do not fully understand or for which they

[18] using a higher-level notification scheme appropriate for the semantics of the particular connection. An example of this is OLE 2.0 is broadcasting IAdviseSink::OnClose() to connected links.

 Copyright © 1995 Microsoft Corporation All Rights Reserved

choose not to take special action. The standard marshaler is also used in the case that the object being marshaled does not support custom marshaling.

Argument	Type	Description
iid	REFIID	the interface id we would like to marshal.
pUnkObject	IUnknown*	the object that we wish to marshal. It is specifically *not* the case that this interface is known to be of shape iid; rather, it can be any interface on the object which conforms to IUnknown. The standard marshaler will internally do a QueryInterface().
dwDestContext	DWORD	the destination context in which the unmarshaling will occur.
pvDestContext	void *	associated with the destination context.
mshlflags	DWORD	the marshal flags for the marshaling operation.
ppmarshal	IMarshal **	the place at which the standard marshaler should be returned.
return value	HRESULT	S_OK, E_FAIL

CoGetMarshalSizeMax

HRESULT CoGetMarshalSizeMax(riid, pUnk, dwDestContext, pvDestContext, mshlfags, pulSize)

Return the number of bytes needed to marshal the given interface on the given object. On successful exit, the value pointed to by *pulSize will have been *incremented* by the number of bytes required.

This function is useful to custom marshaling implementations which themselves internally marshal interface pointers as part of their state.

Argument	Type	Description
riid	REFIID	the interface on the object which is to be marshaled.
pUnk	IUnknown*	an IUnknown (any old one) on the object.
dwDestContext	DWORD	the context into which the object is to be marshaled.
pvDestContext	void *	the context into which the object is to be marshaled.
mshlflags	DWORD	the marshal flags for the marshaling operation
pulSize	ULONG *	the place at which the required size is to be returned.
return value	HRESULT	S_OK, E_NOINTERFACE, E_OUTOFMEMORY, E_UNEXPECTED

1.9IMarshal interface

IMarshal interface is the mechanism by which an object is custom-marshaled. IMarshal is defined as follows:

```
interface IMarshal : IUnknown {
    HRESULT    GetUnmarshalClass(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pclsid);
    HRESULT    GetMarshalSizeMax(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pcb);
    HRESULT    MarshalInterface(pstm, iid, pvInterface, dwDestContext, pvDestContext, mshlflags);
    HRESULT    UnmarshalInterface(pstm, iid, ppvInterface);
    HRESULT    DisconnectObject(dwReserved);
    HRESULT    ReleaseMarshalData(pstm);
    };
```

The process of custom marshaling an interface pointer involves two steps, with an optional third:

1. The code doing the marshaling calls IMarshal::GetUnmarshalClass(). This returns the class id that will be used to create an uninitialized proxy object in the unmarshaling context.
2. (optional) The marshaler calls IMarshal::GetMarshalSizeMax() to learn an upper bound on the amount of memory that will be required by the object to do the marshaling.
3. The marshaler calls IMarshal::MarshalInterface() to carry out the marshaling.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

The class id and the bits that were marshaled into the stream are then conveyed by appropriate means to the destination, where they are unmarshaled. Unmarshaling involves the following essential steps:

1. Load the class object that corresponds to the class that the server said to use in GetUnmarshalClass().

```
IClassFactory * pcf;
CoGetClassObject(clsid, CLSCTX_INPROCSERVER, IID_IClassFactory, &pcf);
```

2. Instantiate the class, asking for IMarshal interface:

```
IMarshal * proxy;
pcf->CreateInstance(NULL, IID_IMarshal, &proxy);
```

3. Initialize the proxy with IMarshal::UnmarshalInterface() using a copy of the bits that were originally produced by IMarshal::MarshalInterface() and asking for the interface that was originally marshaled.

```
IOriginal * pobj;
proxy->UnmarshalInterface(pstm, IID_Original, &pboj);
proxy->Release();
pcf->Release();
```

The object proxy is now ready for use.

IMarshal::GetUnmarshalClass

HRESULT IMarshal::GetUnmarshalClass(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pclsid)

Answer the class that should be used in the unmarshaling process to create an uninitialized object proxy.

dwDestContext is described in the API function CoMarshalInterface. The implementation of GetUnmarshalClass may wish for some destination contexts for which it takes no special action to delegate to the standard marshaling implementation, which is available through CoGetStandardMarshal. In addition, this delegation should *always* be done if the dwDestContext parameter contains any flags that the GetUnmarshalClass does not fully understand; it is by this means that we can extend the richness of destination contexts in the future. For example, in the future, one of these bits will likely be defined to indicate that the destination of the marshaling is across the network.

If the caller already has in hand the iid interface identified as being marshaled, he should pass the interface pointer through pvInterface. If he does not have this interface already, then he should pass NULL. This pointer can sometimes, though rarely, be used in order to determine the appropriate unmarshal class. If the IMarshal implementation really needs it, in can always QueryInterface on itself to retrieve the interface pointer; we optionally pass it here only to improve efficiency.

