[MS-IOI]:
IManagedObject Interface Protocol

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation for protocols, file formats, languages, standards as well as overviews of the interaction among each of these technologies.
- **Copyrights.** This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the technologies described in the Open Specifications and may distribute portions of it in your implementations using these technologies or your documentation as necessary to properly document the implementation. You may also distribute in your implementation, with or without modification, any schema, IDL's, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications.
- **No Trade Secrets.** Microsoft does not claim any trade secret rights in this documentation.
- **Patents.** Microsoft has patents that may cover your implementations of the technologies described in the Open Specifications. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. However, a given Open Specification may be covered by Microsoft Open Specification Promise or the Community Promise. If you would prefer a written license, or if the technologies described in the Open Specifications are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **Trademarks.** The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights. For a list of Microsoft trademarks, visit www.microsoft.com/trademarks.
- **Fictitious Names.** The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications do not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them. Certain Open Specifications are intended for use in conjunction with publicly available standard specifications and network programming art, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.

Revision Summary

Date	Revision History	Revision Class	Comments
07/20/2007	0.1	Major	MCPP Milestone 5 Initial Availability
09/28/2007	1.0	Major	Updated and revised the technical content.
10/23/2007	1.1	Minor	Updated the technical content.
11/30/2007	2.0	Major	Clarified the state requirements of .NET object versioning.
01/25/2008	2.0.1	Editorial	Revised and edited the technical content.
03/14/2008	3.0	Major	Updated and revised the technical content.
05/16/2008	4.0	Major	Updated and revised the technical content.
06/20/2008	5.0	Major	Updated and revised the technical content.
07/25/2008	6.0	Major	Updated and revised the technical content.
08/29/2008	6.0.1	Editorial	Fix capitalization issues
10/24/2008	7.0	Major	Updated and revised the technical content.
12/05/2008	8.0	Major	Updated and revised the technical content.
01/16/2009	9.0	Major	Updated and revised the technical content.
02/27/2009	9.0.1	Editorial	Revised and edited the technical content.
04/10/2009	9.0.2	Editorial	Revised and edited the technical content.
05/22/2009	10.0	Major	Updated and revised the technical content.
07/02/2009	11.0	Major	Updated and revised the technical content.
08/14/2009	12.0	Major	Updated and revised the technical content.
09/25/2009	12.1	Minor	Updated the technical content.
11/06/2009	12.1.1	Editorial	Revised and edited the technical content.
12/18/2009	12.1.2	Editorial	Revised and edited the technical content.
01/29/2010	13.0	Major	Updated and revised the technical content.
03/12/2010	13.0.1	Editorial	Revised and edited the technical content.
04/23/2010	13.0.2	Editorial	Revised and edited the technical content.
06/04/2010	14.0	Major	Updated and revised the technical content.
07/16/2010	15.0	Major	Significantly changed the technical content.

Date	Revision History	Revision Class	Comments
08/27/2010	15.0	No change	No changes to the meaning, language, or formatting of the technical content.
10/08/2010	15.0	No change	No changes to the meaning, language, or formatting of the technical content.
11/19/2010	15.0	No change	No changes to the meaning, language, or formatting of the technical content.
01/07/2011	15.0	No change	No changes to the meaning, language, or formatting of the technical content.
02/11/2011	15.0	No change	No changes to the meaning, language, or formatting of the technical content.
03/25/2011	15.0	No change	No changes to the meaning, language, or formatting of the technical content.
05/06/2011	15.0	No change	No changes to the meaning, language, or formatting of the technical content.
06/17/2011	15.1	Minor	Clarified the meaning of the technical content.
09/23/2011	15.1	No change	No changes to the meaning, language, or formatting of the technical content.
12/16/2011	16.0	Major	Significantly changed the technical content.
03/30/2012	16.0	No change	No changes to the meaning, language, or formatting of the technical content.
07/12/2012	16.0	No change	No changes to the meaning, language, or formatting of the technical content.
10/25/2012	17.0	Major	Significantly changed the technical content.
01/31/2013	17.0	No change	No changes to the meaning, language, or formatting of the technical content.
08/08/2013	17.0	No change	No changes to the meaning, language, or formatting of the technical content.

Contents

1 Introduction

The IManagedObject Interface Protocol provides interoperability support for the **common language runtime (CLR)**. The common language runtime (CLR) is a virtual machine for the execution of software. The IManagedObject interface provides a bridge between existing computer systems and the virtual execution environment.

In particular, the CLR supports interoperability with the Component Object Model (COM).<1> The CLR supports exposing its own **objects** to COM for use as native COM objects and supports consuming COM objects.

In order to determine whether a COM object that enters the CLR is actually one of its own managed objects, the **IManagedObject** interface was created to allow for the CLR to identify its own objects. The IManagedObject Interface Protocol mechanism is detailed in this specification.

Sections 1.8, 2, and 3 of this specification are normative and can contain the terms MAY, SHOULD, MUST, MUST NOT, and SHOULD NOT as defined in RFC 2119. Sections 1.5 and 1.9 are also normative but cannot contain those terms. All other sections and examples in this specification are informative.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

activation
application domain
application domain identifier (ID)
authentication level
class identifier (CLSID)
dynamic endpoint
endpoint
garbage collection
globally unique identifier (GUID)
Interface Definition Language (IDL)
interface pointer
.NET Framework
object
object class
opnum
process identifier (PID)
remote procedure call (RPC)
universally unique identifier (UUID)

The following terms are specific to this document:

common language runtime (CLR): The Microsoft implementation of the Common Language Infrastructure (CLI), as specified in [ECMA-335].

deactivation: Resetting the state of the server object instance such that a new server object instance is created when the object instance is called again by the client.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

References to Microsoft Open Specifications documentation do not include a publishing year because links are to the latest version of the documents, which are updated frequently. References to other documents include a publishing year when one is available.

A reference marked "(Archived)" means that the reference document was either retired and is no longer being maintained or was replaced with a new document that provides current implementation details. We archive our documents online [Windows Protocol].

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, http://www.opengroup.org/public/pubs/catalog/c706.htm

[ECMA-335] ECMA International, "Common Language Infrastructure (CLI) Partitions I to VI", ECMA-335, June 2006, http://www.ecma-international.org/publications/standards/Ecma-335.htm

[MS-DCOM] Microsoft Corporation, "Distributed Component Object Model (DCOM) Remote Protocol".

[MS-DTYP] Microsoft Corporation, "Windows Data Types".

[MS-ERREF] Microsoft Corporation, "Windows Error Codes".

[MS-NRBF] Microsoft Corporation, ".NET Remoting: Binary Format Data Structure".

[MS-NRTP] Microsoft Corporation, ".NET Remoting: Core Protocol".

