0Part V: The COM Library

It should be clear by this time that COM itself involves some systems-level code, that is, some implementation of its own. However, at the core the Component Object Model by itself is a specification (hence "Model") for how objects and their clients interact through the binary standard of interfaces. As a specification it defines a number of other standards for interoperability:

- The fundamental process of interface negotiation through QueryInterface.
- A *reference counting* mechanism through objects (and their resources) are managed even when connected to multiple clients.
- Rules for memory allocation and responsibility for those allocations when exchanged between independently developed components.
- Consistent and rich error reporting facilities.

In addition to being a specification, COM is also an implementation contained what is called the "COM Library." The implementation is provided through a library (such as a DLL on Microsoft Windows) that includes:

- A small number of fundamental API functions that facilitate the creation of COM applications, both clients and servers. For clients, COM supplies basic object creation functions; for servers the facilities to expose their objects.
- Implementation locator services through which COM determines from a class identifier which server implements that class and where that server is located. This includes support for a level of indirection, usually a system registry, between the identity of an object class and the packaging of the implementation such that clients are independent of the packaging which can change in the future.
- Transparent remote procedure calls when an object is running in a local or remote server. This includes the implementation of a standard network wire-protocol.
- A standard mechanism to allow an application to control how memory is allocated within its process.

In general, only one vendor needs to, or should, implement a COM Library for any particular operating system. For example, Microsoft has implemented COM on Microsoft Windows 3.1, Microsoft Windows 95, Microsoft Windows NT, and the Apple Macintosh.

The following chapter describes elements of the COM Library that a vendor implementing COM on a previously unsupported platform would require.

1Component Object Model Network Protocol

The COM network protocol is a protocol for object-oriented remote procedure calls and is thus also called Object RPC or ORPC. The Object RPC protocol consists of a set of extensions, layered on the distributed computing environment (DCE) RPC specification. The Object RPC protocol specifies:

- How calls are made on an object
- How object references are represented, communicated, and maintained

1.1Overview

The Object RPC protocol highly leverages the OSF DCE RPC network protocol (see the reference [CAE RPC]). This leverage occurs at both the specification level and the implementation level: the bulk of the implementation effort involved in implementing the COM network protocol is in fact that of implementing the DCE RPC network protocol on which it is built.

1.1.1Object Calls

An actual COM network remote procedure call (hereinafter referred to as "an ORPC") is in fact a true DCE remote procedure call (herein termed "a DCE RPC"), a "Request PDU" conforming to the specification for such calls per [CAE RPC].

In an ORPC, the object ID field of the invocation header as specified in [CAE RPC] contains an "IPID". An IPID is a 128-bit identifier known as an *interface pointer identifier* which represents a particular interface on a particular object in a particular server. As it is passed in the object ID fields of a DCE RPC, the static type of an IPID is in fact a UUID. However, IPIDs are scoped not globally but rather only relative to the machine on which the server is located (and which thus originally allocated them); IPIDs do not necessarily use the standard UUID allocation algorithm, but rather may use a machine-specific

This page intentionally left blank.

algorithm which can assist with dispatching.[1]

In an ORPC, the interface ID field of the RPC header specifies the IID, and arguments are found in the body, as usual.[2] However, when viewed from the DCE RPC perspective an additional first argument is always present that is absent in the corresponding COM interface specification. This argument is of type ORPCTHIS, which is described below. It is placed first in the body of the Request PDU, before the actual arguments of the ORPC.

It is specifically legal for an ORPC to attempt a call a method number on a given interface which is beyond the number of methods believed by the server to be in that interface. Such calls should cause a fault.

Similarly, in a reply to an ORPC (a DCE RPC "Response PDU"), when viewed from the DCE RPC perspective, an additional first return value is always present that is absent in the corresponding COM interface specification. This argument is of type ORPCTHAT, which is also described below. It is placed first in the body of the Response PDU, before the actual return values of the ORPC.

An ORPCTHAT may also be present in a "Fault PDU." In the Connectionless (CL) Fault PDU,[3] it is placed four bytes after the 32-bit fault code which normally comprises the entire body of the PDU, thus achieving eight byte alignment for the ORPCTHAT; the intervening padding bytes are presently reserved and must be zero. The PDU body length[4] is of course set to encompass the entire body of the Fault PDU, including the ORPCTHAT. In the Connection-oriented (CO) Fault PDU, the ORPCTHAT is placed in the

[1] As in DCE RPC object IDs are indeed only ever interpreted relative to a given machine, this relaxing of the DCE specification is not problematic.

[2] The IID in the interface ID field is from a logical perspective actually redundant because the IPID uniquely specifies an interface pointer (though the IID is not recoverable from just the IPID). However, an additional (optional) check to verify that the caller and callee agree on the type of the interface pointer would make the system more robust. Also, the specification of the actual IID in question eases the integration of the COM network protocol with the DCE RPC network protocol. Note that it is not expensive for callers to provide the IID since the space for the IID is allocated in the DCE RPC header, which is always transmitted anyway.

[3] For the specification of the Connectionless Fault PDU, see [CAE RPC], page 520. Page 535 of the same work describes the Connection-oriented Fault PDU.

[4] Ibid, page 516.

standard location allocated for the "stub data."[5] In a Fault PDU of either form that results from an ORPC, if an ORPCTHAT is not present then no other data may be substituted in its here-specified location in the PDU.

1.1.2OXIDs, Object Exporters, & Machines

Although an IPID from a logical perspective semantically determines the server, object and interface to which a particular call should be directed, it does not by itself indicate the binding information necessary to actually carry out an invocation.

The protocol represents this "how-to" communication information in a UUID called an object exporter identifier, otherwise known as an OXID. Conceptually, an OXID can be thought of as an implementation scope for an COM object, which may be a whole machine, a given process, a thread within that process, or other more esoteric implementation scope, but the exact definition of such scopes has no bearing on the COM network protocol.

A given machine at any moment may support several OXIDs; however there is always a unique *Object Exporter service* per machine which coordinates the management of all the OXIDs on the machine. Data structures in each Object Exporter keep track of the IPIDs exported and imported by that Object Exporter. The Object Exporter resides at well-known endpoints (one per protocol, of course) on the machine. It supports a DCE RPC interface known as IObjectExporter, which is described below.

An OXID is used to determine the RPC string bindings that allow calls to reach their target IPID. Before making a call, the calling process must translate an OXID into a set of bindings that the underlying RPC implementation understands. It accomplishes by maintaining a cache of these mappings. When the destination application receives an object reference, it checks to see if it recognizes the OXID. If it does not, then it asks the source of the object reference (the server machine from which the object reference was acquired, which is not necessarily the home machine for the interface pointer) for the translation, and saves the resulting set of string bindings in a local table that maps OXIDs to string bindings.

Associated with each OXID (not each Object Exporter) is COM object termed an "OXID object." OXID objects implement (at least) the IRemUnknown interface, through which remote management of reference counts and requests for interfaces are returned.

Each machine is represented by a MID. MIDs are UUIDs and thus universally unique. The MID for a machine may (should) change when the machine reboots. However, when the MID for a machine changes, all OXIDs, OIDs, and IPIDs on that machine become invalid. MIDs are an optimization to simplify the task of determining which OXIDs are exported and pinged by which object exporters.

1.1.3Marshaled Interface References

The COM network protocol extends the Network Data Representation standard specified in [CAE RPC] by defining what can be thought of as a new primitive data type that can be marshaled: that of an interface reference to a COM object.[6] This is the only extension to NDR made by the COM network protocol.

A marshaled interface references is described by a type known as an OBJREF, which is described in detail below. An OBJREF in actuality has several variations:

- Null.
- Standard. A standard remote reference. Known as a STDOBJREF. A STDOBJREF contains:
    - An IPID, which uniquely specifies the interface and object.
    - An object ID (OID), which uniquely specifies the identity of the object on which the IPID is found. OIDs are UUIDs; they are universally unique.
    - An OXID, which identifies the scope where the implementation of the object is active, and can be used to reach the interface pointer.[7]

[5] That is, in the Connection-oriented case, the ORPCTHAT also follows four bytes of padding after the fault code; however, the fault code in the Connection-oriented Fault PDU is preceded other data not found in the Connectionless Fault PDU. Consult [CAE RPC] for further details.

[6] Whether one actually thinks of this as a new primitive data type or new compositional operator over existing data types depends on one's point of view. Both positions have some merit.

[7] Note that the object reference does not include the interface identifier (IID), although you can't do much with the object reference without knowing the IID. The IPID does uniquely specify an IID; however, you can't algorithmically derive the IID from the IPID.

    - A reference count, indicating the number of references to this IPID that are conveyed by this marshaling. This count, though typically a value of one, may in fact be zero, one, or more (see the next section).
    - Some flags, explained later.
- Long. A standard reference, along with a set of protocol sequences and network addresses that can be useful when marshaling a proxy to give to another machine (a.k.a. the middle-man case).
- Custom. Contains a class ID (CLSID) and class-specific information.

  The Custom format gives an object control over the representation of references to itself. For example, an immutable object might be passed by value, in which case the class-specific information would contain the object's immutable data.
- Handler. A sub-case of the custom reference in which the class-specific information is standardized.

  For example, an object wishes to be represented in client address spaces by a proxy object that caches state. In this case, the class-specific information is just a standard reference to an interface pointer that the handler (proxy object) will use to communicate with the original object.
- Long Handler: Contains the same information as the handler case as well as the object resolver address. This form is needed for the same reason the long form is needed.

Interface references are *always* marshaled in little-endian byte order, irrespective of the byte order prevailing in the remainder of the data being marshaled.

1.1.4Reference Counting

In the COM network protocol, remote reference counting is conducted on per interface (per IPID), just as local reference counting is carried out on a per interface basis.