Argument	Type	Description
iid	REFIID	the interface on this object that we are going to marshal.
pvInterface	void *	the actual pointer that will be marshaled. May be NULL.
dwDestContext	DWORD	the destination context relative to the current context in which the unmarshaling will be done.
pvDestContext	void*	non-NULL for some dwDestContext values.
mshlflags	DWORD	as in CoMarshalInterface().
pclsid	CLSID *	the class to be used in the unmarshaling process.
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE, E_UNEXPECTED

IMarshal::MarshalInterface

HRESULT IMarshal::MarshalInterface(pstm, iid, pvInterface, dwDestContext, pvDestContext, mshlflags)

Marshal a reference to the interface iid of this object into the given stream. The interface actually marshaled is the one that would be returned by this->QueryInterface(iid, ...). Once the contents of this stream are conveyed to the destination by whatever means, the interface reference can be reconstituted by instantiating with IMarshal interface the class here retrievable with GetUnmarshalClass and then calling

 Copyright © 1995 Microsoft Corporation All Rights Reserved

IMarshal::UnmarshalInterface. The implementation of IMarshal::MarshalInterface writes in the stream any data required for initialization of this proxy.

If the caller already has in hand the iid interface identified as being marshaled, he should pass the interface pointer through pvInterface. If he does not have this interface already, then he should pass NULL; the IMarshal implementation will QueryInterface on itself to retrieve the interface pointer.

On exit from this function, the seek pointer in the stream must be positioned immediately after the last byte of data written to the stream.

Argument	Type	Description
pstm	IStream *	the stream onto which the object should be marshaled.
iid	REFIID	the interface of this object that we wish to marshal.
pvInterface	void *	the actual pointer that will be marshaled. May be NULL.
dwDestContext	DWORD	as in CoMarshalInterface().
pvDestContext	void *	as in CoMarshalInterface().
mshlflags	DWORD	as in CoMarshalInterface().
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE, STG_E_MEDIUMFULL, E_UNEXPECTED

IMarshal::GetMarshalSizeMax

HRESULT IMarshal::GetMarshalSizeMax(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pcb)

Return an upper bound on the amount of data that would be written into the marshaling stream in an IMarshal::MarshalInterface() stream. The value returned must be an upper bound in the sense that it must be the case that a subsequent call to MarshalInterface() in fact require no more than the indicated number of bytes of marshaled data.

Callers can optionally use the returned upper bound to pre-allocate stream buffers used in the marshaling process. Note that when IMarshal::MarshalInterface() is ultimately called, the IMarshal implementation cannot rely on the caller actually having called GetMarshalSizeMax() beforehand; it must still be wary of STG_E_MEDIUMFULL errors returned by the stream.

The value returned by this function is guaranteed by the callee to be a conservative estimate of the amount of data needed to marshal the object; it is valid so long as the object instance is alive. Violation of this can be treated as a catastrophic error. To repeat for emphasis: an object *must* return a reasonable maximum size needed for marshaling: callers have the option of allocating a fixed-size marshaling buffer.

Argument	Type	Description
iid	REFIID	the interface of this object that we wish to marshal.
pvInterface	void *	the actual pointer that will be marshaled. May be NULL.
dwDestContext	DWORD	as in CoMarshalInterface().
pvDestContext	void *	as in CoMarshalInterface().
mshlflags	DWORD	as in CoMarshalInterface().
pcb	ULONG *	the place at which the maximum marshal size should be returned. A return of zero indicates “unknown maximum size.”
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE, E_UNEXPECTED

IMarshal::UnmarshalInterface

HRESULT IMarshal::UnmarshalInterface(pstm, iid, ppvInterface)

This is called as part of the unmarshaling process in order to initialize a newly created proxy; see the above sketch of the unmarshaling process for more details.

iid indicates the interface that the caller in fact would like to retrieve from this object; this interface instance is returned through ppvInterface. In order to support this, UnmarshalInterface will often merely do a QueryInterface(iid, ppvInterface) on itself immediately before returning, though it is free to create a different object (an object with a different identity) if it wishes.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

On successful exit from this function, the seek pointer must be positioned immediately after the data read from the stream. On error exit, the seek pointer should still be in this location: even in the face of an error, the stream should be positioned as if the unmarshal were successful.

See also CoReleaseMarshalData.

Argument	Type	Description
pstm	IStream *	the stream from which the interface should be unmarshaled.
iid	REFIID	the interface that the caller ultimately wants from the object.
ppvInterface	void **	the place at which the interface the caller wants is to be returned.
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE, E_UNEXPECTED

IMarshal::Disconnect

HRESULT IMarshal::DisconnectObject(dwReserved)

This function is called by the implementation of CoDisconnectObject in the event that the object attempting to be disconnected in fact supports custom marshaling. This is completely analogous to how CoMarshalInterface defers to IMarshal::MarshalInterface in if the object supports IMarshal.

Argument	Type	Description
dwReserved	DWORD	as in CoDisconnectObject().
return value	HRESULT	as in CoDisconnectObject().

IMarshal::ReleaseMarshalData

HRESULT IMarshal::ReleaseMarshalData(pstm)

This function is called by CoReleaseMarshalData() in order to actually carry out the destruction of a marshaled-data-packet. See that function for more details.

Note that whereas the IMarshal methods

GetUmarshalClass
GetMarshalSizeMax
MarshalInterface
Disconnect

are always called on the IMarshal interface instance in the originating side (server side), the method

UnmarshalInterface

is called on the receiving (client) side. (This should be no surprise.) However, the function

ReleaseMarshalData

(when needed) will be called on the receiving (client) side if the appropriate IMarshal instance can be successfully created there; otherwise, it is invoked on the originating (server) side.

Argument	Type	Description
pstm	IStream*	as in CoReleaseMarshalData().
return value	HRESULT	as in CoReleaseMarshalData().