[MS-OAUT] Microsoft Corporation, "OLE Automation Protocol".

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

[RFC3986] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005, http://www.ietf.org/rfc/rfc3986.txt

1.2.2 Informative References

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary".

[MSDN-CCW] Microsoft Corporation, "COM Callable Wrapper", http://msdn.microsoft.com/en-us/library/f07c8z1c.aspx

[MSDN-RCW] Microsoft Corporation, "Runtime Callable Wrapper", http://msdn.microsoft.com/en-us/library/8bwh56xe.aspx

[MSDN-.NETFROVW] Microsoft Corporation, ".NET Framework Remoting Overview", http://msdn.microsoft.com/en-us/library/kwdt6w2k

[MSFT-DCOMTECHOVW] Microsoft Corporation, "DCOM Technical Overview", http://technet.microsoft.com/en-us/library/cc722925.aspx

1.3 Overview

The **IManagedObject** interface is a COM interface used by the common language runtime (CLR) to identify managed objects (objects created by the CLR) that are exported for interoperability with the Component Object Model (COM). The **IManagedObject** interface allows these objects to be identified when they reenter the CLR.

The **IManagedObject** interface is used specifically for scenarios in which managed code uses COM and interacts with a managed object. This interface is an optimization that allows managed code to avoid going through COM to interact with the managed object. There are two different scenarios in which this can occur: Either the managed object is within the same process division, the same **application domain**, or the managed object is in a different process division (application domain). In either case, this document discusses what is done instead of using DCOM [MS-DCOM] to interact between the CLR and managed objects.

When using COM, the COM Callable Wrapper (CCW) is the view of the object to COM, as defined in [MSDN-CCW]. When the CLR identifies a COM object that includes a CCW, a Runtime Callable Wrapper (RCW) is required in order to interact with the COM object, as defined in [MSDN-RCW]. If an RCW doesn't exist, the CLR attempts to create an RCW. If the object implements **IManagedObject**, the CLR determines that it is a .NET object. For more information on CCW and RCW, see [MSDN-CCW] and [MSDN-RCW].

In cases in which the .NET object is in the same process division, the CLR interacts directly with the .NET object.

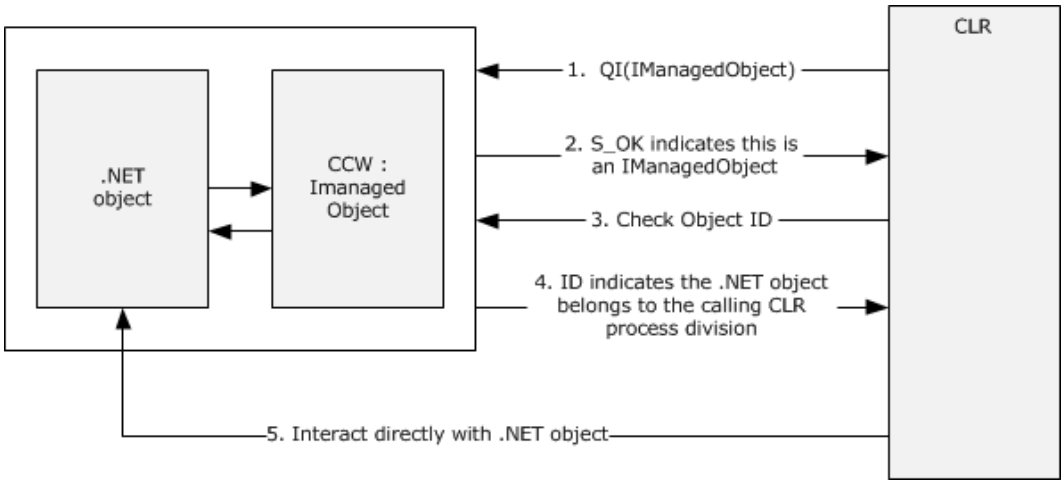

Figure 1: CLR interacts directly with .NET object

The steps describing the CLR and the .NET object interaction are as follows:

1. A COM object, which is a CCW wrapping a local CLR object, reenters the CLR through a COM call. The CLR calls the **QueryInterface** method on the CCW for the **IManagedObject** interface in order to determine whether this COM object is a CCW or not.

2. Since the COM object is a CCW, it returns the **IManagedObject** call with S_OK and a pointer to its **IManagedObject** interface.

3. The CLR calls **IManagedObject::GetObjectIdentity** on the **IManagedObject** interface obtained in step 2 in order to determine if the object is local to the current process and application domain.

4. The CCW responds back with its ID, and the CLR notes that this ID is local to the current process and application domain.

5. As established in step 4, the wrapped object belongs to this instance of the CLR, and the CLR can interact with the object directly instead of going through an RCW / CCW pair and communicating over a COM channel.

In cases in which the .NET object is in a different process division (different application domain or process), a remoting proxy is used to interact with the .NET object. In such a case, the CCW implementing **IManagedObject** returns a **Server Object Identity** / *AppDomainID* from **IManagedObject::GetObjectIdentity** that does not correspond to the current process. The CLR will then ask for the remoting ID for the object (**IManagedObject::GetSerializedBuffer**). This can be used to generate a transparent remoting proxy to communicate to the original object. At this point, the communication endpoints are now remoting .NET objects, and CCWs / RCWs are not used:

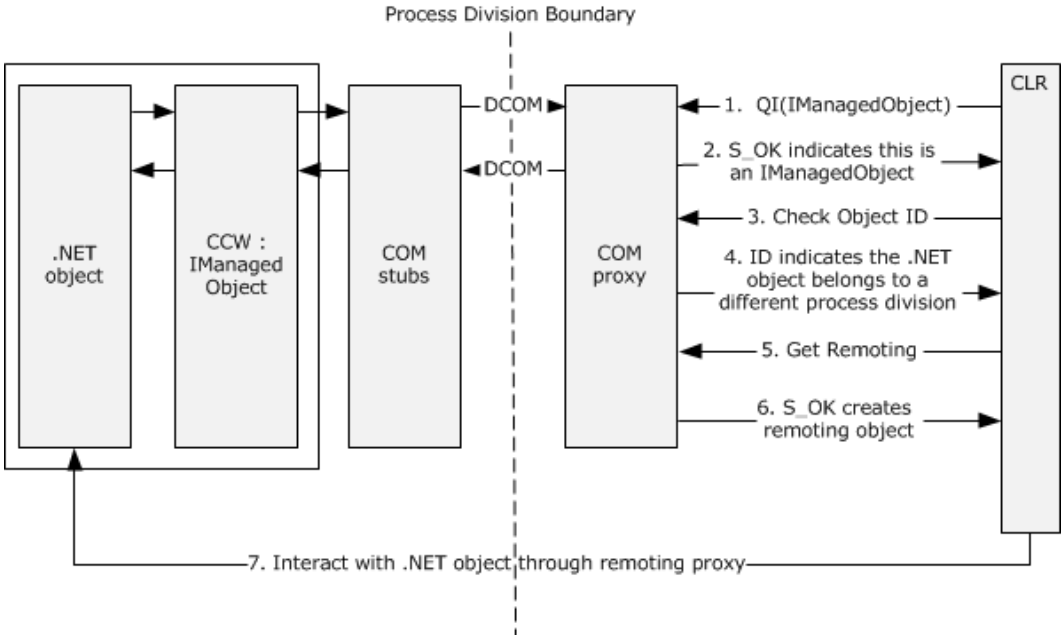

Figure 2: CLR interacts via remoting proxy

The steps describing the CLR interaction via remoting proxy are as follows:

1. A COM object, which is a CCW wrapping a CLR object, from a different application domain or process enters the CLR through a COM call. The CLR calls the **QueryInterface** method on the CCW for the **IManagedObject** interface in order to determine whether this COM object is a CCW or not.

2. Because the COM object is a CCW, it responds to the **QueryInterface** call with S_OK and a pointer to its **IManagedObject** interface.

3. The CLR calls **IManagedObject::GetObjectIdentity** on the **IManagedObject** interface obtained in step 2 in order to determine whether the object is local to the current process and application domain.

4. The CCW responds back with its ID, and the CLR notes that this ID is not local to the current process and application domain.

5. The CLR calls **IManagedObject::GetSerializedBuffer** to get information to set up a .NET Remoting connection to the remote CLR object.

6. The CCW responds back with the remoting information.

7. The CLR uses the remoting information obtained in step 6 to create a remoting proxy that communicates with the .NET object via .NET Remoting, rather than using an RCW/CCW pair and communicating over a COM channel.

CLR-managed objects can be exposed to COM clients as COM objects. They can implement any number of COM interfaces, but all such exported objects implement **IManagedObject**.

The CLR also allows COM objects to be imported and used as managed objects. In this case, **IManagedObject** is used to determine if an object is truly a COM object or if it is actually originated as a CLR-managed object.

When a COM object enters the CLR, the CLR uses the standard COM interface querying mechanism (**QueryInterface**) to determine if the given object implements **IManagedObject**. If the object supports **IManagedObject**, **IManagedObject::GetObjectIdentity** is called.

At CLR instantiation, the CLR creates a unique **GUID** to identify a specific CLR instance within a given process. This GUID is formatted as a string ([MS-DTYP] section 2.3.4.3) and is saved. All CLR-managed objects originating from this specific instance of the CLR will return this unique identifier as the first parameter of the call to **IManagedObject::GetObjectIdentity**. This GUID is used to recognize that an imported managed object originated in this runtime.

The CLR can support even finer-grained levels of grouping than the process. Objects exported from a given process division are tagged and return the identifier used for process division in their second parameter to **IManagedObject::GetObjectIdentity**. This identifier is also used to indicate whether or not the given object originated in the correct process division. If the process identifier and process division match, the last parameter of **IManagedObject::GetObjectIdentity** is a pointer to the implementation-specific representation of the managed object.

If the given object does not match the current CLR instance and process division, **IManagedObject::GetSerializedBuffer** is called to return a binary representation of a managed object, as specified by the .NET Remoting: Binary Format Data Structure [MS-NRBF]. It is the responsibility of the caller on the client CLR to interpret the deserialized opaque object reference.

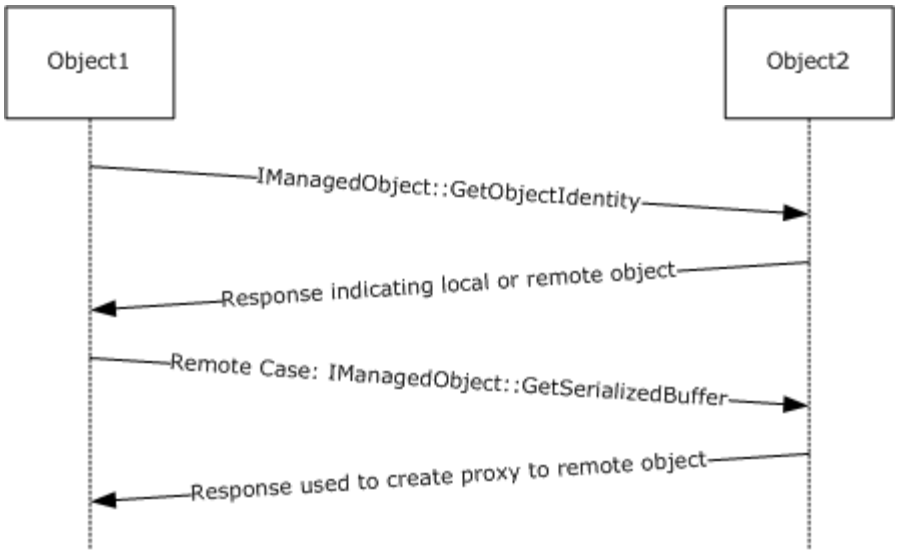

Figure 3: IManagedObject request-response

1.3.1 IRemoteDispatch Interface and IServicedComponentInfo Interface

A server object instance can associate a unique identity with itself. This identity can be used by the client to track multiple instances of the server object. The server can use **IServicedComponentInfo** to allow the client to query for its identity.

The **IRemoteDispatch** interface can be used by the server to provide an alternative way to dispatch method calls on its object instance. A client can further use this interface to perform **deactivation** of the server object instance.

1.4 Relationship to Other Protocols

This protocol uses the OLE Automation Protocol [MS-OAUT], making use of the **BSTR** and **VARIANT** types from the **IDispatch** interface.

The **IManagedObject** interface uses the Distributed Component Object Model (DCOM) Remote Protocol [MS-DCOM].

The **IRemoteDispatch** and **IServicedComponentInfo** interfaces use DCOM [MS-DCOM] to communicate over the wire and to authenticate all requests issued against the infrastructure.

This protocol allows for encodings defined in [MS-NRTP] and [MS-NRBF].

1.5 Prerequisites/Preconditions

This protocol requires the Distributed Component Object Model (DCOM) Remote Protocol [MS-DCOM] and the OLE Automation Protocol [MS-OAUT]. This protocol requires the CLR to be installed on the client machine.

All interfaces assume that the client is in possession of valid credentials recognized by the server that is accepting the client requests.

This protocol assumes that the client has relied on **QueryInterface** to determine if the server supports the **IManagedObject** interface or the **IRemoteDispatch** interface. The protocol also

assumes that the client has the server object Microsoft .NET Framework type information prior to initialization.

1.6 Applicability Statement

IManagedObject is useful as part of the infrastructure for allowing the CLR to interoperate with COM.

