Network Working Group
INTERNET-DRAFT
<draft-brown-dcom-v1-spec-03.txt>
Expires in six months

Nat Brown
Charlie Kindel
Microsoft Corporation
January, 1998

Distributed Component Object Model Protocol -- DCOM/1.0

Status of this Memo

Abstract

The Distributed Component Object Model protocol is an application-level protocol for object-oriented remote procedure calls useful for distributed, component-based systems of all types. It is a generic protocol layered on the
distributed computing environment (DCE) RPC specification and facilitates

the
construction of task-specific communication protocols through features such
as: a platform neutral argument/parameter marshaling format (NDR), the

ability for objects to support multiple interfaces with a safe, interface-
level versioning scheme suited to independent evolution by multiple parties,
the ability to make authenticated connections and to choose levels of
channel
security, and a transport-neutral data representation for references
(including by-value) to objects.

Table of Contents

1. Introduction

The Distributed Component Object Model protocol (DCOM) is an application-
level protocol for object-oriented remote procedure calls and is thus also
called "Object RPC" or ORPC. The protocol consists of a set of extensions,
layered on the distributed computing environment (DCE) RPC specification
[CAE
RPC], with which familiarity is assumed. Familiarity is also assumed with
the
COM (Component Object Model) specification [COM].

Object RPC specifies:

@ How calls are made on an object

@ How object references are represented, communicated, and maintained

1.1 Purpose

There is a natural tendency in a networked environment to create entirely
new
application-level protocols as each new or seemingly unique combination of
client, user agent, and server requirement arises.

While in many situations the definition of a new protocol is useful and
justifiable, there are numerous features which have eventually been added to

or required from each new protocol (or which become layered above them) as they evolve and become used in broader contexts.

A design goal of the DCOM protocol is the inherent support of standard features required by any distributed application communication protocol. In other words, to act as a framework to facilitate the construction of task-specific communication paths between distributed applications.

Data Marshaling

A common occurrence among user agents using the HTTP protocol today is the use of complex, task-specific Query URL syntax and HTTP POSTs. Also increasingly common is the POSTing and response with custom MIME types to and
from resources which interpret the format and reply in same. While workable, there are drawbacks to this approach including increased complexity and work to produce and consume each new (and unique) format in the client and server,
lessened ability to build task-specific firewalls for administration and security purposes, and in many cases definition of platform-centric formats.

DCOM utilizes the Network Data Representation (NDR) for arbitrary data types supported by DCE RPC.

Security

DCOM leverages the authentication, authorization, and message integrity capabilities of DCE RPC. An implementation may support any level of DCE RPC security. Any connection or call can be made as secure or as insecure as negotiated by the client and the server.

Safe Non-Coordinated Versioning of Interfaces

In DCOM versioning of interfaces is done through identifiers which are universally unique (UUID's). To version a published interface, a new interface is defined with a different UUID to the updated specification. Multiple parties can simultaneously introduce "revisions" to interfaces by defining related but distinct interfaces without fear of colliding with each other's version numbers and without fear of breaking each other's down-level or up-level clients.

To date, the bulk of task-specific protocols (such as custom POSTs or MIME types using HTTP) have little or no concept of versioning at all, and simply "narrow" the incompatibility window by updating clients (typically pages which are being downloaded anyway) and servers (CGI scripts or other HTTP server infrastructure) simultaneously.

2. Overall Operation

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 DCOM network protocol is in fact that of implementing the DCE RPC network protocol on which it is built.

2.1 Object 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 server process

which originally allocated them; IPIDs do not necessarily use the standard UUID allocation algorithm, but rather may use a machine-specific algorithm which can assist with dispatching.

In an ORPC, the interface ID field of the RPC header specifies the IID, and arguments are found in the body, as usual. 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 in Section 3.7. 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 described in Section 3.8. 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, 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 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 standard location allocated for the "stub data." 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.

2.2 OXIDs and Object Exporters

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 64-bit value called an object exporter identifier, otherwise known as an OXID. Conceptually, an OXID can be thought of as an implementation scope for an 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 protocol itself. Data

structures in each Object Exporter keep track of the IPIDs exported and imported by that Object Exporter.

A given machine at any moment may support several OXIDs; however there is always a unique OXID Resolver service per machine which coordinates the management of all the OXIDs on the machine. The OXID Resolver typically (but not necessarily) resides at well-known ports (or endpoints, depending on your
terminology -- one per protocol, of course) on the machine. It supports a DCE
RPC interface known as IOXIDResolver, described in Section 5.2.

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 this 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 OXID Resolver which scopes the OXID specified in the object reference 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 (ie each Object Exporter) is COM object termed an "OXID object." OXID objects implement (at least) the IRemUnknown interface, a
COM interface through which remote management of reference counts and requests for interfaces are returned.

2.3 Class Activation

The server-provided DCOM class activation facility provides a way for a client to instantiate a new class object on a host machine, obtain its OXID, and obtain one or more IPIDs for the interfaces that it is interested in communicating through, all in one network round-trip. While it is primarily intended for the creation of new objects, the activation facility is free to return the same OXID and IPIDs for all activation requests of a particular class of object; this gives clients an easy way to access a well-known service without having to have a priori knowledge of a specific object (i.e. OXID & IPID) on the host, knowledge which can be invalidated by a reboot or a
network failure.

Clients are expected to know beforehand either the CLSID or a file moniker which will result in the creation of the class object that they are interested in communicating with. Clients also have the option of providing a
client-side IStorage object which will be used to instantiate and initialize the server-side object.

2.4 Marshaled Interface References

The DCOM protocol extends the Network Data Representation (NDR) 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 an object.
This is the only extension to NDR made by the DCOM protocol.

A marshaled interface reference is described by a type known as an OBJREF, which is described in detail in Section 3.3. An OBJREF in actuality has several variations:

NULL

This is a reference to no object.

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 (scoped to the object exporter with which the object is associated).

@ An OXID, which identifies the scope where the implementation of the object is active, and can be used to reach the interface pointer.