1.10IStdMarshalInfo interface

IStdMarshalInfo is implemented by objects wishing to support handler marshaling in their remote client process. This is common, for example, for OLE 2 compound document embedded objects which for example support client-side drawing related interfaces using the IViewObject interface, an interface which is not (usually) supported on the actual embedding itself.

```
interface IStdMarshalInfo : IUnknown {
    HRESULT      GetClassForHandler(DWORD dwDestContext, void* pvDestContext, CLISD* pclsid);
    };
```


Copyright © 1995 Microsoft Corporation
All Rights Reserved

IStdMarshalInfo::GetClassForHandler

HRESULT IStdMarshalInfo::GetClassForHandler(dwDestContext, dwDestContext, pclsid)

Return the CLSID whose handler is to be used in the remote client process.

Argument	Type	Description
dwDestContext	DWORD	As in CoMarshalInterface.
pvDestContext	void*	As in CoMarshalInterface.
pclsid	CLSID*	The place at which the requested CLSID is returned.

Return Value	Meaning
S_OK	Success. The required CLSID is returned.
E_UNEXPECTED	An unspecified error occurred.

1.11 Support for Remote Debugging

The COM Library and the COM Network Protocol provide support for debugging engines on the client and the server side of a remote COM invocation to cooperate in allowing the overall application to be debugged. This section describes the runtime infrastructure provided by the Microsoft Windows implementation of the COM Library by which that is accomplished; other implementations will provide similar infrastructures, though in practice the details of such support will be highly sensitive to the mechanisms by which debugging engines are supported on the given platform. This section also specifies the standard data formats transmitted between client and server by which this cooperation is carried out.

The following a brief example of the sort of debugging session scenario which can be supported with this infrastructure.

Suppose the programmer is debugging an application with is an OLE document container, and that the application is presently stopped in the debugger at a point in the code where the container is about to invoke a method in some interface on one of its contained objects, the implementation of which happens to be in another executable. That is, the pointer that the container has in hand actually points to an occurrence of part of the remoting infrastructure known as an "interface proxy" (see above). Interface proxies and the rest of the remoting infrastructure are not (normally) part of the programmer's concern when debugging client and server applications, as the whole *raison d'être* of the RPC infrastructure is to be *transparent*, is to make remote object invocations appear to be local ones. Unless the programmer is debugging the remoting infrastructure himself, this should apply to debugging as well.

This perspective leads to some of the following scenarios that need to be supportable by the debugger. If the programmer Single Steps into the function invocation, then the debugger should next stop just inside the real implementation of the remote server object, having transparently passed through the RPC infrastructure. (Notice that before the Step command is executed, the remote process may not presently have the debugger[19] attached to it, and so the act of doing the step may need to cause the debugger to attach itself.) The programmer will now be able to step line by line through the server's function. When he steps past the closing brace of the function, he should wind up back in the debugger of the client process immediately after the function call.

A similar scenario is one where we skip the incoming single step but instead, out of the blue, hit a break-point in the server, then start single stepping. This, too, should single step over the end of the server function back into the client process. The twist is that this time, the *client* debugger may not presently be running, and therefore may need to be started.

1.11.1Implementation

The ability for debuggers to support scenarios such as these is provided by hooks in the client and server side RPC infrastructure. If requested by the debugger, at certain important times, these hooks inform the debugger of the fact that a transmission of a remote call about to be made or that transmission of return values is about to occur. That is, when the COM Library is about to make or return from a call to an

[19] More precisely, it may not have *a* debugger attached to it: depending on the debugger's implementation and the relative location of the two processes with respect to machine boundaries, a new debugger instance may or may not need to be created. The main point is that the process wasn't being debugged.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

object, it notifies the debugger of what is happening, so that the debugger can take any special actions it desires.

DllDebugObjectRPCHook

BOOL WINAPI DllDebugObjectRPCHook(BOOL fTrace, LPORPC_INIT_ARGS lpOrpcInitArgs)

This function is to be exported by name from one or more DLLs that wish to be informed when from the user's point of view that debugging is engaged. Debuggers will should call this function to inform each of their loaded DLLs that export this function as to whether they are presently being debugged or not. When the debugger wants to enable debugging, it calls DllDebugObjectRpcHook with fTrace=TRUE and when it wants to disable it, it calls DllDebugObjectRpcHook with fTrace=FALSE. When enabled, debugging support such as the tracing described herein should be enabled.

Certain of the COM Library DLLs, for example, implement this function. When debugging is enabled, they turn on what is here called COM remote debugging, and which is the focus of this section.

The second argument points to an ORPC_INIT_ARGS structure whose definition is given below. The pvPSN member is used only on the Macintosh, where the calling debugger is required in this field to pass the process serial number of the debuggee's process. On other systems pvPSN should be NULL.