Interoperability between the CLR and COM offers the following benefits.

- Existing COM objects can be used from the CLR.
- Managed objects created in the CLR can be used from existing COM applications.
- The managed identity of an object is not lost when it is passed out to COM and then back to the CLR.

The **IRemoteDispatch** interface is used for method call dispatch and deactivation.

The **IServicedComponentInfo** interface is used for determining server object instance identity.

1.7 Versioning and Capability Negotiation

Supported Transports: This protocol uses the DCOM Remote Protocol as its transport, as specified in [MS-DCOM].

Protocol Version: The **IManagedObject** protocol consists of one DCOM interface, **IManagedObject** version 0.0. The interfaces defined in this specification have no versioning or capability negotiation beyond those of the underlying transport.

For both of these interfaces, it is assumed that the client has the Microsoft .NET Framework information (such as type information of server objects) prior to initialization.

The client relies on **QueryInterface** to determine if the server supports **IManagedObject** or **IRemoteDispatch**.

1.8 Vendor-Extensible Fields

This protocol uses **universally unique identifiers (UUIDs)**. Vendors can create their own UUIDs, as described in [MS-DTYP] section 2.3.4.

This protocol uses **HRESULT** values as defined in [MS-DTYP] section 2.2.18. Vendors can define their own **HRESULT** values, provided that they set the *C* bit (0x20000000) for each vendor-defined value, indicating that the value is a customer code.

This protocol uses Win32 error codes. These values are taken from the Windows error number space, as specified in [MS-ERREF] section 2.2. It is recommended that vendors reuse those values with their indicated meaning. Choosing any other value runs the risk of a collision in the future.

1.9 Standards Assignments

Constant/value	Description
IManagedObject {C3FCC19E-A970-11D2-8B5A-00A0C9B7C9C4}	The GUID associated with the **IManagedObject** interface.

Constant/value	Description
IRemoteDispatch {6619a740-8154-43be-a186-0319578e02db}	The GUID associated with the **IRemoteDispatch** interface.
IServicedComponentInfo {8165B19E-8D3A-4d0b-80C8-97DE310DB583}	The GUID associated with the **IServicedComponentInfo** interface.

2 Messages

2.1 Transport

This protocol uses **RPC dynamic endpoints** ([C706] Part 4) and DCOM [MS-DCOM].

To access an interface, the client MUST request a DCOM connection to its well-known object UUID **endpoint** on the server, as specified in section 1.9.

The RPC version number for all interfaces MUST be 0.0.

2.2 Common Data Types

This protocol MUST indicate to the RPC runtime that it is to support the **NDR** transfer syntax only, as specified in [C706] part 4.

In addition to RPC base types and definitions specified in [C706] and [MS-DTYP], additional data types are defined in the following subsection.

2.2.1 CCW_PTR

CCW_PTR is an opaque pointer that is up to the implementation to interpret.

The wire representation will consist of the pointer representation used for the transfer syntax in use (either NDR or NDR64) followed by the wire representation of the content of the **int** or **__int64**.

The pointer representation used in NDR transfer syntax is 4 octets in length. NDR transfer syntax is specified in [C706] Chapter 14, section 14.3.10. The pointer representation used in NDR64 transfer syntax is specified in [MS-RPCE] 2.2.5.3.5, and is 8 octets in length.

If _64BIT is defined, the pointer representation will be followed by the wire representation of the **__int64**. The wire representation of **__int64** is specified in [MS-RPCE] 2.2.4.1.3. The **__int64** is synonymous to **hyper** in [C706], which is 8 octets in length.

If _64BIT is not defined, the pointer representation will be the same, but followed by the wire representation of **int**. This **int** will be treated as a long, as specified in [C706] section 14.2.5, and is 4 octets in length.

No negotiation occurs to determine whether _64BIT has been defined. The wire syntax for **CCW_PTR** can be made consistent using any implementation-specific method.

The type definition of **CCW_PTR** is as follows.

```
#ifdef _64BIT

typedef __int64* CCW_PTR;

#else

typedef int* CCW_PTR;

#endif
```

3 Protocol Details

The following sections specify details of the IManagedObject Interface Protocol, including abstract data model, interface method syntax, and message processing rules.

The **IRemoteDispatch** and **IServicedComponentInfo** client applications initiate the conversation with the server by performing DCOM **activation** ([MS-DCOM] section 3.2.4.1.1) of an application-specific **CLSID** of an object that supports these interfaces. After the client application uses activation to get the **interface pointer** to the DCOM object, it works with this object by making calls on the DCOM interface supported by the object. After it has finished making calls, the client application does a release on the interface pointer.

3.1 IManagedObject Server Details

A CLR-managed object that has been exposed to COM will expose the COM interface **IManagedObject**. This interface is used to determine whether COM objects that enter the CLR are actually CLR-managed objects and can be mapped directly to the managed object. This allows CLR-managed objects to roundtrip from managed to COM and back to managed while maintaining their original identity.

3.1.1 Abstract Data Model

The CLR implementation that exposes objects to COM MUST maintain a unique UUID to differentiate its objects from those of other CLR instances and implementations. If the CLR supports per-process divisions, it will also need to maintain unique identifiers for each division to map objects back to their originating process division. In addition, the CLR will also need to use an opaque identifier that is used to map back internally from the COM interface pointer to **IManagedObject** to the underlying managed object.

In the case that the CLR-managed object does not belong to the given CLR process instance and process subdivision, the implementation of **IManagedObject** MUST provide a mechanism that returns a binary-formatted version of the underlying managed object.

Server Object Identity: The remote server object instance MUST have a unique Uniform Resource Identifier (URI), as specified in [RFC3986]. This URI represents the unique identity of the server object instance. The client uses this identity to track multiple instances of the server object.

3.1.2 Timers

None.

3.1.3 Initialization

The server MUST create a unique UUID to identify this CLR instance upon startup. Upon the startup of each process division, a unique identifier also needs to be generated.

3.1.4 Message Processing Events and Sequencing Rules

3.1.4.1 IManagedObject

The **IManagedObject** interface includes the following methods.

The client MUST be implemented with the type information for the remote object.<2>

Methods in RPC Opnum Order

Method	Description
GetSerializedBuffer	Returns a binary-formatted representation of a managed object, as specified in [MS-NRBF] section 2.3. Opnum: 3
GetObjectIdentity	Used to determine if a COM object is a managed object that belongs to this CLR instance and process subdivision. Opnum: 4

3.1.4.1.1 GetSerializedBuffer (Opnum 3)

The **GetSerializedBuffer** method converts the given managed object to a binary-formatted string representation that can be used to create a managed object.

```
HRESULT GetSerializedBuffer(
  [out] BSTR* pbSTR
);
```

pbSTR: The value MUST contain a binary-formatted string representation of the class record for the underlying managed object, as specified in [MS-NRBF] section 2.3. For more information on binary format mapping, see [MS-NRTP] section 3.1.5.1.