The actual increment and decrement calls are carried out using (respectively) the RemAddRef and RemRelease methods in a COM interface known as IRemUnknown found on an object associated with the each OXID, the IPID of which is returned from the function IObjectExporter::GetStringBindings (see below). In contrast to their analogues in IUnknown, RemAddRef and RemRelease can in one call increment or decrement the reference count of many different IPIDs by an arbitrary amount; this allows for greater network efficiency.

In the interests of performance, client COM implementations typically do *not* immediately translate each local AddRef and Release into a remote RemAddRef and RemRelease. Rather, the actual remote release of all interfaces on an object is typically deferred until all local references to all interfaces on that object have been released. Further, one actual remote reference count may be used to service many local reference counts; that is, the client infrastructure may multiplex zero or more local references to an interface into zero or one remote references on the actual IPID.[8]

1.1.5Pinging

The above reference counting scheme would be entirely adequate on its own if clients never crashed, but in fact they do, and the system needs to be robust in the face of clients crashing when they hold remote references. In a DCE RPC, one typically addresses this issue through the use of context handles. Context handles are *not* used, however, by the COM network protocol, for reasons of expense. The basic underlying technology used in virtually all protocols for detecting remote crashes is that of periodic pings. Naive use of RPC context handles would result in per object per client process pings being sent to the server. The COM network protocol architects its pinging infrastructure to reduce network traffic by relying on the client Object Exporter implementation to do local management of client liveness detection, and having the actual pings be sent only on a machine to machine basis.

Pinging is carried out on a per-object (per OID), not a per-interface (per IPID) basis. Architecturally, at its server machine, each exported object (each exported OID) has associated with it a pingPeriod time value

This is not a problem because the IID does not have to be explicitly specified; it is either implicitly specified (by the type of the argument in a MIDL declaration) or available explicitly as another argument in the call that is carrying the polymorphic object reference (for example, IUnknown::QueryInterface).

[8] A consequence of allowing for this optimizations is that the reference count on each IPID as seen by the server may be in fact less than the total number of references on that interface as seen by all the extant clients of that interface.

and a numPingsToTimeOut count which together (through their product) determine the overall amount of time known as the "ping period" that must elapse without receiving a ping on that OID or an invocation on one of its IPIDs before all the remote references to IPIDs associated with that OID can be considered to have expired. Once expiration has occurred, the interfaces behind the IPIDs can as would be expected be reclaimed solely on the basis local knowledge, though the timeliness with which this is carried out, if at all, is an implementation specific. If the server COM infrastructure defers such garbage collection in this situation (perhaps because it has local references keeping the interface pointer alive) and it later hears a ping or receives a remote call, then it knows a network partition healed. It can consider the extant remote references to be reactivated and can continue remote operations.

When interface pointers are conveyed from one client to another, such as being passed as either [in] or [out] parameters to a call, the interface pointer is marshaled in one client and unmarshaled in the other. In order to successfully unmarshal the interface, the destination client must obtain at least one reference count on the interface. This is usually accomplished by passing in the marshaled interface STDOBJREF a cRefs of (at least) one; the destination client then takes ownership of that many (more) reference counts to the indicated IPID, and the source client then owns that many fewer reference counts on the IPID. It is legal, however, for zero reference counts to be passed in the STDOBJREF; here, the destination client must (if it does not already have access to that IPID and thus have a non-zero reference count for it) before it successfully unmarshals the interface reference (concretely, e.g., before CoUnmarshalInterface returns) call to the object's exporter using IRemUnknown::RemAddRef to obtain a reference count for it.

If the destination client is in fact the object's server, then special processing is required by the destination client. The remote reference counts being passed to it should, in effect, be "taken out of circulation," as what where heretofore remote references are being converted into local references. Thus, the reference counts present in the STDOBJREF are in fact decremented from the remote reference count for the IPID in question.

Some objects have a usage model such that they do not need to be pinged at all; such objects are indicated by the presence of a flag in a STDOBJREF to an interface on the object. Objects which are not pinged in fact need not be reference counted either, though it is legal (but pointless) for a client to reference count the IPIDs of such objects.

For all other objects, assuming a non-zero ping period, it is the responsibility of the holder of an interface reference on some object to ensure that pings reach the server frequently enough to prevent expiration of the object. The frequency used by a client depends on the ping period, the reliability of the channel between the client and the server, and the probability of failure (no pings getting through and possible premature garbage-collection) that the client is willing to tolerate. The ping packet and / or its reply may both request changes to the ping period. Through this mechanism, network traffic may be reduced in the face of slow links busy servers.

Delta Pinging

Without any further refinements, ping messages could be quite hefty. If machine A held 1024 remote objects (OIDs) on machine B, then it would send 16K byte ping messages. This would be annoying if the set of remote objects was relatively stable and the ping messages were the same from ping to ping.

The delta mechanism reduces the size of ping messages. It uses a ping-set interface that allows the pinging of a single set to replace the pinging of multiple OIDs.

Instead of pinging each OID, the client defines a set. Each ping contains only the set id and the list of additions and subtractions to the set. Objects that come and go within one ping period are removed from the set without ever having been added.

The pinging protocol is carried out using two methods in the (DCE) interface IObjectExporter on the Object Exporter: ComplexPing, and SimplePing. ComplexPing is used to by clients to group the set of OIDs that they must ping into UUID-tagged sets known to the server. These entire sets of OIDs can then be subsequently pinged with a single, short, call to SimplePing.

1.1.6QueryInterface

The IRemUnknown interface on the object-exporter specified object, in addition to servicing reference counting as described above also services QueryInterface calls for remote clients for IPIDs managed by that exporter. IRemUnknown::RemQueryInterface differs from IUnknown::QueryInterface in much the same way as RemAddRef and RemRelease differ from AddRef and Release, in that it is optimized for network access by being able to retrieve many interfaces at once.

1.1.7Causality ID

Each ORPC carries with it a UUID known as the causality id that connects together the chain of ORPC calls that are causally related. If an outgoing ORPC is made while servicing an incoming ORPC, the outgoing call is to have the same causality id as the incoming call. If an outgoing ORPC is made while not servicing an incoming ORPC, then a new causality id is allocated for it.

Causality ids may in theory be reused as soon as it is certain that no transitively outstanding call is still in progress which uses that call. In practice, however, in the face of transitive calls and the possibility of network failures in the middle of such call chains, it is difficult to know for certain when this occurs. Thus, pragmatically, causality ids are not reusable.

The causality id can be used by servers to understand when blocking or deferring an incoming call (supported in some COM server programming models) is very highly probable[9] to cause a deadlock, and thus should be avoided.

The causality id for maybe, idempotent, and broadcast calls must be set to null.[10] If a server makes a ORPC call while processing such a call, a new causality id must be generated as if it were a top level call.

1.2Data types and structures

This following several sections present the technical details of the COM network protocol. The notation used herein is that of the Microsoft Interface Definition Language, version 2 (MIDL). MIDL is a upwardly compatible extension to the DCE IDL language specified in [CAE RPC]. Details of the MIDL language specification are available from Microsoft Corporation.

1.2.1DCE Packet Headers

Object RPC sits entirely on top of DCE RPC. The following list describes the elements of ORPC that are specified above and beyond DCE RPC.

- The object id field of the header must contain the IPID.
- The interface id of the RPC header must contain the IID, even though it is not needed given the IPID. This allows ORPC to sit on top of DCE RPC. An unmodified DCE RPC implementation will correctly dispatch based on IID and IPID. An optimized RPC need only dispatch based on IPID.
- An IPID uniquely identifies a particular interface on a particular object on a machine. The converse is not true; a particular interface on a particular object may only be represented by multiple IPIDs. IPIDs are unique on their OXID. IPIDs may be reused, however reuse of IPIDs should be avoided.
- Datagram, maybe, and idempotent calls are all allowed in ORPC. Interface pointers may not be passed on maybe or idempotent calls.
- Datagram broadcasts are not allowed in ORPC.
- Remote COM input synchronous calls are not allowed in ORPC.
- COM asynchronous calls are synchronous RPC calls.
- COM faults are returned in the stub fault field of the DCE RPC fault packet. Any 32 bit value may be returned. Only the following value is pre-specified:

[9] This has high probability and not certainty due to some pathological cases involving network failures. Suppose A calls B calls C calls D which will call back to A, and while D is processing its call, the link from B to C goes down, causing B to try to obtain C's services through another party E, which, as D would, calls back to A. In such situations, A may receive incoming calls from both D and E. Only the call from E is actually a potential for deadlock.

[10] Though this decision is subject to review.

RPC_E_VERSION_MISMATCH

- COM will allow DCE cancel.

All interface version numbers must be 0.0.