The lpIntfOrpcDebug member is a pointer to an interface. This is used by in-process debuggers and is discussed in more detail later. Debuggers that are neither in-process debuggers nor are Macintosh debuggers should pass NULL for lpIntfOrpcDebug.

```
typedef struct ORPC_INIT_ARGS {
    IOrpcDebugNotify __RPC_FAR *  lpIntfOrpcDebug;
    void *                        pvPSN;          // contains ptr to Process Serial No. for Mac COM debugging.
    DWORD                         dwReserved1;    // For future use, must be 0.
    DWORD                         dwReserved2;    // For future use, must be 0.
    } ORPC_INIT_ARGS;

typedef ORPC_INIT_ARGS  __RPC_FAR * LPORPC_INIT_ARGS;

interface IOrpcDebugNotify : IUnknown {
    VOID    ClientGetBufferSize(LPORPC_DBG_ALL);
    VOID    ClientFillBuffer(LPORPC_DBG_ALL);
    VOID    ClientNotify(LPORPC_DBG_ALL);
    VOID    ServerNotify(LPORPC_DBG_ALL);
    VOID    ServerGetBufferSize(LPORPC_DBG_ALL);
    VOID    ServerFillBuffer(LPORPC_DBG_ALL);
    };
```

As one would expect, a debugger calls DllDebugObjectRPCHook within the context (that is, within the process) of the relevant debuggee. Thus, the implementation of this function most often will merely store the arguments in global DLL-specific state.

Further, as this function is called from the debugger, the function can be called when the DLL in which it is implemented is in pretty well any state; no synchronization with other internal DLL state can be relied upon. Thus, it is recommended that the implementation of this function indeed do nothing *more* than set internal global variables.

Argument	Type	Description
fTrace	BOOL	TRUE if debugging is enabled, FALSE otherwise
lpOrpcInitArgs	LPORPC_INIT_ARGS	typically NULL; see comments above for MAC COM debuggers or in-process debuggers.
return value	BOOL	TRUE if the function was successful (the DLL understood and executed the request), FALSE otherwise

1.11.2Architectural Overview

When COM remote debugging is enabled, there are a total of six notifications that occur in the round-trip of one COM RPC call: three on the client side and three on the server side. The overall sequence of events is as follows.

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

Suppose the client has an interface pointer pFoo of type IFoo* which happens to be a proxy for another object in a remote server process.

```
interface IFoo : IUnknown {
    HRESULT Func();
    };
IFoo *pFoo;
```

When the client invokes pFoo->Func(), it executes code in the interface proxy. This code is responsible for marshaling the arguments into a buffer, calling the server, and unmarshaling the return values. To do so, it draws on the services of an IRpcChannelBuffer instance with which it was initialized by the COM Library.

To get the buffer, the interface proxy calls IRpcChannelBuffer::GetBuffer(), passing in (among other things) the requested size for the buffer. Before actually allocating the buffer, the GetBuffer() implementation (normally[20]) checks to see if debugging is enabled per DllDebugObjectRPCHook(). If so, then the channel calls DebugORPCClientGetBufferSize() (see below for details) to inform the debugger that an COM RPC call is about to take place and to ask the debugger how many bytes of information *it* would like to transmit to the remote server debugger. The channel then, unbeknownst to the interface proxy, allocates a buffer with this many additional bytes in it.

The interface proxy marshals the incoming arguments in the usual way into the buffer that it received, then calls IRpcChannelBuffer::SendReceive(). Immediately on function entry, the channel again checks to see if debugging is enabled. If so, then it calls DebugORPCClientFillBuffer() passing in the pointer to (the debugger's part of) the marshaling buffer. The debugger will write some information into the buffer, but this need be of no concern to the channel implementation other than that it is to ferry the contents of the buffer to the server debugger. Once DebugORPCClientFillBuffer() returns, the channel implementation of SendReceive() proceeds as in the normal case.

We now switch context in our explanation here to the server-side RPC channel. Suppose that it has received an incoming call request and has done what it normally does just up to the point where it is about to call IRpcStubBuffer::Invoke(), which when will cause the arguments to be unmarshaled, etc. Just before calling Invoke(), if there was any debugger information (i.e.: it exists in the incoming request and is of non-zero size) in the incoming request *or* if debugging is presently *already* enabled per DllDebugObjectRPCHook() (irrespective of the presence or size of the debug info), then the channel is to call DebugORPCServerNotify().[21] The act of calling this function may in fact *start* a new debugger if needed and attach it to this (the server) process; however, this need not be of concern to the channel implementation. Having made the request, the channel proceeds to call Invoke() as in the normal case.

The implementation of Invoke() will unmarshal the incoming arguments, then call the appropriate method on the server object. When the server object returns, Invoke() marshals the return values for transmission back to the client. As on the client side, the marshaling process begins by calling IRpcChannelBuffer::GetBuffer() to get a marshaling buffer. As on the client side, the server side channel GetBuffer() implementation when being debugged (per the present setting of DllDebugObjectRPCHook(), *not* per the presence of the incoming debug info) asks the debugger how many bytes it wishes to transmit back to the client debugger. The channel allocates the buffer accordingly and returns it to the Invoke() implementation who marshals the return values into it, then returns to its caller.

The caller of IRpcStubBuffer::Invoke() then checks to see if he is presently being debugged. If so, then he at this time calls DebugORPCServerFillBuffer(), passing in the pointer to the debug-buffer that was allocated in the (last, should there erroneously be more than one) call to GetBuffer() made inside Invoke(); should no such call exist, and thus there is no such buffer, NULL is passed.[22] The bytes written into the buffer (if any) by the debugger are ferried to the client side.