@ 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.

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.

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

2.5 Reference Counting

In the DCOM protocol, remote reference counting is conducted per interface (per IPID).

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 the OXID object associated with each OXID, the IPID of which is returned from the function IOXIDResolver::ResolveOxid (section 5.2.1) or IRemoteActivation::RemoteActivation (section 6.2.1). 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.

To prevent a malicious application from calling RemRelease incorrectly, an
application may request secure references. In that case the application
must
call RemAddRef (and RemRelease later on) securely and must request private
references. Private references are stored by client identity so one client
cannot release another client s references. DCOM requires that each client
make a call to get his own secure references, rather then receiving a secure
reference from someone who already has one. This reduces the efficiency of
interface marshalling because the client must make a callback.

2.6 Pinging

The above reference counting scheme would be entirely adequate on its own if
clients never terminated abnormally, but in fact they do, and the system

needs to be robust in the face of clients terminating abnormally 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 DCOM protocol, for reasons of expense. The basic underlying technology
used in virtually all protocols for detecting remote abnormal termination 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 DCOM protocol
includes a pinging infrastructure to significantly reduce network traffic by
relying on the client OXID Resolver implementation to do local management of
client liveness detection, and having the actual pings be sent only on a
machine by 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 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 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 of local knowledge, though the timeliness with
which this is carried out, if at all, is implementation specific detail of
the server. 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, 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 cPublicRefs 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
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 to busy
servers.

2.6.1 Delta Pinging

Without any further refinements, ping messages could be quite hefty. If
machine A held 1024 remote object references (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 RPC)
interface IOXIDResolver on the OXID Resolver: ComplexPing, and SimplePing.
ComplexPing is used by clients to group the set of OIDs that they must ping
into sets known to the server. These entire sets of OIDs can then be
subsequently pinged with a single, short, call to SimplePing.

2.7 QueryInterface

zThe IRemUnknown interface on the OXID object, in addition to servicing

reference counting as described above also services QueryInterface calls for
remote clients for IPIDs managed by that object 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.

2.8 Causality 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 to cause a deadlock, and thus should be avoided.

The causality id for maybe, idempotent, and broadcast calls must be set to
null (e.g., all zeros). 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.

3. Data Types and Structures

This following several sections present the technical details of the DCOM
protocol.

1998

3.1 DCE 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 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.

@ Faults are returned in the stub fault field of the DCE RPC fault packet. Any 32 bit value may be returned. Only RPC_E_VERSION_MISMATCH is pre-specified.

@ DCE RPC cancel is supported.

@ All interface version numbers must be 0.0.

@ The transfer syntax GUID is {8a885d04-1ceb-11c9-9fe8-08002b104860}, version 2 (0x02).

3.2 ORPC Base Definitions

There are several fundamental data types and structures on which the COM network protocol is built. These types are shown here in standard C header format.

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

```
interface ObjectRpcBaseTypes
{
    ////////////////////////////////////////////////////////////
    //
    //  Identifier Definitions
    //
    ////////////////////////////////////////////////////////////

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

    typedef const IPID &REFIPID;
    typedef REFGUID REFIPID;

    //////////////////////////////////////////////////////////////////
    //
    //  ORPC Call Packet Format
    //
    //////////////////////////////////////////////////////////////////

    // COM_MINOR_VERSION = 1   (NT4.0, SP1, SP2, DCOM95).
    //      - Initial Release
    //      - Must be used when talking to downlevel machines, including
    //        on Remote Activation calls.
    // COM_MINOR_VERSION = 2   (NT4.0 SP3 and beyond).
    //      - Added ResolveOxid2 to IObjectExporter to retrieve the
    //        COM version number of the server. Passed to the NDR engine
    //        to fix fatal endian-ness flaw in the way OLEAUTOMATION
marshals
    //        BSTRS. Previous way used trailing padding, which is not NDR