1.2.2Object RPC Base Definitions

There are several fundamental data types and structures on which the COM network protocol is built. These are defined in the MIDL file OBASE.IDL, which is included below:

```
[
    uuid(99fcfe60-5260-101b-bbcb-00aa0021347a),
    pointer_default(unique)
]

interface ObjectRpcBaseTypes
{
#ifndef DO_NO_IMPORTS
    import "wtypes.idl";
#endif

    // the object id specifed does not exist.
    const unsigned long RPC_E_INVALID_OBJECT = 0x80010150;
    // the objects exporter specifed does not exist.
    const unsigned long RPC_E_INVALID_OXID = 0x80010151;
    // the set id specifed does not exist.
    const unsigned long RPC_E_INVALID_SET = 0x80010152;
    //
    // Marshalling constants.
    const unsigned int MSHLFLAGS_NOPING     = 4;
    const unsigned int MSHLFLAGS_SIMPLEIPID = 8;
    const unsigned int MSHLFLAGS_KEEPALIVE  = 16;
    ////////////////////////////////////////////////////////////

    typedef GUID MID;      // Machine Identifier
    typedef GUID OXID;     // Object Exporter Identifier
    typedef GUID OID;      // Object Identifer
    typedef GUID IPID;     // Interface Pointer Identifier
    typedef GUID SETID;    // Ping Set Identifier
    typedef GUID CID;      // Causality Identifier

    typedef REFGUID REFIPID;
    typedef REFGUID REFOXID;
    typedef REFGUID REFOID;

    const unsigned short COM_MAJOR_VERSION = 1;
    const unsigned short COM_MINOR_VERSION = 1;

    // Component Object Model version number
    typedef struct tagCOMVERSION
    {
       unsigned short MajorVersion;  // Major version number
       unsigned short MinorVersion;  // Minor version number
    } COMVERSION;


    // STRINGARRAYS are the return type for arrays of network addresses,
    // arrays of endpoints and arrays of both used in many ORPC interfaces

    const unsigned short NCADG_IP_UDP   = 0x08;
    const unsigned short NCACN_IP_TCP   = 0x07;
    const unsigned short NCADG_IPX      = 0x0E;
    const unsigned short NCACN_SPX      = 0x0C;
    const unsigned short NCACN_NB_NB    = 0x12;
    const unsigned short NCACN_DNET_NSP = 0x04;
    const unsigned short NCALRPC        = 0x10;
    // const unsigned short MSWMSG         = 0x01; // note: not a real tower id.

    // this is the return type for arrays of string bindings or protseqs
    // used by many ORPC interfaces
```

```
typedef struct tagSTRINGARRAY
{
   unsigned long size;   // total size of array

    // array of NULL terminated wchar_t strings with two NULLs at the end.
    // The first word of each string is the protocol ID (above) and the
    // rest of the string is the network address[endpoint].

   [size_is(size)] unsigned short awszStringArray[];
} STRINGARRAY;

// flag values for OBJREF
const unsigned long OBJREF_STANDARD  = 1;   // standard marshalled objref
const unsigned long OBJREF_HANDLER   = 2;   // handler marshalled objref
const unsigned long OBJREF_LONGSTD   = 4;   // long form objref
const unsigned long OBJREF_LONGHDLR  = 8;   // long form handler objref
const unsigned long OBJREF_CUSTOM = 16;  // custom marshalled objref

// flag values for a STDOBJREF.
// Should be an enum but DCE IDL does not support sparse enumerators.
// OXRES1 - OXRES4 are reserved for the object exporters use only,
// object importers should ignore them and not enforce MBZ.
const unsigned long SORF_NOPING  = 1;   // Pinging is not required
const unsigned long SORF_OXRES1  = 8;   // reserved for exporter
const unsigned long SORF_OXRES2  = 16;  // reserved for exporter
const unsigned long SORF_OXRES3  = 32;  // reserved for exporter
const unsigned long SORF_OXRES4  = 64;  // reserved for exporter

// Reserved flag values for a STDOBJREF.
const unsigned long SORF_FREETHREADED = 2;  // Proxy may be used on any thread

// standard object reference
typedef struct tagSTDOBJREF
{
   unsigned long  flags;     // STDOBJREF flags
   unsigned long  cRefs;     // count of references passed
   IPID          ipid;      // ipid of Interface
   OID           oid;       // oid of object with this ipid
   OXID          oxid;      // oxid of server with this oid
} STDOBJREF;

// format of a marshalled interface pointer
typedef struct tagOBJREF
{
unsigned long flags; // OBJREF flags (see above)

[switch_is(flags), switch_type(unsigned long)] union
{
    [case(OBJREF_STANDARD)]
   STDOBJREF std;     // standard objref

    [case(OBJREF_LONGSTD)] struct
    {
   STDOBJREF   std;   // standard objref
   STRINGARRAY saResAddr; // resolver address
    } longstd;

    [case(OBJREF_HANDLER)] struct
    {
   STDOBJREF std;     // standard objref
   CLSID    clsid;   // Clsid of handler
    } handler;

    [case(OBJREF_LONGHDLR)] struct
    {
   STDOBJREF   std;   // standard objref
   CLSID      clsid; // Clsid of handler (or GUID_NULL)
   STRINGARRAY saResAddr; // resolver address
    } longhdlr;

    [case(OBJREF_CUSTOM)] struct
    {
            CLSID         clsid;   // Clsid of unmarshaling code
            unsigned long size;    // size of data that follows
            [size_is(size), ref] byte *pData;
            } custom;

        } u_objref;
        } OBJREF;


        // enumeration of additional information present in the call packet.
        // Should be an enum but DCE IDL does not support sparse enumerators.

        const unsigned long INFO_NULL      = 0;  // no additional info in packet
        const unsigned long INFO_LOCAL     = 1;  // call is local to this machine
        const unsigned long INFO_RESERVED1 = 2;  // reserved for local use
        const unsigned long INFO_RESERVED2 = 4;  // reserved for local use
        const unsigned long INFO_RESERVED3 = 8;  // reserved for local use
        const unsigned long INFO_RESERVED4 = 16; // reserved for local use


        // Extension to implicit parameters.
        typedef struct tagORPC_EXTENT
        {
            GUID                    id;     // Extension identifier.
        unsigned long               size;   // Extension size.
        [size_is((size+7)&~7)] byte data[]; // Extension data.
        } ORPC_EXTENT;


        // Array of extensions.
        typedef struct tagORPC_EXTENT_ARRAY
        {
        unsigned long size;   // num extents
        [size_is((size+1)&~1,), unique] ORPC_EXTENT **extent; // extents
        } ORPC_EXTENT_ARRAY;


        // implicit 'this' pointer which is the first [in] parameter on
        // every ORPC call.
        typedef struct tagORPCTHIS
        {
        COMVERSION version;   // COM version number
        unsigned long  flags;  // INFO flags for presence of other data
        unsigned long  reserved1; // set to zero
        CID    cid;       // causality id of caller

            // Extensions.
        [unique] ORPC_EXTENT_ARRAY *extensions;
        } ORPCTHIS;


        // implicit 'that' pointer which is the first [out] parameter on
        // every ORPC call.
        typedef struct tagORPCTHAT
        {
        unsigned long  flags; // INFO flags for presence of other data

            // Extensions.
        [unique] ORPC_EXTENT_ARRAY *extensions;
        } ORPCTHAT;


        // OR information associated with each OXID.
        typedef struct tagOXID_INFO
        {
        DWORD       dwTid;     // thread id of object exporter
        DWORD       dwPid;     // process id of object exporter
        IPID        ipidRemUnknown;  // IRemUnknown IPID for object exporter
        [unique] STRINGARRAY *psa;   // protocol id's and partial string bindings
        } OXID_INFO;
    }
```

1.2.3OBJREF

An OBJREF is the data type used to represent an actual marshaled object reference. An OBJREF can either be empty or assume one of five variations, depending on the degree to which the object being marshaled uses the hook architecture (IMarshal, etc.) in the marshaling infrastructure. The OBJREF structure is a union consisting of a switch flag followed by the appropriate data.

OBJREF_STANDARD

Contains one interface of an object marshaled in standard form. The data that follows the switch flag is a STDOBJREF structure (described below).

OBJREF_HANDLER

A marshaling of an object that wishes to use handler marshaling. For example, an object wishes to be represented in client address spaces by a proxy object that caches state. In this case, the class-specific information is just a standard reference to an interface pointer that the handler (proxy object) will use to communicate with the original object. See the IStdMarshalInfo interface.

Member	Type	Semantic
std	STDOBJREF	A standard object reference used to connect to the source object.
Clsid	CLSID	The CLSID of handler to create in the destination client.

OBJREF_LONGSTD

An interface marshaled on an object in long form. Contains a standard reference, along with a set of protocol sequences and network addresses that can be used to bind to an OXID resolver that is able to resolve the OXID in the STDOBJREF. This is useful when marshaling a proxy to give to another machine (a.k.a. the "middleman" case). The marshaling machine can specify the saResAddr for the resolver on the server machine so that the unmarshaler does not need to call the marshaler (middleman) back to get this information. Further, the marshaler does not need to keep the OXID in its cache beyond the lifetime of its own references in order to satisfy requests from parties that it just gave the OBJREF to.

Member	Type	Semantic
std	STDOBJREF	A standard object reference used to connect to the source object.
SaResAddr	STRINGARRAY	The resolver address.

OBJREF_LONGHDLR

Contains the same information as the handler case as well as the object resolver address. This form is needed for the same reason OBJREF_LONGSTD is needed.

Member	Type	Semantic
std	STDOBJREF	A standard object reference used to connect to the source object.
Clsid	CLSID	The class ID of the handler.
SaResAddr	STRINGARRAY	The resolver address.

OBJREF_CUSTOM

A marshaling of an object which supports custom marshaling. The Custom format gives an object control over the representation of references to itself. For example, an immutable object might be passed by value, in which case the class-specific information would contain the object's immutable data. See the IMarshal interface.

Member	Type	Semantic
clsid	CLSID	The CLSID of the object to create in the destination client.
size	unsigned long	The size of the marshaled data provided by the source object and passed here in pData.
pData	byte*	The data bytes that should be passed to IMarshal::UnmarshalInterface on a new instance of class clsid in order to initialize it and complete the unmarshal process.

1.2.4STDOBJREF

An instance of a STDOBJREF represents a COM interface pointer that has been marshaled using the standard COM network protocol. A STDOBJREF in general can only be interpreted in the context of an outstanding ORPC, for it may contain an OXID unknown to the machine on which it is unmarshaled, and it is the only the machine which is making outstanding call which is guaranteed to be able to provide the binding information for the OXID.

The members and semantics of the STDOBJREF structure are as follows:

Member	Semantic
flags	Flag values taken from the enumeration SORFFLAGS. These are described below.
Crefs	The number of reference counts on ipid that being transferred in this marshaling.
Ipid	The IPID of the interface being marshaled.
Oid	The OID of the object to which ipid corresponds.
Oxid	The OXID of the server that owns this OID.