Return Values: The method MUST return a positive value or 0 to indicate successful completion or a negative value to indicate failure.

Return value/code	Description
0x00000000 ERROR_SUCCESS	Success.

Exceptions Thrown: No exceptions are thrown from this method beyond those thrown by the underlying RPC protocol.

3.1.4.1.2 IManagedObject::GetObjectIdentity (Opnum 4)

The **IManagedObject::GetObjectIdentity** method is used by a CLR instance to determine whether a COM object entering the system is really a managed object that originated in this CLR instance and within the current process division.

```
HRESULT GetObjectIdentity(
  [out] BSTR* pBSTRGUID,
  [out] int* AppDomainID,
  [out] CCW_PTR pCCW
);
```

pBSTRGUID: The *pBSTRGUID* parameter is a GUID ([MS-DTYP] section 2.3.4.3). The *pBSTRGUID* parameter MUST indicate the CLR instance in which this object was created.

AppDomainID: Optional parameter that contains implementation-specific, opaque, process-unique identifiers. If present, the *AppDomainID* parameter MUST denote the process subdivision in which this object resides.

pCCW: Optional field. Implementation-specific, opaque value that helps identify the managed object. If present, this field MUST map back to the implementation's internal representation of a managed object.

Return Values: The method MUST return a positive value or 0 to indicate successful completion or a negative value to indicate failure.

Return value/code	Description
0x00000000 ERROR_SUCCESS	Success

Exceptions Thrown: No exceptions are thrown from this method beyond those thrown by the underlying RPC protocol.

3.1.4.2 IRemoteDispatch Interface

The **IRemoteDispatch** interface provides methods to dispatch calls on the server object. A client can optionally use this interface to deactivate the server object instance after the method call completes. The interface inherits **opnums** 0 to 6 from **IDispatch** ([MS-OAUT] section 3.1.4). The version for this interface is 0.0.

To receive incoming remote calls for this interface, the server MUST implement a DCOM **object class** that supports this interface by using the UUID {6619a740-81c4-43be-a186-0319578e02db} for this interface.

The client MUST be implemented with the type information for the remote object.

The interface includes the following methods beyond those in **IDispatch**.<3>

Methods in RPC Opnum Order

Method	Description
RemoteDispatchAutoDone	Invokes a call on the server object and deactivates it when the call completes. Opnum: 7
RemoteDispatchNotAutoDone	Invokes a call on the server object without deactivating it when the call completes. Opnum: 8

3.1.4.2.1 RemoteDispatchAutoDone (Opnum 7)

The **RemoteDispatchAutoDone** method is called by the client to invoke a method on the server.

```
[id(0x60020000)] HRESULT RemoteDispatchAutoDone(
  [in] BSTR s,
  [out, retval] BSTR* pRetVal
);
```

s: The *s* parameter contains binary data representing the input parameters of the method called on the server. The binary data MUST be marshaled as specified in [MS-NRTP] section 3.1.5.1.1. The data is specified as is in the **BSTR**, such that the length of the **BSTR** is the size of the data divided by 2 (rounded up if necessary).

pRetVal: The *pRetVal* parameter contains the binary data representing the output parameters of the method called on the server. The binary data MUST be marshaled as specified in [MS-NRTP] section 3.1.5.1.1. The data is specified as is in the **BSTR**, such that the length of the **BSTR** is the size of the data divided by 2 (rounded up if necessary).

Return Values: An **HRESULT** that specifies success or failure. All success **HRESULT** values MUST be treated as success and all failure **HRESULT** values MUST be treated as failure.

When this method is invoked, the server MUST unmarshal the method input parameters and formulate a method call request. If the payload is a valid method call request for the given server object instance, the server MUST dispatch the method on the server object instance. Otherwise it MUST fail the call. After the server object instance completes the method call, the server MUST marshal the output parameters as specified in [MS-NRTP] section 3.1.5.1.1, and return the encoded reply through the *pRetVal* argument. It MUST then deactivate the instance of the server object that services the call.

3.1.4.2.2 RemoteDispatchNotAutoDone (Opnum 8)

The **RemoteDispatchNotAutoDone** method is called by the client to invoke a method on the server.

```
[id(0x60020001)] HRESULT RemoteDispatchNotAutoDone(
  [in] BSTR s,
  [out, retval] BSTR* pRetVal
);
```

s: The *s* parameter contains binary data representing the input parameters of the method called on the server. The binary data MUST be marshaled as specified in [MS-NRTP] section 3.1.5.1.1. The data is specified as is in the **BSTR**, such that the length of the **BSTR** is the size of the data divided by 2 (rounded up if necessary).

pRetVal: The *pRetVal* parameter contains the binary data representing the output parameters of the method called on the server. The binary data MUST be marshaled as specified in [MS-NRTP] section 3.1.5.1.1. The data is specified as is in the **BSTR**, such that the length of the **BSTR** is the size of the data divided by 2 (rounded up if necessary).

Return Values: An **HRESULT** that specifies success or failure. All success **HRESULT** values MUST be treated as success and all failure **HRESULT** values MUST be treated as failure.

When this method is invoked, the server MUST unmarshal the method input parameters and formulate a method call request. If the payload is a valid method call request for the given server object instance, the server MUST dispatch the method on the server object instance. Otherwise it MUST fail the call. After the server object instance completes the method call, the server MUST marshal the output parameters as specified in [MS-NRTP] section 3.1.5.1.1 and return the encoded reply through the *pRetVal* argument.

3.1.4.3 IServicedComponentInfo Interface

The **IServicedComponentInfo** interface is used to get the object identity of the server object instance that supports this interface. Because this is a DCOM interface, opnum 0 to opnum 2 are

IUnknown methods, as specified in [MS-DCOM] section 3.1.1.5.8. The version for this interface is 0.0.

To receive incoming remote calls for this interface, the server MUST implement a DCOM object class that supports this interface by using the UUID {8165B19E-8D3A-4d0b-80C8-97DE310DB583} for this interface.

The interface contains the following methods beyond those of IUnknown.

Methods in RPC Opnum Order

Method	Description
GetComponentInfo	Gets the server object identity associated with the server object instance. Opnum: 3

3.1.4.3.1 GetComponentInfo (Opnum 3)

The **GetComponentInfo** method is used to determine the environment of the server object.

```
HRESULT GetComponentInfo(
  [in, out] int* infoMask,
  [out] SAFEARRAY(BSTR)* infoArray
);
```

infoMask: A bitwise OR of zero of more of the following values:

Value	Meaning
0x00000001	The serviced component's **process identifier (PID)**.
0x00000002	The serviced component's **application domain identifier (ID)**.
0x00000004	The serviced component's remote **URI** [RFC3986], which represents the server object identity.