We now switch our explanatory context back to the client side. Eventually the client channel either receives a reply from the server containing the marshaled return values (and possibly debug info), receives an error indication from the server RPC infrastructure, or decides to stop waiting. That is, even-

[20] That is, in the channel implementation approach described here, which uses only one memory buffer. Another channel implementation approach would use two separate buffers, one to give back to the interface proxy, and another independent one for the debug information. Such an implementation would only need to call DebugORPCClientGetBufferSize() in its IRpcChannelBuffer::SendReceive() implementation immediately before calling DebugORPCClientFillBuffer(). While perfectly legal, this will not be elaborated further here, though in fact this is the implementation likely to be used in practice, given how the debug data is to be transmitted in the COM Network Protocol. We trust that readers can accommodate our pedagogical style; apologies to those who cannot.

[21] Some control as to whether this is to be actually carried out is provided by the first four bytes of the incoming debug data; see later in this specification.

[22] This is important in error handling cases to allow us to ensure that breakpoints are always cleared correctly.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

tually the client channel decides that it is about to return from IRpcChannel::SendReceive(). Immediately before doing so, it checks to see if it is either already presently being debugged *or* if in the reply it received any (non-zero sized) information from the server debugger. If so, then it calls DebugORPCClientNotify(), passing in the server-debugger's info if it has any; doing so may start and attach the debugger if needed. The channel then returns from SendReceive().

1.11.3Calling Convention for Notifications

The preceding discussion discussed the COM RPC debugging architecture in terms of six of debugger-notification APIs (DebugORPC...()). However, rather than being actual API-entry points in a a static-linked or dynamically-linked library, these notifications use an somewhat unusual calling convention to communicate with the notification implementations, which are found inside debugger products. This somewhat strange calling convention is used for the following reasons:

- Two of the six notifications need to start and attach the debugger if it is not already attached to the relevant process.
- The convention used transitions into the debugger code with the least possible disturbance of the debuggee's state and executing the minimal amount of debuggee code. This increases robustness of debugging.
- The debugger is necessarily equipped to deal with concurrency issues of other threads executing in the same process. Therefore, it is important to transition to the debugger as fast as possible to avoid inadvertent concurrency problems.

The actual calling convention used is by its nature inherently processor and operating-system specific. On Win32 implementations, the default calling convention for notifications takes the form of a software exception, which is raised by a call to the RaiseException Win32 API:

```
VOID RaiseException(
    DWORD  dwExceptionCode,         // exception code
    DWORD  dwExceptionFlags,        // continuable exception flag
    DWORD  cArguments,              // number of arguments in array
    CONST DWORD *  lpArguments      // address of array of arguments
    );
```

As used here, the arguments to this raised exception call in order are:

- dwExceptionCode: An exception code EXCEPTION_ORPC_DEBUG (0x804F4C45) is used. The debugger should recognize this exception as a special one indicating an COM RPC debug notification.
- dwExceptionFlags: This is zero to indicate a continuable exception.
- cArguments: One
- lpArguments: The array contains one argument. This argument is a pointer to a structure which contains the notification specific information that the COM RPC system passes to the debugger. The definition of this structure ORPC_DBG_ALL is given below. The same structure is used for all the notifications. The structure is just the union of the arguments of the six debugger notification APIs. For a particular notification not all the fields in the structure are meaningful and those that are not relevant have undefined values; details on this are below:

```
typedef struct ORPC_DBG_ALL  {
    BYTE *                  pSignature;
    RPCOLEMESSAGE *         pMessage;
    const IID *             iid;
    void*                   reserved1;
    void*                   reserved2;
    void*                   pInterface;
    IUnknown *              pUnkObject;
    HRESULT                 hresult;
    void *                  pvBuffer;
    ULONG                   cbBuffer;
    ULONG *                 lpcbBuffer;
    void *                  reserved3;
    } ORPC_DBG_ALL;
```

 Copyright © 1995 Microsoft Corporation
All Rights Reserved

The pSignature member of this structure points to a sequence of bytes which contains:

- a four-byte sanity-check signature of the ASCII characters "MARB" in increasing memory order.[23]
- a 16-byte GUID indicating which notification this is. Each of the six notifications defined here has a different GUID. More notifications and corresponding GUIDs can be defined in the future and be known not to collide with existing notifications.
- a four-byte value which is reserved for future use. This value is NULL currently.

The notifications specified here pass their arguments by filling in the appropriate structure members. See each notification description for details.

Using software exceptions for COM debugging notifications is inconvenient for "in-process" debugging. In-process debuggers can alternately get these notifications via direct calls into the debugger's code. The debugger which wants to be notified by a direct call passes in an IOrpcDebugNotify interface in the LPORPC_INIT_ARGS argument to DllDebugObjectRPCHook. If this interface pointer is available, COM makes the debug notifications by calling the methods on this interface. The methods all take an LPORPC_DBG_ALL as the only argument. The information passed in this structure is identical to that passed when the notification is done by raising a software exception.

1.11.4Notifications

What follows is a detailed description of each of the relevant notifications.

Note that in the network case, depending on the notification in question the byte order used may be different than that of the local machine. The byte order, etc., of the incoming data is provided from the dataRep contained the passed RPCOLEMESSAGE structure.

Though each function is documented here for purely historical reasons as if it were in fact a function call, we have seen above that this is not the case. Unless otherwise specified, the name of the argument to the DebugORPC... notification call is the same as the name of the structure member in ORPC_DBG_ALL used to pass it to the debugger. So for example the pMessage argument of the DebugORPCClientGetBufferSize notification is passed to the debugger in the pMessage structure member of ORPC_DBG_ALL. We trust that readers will not be too confused by this, and apologize profusely should this prove not to be the case.