The various SORFLAGS values have the following meanings. The SORF_OXRESx bit flags are reserved for the object exporter's use only, and must be ignored by object importers. They need not be passed through when marshaling an interface proxy.

Flag	Value	Meaning
SORF_NOPING	1	This OID does not require pinging. Further, all interfaces on this OID, including this IPID, need not be reference counted. Pinging and reference counting on this object and its interfaces are still permitted, however, though such action is pointless.
SORF_OXRES1	8	Reserved for exporter.
SORF_OXRES2	16	Reserved for exporter.
SORF_OXRES3	32	Reserved for exporter.
SORF_OXRES4	64	Reserved for exporter.

1.2.5ORPCTHIS

In every Request PDU that is an ORPC, the body (CL case) or the stub data (CO case) which normally contains the marshaled arguments in fact begins with an instance of the ORPCTHIS structure. The marshaled arguments of the COM interface invocation follow the ORPCTHIS; thus, viewed at the DCE RPC perspective, the call has an additional first argument. The ORPCTHIS is padded with zero-bytes if necessary to achieve an overall size that is a multiple of eight bytes; thus, the remaining arguments are as a whole eight byte aligned.

As in regular calls, the causality id must be propagated. If A calls ComputePi on B, B calls Release on C (which gets converted to RemRelease), and C calls Add on A, A will see the same causality id that it called B with.

Member	Type	Semantic
version	COMVERSION	The version number of the COM protocol used to make this particular ORPC. The initial value will be 1.1. Each packet contains the sender's major and minor ORPC version numbers. The client's and server's major versions must be equal. Backward compatible changes in the protocol are indicated by higher minor version numbers. Therefore, a server's minor version must be greater than or equal to the client's. However, if the server's minor version exceeds the client's minor version, it must return the client's minor version and restrict its use of the protocol to the minor version specified by the client. A protocol version mismatch causes the RPC_E_VERSION_MISMATCH ORPC fault to be returned.
flags	unsigned long	Flag values taken from the enumeration ORPCINFOFLAGS. These are elaborated below.
reserved	unsigned long	Must be set to zero.
cid	CID	The causality id of this ORPC. See comments below.
extensions	ORPC_EXTENT_ARRAY*	The body extensions, if any, passed with this call. Body extensions are GUID-tagged blobs of data which are marshaled as an array of bytes. Extensions are always marshaled with initial eight byte alignment. Body extensions which are presently defined are described below.

The various ORPCINFOFLAGS have the following meanings.

Flag	Meaning
INFO_NULL	(Not a real flag. Merely a defined constant indicating the absence of any flag values.)
INFO_LOCAL	The destination of this call is on the same machine on which it originates. This value is never to be specified in calls which are not in fact local.
INFO_RESERVED1	If INFO_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
INFO_RESERVED2	If INFO_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
INFO_RESERVED3	If INFO_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
INFO_RESERVED4	If INFO_LOCAL is set, then reserved for local use; otherwise, reserved for future use.

Implementations may use the local and reserved flags to indicate any extra information needed for local calls. Note that if the INFO_LOCAL bit is not set and any of the other bits *are* set then the receiver should return a fault.

Comments

The cid field contains the causality id.[11] Each time a client makes a call, a new causality id is generated. If a server makes a call while processing a request from a client, the new call must have the same causality id. This allows simple servers to avoid working on more then one thing at a time (for example A calls B calls A again, meanwhile C tries to call A with a new causality id). It tells the server that he is being called because he asked someone to do something for him. There are several interesting exceptions.

- The causality id for maybe and idempotent calls must be set to CID_NULL. If a server makes a ORPC call while processing such a call, a new causality id must be generated.
- In the face of network failures, the same causality id may end up in use by two independent processes at the same time. If A calls B calls C calls D and C fails, both B and D can independently, simultaneously make calls to E with the same causality id.

The extensions field contains extensions to the channel header. Two are currently defined for Microsoft's implementation of this protocol (described below). Other implementations may define their own extensions with their own UUIDs. Implementations should skip over extensions they do not recognize or wish to support. Note that in order to force the ORPCTHIS header to be 8 byte aligned an even number of extensions must be present and the size of the extension data must be a multiple of 8.

1.2.6ORPCTHAT

In every Response PDU that is an ORPC, the body (CL case) or the stub data (CO case) which normally contains the marshaled output parameters in fact begins with an instance of the ORPCTHAT structure. The marshaled output parameters of the COM interface invocation follow the ORPCTHAT; thus, viewed at the DCE RPC perspective, the call has an additional output parameters. The ORPCTHAT is padded with zero-bytes if necessary to achieve an overall size that is a multiple of eight bytes; thus, the remaining output parameters as a whole eight byte aligned.

Member	Type	Semantic
flags	unsigned long	Flag values taken from the enumeration ORPCINFOFLAGS. These are elaborated above.
extensions	ORPC_EXTENT_ARRAY*	The body extensions, if any, returned by this call. Body extensions are GUID-tagged blobs of data which are marshaled as an array of bytes. Extensions are always marshaled with initial eight byte alignment.

1.2.7Debug Information Body Extension

This extension aids in debugging ORPC. In particular it is designed to allow single stepping over an ORPC call into the server and out of the server into the client. See << REFERENCE >> for more details. This extension is identified by the UUID {f1f19680-4d2a-11ce-a66a-0020af6e72f4}.

[11] In several early specifications the term *logical thread id* was used instead of causality id. The name was changed because the term logical thread id implies a single threaded model that is not guaranteed.

1.2.8Extended Error Info Body Extension

The extended error information body extension conveys extended error information concerning the original root cause of a error back to a caller can deal with it. It is intended that this error information is suitable for displaying information to a human being who is the user; this information is not intended to be the basis for logic decisions in a piece of client code, for doing so couples the client code to the implementation of the server. Rather, client code should act semantically only on the information returned through the interface that it invokes. See also the ISupportErrorInfo, IErrorInfo, and ICreateErrorInfo interfaces.

This extension is identified by the UUID f1f19681-4d2a-11ce-a66a-0020af6e72f4. It is only semantically useful in Response and Fault PDUs.

There are three variations of the error information. The first of these can be used in a local context, where a) the server knows it can reliably specify a path name to a file that can in fact be understood by the client, and b) the server knows the language of interest to the client. This first variation specifies the following members:

Member	Description
wszErrorString	An error string suitable for display to a human user.
wszHelpFile	A path to a help file that can give additional information concerning the error.
ulHelpContext	A help context topic within that help file.

The second variation is suitable for use in remote situations where one or the other of the requirements of the use of the first variation cannot be upheld. This second variation specifies the following members:

Member	Description
uuidErrorSemantic	A UUID signifying the semantic of the error.
ulErrorSemantic	A four-byte quantity that qualifies the error semantic. The semantics of these four bytes are completely determined by the uuidErrorSemantic.
cbData	The size of the data passed in pbData.
pbData	Data associated with the error marshaled as an array of bytes. The interpretation of these bytes is governed by the uuidErrorSemantic.
wszErrorString	An error string suitable for display to a human user. This is in fact of somewhat little use, as the server returning the error can usually only guess as to the appropriate language in which to form this string. However, the ability to pass such as string as a last resort is provided here.
lcidErrorString	The locale context in which the error string, if any, is formed. Locale constants are as in the Microsoft Win32 API.

The third variation allows for extensibility of the error information being passed. It specifies an object reference (an OBJREF). In practice, this reference most always contains a custom marshaled object, though this is not required.

1.3IRemUnknown interface

The IRemUnknown interface is used by remote clients for manipulating reference counts on the IPIDs that they hold and for obtaining additional interfaces on the objects on which those IPIDs are found. This interface is implemented by the COM “OXID object” associated with each OXID (nb. not each Object Exporter). The IPID for the IRemUnknown interface on this object is returned from IObjectExporter::-ResolveOxid; see below. An OXID object need never be pinged; its interfaces (this IPID included) need never be reference counted.

IRemUnknown is specified as follows (REMUNK.IDL):

```
//+-------------------------------------------------------------------------
//
//  Microsoft Windows
//  Copyright (C) Microsoft Corporation, 1992 - 1995.
//
//  File:   remunk.idl
//
//  The remote version of IUnknown.  Once instance of this interface exists
//  per OXID (whether an OXID represents either a thread or a process is
//  implementation specific). This interface is passed along during OXID
//  resolution.  It is used by clients to query for new interfaces, get
//  additional references (for marshalling), and release outstanding
//  references.
//
//+-------------------------------------------------------------------------
[
    object,
    uuid(99fcff28-5260-101b-bbcb-00aa0021347a)
]

import "unknwn.idl";

interface IRemUnknown : IUnknown
{
    // return structure from a QI call
    typedef struct tagREMQIRESULT
    {
        HRESULT    hResult;   // result of call
        STDOBJREF  std;       // data for returned interface
    } REMQIRESULT;

    HRESULT RemQueryInterface
    (
        [in] IPID              ipid,  // interface to QI on
        [in] unsigned long cRefs,  // count of AddRefs requested for each interface
        [in] unsigned short    cIids, // count of IIDs that follow
        [in, size_is(cIids)] IID *iids, // IIDs to query for
        [out, size_is(,cIids)] REMQIRESULT **ppQIResults // results returned
    );


    // structure passed to AddRef/Release to specify interface and count of
    // references to Add/Release.
    typedef struct tagREMINTERFACEREF
    {
        IPID       ipid;      // ipid to AddRef/Release
        unsigned long  cRefs;     // number of refs to add/release
    } REMINTERFACEREF;

    HRESULT RemAddRef
    (
        [in] unsigned short cInterfaceRefs,
        [in, size_is(cInterfaceRefs)] REMINTERFACEREF InterfaceRefs[]
    );

    HRESULT RemRelease
    (
        [in] unsigned short cInterfaceRefs,
        [in, size_is(cInterfaceRefs)] REMINTERFACEREF InterfaceRefs[]
    );
}
```

Comments

References are kept per interface rather then per object.