    //        compatible. See Bug# 69189.
    // COM_MINOR_VERSION = 3   (NT4.0 SP4 and DCOM95 builds 1018 and beyond)
    //      - OLEAUT32 added two new types to the SAFEARRAY, but SAFEARRAY
    //        previously included the "default" keyword, which prevented
    //        downlevel NDR engines from correctly handling any extensions.
    //        Machines with version >=5.3 don't use "default" and will
    //        gracefully handle future extensions to SAFEARRAY.
    // old constants (for convenience)
    const unsigned short COM_MINOR_VERSION_1 = 1;
    const unsigned short COM_MINOR_VERSION_2 = 2;
```

```
    // current version
    const unsigned short COM_MAJOR_VERSION = 5;
    const unsigned short COM_MINOR_VERSION = 3;

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


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

    const unsigned long ORPCF_NULL      = 0;  // no additional info in
packet
    const unsigned long ORPCF_LOCAL     = 1;  // call is local to this
machine
    const unsigned long ORPCF_RESERVED1 = 2;  // reserved for local use
    const unsigned long ORPCF_RESERVED2 = 4;  // reserved for local use
    const unsigned long ORPCF_RESERVED3 = 8;  // reserved for local use
    const unsigned long ORPCF_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.
        unsigned long reserved; // Must be zero.
        [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;      // ORPCF 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;       // ORPCF flags for presence of other
data

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


    ///////////////////////////////////////////////////////////////////
    //
    //  Marshaled COM Interface Wire Format
    //
    ///////////////////////////////////////////////////////////////////

    // DUALSTRINGARRAYS 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_NB_IPX   = 0x0D;
    const unsigned short NCACN_DNET_NSP = 0x04;
    const unsigned short NCACN_HTTP     = 0x1F;
```

```
    typedef struct tagSTRINGBINDING
    {
         unsigned short wTowerId;     // Cannot be zero.
         unsigned short aNetworkAddr; // Zero terminated.
    } STRINGBINDING;

    const unsigned short COM_C_AUTHZ_NONE = 0xffff;

    typedef struct tagSECURITYBINDING
    {
        unsigned short wAuthnSvc;  // Cannot be zero.
        unsigned short wAuthzSvc;  // Must not be zero.
        unsigned short aPrincName; // Zero terminated.
    }  SECURITYBINDING;

    typedef struct tagDUALSTRINGARRAY
    {
        unsigned short wNumEntries;     // Number of entries in array.
        unsigned short wSecurityOffset; // Offset of security info.
        // The array contains two parts, a set of STRINGBINDINGs
        // and a set of SECURITYBINDINGs.  Each set is terminated by an
        // extra zero.  The shortest array contains four zeros.

        [size_is(wNumEntries)] unsigned short aStringArray[];
    } DUALSTRINGARRAY;

    // signature value for OBJREF (object reference, actually the
    // marshaled form of a COM interface).
    const unsigned long OBJREF_SIGNATURE = 0x574f454d;  // 'MEOW'

    // flag values for OBJREF
    const unsigned long OBJREF_STANDARD = 0x1;  // standard marshaled objref
    const unsigned long OBJREF_HANDLER  = 0x2;  // handler marshaled objref
    const unsigned long OBJREF_CUSTOM   = 0x4;  // custom marshaled objref

    // Flag values for a STDOBJREF (standard part of an OBJREF).
    // SORF_OXRES1 - SORF_OXRES8 are reserved for the object exporters
    // use only, object importers must ignore them and must not enforce MBZ.

    const unsigned long SORF_OXRES1     = 0x1;  // reserved for exporter
    const unsigned long SORF_OXRES2     = 0x20; // reserved for exporter
```

```
    const unsigned long SORF_OXRES3     = 0x40; // reserved for exporter
    const unsigned long SORF_OXRES4     = 0x80; // reserved for exporter
    const unsigned long SORF_OXRES5     = 0x100;// reserved for exporter
    const unsigned long SORF_OXRES6     = 0x200;// reserved for exporter
    const unsigned long SORF_OXRES7     = 0x400;// reserved for exporter
    const unsigned long SORF_OXRES8     = 0x800;// reserved for exporter

    const unsigned long SORF_NULL       = 0x0;   // convenient for
initializing SORF
    const unsigned long SORF_NOPING     = 0x1000;// Pinging is not required


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

    // OBJREF is the format of a marshaled interface pointer.
    typedef struct tagOBJREF
    {
        unsigned long signature;           // must be OBJREF_SIGNATURE
        unsigned long flags;               // OBJREF flags (see above)
        GUID          iid;                 // interface identifier

        [switch_is(flags), switch_type(unsigned long)] union
        {
            [case(OBJREF_STANDARD)] struct
            {
                STDOBJREF       std;       // standard objref
                DUALSTRINGARRAY saResAddr; // resolver address
            } u_standard;

            [case(OBJREF_HANDLER)] struct
            {
                STDOBJREF       std;       // standard objref
                CLSID           clsid;     // Clsid of handler code
                DUALSTRINGARRAY saResAddr; // resolver address
```

```
            } u_handler;

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