DebugORPCClientGetBufferSize

ULONG DebugORPCClientGetBufferSize(pMessage, iid, reserved, pUnkProxyObject)

Called on the client side in IRpcChannel::GetBuffer().

The GUID for this notification is 9ED14F80-9673-101A-B07B-00DD01113F11

```
GUID __private_to_macro__ = { /* 9ED14F80-9673-101A-B07B-00DD01113F11 */
  0x9ED14F80,
  0x9673,
  0x101A,
  0xB0,
  0x7B,
  {0x00, 0xDD, 0x1, 0x11, 0x3F, 0x11}
 };
```

[23] "MARB" is "Mike Alex Rico Bob," arranged in an order such that it makes a goofy-sounding syllable. Call us whimsical.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	identification of the method being invoked, etc.
iid	REFIID	contains the IID of the interface being called.
reserved	void *	reserved for future use.
pUnkProxyObject	IUnknown *	an IUnknown (no particular one) on the object involved in this invocation. May legally be NULL, though this reduces debugging functionality. Further, this and like-named parameters must consistently be either NULL or non-NULL in all notifications in a given client side COM RPC implementation.
"return value"	ULONG	the number of bytes that the client debugger wishes to transmit to the server debugger. May legitimately be zero, which indicates that no information need be transmitted. The lpcbBuffer field in the ORPC_DBG_ALL structure holds a pointer to a ULONG. The debugger writes the number of bytes it wants to transmit with the packet in that location.

DebugORPCClientFillBuffer

void DebugORPCClientFillBuffer(pMessage, iid, reserved, pUnkProxyObject, pvBuffer, cbBuffer)

Called on the client side on entry to IRpcChannel::SendReceive(). See the above overview for further details.

The GUID for this notification is DA45F3E0-9673-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* DA45F3E0-9673-101A-B07B-00DD01113F11 */
  0xDA45F3E0,
  0x9673,
  0x101A,
  0xB0,
  0x7B,
  {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
 };
```

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in DebugORPCClientGetBufferSize().
iid	REFIID	as in DebugORPCClientGetBufferSize().
reserved	void *	as in DebugORPCClientGetBufferSize().
pUnkProxyObject	IUnknown *	as in DebugORPCClientGetBufferSize().
pvBuffer	void *	the debug-data buffer which is to be filled. Is undefined (may or may not be NULL) if cbBuffer is zero.
cbBuffer	ULONG	the size of the data pointed to by pvBuffer.

DebugORPCServerNotify

void DebugORPCServerNotify(pMessage, iid, pChannel, pInterface, pUnkObject, pvBuffer, cbBuffer)

Called on the server side immediately before calling IRpcStubBuffer::Invoke() to inform it that there is an incoming request. Will start the debugger in this process if need be. See the above overview for further details.

The GUID for this notification is 1084FA00-9674-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* 1084FA00-9674-101A-B07B-00DD01113F11 */
  0x1084FA00,
  0x9674,
  0x101A,
  0xB0,
  0x7B,
  {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
 };
```

On entry, the members of pMessage are set as follows:

Copyright © 1995 Microsoft Corporation
All Rights Reserved

Member Name	Value on entry to Invoke()
reserved members	indeterminate. These members are neither to be read nor to be changed by the callee.
dataRepresentation	this indicates the byte order, etc., of the client debugger
pvBuffer	points to a buffer which contains the marshaled incoming arguments. In the case that there are no such arguments (i.e.: cbBuffer == 0), pvBuffer may be NULL, but will not necessarily be so.
cbBuffer	the size in bytes of the memory buffer to which pvBuffer points. If pvBuffer is NULL, then cbBuffer will be zero (but the converse is not necessarily true, as was mentioned in pvBuffer).
iMethod	the zero-based method number in the interface which is being invoked.
rpcFlags	indeterminate. Neither to be read nor to be changed by the callee.

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in IRpcStubBuffer::Invoke().
iid	REFIID	contains the iid of the interface being called.
pChannel	IRpcChannelBuffer*	as in IRpcStubBuffer::Invoke(). The COM RPC channel implementation on the server side.
pInterface	void *	This contains the pointer to the COM interface instance which contains the pointer to the method that will be invoked by this particular remote procedure call. Debuggers can use this information in conjunction with the iMethod field of the pMessage structure to get to the address of the method to be invoked. May not be NULL.
pUnkObject	IUnknown *	this pointer is currently NULL. In the future this might be used to pass the controlling IUnknown of the server object whose method is being invoked.
pvBuffer	void *	the pointer to the incoming debug information. Is undefined (may or may not be NULL) if cbBuffer is zero.
cbBuffer	ULONG	the size of the data pointed to by pvBuffer. May be zero, but as described above, a size of zero can only passed in the case that debugging is already enabled.

DebugORPCServerGetBufferSize

ULONG DebugORPCServerGetBufferSize(pMessage, iid, pChannel, pInterface, pUnkObject)

Called on the server side from within IRpcChannelBuffer::GetBuffer(). See the above overview for further details.