IRemUnknown::RemQueryInterface

HRESULT IRemUnknown::RemQueryInterface(ipid, cIids, cRefs, iids, ppQIResults)

QueryInterface for and return the result thereof for zero or more interfaces from the interface behind the IPID ipid. ipid must designate an interface derived from IUnknown (recall that all remoted interfaces must derive from IUnknown). The QueryInterface calls on the object that are used to service this request are conducted on this interface instance, not any other IUnknown instance that the object may happen to have. Thus if the client calls IFoo->QueryInterface rather then pIUnknown->QueryInterface, RemQueryInterface will also call QueryInterface on IFoo.

Argument	Type	Description
ipid	IPID	The interface on an object from whom more interfaces are desired.
cRefs	REFCNT	The number of references sought on each of the returned IIDs.
cIids	USHORT	The number of interfaces being requested.
iids	IID*	The list of IIDs that name the interfaces sought on this object.
ppQIResults	REMQIRESULT**	The place at which the array of the results of the various QueryInterface calls are returned.

Return Value	Meaning
S_OK	Success. An attempt was made to retrieve each of the requested interfaces from the indicated object; that is, QueryInterface was actually invoked for each IID.
E_INVALIDARG	One or more arguments (likely ipid) were invalid. No result values are returned.
E_UNEXPECTED	An unspecified error occurred. No result values are returned.

The REMQIRESULT structure contains the following members:

Member	Type	Semantic
hResult	HRESULT	The result code from the QueryInterface call made for the requested IID.
std	STDOBJREF	The data for the returned interface. Note that if hResult indicates failure then the contents of STDOBJREF are undefined.

IRemUnknown::RemAddRef

Obtain and grant ownership to the caller of one or more reference counts on one or more IPIDs managed by the corresponding OXID.

HRESULT IRemUnknown::RemAddRef(cInterfaceRefs, rgRefs)

Argument	Type	Description
cInterfaceRefs	unsigned short	The size of the rgRefs array.
rgRefs	REMINTERFACEREF	An array of IPID, cRefs pairs, cInterfaceRefs large. Each IPID indicates an interface managed by this OXID on whom more reference counts are sought. The corresponding reference count (cRefs), which may not be zero (and thus is one or more), indicates the number of reference counts sought on that IPID.

Return Value	Meaning
S_OK	Success. An attempt was made to retrieve each of the requested interfaces.
E_INVALIDARG	One or more of the IPIDs indicated were not in fact managed by this OXID, or one or more of the requested reference counts was zero. **None** of the requested reference counts have been granted to the caller; the call is a no-op.
E_UNEXPECTED	An unspecified error occurred. It is unknown whether any or all of the requested reference counts have been granted.

Comments

A useful optimization is for a caller to RemAddRef more than needed. When a process receives an out marshaled interface, it receives one reference count. If the process wishes to pass that interface as an out parameter, it must get another reference to pass along. Instead, the process (or middleman) should get a large number of references. Then if the interface is passed out multiple times, no new remote calls are needed to gain additional references.

A marshaler may optionally specify more than one reference in the STDOBJREF when marshaling an interface. This allows the middle man case to pre-fill its cache of references without making an extra RemAddRef call. The number of references passed is always specified in the STDOBJREF field.

IRemUnknown::RemRelease

HRESULT IRemUnknown::RemRelease(cInterfaceRefs, rgRefs)

Argument	Type	Description
cInterfaceRefs	USHORT	The size of the rgRefs array.
rgRefs	REMINTERFACEREF	An array of IPID, cRefs pairs, cInterfaceRefs large. Each IPID indicates an interface managed by this OXID on whom more reference counts are being returned. The corresponding reference count, which may not be zero (and thus is one or more), indicates the number of reference counts returned on that IPID.

Return Value	Meaning
S_OK	Success. An attempt was made to retrieve each of the requested interfaces.
E_INVALIDARG	One or more of the IPIDs indicated were not in fact managed by this OXID, or one or more of the requested reference counts was zero. **None** of the offered reference counts have been accepted by the server; the call is a no-op.
E_UNEXPECTED	An unspecified error occurred. It is unknown whether any or all of the offered reference counts have been accepted.

1.4The Object Exporter

Each machine that supports the COM network protocol supports a one-per-machine service known as the

Release ownership of one or more reference counts on one or more IPIDs managed by the corresponding OXID.

machine's 'Object Exporter.' Communication with an Object Exporter is via a DCE RPC, not an ORPC. To ensure connectivity, the Object Exporter resides at well-known endpoints. It is proposed that the Object Exporter either (1) make use of the same endpoints allocated for the DCE RPC Endpoint Mapper (listed below)[12], implying typically that these services are written within the same process on a given machine, or alternately and less preferably (2) that the Object Exporter reside at a different set of well-known endpoints TBD.

The Object Exporter performs several services:

- It caches and returns to clients when asked the string bindings necessary to connect to OXIDs of exported objects for which this machine is it either itself a client or is the server.
- It receives pings from remote client machines to keep its own objects alive.

These services are carried out through an RPC interface (not a COM interface) known as IObjectExporter.

An Object Exporter may be asked for the information required to connect to one of two different kinds of OXIDs, either the OXIDs associated with its own objects, or the OXIDs associated with objects for which it itself is a client, and which it has passed on to a second client machine. This second case, where one marshals an object from one client machine to a second, is colloquially referred to the "middleman" case. In the middleman case, the exporter is required to retain the connection information associated with the OXIDs that it passes on until it is certain that that the second client machine no longer needs them. More on this below.

[12] See [CAE RPC], Appendix H, p 613.

Protocol String Name(s)	Description	Object Exporter End Point	Endpoint Mapper End Point
ncadg_ip_udp, ip	CL over UDP/IP	TBD	135
ncacn_ip_tcp	CO over TCP/IP	TBD	135
ncadg_ipx	CL over IPX	TBD	not yet listed
ncacn_spx	CO over SPX	TBD	not yet listed
ncacn_nb_nb	CO over NetBIOS over NetBEUI	TBD	not yet listed
ncacn_nb_tcp	CO over NetBIOS over TCP/IP	TBD	135
ncacn_np	CO over Named Pipes	TBD	not yet listed
ncacn_dnet_nsp	CO over DECNet Network Services Protocol (DECnet Phase IV)	TBD	69
ncacn_osi_dna	CO over Open Systems Interconnection (DECNet Phase V)	TBD	69
ncadg_dds, dds	CL over Domain Datagram Service	TBD	12
ncalrpc	Local procedure call	N/A	N/A

Table 1. Object Exporter Well-known Endpoints

IObjectExporter interface is defined as follows (OBJEX.IDL):

```
//+-----------------------------------------------------------------------
//
//  Microsoft Windows
//  Copyright (C) Microsoft Corporation, 1992 - 1995.
//
//  File:      objex.idl
//
//  Synopsis:      Interface implemented by object exporters.
//
//  This is the interface that needs to be supported by hosts that export
//  objects. Only one instance of this interface can be exported by the host.
//
//  An object exporter needs to be able to:
//  1. return string bindings that can be used to talk to objects it
//     has exported
//  2. receive pings from object importers to keep the objects alive
//
//------------------------------------------------------------------------
[
    uuid(99fcfec4-5260-101b-bbcb-00aa0021347a),
    pointer_default(unique)
]

interface IObjectExporter
{
    import "obase.idl";

    // Method to get the protocol sequences, string bindings and machine id
    // for an object server given its OXID.

    [idempotent] error_status_t ResolveOxid
    (
    [in]       handle_t       hRpc,
    [in]       OXID           *pOxid,
    [in]       unsigned short cRequestedProtseqs,
    [in,  ref, size_is(cRequestedProtseqs)]
               unsigned short arRequestedProtseqs[],
    [out, ref] MID            *pmid,
    [out, ref] STRINGARRAY   **psaOxidBindings,
    [out, ref] IPID           *pipidRemUnknown
    );

    // Simple ping is used to ping a Set. Client machines use this to inform
    // the object exporter that it is still using the items inside the set.
    // Returns S_TRUE if the SetId is known by the object exporter,
    // S_FALSE if not.

    [idempotent] error_status_t SimplePing
    (
    [in]  handle_t  hRpc,
    [in]  SETID    *pSetId
```

```
        );

        // Complex ping is used to create sets of OIDs to ping. The whole set
        // can subsequently be pinged using SimplePing, thus reducing network
        // traffic.

        [idempotent] error_status_t ComplexPing
        (
        [in]  handle_t        hRpc,
        [in]  SETID          *pSetId,
        [in]  unsigned short  SequenceNum,
        [in]  unsigned short  SetPingPeriod,
        [in]  unsigned short  SetNumPingsToTimeout,
        [out] unsigned short *pReqSetPingPeriod,
        [out] unsigned short *pReqSetNumPingsToTimeout,
        [in]  unsigned short  cAddToSet,
        [in]  unsigned short  cDelFromSet,
        [in,  unique, size_is(cAddToSet)]   GUID AddToSet[],   // add these OIDs to the set
        [in,  unique, size_is(cDelFromSet)] GUID DelFromSet[]  // remove these OIDs from the set
        );
    }
```

IObjectExporter::ResolveOxid

```
    [idempotent] error_status_t ResolveOxid
        (
        [in]       handle_t        hRpc,
        [in]       OXID           *pOxid,
        [in]       unsigned short  cRequestedProtseqs,
        [in,  ref, size_is(cRequestedProtseqs)]
                   unsigned short  arRequestedProtseqs[],
        [out, ref] MID            *pmid,
        [out, ref] STRINGARRAY   **psaOxidBindings,
        [out, ref] IPID           *pipidRemUnknown
        );
```

Return the string bindings necessary to connect to a given OXID object.