```
specific data
            } u_custom;

        } u_objref;

    } OBJREF;

    // wire representation of a marshalled interface pointer
    typedef struct tagMInterfacePointer
    {
        ULONG           ulCntData;          // size of data
        [size_is(ulCntData)] BYTE abData[]; // data (OBJREF)
    } MInterfacePointer;

    typedef [unique] MInterfacePointer * PMInterfacePointer;
}


//////////////////////////////////////////////////////////////////
```

3.3 OBJREF

An OBJREF is the data type used to represent an actual marshaled object reference. An OBJREF can either be empty or assume one of three 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.

3.3.1 OBJREF_STANDARD

Contains one interface of an object marshaled in standard 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
specifies the saResAddr for the OXID Resolver on the server machine, eliminating the need for the unmarshaler 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
signature	unsigned long	Must be OBJREF_SIGNATURE
flags	unsigned long	OBJREF flags (section 3.5)
iid	GUID	Interface identifier
std	STDOBJREF	A standard object reference used to connect to the source object (Section 3.4).
saResAddr	STRINGARRAY	The resolver address.

3.3.2 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
signature	unsigned long	Must be OBJREF_SIGNATURE
flags	unsigned long	OBJREF flags (section 3.5)
iid	GUID	Interface identifier
std	STDOBJREF	A standard object reference used to connect to the source object (Section 3.4).
clsid	CLSID	The CLSID of handler to create in the destination client.
saResAddr	STRINGARRAY	The resolver address.

3.3.3 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
signature	unsigned long	Must be OBJREF_SIGNATURE
flags	unsigned long	OBJREF flags (section 3.5)
GUID	iid	Interface identifier
clsid	CLSID	The CLSID of the object to create in the destination client.
cbExtension	unsigned long	The size of the extension data.
size	unsigned long	The size of the marshaled data provided by the source object, plus the size of the extension data, and passed 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 (class specific data). The first cbExtension bytes are the reserved for future extensions to the protocol, and should not be passed into the custom unmarshaler. CoUnmarshalInterface should skip the extension data, and the data starting at pData+cbExtension should be given to the custom unmarshaler.

3.4 STDOBJREF

An instance of a STDOBJREF represents a COM interface pointer that has been
marshaled using the standard COM network protocol.

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

Member Semantic

flags	Flag values taken from the enumeration SORFFLAGS. These are described in Section 3.5.

cPublicRefs	The number of reference counts on ipid that are being transferred in this marshaling.
oxid	The OXID of the server that owns this OID.
oid	The OID of the object to which ipid corresponds.
ipid	The IPID of the interface being marshaled.

3.5 SORFLAGS

The various SORFLAGS values have the following meanings. The SORF_OXRESxxx 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	Meaning
SORF_NULL	Convenient for initialization.
SORF_OXRES1	Reserved for exporter.
SORF_OXRES2	Reserved for exporter.
SORF_OXRES3	Reserved for exporter.
SORF_OXRES4	Reserved for exporter.
SORF_OXRES5	Reserved for exporter.
SORF_OXRES6	Reserved for exporter.
SORF_OXRES7	Reserved for exporter.
SORF_OXRES8	Reserved for exporter.
SORF_NOPING	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.

3.6 ORPCINFOFLAGS

The various ORPCINFOFLAGS have the following meanings.

Flag	Meaning
ORPCF_NULL	(Not a real flag. Merely a defined constant indicating the absence of any flag values.)
ORPCF_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.

ORPCF_RESERVED1	If ORPCF_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
ORPCF_RESERVED2	If ORPCF_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
ORPCF_RESERVED3	If ORPCF_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
ORPCF_RESERVED4	If ORPCF_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 ORPCF_LOCAL bit is not set and any of the other bits are set then the receiver should return a fault.

3.7 ORPCTHIS

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 5.1. 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 (section 3.6). Reserved unsigned long Must be set to zero.
reserved1	unsigned long	Set to zero.
cid	CID	The causality id of this ORPC.
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 in Section 3.10.

The cid field contains the causality id. Each time a client makes a unique

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 as the call currently being processed. 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, described in
Section 3.10. 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.

3.8 ORPCTHAT

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 are eight byte aligned.

Member	Type	Semantic
flags	unsigned long	Flag values taken from the enumeration ORPCINFOFLAGS (section 3.6).
extensions	ORPC_EXTENT_ARRAY*	The body extensions, if any, returned by this call. See Section 3.10 for a general description of body extensions as well as a description of existing well-known extensions.

3.9 HRESULTs

HRESULTs are the 32-bit return value from ORPC methods. The following is a partial list of already defined HRESULTs.

Value	Meaning
S_OK	Success. (0x00000000)
E_OUTOFMEMORY	Insufficient memory to complete the call. (0x80000002)
E_INVALIDARG	One or more arguments are invalid. (0x80000003)
E_NOINTERFACE	No such interface supported (0x80000004)
E_ACCESSDENIED	A secured operation has failed due to inadequate security privileges. (0x80070005)
E_UNEXPECTED	Unknown, but relatively catastrophic error. (0x8000FFFF)
S_FALSE	False. (0x00000001)
RPC_S_PROCNUM_OUT_OF_RANGE	The procedure number is out of range. (0xC002002E)
RPC_E_INVALID_OXID	The object exporter was not found. (0x80070776)
RPC_E_INVALID_OID	The object specified was not found or recognized.

Value	Meaning
	(0x80070777)
RPC_E_INVALID_SET	The object exporter set specified was not found. (0x80070778)

RPC_E_INVALID_OBJECT (0x80010114)	The requested object does not exist.
RPC_E_VERSION_MISMATCH	The version of COM on the client and server machines does not match. (0x80010110)

Further details TBS.

3.10 Body Extensions

Body Extensions are UUID-tagged blocks of data which are useful for conveying
additional, typically out-of-band, information on incoming invocations (within ORPCTHIS, Section 3.7) and in replies (within ORPCTHAT, Section 3.8).

Any implementations of the DCOM protocol may define its own extensions with their own UUIDs. Implementations should skip over extensions which they do not recognize or do not wish to support.

Body Extensions are marshaled as an array of bytes with initial eight byte alignment. The following sections describe several existing body extensions.

3.10.1 Debugging Extension

{f1f19680-4d2a-11ce-a66a-0020af6e72f4}

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.

Further details TBS.

3.10.2 Extended Error Extension

{f1f19681-4d2a-11ce-a66a-0020af6e72f4}

The extended error information body extension conveys extended error information concerning the original root cause of a error back to a caller so
that the caller can deal with it. This extension is only semantically useful in Response and Fault PDUs.

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.

Further details TBS.

4. IRemUnknown and IRemUnknown2 interfaces

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.

References are kept per interface rather then per object.

This interface is implemented by the COM "OXID object" associated with each
OXID (i.e. each Object Exporter). The IPID for the IRemUnknown interface on
this object is returned from IOXIDResolver::ResolveOxid (see Section 5.2.1),
or when an object is activated with IRemoteActivation::RemoteActivation (see
section 6.2.1). An OXID object need never be pinged; its interfaces (this
IPID included) need never be reference counted. IRemUnknown is specified as
follows:

The IRemUnknown2 interface introduced in version 5.2 of the DCOM protocol
inherits from IRemUnknown and adds an extra method - RemoteQueryInterface2

- which allows clients to retrieve interface pointers to objects which
supply
additional data beyond the STDOBJREF in their marshaled interface packets.

```
//  The remote version of IUnknown.  This interface exists on every
//  OXID (whether an OXID represents either a thread or a process is
//  implementation specific).  It is used by clients to query for new
//  interfaces, get additional references (for marshaling), and release
//  outstanding references.
//  This interface is passed along during OXID resolution.
//
[
   object,
   uuid(00000131-0000-0000-C000-000000000046)
```

```
]
interface IRemUnknown : IUnknown
{
    typedef struct tagREMQIRESULT
```

```
    {
       HRESULT      hResult;            // result of call
       STDOBJREF    std;                // data for returned interface
    } REMQIRESULT;

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

    typedef struct tagREMINTERFACEREF
    {
        IPID           ipid;           // ipid to AddRef/Release
        unsigned long  cPublicRefs;
        unsigned long  cPrivateRefs;
    } REMINTERFACEREF;

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

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

//  Derived from IRemUnknown, this interface supports Remote Query interface
//  for objects that supply additional data beyond the STDOBJREF in their
```

```
//  marshaled interface packets.