The GUID for this notification is 22080240-9674-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* 22080240-9674-101A-B07B-00DD01113F11 */
  0x22080240,
  0x9674,
  0x101A,
  0xB0,
  0x7B,
  {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
};
```

Copyright © 1995 Microsoft Corporation
All Rights Reserved

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in DebugORPCServerNotify().
iid	REFIID	as in DebugORPCServerNotify().
pChannel	IRpcChannelBuffer*	as in DebugORPCServerNotify().
pInterface	void *	as in DebugORPCServerNotify().
pUnkObject	IUnknown *	as in DebugORPCServerNotify().
return value	ULONG	the number of bytes that the client debugger wishes to transmit to the server debugger. May legitimately be zero, which indicates that no information need be transmitted. Value is actually returned through lpcbBuffer member of an ORPC_DBG_ALL.

DebugORPCServerFillBuffer

void DebugORPCServerFillBuffer(pMessage, iid, pChannel, pInterface, pUnkObject, pvBuffer, cbBuffer)

Called on the server side immediately after calling IRpcStubBuffer::Invoke(). See the above overview for further details.

The GUID for this notification is 2FC09500-9674-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* 2FC09500-9674-101A-B07B-00DD01113F11 */
  0x2FC09500,
  0x9674,
  0x101A,
  0xB0,
  0x7B,
  {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
};
```

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in DebugORPCServerNotify().
iid	REFIID	as in DebugORPCServerNotify().
pChannel	IRpcChannelBuffer*	as in DebugORPCServerNotify().
pInterface	void *	as in DebugORPCServerNotify().
pUnkObject	IUnknown *	as in DebugORPCServerNotify().
pvBuffer	void *	the debug-data buffer which is to be filled. Is undefined (may or may not be NULL) if cbBuffer is zero.
cbBuffer	ULONG	the size of the data pointed to by pvBuffer.

DebugORPCClientNotify

void DebugORPCClientNotify(pMessage, iid, reserved, pUnkProxyObject, hresult, pvBuffer, cbBuffer)

Called on the client side immediately before returning from IRpcChannelBuffer::SendReceive(). See the above overview for further details.

The GUID for this notification is 4F60E540-9674-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* 4F60E540-9674-101A-B07B-00DD01113F11 */
  0x4F60E540,
  0x9674,
  0x101A,
  0xB0,
  0x7B,
  {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
};
```


Copyright © 1995 Microsoft Corporation
All Rights Reserved

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in DebugORPCClientGetBufferSize().
iid	REFIID	as in DebugORPCClientGetBufferSize().
reserved	void *	reserved for future use.
pUnkProxyObject	IUnknown *	as in DebugORPCClientGetBufferSize().
hresult	HRESULT	the HRESULT of the RPC call that just happened.
pvBuffer	void *	the pointer to the incoming debug information. Is undefined (may or may not be NULL) if cbBuffer is zero.
cbBuffer	ULONG	the size of the data pointed to by pvBuffer.

1.11.5Special Segments

The COM Library system DLLs have code in specially named segments (sections in COFF terminology) to aid debuggers. The remoting code in the COM interface proxy and interface stub DLLs and other appropriate parts of the runtime are put in segments whose name begins with ".orpc"[24]. These segments are henceforth referred to as .orpc segments. A transition of the instruction pointer from a non .orpc segment to a .orpc segment indicates that the program control is entering the RPC layer. On the client side such a transition implies that a RPC call is about to happen.[25] On the server side if a function is returning back to a .orpc segment it implies that the call is going to return back to the client side. Application writers who write their own remoting code can also avail of this feature by putting their remoting specific code in a .orpc segment.

Debuggers can use this naming convention regarding which code lies in COM RPC to aid in their user interface as to what code they choose to show the user and what code they do not. When the debugger reaches the code address after handling the DebugOrpcServerNotify() exception it should check if it is still in a .orpc segment. This implies that the instruction pointer is still in code that to the programmer is part of the local-remote transparency magic provided by COM, and so should be skipped by the debugger.

Similar behavior on the client side after the DebugOrpcClientNotify() exception is also desirable.

1.11.6Registry specific information

Windows NT and Windows '95 provide facilities to spawn a debugger when an application faults. Familiarity with the post-mortem debugging support on these systems is assumed in this section.

COM RPC debuggers make use of this mechanism in order to start the debugging of a client or server application that is not presently being debugged. A common scenario is that of a user wanting to step into a RPC call as she is debugging. The client side debugger is notified about the RPC call and sends debugger specific information with the packet. A DebugOrpcServerNotify() notification is raised in the server process. If the server application is already being debugged, it recognizes this as a COM RPC notification and handles it. However if the server application is not being debugged, the system will launch the debugger specified in the AeDebug entry. The debugger will then get the exception notification and handle it.

To avoid having malicious clients being able to force the debugging of a remote server, additional safeguards are required. The COM RPC system checks that the registry key DebugObjectRPCEnabled exists on the system.[26] If this key does not exist, the debug notifications are disabled. Thus, debugging will only take place if explicit action has been taken on a given machine to enable it, and so a remote client cannot cause debugging (and thus denial of service) to occur on an otherwise secure machine.

The full path to this key for a Windows NT system is:

Software\Microsoft\Windows NT\CurrentVersion\DebugObjectRPCEnabled.

For Windows '95 the path to this key is:

Software\Microsoft\Windows\CurrentVersion\DebugObjectRPCEnabled.

[24] This is so segment names such as .orpc1, .orpc2... can be used if the remoting code needs to be split up into different segments for swap tuning, etc.

[25] It is not guaranteed that a RPC call will happen for every such transition. The debugger should deal with the case where it receives no notification about an RPC call.

[26] In Windows NT, the registry is securable.

Copyright © 1995 Microsoft Corporation
All Rights Reserved

The client side debugger should also ensure that the AeDebug\Debugger entry on its machine is set appropriately.

Before sending any notification, COM sets the AeDebug\Auto entry to 1. This is done in order that the system does not put up a dialog box to ask the user if she wants to debug the server application. Instead it directly launches the debugger.

The scenario where the user steps out of the server application into to a client application which is not being debugged currently is symmetrically identical the preceding insofar as launch of the debugger is concerned.

1.11.7 Format of Debug Information

This section discusses the format of the debug information which the debugger puts into the buffer in the DebugORPCClientFillBuffer and DebugORPCServerFillBuffer calls. The structure of this data is as follows, here specified in an IDL-like manner.[27] For historical reasons, this structure has 1-byte alignment of its internal members. Again, for historical reasons, the data is always transmitted in little-endian byte order.