On entry, arRequestedProtseqs contains the protocol sequences the client is willing to use to reach the server. These should be decreasing order of protocol preference, with no duplicates permitted. Local protocols (such as “ncalrpc”) are not permitted.

On exit, psaOxidBindings contains the string bindings that may be used to connect to the indicated OXID; if no such protocol bindings exist which match the requested protocol sequences, NULL may be returned. The returned string bindings are in decreasing order of preference of the server, with duplicate string bindings permitted (and not necessarily of the same preferential priority), though of course duplicates are of no utility. Local protocol sequences may not be present; however, protocol sequences that were not in the set of protocol sequences requested by the client may be. The string bindings returned need not contain endpoints; the endpoint mapper will be used as usual to obtain these dynamically.

If a ResolveOxid call is received for which the recipient Object Exporter is a middleman, the action required of the middleman depends on how the ordered list of requested protocol sequences (arRequested-Protseqs) relate to lists of protocol sequences previously known by the middleman to have been previously requested of the server. If the list of requested protocol sequences is a (perhaps non-proper) subset in order of a protocol sequence list previously requested of the server, then the corresponding cached string bindings may be returned immediately to the caller without actually communicating with the server. Otherwise, the actual psaRequestedProtseqs must be forwarded to the server, and the returned string bindings propagated back to the client. In such cases, it behooves the middleman to cache the returned string bindings for use in later calls.

In order to support the middleman case, Object Exporters are required to remember the OXID mapping information for remote OXIDs they have learned for some period of time beyond when they themselves have released all references to objects of this OXID. Let *t* be the full time-out period (ping period × number of pings to time-out) for some OID (any OID) for which this Object Exporters is a client and which resides in OXID. Then the Object Exporter must keep the binding information for OXID for at least an amount of time *t* following the release of a the local reference to any object in OXID.[13]

[13] This time-out period does not *guarantee* that OXID information will not be discarded before clients may need it, but is a very good heuristic, and indeed better than a hard-coded time-out value.

Returned through pmid is an identifier that uniquely identifies the machine. Clients can use this to learn which OXIDs are collocated on the same machine, and thus which OIDs may be appropriately grouped together in ping sets (see ComplexPing). The machine identifier is guaranteed not to change so long as there are remote references to objects on the machine which remain valid. Thus, specifically, the machine id may change as the machine reboots.

ResolveOxid also informs the caller of the IPID of the OXID object associated with this OXID.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
pOxid	OXID*	The OXID for whom string bindings are requested. . The OXID may or may not represent a process on the machine that receives the ResolveOxid call
cRequestedProtseqs	unsigned short	The number of protocol sequences requested.
arRequestedProtseqs	unsigned short[]	arRequestedProtseqs must be initialized with all the protocol id's the client is willing to use to reach the server. It cannot contain local protocol sequences. The object exporter must take care of local lookups privately. The protocol sequences are in order of preference or random order. No duplicates are allowed. See the Lazy Use Protseq section for more details.
pmid	MID*	The machine identifier associated with OXID.
psaOxidBindings	STRINGARRAY**	The string bindings supported by this OXID, in preferential order. Note that these are Unicode strings.
pipidRemUnknown	IPID*	The IPID to the IRemUnknown interface the OXID object for this OXID.

Return Value	Meaning
S_OK	Success. The requested information was returned.
RPC_E_INVALID_OBJECT	
RPC_E_INVALID_OXID	This OXID is unknown to this Object Exporter, and thus no information was returned.
RPC_E_SERVER_DIED	
E_OUTOFMEMORY	
E_UNEXPECTED	An unspecified error occurred. Some of the requested information may not be returned.

Comments

Since the object exporter ages string bindings and discards them, object references are transient things. They are not meant to be stored in files or otherwise kept persistently. The preferred method of storing persistent references will depend on the activation models available. On platforms that support them, monikers should be used for any persistent reference. In any case, well known object references can be constructed from well known string bindings, IPIDs and OIDs.

Conversely, since object references are aged, it is the responsibility of each client to unmarshal them and begin pinging them in a timely fashion.

The basic use of the ResolveOxid method is to translate an OXID to string bindings. Put another way, this method translates an opaque process and machine identifier to the information needed to reach that machine and process. There are four interesting cases: looking up an OXID the first time an interface is unmarshaled on a machine, looking up an OXID between a pair of machines that already have connections, looking up an OXID from a middleman, and looking up string bindings with unresolved endpoints (lazy use protseq). Another interesting topic is garbage collection of stored string binding vectors.

Lookup Between Friends

OX C	Process D		OX E	Process F	Process G
	call F				
				pass out ref to G	

	receive out ref to G				
	ask local OX to resolve OXID G				
ask OX E to resolve OXID G					
			look up G and return G's endpoints.		
Construct and cache string binding vector for G					
return results to D					
	ready to call G directly				

The case of a lookup between two machines that have already established communication is the easiest. In this scenario there are two machines, A and B. Process D already has an interface pointer to process F. Object exporter C already knows the string bindings for object exporter E and process F, but not process G. Object exporter E knows the string bindings for all the servers on its machine, i.e. processes F and G. Process D calls process F and gets a reference to process G. Since process D has never seen the OXID for G before, it asks its local object exporter to resolve G. Process D also has to tell object exporter C where it got the reference from, in this case, process F. Object exporter C does not recognize the OXID G. However it does recognize the OXID F and knows the object exporter E is on the same machine as process F. So OX C calls ResolveOxid on OX E. OX E recognizes G and passes the string bindings back to OX C with the machine id B. OX C caches this information so that if D ever gets a reference from G, it knows who to ask to resolve that reference.

My First Lookup

The previous example assumes that OX C already knows about OX E and process D is already talking to process F. Setting up the first connection between D and F (as well as C and E) is a tricky business known as activation. ORPC as described in this specification does not include activation models. Thus different vendors may have different activation models. However there is one basic form of activation shared by all ORPC. If two processes can communicate via DCE RPC, they can pass long standard object references. While this is not expected to be a common form of activation, it is a simple one that should certainly work across all ORPC implementations. Thus if D and E have established DCE RPC (or raw RPC) communication, they can bootstrap ORPC communication as follows.

OX C	Process D		OX E	Process F
			register endpoints and OXID for F	
	call F with raw RPC			
				pass an out ref to F
				pass IID as additional parameter
	tell C the OXID_INFO and MID for F. Include network address(es).			
compute the string bindings for OX E from F				
ask E to resolve F to get machine id and endpoints for F				

			return endpoints and machine id	
	ready to call F via ORPC			

This example points out that there has to be a local interface between processes and the local object exporter.

Middleman Lookup

The next case shows how lookup works between multiple machines. Suppose that E has a reference to G and G has a reference to I. Similarly, D knows about F and G and F knows about H and I. What happens if G passes a reference to I over to E?

OX D	Process E	OX F	Process G	OX H	Process I
	call G				
			return a long reference to I		
	ask D to lookup I				
Since it's a long ref to I call ResolveOxid on H.					
				Return endpoints to I	
Compute string bindings to I from endpoints and network addresses.					
return string bindings to E					
	ready to call I				

Note that when process G returned a reference to I, it used he long form of the OBJREF which includes the protocol id's and network addresses of the OXID resolver for process I (in this example, the addresses for OX H). This would results in OX D calling OX H directly, rather than needing to call OX F. The advantage of this is that if no references to process I needed by OX F, it could remove it from its OXID cache at any time, rather than keeping it around at least until OX D has had a chance to call it back to resolve OXID I.

Lazy Use Protseq

In a homogeneous network, all machines communication via the same protocol sequence. In a heterogeneous network, machines may support multiple protocol sequences. Since it is often expensive in resources to allocate endpoints (RpcServerUseProtseq) for all available protocol sequences, ORPC provides a mechanism where they may be allocated on demand. To implement this extension fully, there are some changes in the server. However, changes are optional. If not implemented, ORPC will still work correctly if less optimally in heterogeneous networks.

There are two cases: the server implements the lazy use protocol or it does not.

If the server is using the lazy use protseq protocol, the use of ResolveOxid is modified slightly. When the client OX calls the server OX, it passes the requested protseq vector. If none of the requested protseqs have endpoints allocated in the server, the server OX performs some local magic to get one allocated.

If the server does not implement the lazy use protseq protocol, then all protseqs are registered by the server and contain complete endpoints. However, if they are not, the endpoint mapper can be used to forward calls to the server. This requires that all server IIDs are registered in the endpoint mapper. It also allows a different lazy use protseq mechanism. The endpoint mapper can perform some local magic to force the server to allocate an endpoint. This is less efficient since no OXs ever learn the new endpoints.

The client will always pass in a vector of requested protseqs which the server can ignore if it does not implement the lazy use protseq protocol.

Aging String Bindings

Each object exporter must keep all the string bindings for references to remote machines as well as string bindings for all processes that are ORPC servers on its machines. However, unless the middle man marshaler always marshals proxy interfaces using the long form OBJREF, string bindings cannot be discarded as soon as remote references are. In the middleman example above, a process could pass out a reference to a remote object and immediately release any remaining references to that remote object. When the poor client called back to translate the OXID, the string bindings would be gone. To deal with that case, the object exporter must keep the string binding/OXID translation for one full time-out period (round up if the release occurs in the middle of a ping period) after the last local reference is released. A time-out period is the number of pings times the ping period.

IObjectExporter::SimplePing

```
[idempotent] error_status_t SimplePing
    (
    [in]  handle_t  hRpc,
    [in]  SETID    *pSetId
    );
```

Pings provide a mechanism to garbage collect interfaces. If an interface has references but is not being pinged, it may be released. Conversely, if an interface has no references, it may be released even though it has recently been pinged. SimplePing just pings the contents of a set. The set must be created with ComplexPing (see below).