On input, the bits set indicate the information the client is requesting that the server return. On output, the bits set indicate the information actually returned in the *infoArray*.

infoArray: An array that contains a set of values returned by the server corresponding to the bits set in *infoMask*.

Return Values: An **HRESULT** that specifies success or failure. All success **HRESULT** values MUST be treated as success and all failure **HRESULT** values MUST be treated as failure.

When this method is invoked, the server MUST do the following. If any bits not defined above in *infoMask* are set, the server first MUST update *infoMask* to clear those bits.

The server MUST return in *infoArray* a **SAFEARRAY** ([MS-OAUT] section 2.2.30.10) of type VT_BSTR. This **SAFEARRAY** MUST contain, in order, the (possibly empty) subset of the following items, corresponding to the bits set in *infoMask*.

- Process ID
- Application Domain ID

- The serviced component's remote URI [RFC3986]

The type of each element MUST be a **BSTR** ([MS-OAUT] section 2.2.23.2). The server then MUST return success.

3.1.5 Timer Events

None.

3.1.6 Other Local Events

There are no protocol-specific local events.

3.2 IManagedObject Client Details

3.2.1 Abstract Data Model

The client is essentially the same as the server. The **IManagedObject** interface is used to identify CLR-mapped COM objects after they are exported to COM and returned as COM objects. Implementation of the **IManagedObject** class denotes that a given COM object is really a CLR-managed COM object. The methods of **IManagedObject** are used to determine if the COM object lives in this CLR instance and process subdivision. These methods will otherwise return a CLR-managed object underlying the COM object. The deserialized opaque object reference is returned to the caller on the client CLR for interpretation.

3.2.2 Timers

There are no protocol-specific timers.

3.2.3 Initialization

The initialization is the same as for server. See section 3.1.3.

3.2.4 Message Processing Events and Sequencing Rules

The client determines if it is the server by matching values returned from the **IManagedObject::GetObjectIdentity** method. The client matches the value of **pBSTRGUID** against the GUID of the client CLR instance and uses the implementation-specific helper values in **AppDomainID** and **pCCW** to help decide if the COM object originated in the client CLR instance. Otherwise, the client fetches a binary-formatted string representation ([MS-NRBF] section 2.3) to the underlying managed object by using **GetSerializedBuffer**, which can be used to create a managed object.

When the IServiceComponentInfo interface is used, the client determines the server instance identity by means of the URI returned from the IServiceComponentInfo::GetComponentInfo method.

Note the following:

- A COM reference to a CCW, as defined in [MSDN-CCW], keeps the managed object rooted for the **garbage collection** routine.
- Managed object references also keep the object rooted for the garbage collection.

- The garbage collection only collects an object once there are no longer COM object references to it and there are no further managed references to it.
- In the event that the application domain that hosts the object is torn down, the CCW that implements **IManagedObject** is kept alive by the CLR, but calls to it will return failure **HRESULT**.

3.2.5 Timer Events

There are no protocol-specific timers.

3.2.6 Other Local Events

None.

4 Protocol Examples

4.1 Using the IManagedObject Interface

A CLR instance uses the **IManagedObject** interface in the following manner.

1. A CLR instance starts up and generates a UUID to uniquely identify itself. It later creates a process subdivision and creates a unique value to identify the process subdivision.

2. A COM object enters the CLR, and the CLR then calls the **IUnknown::QueryInterface** method to determine whether the object implements **IManagedObject**. The object returns S_OK and returns a pointer to an **IManagedObject** interface pointer.

3. The CLR then calls the **IManagedObject::GetObjectIdentity** method and matches the *pBSTRGUID* against its UUID and, if they match, compares the *AppDomainID* to the identifier of the current process subdivision. If they match, the CLR converts *pCCW* to the underlying CLR-managed object. If they do not match, it calls the **IManagedObject::GetSerializedBuffer** method and uses the binary-formatted version of the object converted to a string to create a copy of the object that resides in another CLR (which could be a completely different implementation). The caller on the client CLR is then able to interpret the deserialized opaque object reference. For more information about how to create the binary-formatted string representation of an object, see [MS-NRBF] section 2.3.

4.2 Determining Server Object Identity

This example assumes that the client already has an interface pointer to an instance of an object that implements **IServicedComponentInfo**. The example also assumes that the server already has a unique identifier encoded into a URI to identify the server object instance. The following diagram helps to illustrate this example.

Figure 4: Call sequence for determining server object identity

1. The client calls the **IServicedComponentInfo**::**GetComponentInfo** method.

```
HRESULT
GetComponentInfo{
    [in,out] int* infoMask = 0x00000004,
    [out] SAFEARRAY(BSTR)* infoArray = {An uninitialized pointer to
        receive the SAFEARRAY}};
```

2. The server receives the call, verifies the parameters, and returns a **SAFEARRAY** of type **VT_BSTR** into the *infoArray* that contains the URI for the server object instance.

```
HRESULT = S_OK
GetComponentInfo{
    [in,out] int* infoMask = 0x00000004,
    [out] SAFEARRAY(BSTR)* infoArray = {VT_BSTR,
        "http://56C8D6F0ED8B4d658F42148430C65CEE" }};
```

3. The client records the URI of the server object instance, and uses it to distinguish between two different server object instances.

4.3 Dispatching a Call on the Server Using Deactivate

This example assumes that the client already has an interface pointer to an instance of an object that implements **IRemoteDispatch**. The following diagram helps to illustrate this example. The scenario is around a remote server object which has a method named **Method**. This method takes two string parameters. The first parameter *a* is an in only parameter, the second parameter *b* is the out only parameter.

```
Method(string a, out string b);
```

The call graph for this on the client side is given below.