[
    object,
    uuid(00000142-0000-0000-C000-000000000046)
]

interface IRemUnknown2 : IRemUnknown
{
#ifndef DO_NO_IMPORTS
    import "unknwn.idl";
    import "obase.idl";
#endif

    HRESULT RemQueryInterface2
    (
        [in] REFIPID                            ripid,
        [in] unsigned short                     cIids,
        [in, size_is(cIids)] IID                *iids,
        [out, size_is(cIids)] HRESULT           *phr,
        [out, size_is(cIids)] MInterfacePointer **ppMIF
    );
}
```

4.1 IRemUnknown::RemQueryInterface

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 the IUnknown interface of the object.

Argument	Type	Semantic
ripid	REFIPID	The interface on an object from whom more interfaces are desired.
cRefs	unsigned long	The number of references sought on each of the returned IIDs.
cIids	unsigned short	The 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.

ReturnValue	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. QueryInterface returned S_OK for every IID specified.
S_FALSE	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. QueryInterface returned a failure code for at least one of the IIDs specified.
E_NOINTERFACE	QueryInterface returned a failure code for every IID specifed.
E_INVALIDARG	One or more arguments (likely ipid) were invalid. No result values are returned.
E_UNEXPECTED	An unspecified error occurred.
E_OUTOFMEMORY	Insufficient memory to complete the call.
RPC_E_INVALID_OBJECT	The requested object does not exist. No result

values
are returned.

4.1.1 REMQIRESULT

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.

4.2 IRemUnknown2::RemQueryInterface2

Like RemQueryInterface, this method queries for zero or more interfaces from the interface behind the IPID ipid. Instead of returning the STDOBJREF marshaled interface packet, this method can return any marshaled data packet in the form of a blob of bytes (including the traditional STDOBJREF).

Argument	Type	Semantic
ripid	REFIPID	The interface on an object from whom more interfaces are desired.
cIids	unsigned short	The number of references sought on each of the returned IIDs.
iids	IID*	The interfaces being requested.
phr	HRESULT*	The result code returned from QueryInterface() on each interface in the iids array.
ppMIF	MinterfacePointer**	The marshaled interface packet for each of the IIDs requested.

ReturnValue 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. QueryInterface returned S_OK for every IID specified.
S_FALSE	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. QueryInterface returned a failure code for at least one of the IIDs specified.
E_NOINTERFACE	QueryInterface returned a failure code for every IID specifed.
E_INVALIDARG	One or more arguments (likely ipid) were invalid. No result values are returned.
E_UNEXPECTED	An unspecified error occurred.
E_OUTOFMEMORY	Insufficient memory to complete the call.
RPC_E_INVALID_OBJECT	The requested object does not exist. No result values are returned.

4.3 **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.

Argument	Type	Semantic
cInterfaceRefs	unsigned short	The size of the rgRefs array.
InterfaceRefs	REMINTERFACEREF[]	An array of REMINTERFACEREFs, cRefs large. Each IPID indicates an interface managed by this OXID on whom more reference counts are sought. The corresponding reference count

		(cInterfaceRefs), which may not be zero (and thus is one or more), indicates the number of reference counts sought on that IPID.
pResults	HRESULT*	An array of HRESULTs cInterfaceRefs large, each containing the result of

attempting an AddRef on the ipid in the corresponding REMINTERFACEREF.

Return Value	Meaning
S_OK	Success. An attempt was made to retrieve each of the requested interface references.
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.
CO_E_OBJNOTREG	Object is not registered.

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.

If cPrivateRefs is not zero for all IPIDs, the call to RemAddRef must be made securely. DCOM on the server remembers the name of the client and the authentication and authorization service used to make to RemAddRef call.

4.3.1 REMINTERFACEREF

Member	Type	Semantic
ipid	IPID	ipid to AddRef/Release.
cPublicRefs	unsigned long	Number of public references granted.
cPrivateRefs	unsigned long	Number of private references granted. Private references belong only to this client and can

```
                              not be passed to other clients when marshaling
                              the proxy. If a client has only private
                              references and wishes to pass the proxy to
some
                              other client, it must first obtain some public
                              references via IRemUnknown::RemAddRef and then
                              pass one or more of those references in the
                              STDOBJREF cPublicRefs field of the marshaled
                              interface.
```

4.4 IRemUnknown::RemRelease

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

If cPrivateRefs is not zero for all IPIDs, the call to RemRelease must be
made securely. For each IPID, DCOM maintains a table of reference counts
indexed by the client identity (name, authn svc, authz svc). All public
references are stored in one entry. Any call to RemRelease can release
public references. Private references can only be released by the client
that added them.

```
Argument    Type              Semantic

cRefs       unsigned short    The size of the rgRefs array.
rgRefs      REMINTERFACEREF[] An array of REMINTERFACEREFs, cRefs 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 release each of the requested
              interface references.
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.
```

5. The OXID Resolver

Each machine that supports the COM network protocol supports a one- per-
machine service known as the machine's `OXID Resolver.' Communication with
an
OXID Resolver is via a DCE RPC, not an ORPC.

The OXID Resolver 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 either
  itself a client or is the server. Note that it typically returns only to
  client processes on the same machine as itself, the OXIDs for which it is
a
  client.

@ It receives pings from remote client machines to keep its own objects
  alive.

@ May do lazy protocol registration in the servers which it scopes.

These services are carried out through an RPC interface (not a COM
interface)
known as IOXIDResolver. An OXID Resolver 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 is itself a client The second case occurs when two or more client
processes on the same machine ask their local OXID Resolver to resolve a
given OXID. The client OXID Resolver in this case can cache the OXID
information an return it to local clients without having to contact the
server s OXID Resolver again.

5.1 OXID Resolver Ports/Endpoints

The OXID Resolver resides at different endpoints (ports) depending on the transport being used. The OXID Resolver optimally resides at the same endpoints as the DCE RPC Endpoint Mapper (EPM). To accommodate systems where DCOM will coexist with existing DCE RPC installations (i.e., where an EPM and
presumably a complete DCE RPC runtime already exists), the DCOM implementation on that system will register its interfaces with the DCE EPM and all DCOM implementations must be able to fall back if they make DCOM-specific calls on the DCE EPM endpoint which fail.

Protocol String Name(s)	Description	Endpoint
ncadg_ip_udp, ip	CL over UDP/IP	135
ncacn_ip_tcp	CO over TCP/IP	135
ncadg_ipx	CL over IPX	TBD
ncacn_spx	CO over SPX	TBD
ncacn_nb_nb	CO over NetBIOS over NetBEUI	TBD
ncacn_nb_ipx	CO over IPX	TBD
ncacn_http	CO over HTTP	593

5.2 The IOXIDResolver Interface

IOXIDResolver (in earlier DCOM documentation this interface was named IObjectExporter) is defined as follows:

```
[
    uuid(99fcfec4-5260-101b-bbcb-00aa0021347a),
    pointer_default(unique)
]
interface IOXIDResolver
{
    // 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] DUALSTRINGARRAY **ppdsaOxidBindings,
    [out, ref] IPID            *pipidRemUnknown,
    [out, ref] DWORD           *pAuthnHint
    );

    // Simple ping is used to ping a Set. Client machines use this
    // to inform the object exporter that it is still using the
    // members of 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  // Must not be zero
    );

    // 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, out]  SETID          *pSetId,  // In of 0 on first
                                        // call for new set.
    [in]       unsigned short  SequenceNum,
    [in]       unsigned short  cAddToSet,
    [in]       unsigned short  cDelFromSet,
    [in, unique, size_is(cAddToSet)]   OID AddToSet[],
                 // add these OIDs to the set
    [in, unique, size_is(cDelFromSet)] OID DelFromSet[],
                 // remove these OIDs from the set
    [out]      unsigned short *pPingBackoffFactor
                 // 2^factor = multipler
    );

    // In some cases the client maybe unsure that a particular
    // binding will reach the server.  (For example, when the oxid
    // bindings have more then one TCP/IP binding)  This call
    // can be used to validate the binding
    // from the client.
    [idempotent] error_status_t ServerAlive
    (
    [in]       handle_t        hRpc
    );

    // Method to get the protocol sequences, string bindings, RemoteUnknown
IPID
    // and COM version for an object server given its OXID. Supported by
DCOM
    // version 5.2 and above.

    [idempotent] error_status_t ResolveOxid2
    (
    [in]       handle_t        hRpc,
    [in]       OXID           *pOxid,
    [in]       unsigned short  cRequestedProtseqs,
    [in,  ref, size_is(cRequestedProtseqs)]
                unsigned short  arRequestedProtseqs[],
    [out, ref] DUALSTRINGARRAY **ppdsaOxidBindings,
    [out, ref] IPID            *pipidRemUnknown,
```

```
    [out, ref] DWORD           *pAuthnHint,
    [out, ref] COMVERSION      *pComVersion
    );
}
```

5.2.1 IOXIDResolver::ResolveOxid and IOXIDResolver::ResolveOxid2

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 in 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.

Version 5.2 of the DCOM wire protocol introduces a new method called ResolveOXID2 which allows a client to determine the version of a server s COM
implementation when it asks for OXID resolution. All clients must attempt to call this method to make sure that the major version of the server is one which they are capable of supporting. Clients who call this method and get an
RPC_S_PROCNUM_OUT_OF_RANGE error can assume that the server supports version **5.1 of the DCOM wire protocol. If the method call does succeed, the client** should compare pComVersion->MajorVersion with the major version(s) that the client supports. If the client does not explicitly support the major version returned by the server, it should disconnect.

The major version combined with the lower of the client s and server s minor versions should be inserted into the ORPCTHIS structure when issuing an ORPC to the server.

Please see the IDL section above for notes on the differences between the minor versions of the DCOM wire protocol.

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.
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 Protocol Registration section for more details.
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.
pdwAuthnHint	unsigned long*	A value taken from the RPC_C_AUTHN constants. A hint to the caller as to the minimum authentication level which the server will accept.
pComVersion	COMVERSION*	[ResolveOxid2 Only] A structure containing the major and minor version of the COM implementation on the server.

Return Value	Meaning
S_OK	Success. The requested information was returned.
RPC_S_PROCNUM_OUT_OF_RANGE	The procedure number is out of range (i.e. the function is not implemented).
RPC_E_INVALID_OXID	This OXID is unknown to this OXID Resolver, and thus no information was returned.
E_UNEXPECTED	An unspecified error occurred. Some of the requested information may not be returned.

Object references are transient things. They are not meant to be stored in
files or otherwise kept persistently.

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:

1. **Looking up an OXID the first time an interface is unmarshaled on a**
machine,

2. **Looking up an OXID between a pair of machines that already have**
connections,

3. **Looking up an OXID from a middleman, and**

4. **Looking up string bindings with unresolved endpoints (lazy use protseq).**

Another interesting topic is garbage collection of stored string binding
vectors.

5.2.1.2 Lookup Between Friends

Consider the case with two machines A and B. Machine A has a client process
C
and and OXID Resolver process D. Machine B has OXID Resolver process E and
server process F.

Server process F, when it starts up, registers it s RPC string bindings with
its local OXID Resolver process E, and creates an OBJREF to some object
inside process F. At some future time client process C receives that OBJREF
(Note: the mechanism used to acquire this OBJREF is not relevant to this
discussion, it may have come through the object activation protocol (beyond
the scope of this document) or as an [out] interface parameter in some other
ORPC call, or through some other mechanism). The OBJREF contains the OXID
for
process F, and the string bindings for the Resolver process E on the server
machine.

Client Process C asks its local OXID Resolver to resolve the OXID for F. It
passes it the OXID and the string bindings for OXID Resolver E. If Resolver
D
has never seen the OXID before, it calls the OXID Resolver E to ask it to
resolve the OXID. Resolver E looks up the OXID and returns the string
bindings for server process F. Resolver D then caches this information for
future use, and returns the string bindings to client process C. Client
process C then binds to the string bindings and is now able to make ORPC
calls directly to the server process F.

If other client processes on machine A receive an OBJREF for process F, the
OXID resolve can be handled completely by the Resolver process D on machine
A. There is no need to contact Resolver process E on the server again.

If machine A gives an OBJREF for F to a client process on another machine G,
then the Resolver process on G will repeat the same steps as Resolver
process
D did to resolve the OXID.