Ping a set, previously created with IObjectExporter::ComplexPing, of OIDs owned by this Object Exporter. Note that neither IPIDs nor OIDs may be pinged, only explicitly created SETIDs.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
pSetId	SETID*	A SETID previously created with IObjectExporter::ComplexPing on this same Object Exporter.

Return Value	Meaning
S_OK	Success. The set was pinged.
RPC_E_INVALID_SET	This SETID is unknown to this Object Exporter, and thus the ping did not occur.
E_UNEXPECTED	An unspecified error occurred. It is not known whether the ping was done or not.

IObjectExporter::ComplexPing

```
[idempotent] error_status_t ComplexPing
    (
    [in]  handle_t        hRpc,
    [in]  SETID          *pSetId,
    [in]  unsigned short  SequenceNum,
    [in]  unsigned short  SetPingPeriod,
    [in]  unsigned short  SetNumPingsToTimeout,
    [out] unsigned short *pReqSetPingPeriod,
    [out] unsigned short *pReqSetNumPingsToTimeout,
    [in]  unsigned short  cAddToSet,
    [in]  unsigned short  cDelFromSet,
    [in,  unique, size_is(cAddToSet)]   GUID AddToSet[],
    [in,  unique, size_is(cDelFromSet)] GUID DelFromSet[]
    );
```

Ping a ping set. Optionally, add and / or remove some OIDs from the set. Optionally, adjust the ping timing parameters associated with the set. After a set is defined, a SimplePing will mark the entire contents of the set as active. After a set is defined, SimplePing should be used to ping the set. ComplexPing need only be used to adjust the contents of the set (or the time-out).

Ping set ids (SETIDs) are allocated unilaterally by a client Object Exporter. The client Object Exporter then communicates with the server Object Exporter to add (and later remove) OIDs from the ping set. Clients must ensure the SETIDs pinged at a given server are unique over all of that server's clients. Thus, the client must only use SETIDs that it knows not to be in use as SETIDs by other clients on that server. (In practice, clients allocate SETIDs as globally unique). A client may use as many sets as it likes, though using fewer sets is more efficient.

Each OID owned by a server Object Exporter may be placed in zero or more ping sets by the various clients of the OID. The client owner of each such set will set a ping period and a ping time-out count for the set, thus determining an overall time-out period for the set as the product of these two values. The time-out period is implicitly applied to each OID contained in the set and to future OIDs that might add be added to it. The server Object Exporter is responsible for ensuring that an OID that it owns does not expire until at least a period of time *t* has elapsed without that OID being pinged, where *t* is the maximum time-out period over all the sets which presently contain the given OID, or, if OID is not presently in any such sets but was previously, *t* is the time-out period for the last set from which OID was removed at the instant that that removal was done;[14] otherwise, OID has never been in a set, and *t* is a default value (TBD).

Clients are responsible for pinging servers often enough to ensure that they do not expire given the possibility of network delays, lost packets, and so on. If a client only requires access to a given object for what it would consider less than a time-out period for the object (that is, it receives and release the object in that period of time), then unless it is certain it has not itself passed the object to another client it must be sure to nevertheless ping the object (a ComplexPing that both adds and removes the OID will suffice). This ensures that an object will not expire as it is passed through a chain of calls from one client to another.

An OID is said to be pinged when a set into which it was previously added and presently still resides is pinged with either a SimplePing or a ComplexPing, or when it is newly added to a set with ComplexPing. Note that these rules imply that a ComplexPing that removes an OID from a set still counts as a ping on that OID.

In addition to pinging the set SETID, this call sets the time-out period of the set as the product of a newly-specified ping period and a newly-specified "ping count to expiration;" these values take effect immediately. Ping periods are specified in tenths of a second, yielding a maximum allowable ping period of about 1 hr 50 min. Adjustment of the time-out period of the set is considered to happen before the addition of any new OIDs to the set, which is in turn considered to happen before the removal of any OIDs from the set. Thus, an OID that is added and removed in a single call no longer resides in the set, but is considered to have been pinged, and will have as its time-out at least the time-out period specified in that ComplexPing call.

On exit, the server may request that the client adjust the time-out period; that is, ask it to specify a different time-out period in subsequent calls to ComplexPing. This capability can be used to reduce traffic in busy servers or over slow links. The server indicates its desire through the values it returns through the variables pReqSetPingPeriod and pReqSetNumPingsToTimeOut. If the server seeks no change, it simply returns the corresponding values passed by the client; if it wishes a longer time-out period, it indicates larger values for one or both of these variables; if it wishes a smaller period, it indicates smaller values. When indicating a larger value, the server must start immediately acting on that larger value by adjusting the time-out period of the set. However, when indicating a smaller value, it must consider its request as purely advice to the client, and not take any action: if the client wishes to oblige, it will do so in a subsequent call to ComplexPing by specifying an appropriate time-out period.

[14] That is, adjusting the set's time-out period after the OID has been removed from it has no effect on the time-out of the OID.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
pSetId	SETID	The SETID being manipulated.
SequenceNum	USHORT	The sequence number allows the object exporter to detect duplicate packets. Since the call is idempotent, it is possible for duplicates to get executed and for calls to arrive out of order when one ping is delayed.
SetPingPeriod	USHORT	**This parameter is going away.**
SetNumPingsToTimeOut	USHORT	**This parameter is going away**.
pReqSetPingPeriod	USHORT*	**This parameter is changing**. The server uses pReqSetPingPeriod to request a new ping period. If the requested period is shorter then the current period, the server must continue to use the current period until the client calls back. When the client calls back the old requested period will only be used if the client specifies it as the new SetPingPeriod. If the requested period is longer then the current period, the server must immediately begin using the new period. However, if the client doesn't accept it, the next call will contain the old, shorter period.
pReqSetNumPingsToTimeOut	USHORT*	**This parameter is changing**. The server uses pReqSetNumPingsToTimeout to request a new number of pings. If the number of pings is less then the current number of pings, the server must continue to use the current number of pings until the client calls back. When the client calls back the old requested period will only be used if the client specifies it as the new SetNumPingsToTimeout. If the requested number of pings is larger then the current number of pings, the server must immediately being using the new number of pings. However, if the client doesn't accept it, the next call will contain the old, smaller number of pings.
cAddToSet	USHORT	The size of the array AddToSet.
cDelFromSet	USHORT	The size of the array DelFromSet.
AddToSet	OID[]	The list of OIDs which are to be added to this set. Adding an OID to a set in which it already exists is permitted; such an action, as would be expected, is considered to ping the OID.
DelFromSet	OID[]	The list of OIDs which are to be removed from this set. Removal counts as a ping. An IPID removed from a set will expire after the number of ping periods has expired without any pings (not the number of ping periods - 1). If an id is added and removed from a set in the same ComplexPing, the id is considered to have been deleted.

Return Value	Meaning
S_OK	Success. The set was pinged, etc.
RPC_E_INVALID_OBJECT	Indicates that some OID was not recognized. There is no recovery action for this error, it is informational only.
RPC_E_ACCESS_DENIED	
RPC_E_OUT_OF_ORDER	Returned when a call is received with a sequence number which is less then the last sequence number executed successfully.
E_OUTOFMEMORY	There was not enough memory to service the call. The caller may retry adding OIDs to the set on the next ping.
E_UNEXPECTED	An unspecified error occurred. It is not known whether the ping or any of the other actions were done or not.
Security related errors	TBD

1.5Service Control Manager

The Service Control Manager (SCM) is the component of the COM Library responsible for locating class implementations and running them. The SCM ensures that when a client request is made, the appropriate server is connected and ready to receive the request. The SCM keeps a database of class information based on the system registry that the client caches locally through the COM library.

Machines in a COM environment which support the ability to instantiate objects on behalf of a remote client offer SCM services remotely via an ORPC interface.[15] To ensure connectivity, such SCM services

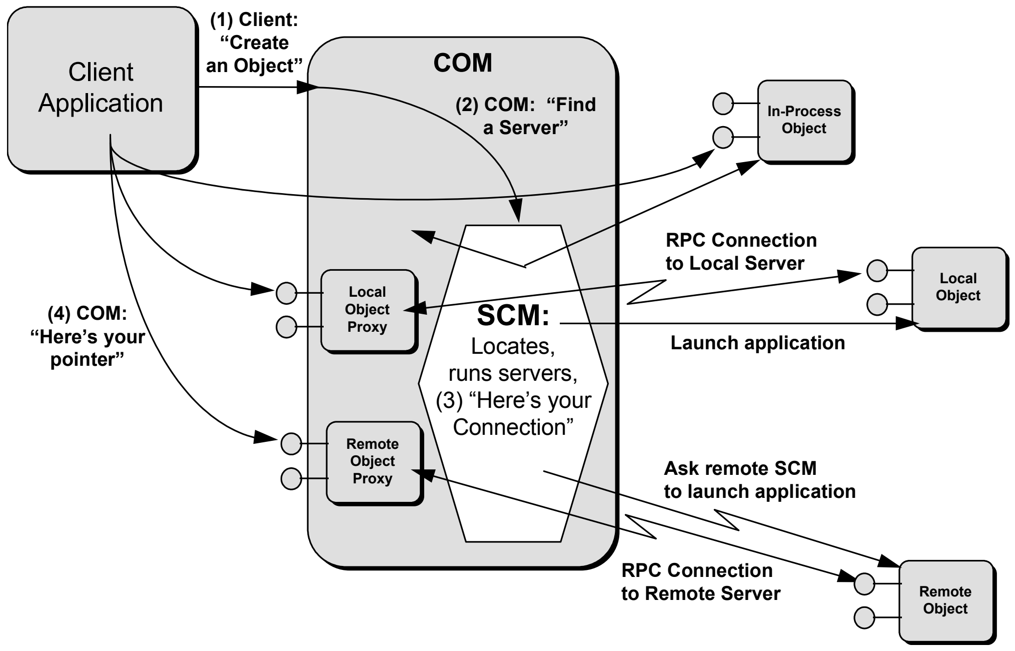
Client Application
(1) Client: "Create an Object"
COM
(2) COM: "Find a Server"
In-Process Object
RPC Connection to Local Server
Local Object
Local Object Proxy
SCM: Locates, runs servers, (3) "Here's your Connection"
Launch application
(4) COM: "Here's your pointer"
Remote Object Proxy
Ask remote SCM to launch application
RPC Connection to Remote Server
Remote Object

Figure 15-1: COM delegates responsibility of loading and launching servers to the SCM.

reside at the same well-known endpoints as the COM Object Exporter[16] on each machine. Note that unlike the Object Exporter service, which is required for a machine to expose COM objects remotely, the exposed SCM service is in fact *optional* and some machines may not offer it. Clients may receive references to existing objects on such a machine or cause objects to be instantiated on that machine through means besides the services offered by the SCM, such as a through a moniker binding mechanism.