Figure 5: Call sequence for dispatching a call with Deactivate from client to server

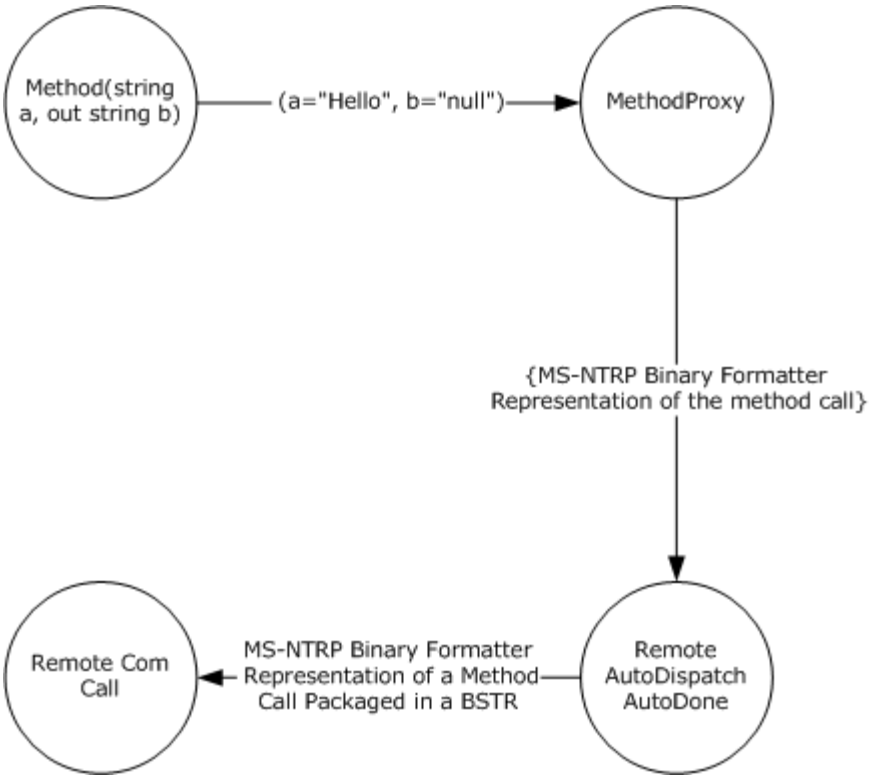

Figure 6: Call graph on the client side for the given scenario

1. The client accepts a call for "**Method**" taking two parameters *a* and *b*.

2. The client transforms the method call into a byte representation as specified in [MS-NRTP] section 3.1.5.1.1.

3. The client then takes the byte representation and passes it as a **BSTR** in the *s* parameter over **RemoteDispatchAutoDone**.

   The representative **BSTR** for the current example would look like the following.

```
00000000 00000000 00000100 00000000  ................
00121500 06120000 6874654d 5112646f  ........Method.Q
74736554 706d6f43 6574202c 202c7473  TestComp, test,
73726556 3d6e6f69 2e302e30 2c302e30  Version=0.0.0.0,
6c754320 65727574 75656e3d 6c617274   Culture=neutral
7550202c 63696c62 5479654b 6e656b6f  , PublicKeyToken
3030313d 66663066 65643064 34336662  =100f0ffd0debf34
00000233 48051200 6f6c6c65 00000b11  3......Hello....
00650073 00000073 78adbf68 00000642  s.e.s...h..xB...
61df552e 800000d1 0000017e 00000006  .U.a....~.......
00000000 00001771 00000002 76726553  ....q.......Serv
20656369 6b636150 202c3120 36322e76  ice Pack 1, v.26
00000037 00000002 0000004c 00000009  7.......L.......
00000002 00000059 00000013 00000004  ....Y...........
0000005d 00000015 00000002 0000006a  ]...........j...
0000001f 00000004 0000006e 00000021  ........n...!...
```

4. The server receives the call and transforms the incoming **BSTR** in parameter *s* into a method call. The server then dispatches the method call on the object.

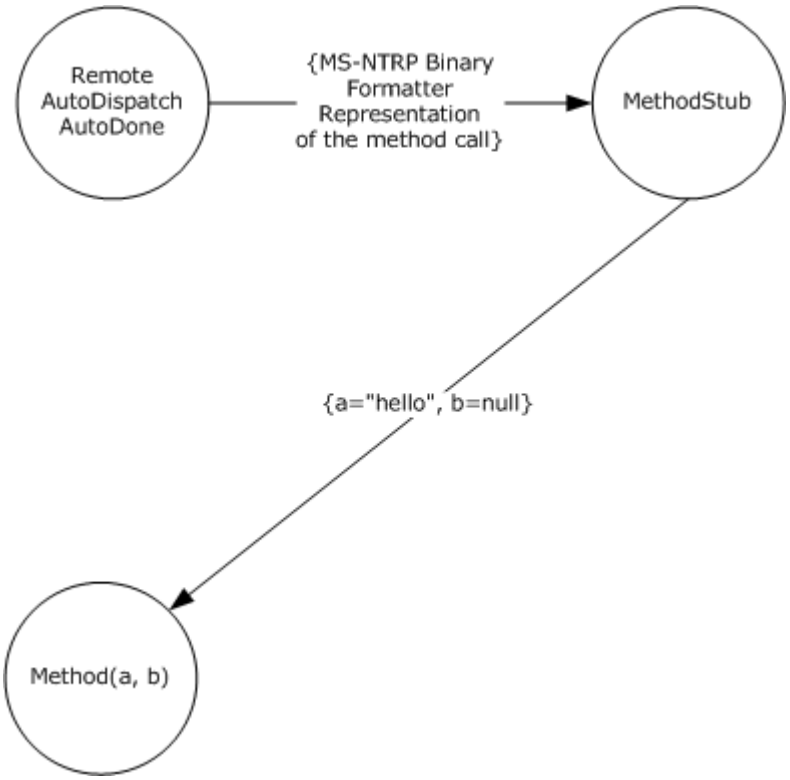

Figure 7: Call graph on the server side for the given scenario.

5. "**Method**" completes. On completion, it populates "**World**" into the out parameter *b*.

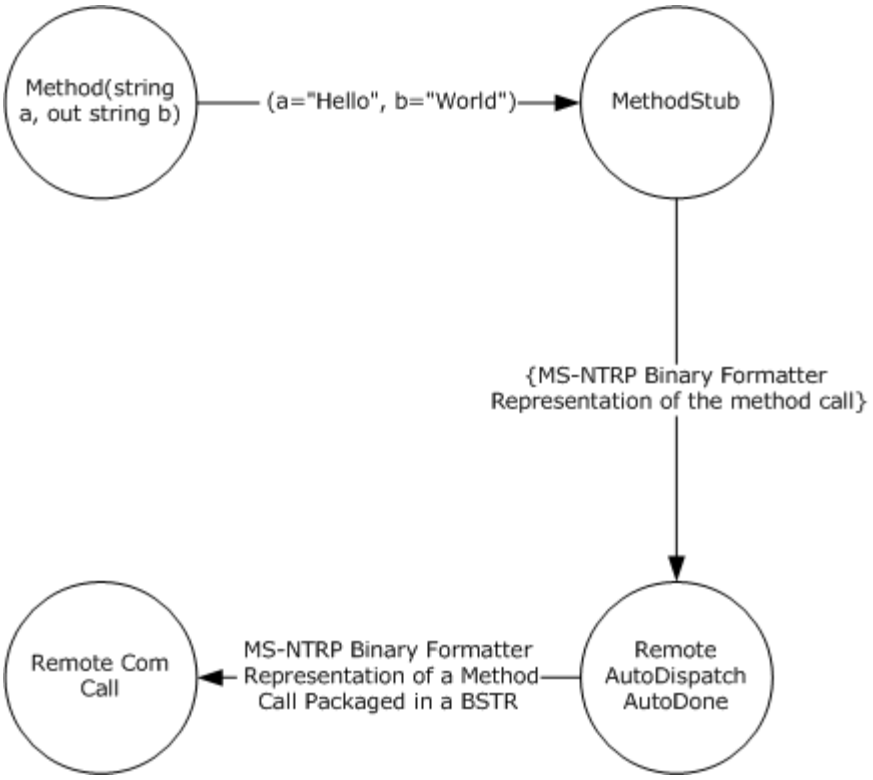

Figure 8: Return call graph on the server side for the given scenario.