```
#pragma pack(1) // this structure defined with 1-byte packing alignment
struct {
    DWORD   alwaysOrSometimes;  // controls spawning of debugger
    BYTE    verMajor;           // major version
    BYTE    verMinor;           // minor version
    DWORD   cbRemaining;        // inclusive of byte count itself
    GUID    guidSemantic;       // semantic of this packet
    [switch_is(guidSemantic)] union { // semantic specific information

    // case "step" semantic, guid = 9CADE560-8F43-101A-B07B-00DD01113F11
        BOOL    fStopOnOtherSide;   // should single step or not?

    // case "general" semantic, guid = D62AEDFA-57EA-11ce-A964-00AA006C3706
        USHORT  wDebuggingOpCode;   // should single step or not, etc.
        USHORT  cExtent;            // offset=28
        BYTE    padding[2];         // offset=30, m.b.z.
        [size_is(cExtent)] struct {
            ULONG cb;               // offset=32
            GUID  guidExtent;       // the semantic of this extent
            [size_is(cb)] BYTE  *rgbData;
            };
        };
    }
```

The first DWORD in the debug packet has a special meaning assigned to it. The rest of the debug packet is treated as a stream of bytes by COM and is simply passed across the channel to the debugger on the other side. If the first DWORD contains the value ORPC_DEBUG_ALWAYS (this is a manifest constant defined in the header files) then COM will *always* raise the notification on the other side (use of the four bytes "MARB" is for historical reasons synonymous with use of ORPC_DEBUG_ALWAYS). If the first DWORD in the debug packet contains the value ORPC_DEBUG_IF_HOOK_ENABLED, then the notification is raised on the other side of the channel only if COM debugging has been enabled in that context; that is only if DllDebugObjectRPCHook has been called in that process with fTrace = TRUE. It is the debugger's responsibility to include enough memory for the first DWORD in its response to the DebugOrpcClientGetBufferSize or DebugOrpcServerGetBufferSize notifications.

The two bytes immediately following the initial DWORD contain the major and minor version numbers of the data format specification.

For packets in the format of the current major version, this is followed by

- A DWORD which holds the count of bytes that follow in this data, and which is inclusive of this byte count itself.
- A GUID that identifies the semantic of the packet.
- Semantic specific information. The layout of this information is dependent on the GUID that specifies the semantic. These are as follows:

[27] One can think of this as IDL with a) default packing override, and b) the ability to have a union keyed by a GUID. This will be made more precise in future drafts of this specification.

 Copyright © 1995 Microsoft Corporation All Rights Reserved

Semantic	Meaning
Step	This semantic indicates that the single stepping is to be performed or not. The GUID of this semantic is 9CADE560-8F43-101A-B07B-00DD01113F11. The data of this semantic consists of a boolean value which indicates in the "step out of a server" case whether execution should continue once the other side is reached or one should remain stopped.
General	This semantic, which has GUID D62AEDFA-57EA-11ce-A964-00AA006C3706, allows for series of tagged bags of data to be passed. Each is byte counted, and has associated with it a GUID. wDebuggingOpCode allows for one of a series of operations to be specified. Existing-defined opcodes are as follows. Future opcodes are to be allocated by a central coordinating body.[28]

Opcode	Meaning
0x0000	No operation
0x0001	Single step, stop on the other side, as in the "Step" semantic.

Extents presently defined for use in the General semantic are as follows:

Extent	Meaning
Interface pointer	This semantic has GUID 53199051-57EB-11ce-A964-00AA006C3706. The contents of rgbData for this extent is simply an OBJREF, which is the data structure which describes a marshaled interface pointer, the data that results from calling CoMarshalInterface (OBREFs are described later in this specification). Usually, this OBJREF is either the self-enclosed LONGOBJREF variation or a custom-marshaled variation, but this is not required. The LONGOBJREF usually contains a reference count of zero, allowing this information to be freely discared without a leakage of state. Remember that OBJREFs are always in little-endian byte order. An OBJREF is converted into its corresponding interface pointer using CoUnmarshalInterface.

With the Interface Pointer extent, an object can be created in the source debugger's space that relates to the call being made. It can then be marshaled, again, in the source debugger's process, not the source debuggee; this yields an OBJREF. The OBJREF is then transmitted in the course of the call as an extent in the passed debug information. On the destination side, it is conveyed to the destination debugger, who unmarshals it in its process. The result is a COM remoting connection from the source debuggers process to the destination debugger's process that is semantically tied to a particular COM call that needs to be debugged. (TBD) Interfaces on this object can be then be used to provide stack walk-backs, remote memory manipulation, or other debugging functionality.

[28] This is presently Microsoft Corporation.

Copyright © 1995 Microsoft Corporation
All Rights Reserved