These capabilities are the basis for COM's implementation locator services as outlined in Figure 15-1.

When a client makes a request to create an object of a CLSID, the COM Library contacts the local SCM (the one on the same machine) and requests that the appropriate server be located or launched, and a class factory returned to the COM Library. After that, the COM Library, or the client, can ask the class factory to create an object.

The actions taken by the local SCM depend on the type of object server that is registered for the CLSID:

In-Process	The SCM returns the file path of the DLL containing the object server implementation. The COM library then loads the DLL and asks it for its class factory interface pointer.
Local	The SCM starts the local executable which registers a class factory on startup. That pointer is then available to COM.
Remote	The local SCM contacts the SCM running on the appropriate remote machine and forwards the request to the remote SCM. The remote SCM launches the server which registers a class factory like the local

[15] Not surprisingly, to get around the chicken and egg problem, getting to a remote machine's SCM interface is not done via normal CoCreateInstance means. Richer mechanisms for creating an interface to a remote SCM are TBD, but note that the clarity about the SCM-to-SCM interface and its endpoints ensure interconnectivity.

[16] This is true despite final word whether the Object Exporter will reuse the DCE RPC Endpoint Mapper's endpoints or a different well-known set TBD, as noted above.

server with COM on that remote machine. The remote SCM then maintains a connection to that class factory and returns an RPC connection to the local SCM which corresponds to that remote class factory. The local SCM then returns that connection to COM which creates a class factory proxy which will internally forward requests to the remote SCM via the RPC connection and thus on to the remote server.

Note that if the remote SCM determines that the remote server is actually an in-process server, it launches a "surrogate" server that then loads that in-process server. The surrogate does nothing more than pass all requests on through to the loaded DLL.

ISCMSCM interface is defined as follows (SCMSCM.IDL):

```
//+------------------------------------------------------------------------
//
//  Microsoft Windows
//  Copyright (C) Microsoft Corporation, 1992 - 1995.
//
//  File:      scmscm.idl
//
//  Synopsis:      Interface for SCM to SCM communication.
//
//  This is the interface that needs to be supported by hosts that allow
//  activation of objects. Only one instance of this interface can be exported
//  by the host.
//
//-------------------------------------------------------------------------
[
  uuid(00000137-0000-0000-C000-000000000046),
  version(1.0),
  pointer_default(unique)
]

interface ISCMtoSCM
{
    HRESULT ActivationRequest(
         [in] handle_t                          hRpc,
         [in] ORPCTHIS *                        orpcthis,
         [out] ORPCTHAT *                       orpcthat,
         [in] const GUID *                      rclsid,
         [in, string, unique] WCHAR *           pwszObjectName,
         [in] DWORD                             clsctx,
         [in] DWORD                             grfMode,
         [in] DWORD                             dwCount,
         [in,unique,size_is(Interfaces)] IID *  pIIDs,
         [out,size_is(Interfaces)] OBJREF **    ppInterfaces,
         [out,size_is(Interfaces)] HRESULT *    pResults
         );
}
```

ISCMtoSCM::ActivationRequest

HRESULT ISCMtoSCM::ActivationRequest(hRpc, orpcthis, orpcthat, rclsid, pwszObjectName, clsctx, grfMode, dwCount, pIIDs, ppInterfaces, pResults);

This single method encapsulates several different forms of activating and instantiating objects on the machine of this SCM.

When pIIDs is NULL, this function behaves roughly as CoGetClassObject(rclsid, clsctx, NULL, IID_IClassFactory, ...), returning a class-object to the caller.

When pwszObjectName and pObjectStorage are NULL, this function behaves roughly as CoCreateInstanceEx(rclsid, NULL, clsctx, ...).

Otherwise, this function acts roughly similar to CreateFileMoniker(pwszObjectName, ...) followed by IMoniker::BindToObject(...) to retrieve the required interfaces.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
orpcthis	ORPCTHIS*	ORPCTHIS identifying this object.
orpcthat	ORPCTHAT*	ORPCTHAT holding return values.
rclsid	CLSID*	Identifies the class to be run to service the request.
pwszObjectName	WCHAR*	Identifies the persistent representation of the object to this machine. Typically, this is a file name which is used to determine the class, as in CreateFileMoniker(pwszFileName, &pmk) followed by BindMoniker(pmk…).
clsctx	DWORD	Values taken from the CLSCTX enumeration.
grfMode	DWORD	Values taken from the STGM enumeration.
dwCount	DWORD	The number of interfaces to return.
pIIDs	IID*	An array of interfaces to QueryInterface for on the new object.
ppInterfaces	OBJREF**	Location to return an array of interfaces on the object.
pResults	HRESULT*	Location to return an array of return codes about the successful retrieval of each of the dwCount interfaces.

Return Value	Meaning
S_OK	Success.
CO_S_NOTALLINTERFACES	Some but not all of the dwCount interfaces were returned in ppInterfaceData. Examine pResults to identify exactly which interf

1.6Wrapping DCE RPC calls to interoperate with ORPC

This is an example of a server side wrapper for the Bar method. It assumes the existence of small helper functions to import and export object references and lookup previously exported object references.

```
RPC_STATUS Bar(handle_t h, short i, OBJREF * prIB, OBJREF ** pprIW) {
            UUID ipid;
            RPC_STATUS status;
            IFoo * pIF;
            IBar * pIB;
            IWaz * PIW;
            HRESULT hR;

            status = RpcBindingInqObject(h, &ipid);
            if (status) return(SOMETHING);

            status = ORpcLookupIPID(ipid, &pIF);
            if (status) return(SOMETHING);

            status = ORpcImportObjRef(prIB, &pIB);
            if (status) return(SOMETHING);

            hR = pIF->Bar(i, pIB, &pIW);          // actual call to the method!

            pIB->Release();
            status = ORpcExportObjRef(pIW, pprIW);
            return(hR ? hR : SOMETHING);
            };
```

This is an example of the client side wrapper for Bar:

```
// assume some class CFoo that implements Bar method
class CFoo : IUnknown, IFoo {
            UUID ipid;      // one for each interface?
            handle_t h;

            virtual HRESULT QueryInterface(UUID iid, void **ppvoid);
            virtual HRESULT AddRef();
            virtual HRESULT Release();
            virtual HRESULT Bar(short i, IFoo * pIF, IWaz ** ppIW);
            };
```

```
HRESULT CFoo::Bar(short i, IFoo * pIF, IWaz ** ppIW) {
            OBJREF * prIF;
            OBJREF * prIW;
            HRESULT hR;
            RPC_STATUS status;

            status = RpcBindingSetObject(this->h, this->ipid);
            if (status) return(SOMETHING);

            status = ORpcExportObjRef(pIF, &prIF);
            if (status) return(SOMETHING);

            hR = Bar(this->h, i, prIF, &prIW);

            status = ORpcImportObjRef(prIW, ppIW);

            ORpcFreeObjRef(prIF);
            ORpcFreeObjRef(prIW);

            return(hR ? hR : SOMETHING);
        };
```

1.7Implementing ORPC in RPC

Since the implicit parameters are specified as IDL, the ORPC header received by RPC will contain many fields inserted by MIDL. Here are the definitions for the header on the wire.

```
/*
  An inbound header would be laid out as follows where the
extent array is optional and there may be zero of more extents.
An outbound header is laid out similarly.

    ORPCTHIS_WITHNOEXTENSIONS
        [
         ORPC_EXTENT_ARRAY
     [ORPC_EXTENT]*
    ]

*/
typedef struct
{
 COMVERSION     version;       // COM version number
 unsigned long  flags;         // INFO flags for presence of other data
 unsigned long  reserved1;     // set to zero
 LTID           ltid;          // logical thread id of caller
 unsigned long  unique;        // tag to indicate presence of extensions
} ORPCTHIS_WITHNOEXTENSION;

typedef struct
{
   unsigned long  rounded_size;   // Actual number of extents.
   uuid_t         id;             // Extension identifier.
   unsigned long  size;           // Extension size.
   byte           data[];         // Extension data.
} ORPC_EXTENT;


// Array of extensions.
typedef struct
{
 unsigned long  rounded_size;   // Actual number of extents
 unsigned long  size;           // Number of extents
 unsigned long  unique_flag[];  // Flags to indicate presense of ORPC_EXTENTs
} ORPC_EXTENT_ARRAY;

typedef struct
{
 unsigned long  flags;         // INFO flags for presence of other data
 unsigned long  unique;        // tag to indicate presence of extensions
} ORPCTHAT_WITH_NOEXTENSIONS;
```

This page intentionally left blank.