6. The server now takes the return call and packages it again into a byte representation as specified in [MS-NRTP] section 3.1.5.1.1.

   RemoteDispatchAutoDone implementation takes the binary representation and makes it a payload for **BSTR** out parameter *pRetVal*.

   The representative **BSTR** for the current example would look like the following.

```
00000000 00000000 00000100 00000000  ................
04121600 00020000 12110000 726f5705  ............Wor
000b646c 00000000 0020f490 00000000  ld..... .....
61df71bf 800000d1 0000003a 00000000  .q.a....:.......
00000000 00000000 ff6c5db0           .........]l.
```

7. **RemoteDispatchAutoDone** implementation deactivates the server object by releasing all references to it such that the particular instance of the server is destroyed.

8. As the client call returns the data received on the client side in the *pRetVal* **BSTR** parameter is converted back to a return call parameters and the control is transferred back to the caller.

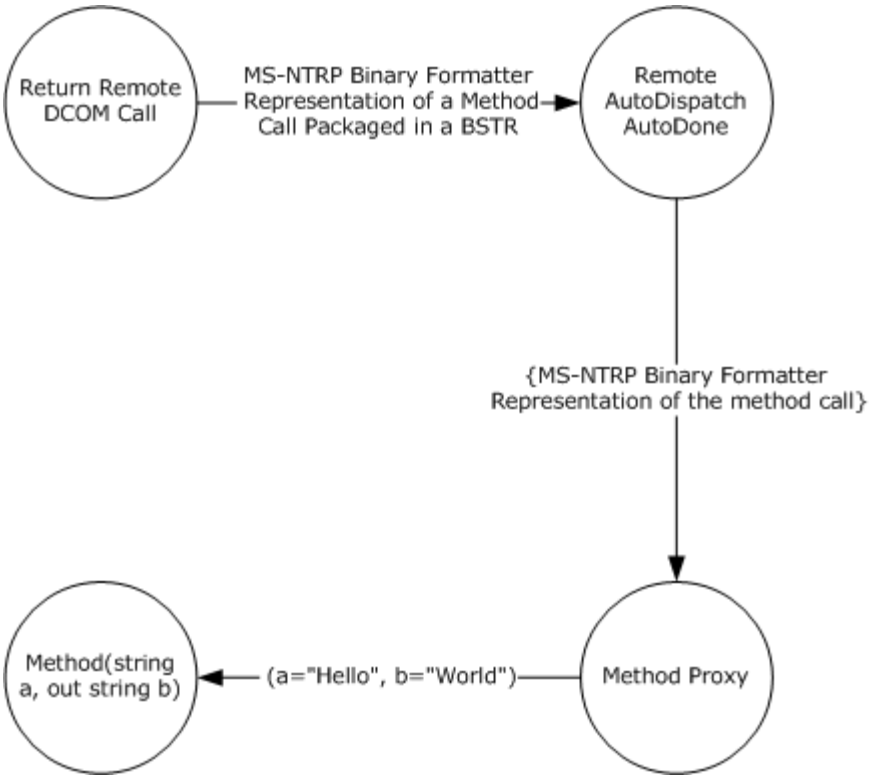

Figure 9: Return call graph on the client side for the given scenario.

5 Security

5.1 Security Considerations for Implementers

None.

5.2 Index of Security Parameters

None.

6 Appendix A: Full IDL

For convenience, the full **IDL** is provided with this specification.

```
import "ms-oaut.idl";

#ifdef _64BIT

typedef __int64* CCW_PTR;

#else

typedef int* CCW_PTR;

#endif

#define SAFEARRAY(type) SAFEARRAY

[
    object,
    oleautomation,
    uuid(C3FCC19E-A970-11d2-8B5A-00A0C9B7C9C4),
helpstring("Managed Object Interface"),
    pointer_default(unique)
]
interface IManagedObject : IUnknown
{
    HRESULT GetSerializedBuffer( [out] BSTR *pBSTR);

    HRESULT GetObjectIdentity([out] BSTR* pBSTRGUID, [out] int* AppDomainID, [out] CCW_PTR
pCCW);
};


[
object,
uuid(6619a740-8154-43be-a186-0319578e02db),
helpstring("RemoteDispatch Interface"),
dual,
pointer_default(unique)
]
interface IRemoteDispatch: IDispatch
{
[id(0x60020000)]
HRESULT RemoteDispatchAutoDone([in] BSTR s, [out, retval] BSTR* pRetVal);
[id(0x60020001)]
HRESULT RemoteDispatchNotAutoDone([in] BSTR s, [out, retval] BSTR* pRetVal);
};



[
object,
uuid(8165B19E-8D3A-4d0b-80C8-97DE310DB583),
helpstring("ServicedComponentInfo Interface"),
pointer_default(unique)
]
interface IServicedComponentInfo : IUnknown{
```

```
  HRESULT GetComponentInfo([in,out] int* infoMask, [out] SAFEARRAY(BSTR)* infoArray);
  };
```

7 Appendix B: Product Behavior

This document specifies version-specific details in the Microsoft .NET Framework. For information about which versions of .NET Framework are available in each released Windows product or as supplemental software, see **.NET Framework**.

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft .NET Framework 1.0
- Microsoft .NET Framework 2.0
- Microsoft .NET Framework 3.0
- Microsoft .NET Framework 3.5
- Microsoft .NET Framework 4.0
- Microsoft .NET Framework 4.5

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

<1> Section 1: The Microsoft common language runtime (CLR) and Component Object Model (COM) are capable of interoperating over the IManagedObject Interface Protocol mechanism. For more information on DCOM and remoting, see [MSFT-DCOMTECHOVW] and [MSDN-.NETFROVW].

<2> Section 3.1.4.1: On the Windows platform, the **IManagedObject** interface is implemented by all .NET Framework components when they are exposed through the COM-Interop feature of the .NET Framework.

<3> Section 3.1.4.2: On the Windows platform, the **IRemoteDispatch** interface is exposed only by components inheriting from **System.EnterpriseServices.ServicedComponent**.

8 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

9 Index

V