```
+============+============+  +===========+===========+
|       Machine A         |  |       Machine B       |
+============+============+  +===========+===========+
+============+============+  +===========+===========+
| Process  C | Resolver D |  | Resolver E| Process F |
+============+============+  +===========+===========+
|            |            |  |           | register  |
|            |            |  |           | endpoints |
|            |            |  |           | with local|
|            |            |  |           | Resolver E|
+------------+------------+  +-----------+-----------+
|            |            |  | cache F   |           |
|            |            |  | and it s  |           |
|            |            |  | endpoints |           |
|            |            |  |           |           |
+------------+------------+  +-----------+-----------+
| receive    |            |  |           |           |
| OBJREF     |            |  |           |           |
| to F       |            |  |           |           |
|            |            |  |           |           |
+------------+------------+  +-----------+-----------+
| ask local  |            |  |           |           |
| Resolver D |            |  |           |           |
| to resolve |            |  |           |           |
| F          |            |  |           |           |
+------------+------------+  +-----------+-----------+
|            | ask remote |  |           |           |
|            | Resolver   |  |           |           |
|            | E to       |  |           |           |
|            | resolve F  |  |           |           |
+------------+------------+  +-----------+-----------+
|            |            |  | lookup F  |           |
|            |            |  | and       |           |
|            |            |  | return    |           |
```

```
|            |            |  | endpoints |           |
+------------+------------+  +-----------+-----------+
|            | cache      |  |           |           |
|            | endpoints  |  |           |           |
|            | to F and   |  |           |           |
|            | return to C|  |           |           |
+------------+------------+  +-----------+-----------+
| bind to    |            |  |           |           |
| endpoint   |            |  |           |           |
| for F      |            |  |           |           |
|            |            |  |           |           |
+------------+------------+  +-----------+-----------+
| invoke     |            |  |           |           |
| method on  |            |  |           |           |
| F          |            |  |           |           |
|            |            |  |           |           |
+------------+------------+  +-----------+-----------+
```

5.2.1.4 Lazy Protocol Registration

In a homogeneous network, all machines communicate 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 lazy protocol registration or it
does not.

If the server is using lazy protocol registration, the implementation of
ResolveOxid is modified slightly. When the client OXID Resolver calls the
server OXID Resolver, it passes the requested protocol sequence vector. If
none of the requested protocol sequences have endpoints allocated in the
server, the server OXID Resolver allocates them according to its own
endpoint
allocation mechanisms.

If the server does not implement the lazy protocol registration, then all
protocol sequences are registered by the server at server initialization
time.

When registering protocol sequences, the server may register endpoints and
the server s string bindings will contain the complete endpoints. However,
if the server chooses not register endpoints when it registers protocol
sequences the endpoint mapper process can be used to forward calls to the
server. Using the endpoint mapper requires that all server IIDs be
registered
in the endpoint mapper. It also allows a different lazy protocol
registration
mechanism. The endpoint mapper can perform some local magic to force the
server to register the protocol sequences.

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

5.2.2 IOXIDResolver::SimplePing

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 (section 5.2.3).

Ping a set, previously created with IOXIDResolver::ComplexPing, of OIDs owned
by this OXID Resolver. 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 IOXIDResolver::ComplexPing on this same OXID Resolver.

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

5.2.3 **IOXIDResolver::ComplexPing**

Ping a ping set. Optionally, add and/or remove some OIDs from the set. Optionally, adjust some 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 to adjust
the time-out).

Ping set ids (SETIDs) are allocated unilaterally by the server OXID Resolver. The client OXID Resolver then communicates with the server OXID

Resolver to add (and later remove) OIDs from the ping set..

Each OID owned by a server OXID Resolver 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 OXID Resolver 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; otherwise, OID has never been in a set, and t is a default value (ping period equals 120 seconds, ping
time-out count equals three (3), t equals 360 seconds, or six (6) minutes).

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.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
pSetId	SETID	The SETID being manipulated. SequenceNum unsigned short 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.
cAddToSet	unsigned short	The size of the array AddToSet.
cDelFromSet	unsigned short	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
                                 OID 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.
pPingBackoffFactor  unsigned     Acts as a hint (only) from the server to
the
                    short*       client in order to reduce ping traffic.
                                 Clients are requested to not ping more
often
                                 than (1<<*pPingBackoffFactor)*
                                 (BasePingInterval=120) seconds, and the
                                 number of pings until timeout remains
                                 unchanged at the default of 3. Clients may
                                 choose to assume that this parameter is
                                 always zero.
```

```
Return Value       Meaning

S_OK               Success. The set was pinged, etc.
RPC_E_INVALID_OID  Indicates that some OID was not recognized. There is no
                   recovery action for this error, it is informational only.
E_ACCESSDENIED     Access is denied.
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.
```

6. Object Activation

Each machine that supports the COM network protocol also supports a one-per-machine interface called IRemoteActivation (which supersedes ISCMtoSCM in previous drafts of this document). This interface contains one method - RemoteActivation - used to activate a class object associated with a caller-
supplied CLSID, file moniker, or IStorage pointer.

6.1 IRemoteActivation Ports/Endpoints

IRemoteActivation resides at the same endpoints as the IOXIDResolver interface. Please see section 5.1 for more information.

6.2 The IRemoteActivation Interface

IRemoteActivation is defined as follows:

```
[
    uuid(4d9f4ab8-7d1c-11cf-861e-0020af6e7c57),
    pointer_default(unique)
]

interface IRemoteActivation
{
    import "obase.idl";

    const unsigned long MODE_GET_CLASS_OBJECT = 0xffffffff;

    HRESULT RemoteActivation(
     [in] handle_t                              hRpc,
     [in] ORPCTHIS                             *ORPCthis,
     [out] ORPCTHAT                            *ORPCthat,
     [in] GUID                                 *Clsid,
     [in, string, unique] WCHAR                 *pwszObjectName,
     [in, unique] MInterfacePointer             *pObjectStorage,
     [in] DWORD                                  ClientImpLevel,
     [in] DWORD                                  Mode,
     [in] DWORD                                  Interfaces,
     [in,unique,size_is(Interfaces)] IID        *pIIDs,
     [in] unsigned short                         cRequestedProtseqs,
     [in, size_is(cRequestedProtseqs)]
          unsigned short                         RequestedProtseqs[],
     [out] OXID                                 *pOxid,
     [out] DUALSTRINGARRAY                     **ppdsaOxidBindings,
     [out] IPID                                 *pipidRemUnknown,
     [out] DWORD                                *pAuthnHint,
     [out] COMVERSION                           *pServerVersion,
     [out] HRESULT                              *phr,
     [out,size_is(Interfaces)] MInterfacePointer **ppInterfaceData,
     [out,size_is(Interfaces)] HRESULT          *pResults
     );
}
```

6.2.1 IRemoteActivation::RemoteActivation

Activates the class object specified by Clsid, associated with the
pwszObjectName file moniker, or associated with pObjectStorage -- a client-
provided IStorage object --, and gives the client the OXID and IPID(s) of
any
interfaces it requested on that object. Implementations are free to return a
new instance of the requested class object every time this API is called or
to return the same class object (i.e. the same OXID and IPIDs) every time.
Implementations may choose to offer launch functionality by loading and
initializing the executable code which implements the requested class object
if it is not already running, or if the programmer wishes to have objects
segregated into different processes because of security or robustness
considerations.

This function incorporates the functionality of both the IRemUnknown and
IOXIDResolver interfaces so that only one network roundtrip is necessary for

object activation.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
ORPCthis	ORPCTHIS*	Client-provided ORPCTHIS
ORPCthat	ORPCTHAT*	Server-provided ORPCTHAT
Clsid	const GUID*	The CLSID of the class object the caller wishes to activate.
pwszObjectName	WCHAR*	Path to a file name which when activated by the server s moniker facility, will return the object that the client is interested in.
pObjectStorage	MInterfacePointer*	Interface pointer to an object on the client which supports IStorage. The server on the remote will use the IStorage s CLSID to determine which object to instantiate on the server.
ClientImpLevel	DWORD	A value taken from the RPC_C_IMP constants. Used to inform the server of the client s default impersonation level.
Mode	DWORD	Set to MODE_GET_CLASS_OBJECT for regular object activations. If pwszObjectName or pObjectStorage are not NULL, this mode is passed to IPersistFile::Load when activating the object on the server.
Interfaces	DWORD	The number of interfaces being requested.
pIIDs	IID*	The list of IIDs that name the interfaces sought on this object.
cRequestedProtseqs	Unsigned short	The number of protocol sequences

		specified in the RequestedProtseqs parameter.
RequestedProtseqs that	Unsigned short[]	The list of protocol sequences the client wants OXID binding handles for.
pOxid	OXID*	The OXID for the object created.
ppdsaOxidBindings	DUALSTRINGARRAY**	Endpoint and security binding strings to reach the OXID
pipidRemUnknown	IPID*	The IPID of the OXID s IRemUnknown interface
pAuthnHint	DWORD*	A value taken from the RPC_C_AUTHN constants. A hint to the caller as to the minimum authentication level which the server will accept.
pServerVersion	COMVERSION*	The version of the COM implementation on the server.
phr	HRESULT*	The HRESULT for the activation operation
ppInterfaceData	MInterfacePointer**	The set of interface pointers requested. Returned in the order they were requested in pIIDs.
pResults	HRESULT*	The HRESULTs for the individual QueryInterface() operations performed on the interfaces specified in Interfaces.

Return Value	Meaning
RPC_S_OK	This method always returns RPC_S_OK. Check phr for the status of this function.

Value of phr	Meaning
E_UNEXPECTED	An unspecified error occurred.
E_ACCESSDENIED	Client attempted to activate but did not have the required permission or the server does not permit

	remote activations.
E_INVALIDARG	One or more arguments are invalid.
E_OUTOFMEMORY	Insufficient memory to complete the operation.

7. Security Considerations

In general, like any generic data transfer protocol, DCOM cannot regulate the
content of the data that is transferred, nor is there any a priori method of determining the sensitivity of any particular piece of information within the
context of any given ORPC.

Specifically, however, DCOM entirely leverages the security infrastructure defined by DCE RPC, which allows for various forms of authentication, authorization, and message integrity.

Further details TBS.

8. Acknowledgements

As previously noted, the DCOM protocol highly leverages the DCE RPC Specification [CAE RPC], and we again acknowledge its usefulness to this specification.

The DCOM protocol itself is the combined effort of a large number of people. The following individuals in particular were critical to the definitions which appear in this specification.

```
    Bob Atkinson                Alex Armanasu (Mitchell)
    Deborah Black               Vibhas Chandorkar
    Richard Draves              Mario Goertzel
    Rick Hill                   Gregory Jensenworth
    David Kays                  Paul Leach
    Michael Nelson              Kevin Ross
    Mark Ryland                 Bharat Shah
    Tony Williams
```

9. References

[CAE RPC] CAE Specification, X/Open DCE: Remote Procedure Call, X/Open
     Company Limited, Reading, Berkshire, UK (xopen.co.uk), 1994. X/Open
     Document Number C309. ISBN 1-85912-041-5. (also available online through
     from the OSF at <http://www.opengroup.org/tech/dce/download>)

[COM] The Component Object Model Specification, Version 0.99, November,
1996,
Microsoft Corporation. (also available online from Microsoft at
<http://www.microsoft.com/oledev/olecom/title.htm>. Note that this
reference is circular because this document (the DCOM wire protocol
specification) is the same chapter 15 of the COM Specification.

10. Author's Addresses

Nat Brown, One Microsoft Way Redmond, WA 98052-6399, U.S.A. Fax: +1 (206)
936
7329 Email: <mailto:natbro@microsoft.com>

Charlie Kindel, One Microsoft Way Redmond, WA 98052-6399, U.S.A. Fax: +1
(206) 936 7329 Email: <mailto:ckindel@microsoft.com>