# [MS-IOI]:
# IManagedObject Interface Protocol Specification



## 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. |

| Date | Revision History | Revision Class | Comments |
|---|---|---|---|
| 01/25/2008 | 2.0.1 | Editorial | Revised and edited the technical content. |

# Table of Contents

# 1 Introduction

The **common language runtime (CLR)** is a Microsoft virtual machine for the execution of software. To bridge between existing computer systems and the virtual execution environment, the CLR supports interoperability via the IManagedObject interface.

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

In order to determine if 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. This IManagedObject mechanism is detailed in this document.

## 1.1 Glossary

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

**Activation**
**Authentication Level**
**CLSID**
**Dynamic Endpoint**
**Endpoint**
**Globally Unique Identifier (GUID)**
**Interface Definition Language (IDL)**
**Interface Pointer**
**IRemUnknown**
**IUnknown**
**Object**
**Object Class**
**Opnum**
**Remote Procedure Call (RPC)**
**Universally Unique Identifier (UUID)**

The following terms are specific to this document:

**Common Language Runtime (CLR):** See Common Language Infrastructure (CLI), as specified in [ECMA-335].

**Deactivation:** Resetting the state of the server object instance such that it is recreated 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

### 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 Specification", March 2007.

[MS-DTYP] Microsoft Corporation, "Windows Data Types", January 2007.

[MS-ERREF] Microsoft Corporation, "Windows Error Codes", January 2007.

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[MS-NRBF] Microsoft Corporation, ".NET Remoting: Binary Format Data Structure", July 2007.

[MS-NRTP] Microsoft Corporation, ".NET Remoting: Core Protocol Specification", September 2007.

[MS-OAUT] Microsoft Corporation, "OLE Automation Protocol Specification", March 2007.

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

[RFC3629] Yergeau, F., "UTF-8, A Transformation Format of ISO 10646", RFC 3629, November 2003, http://www.ietf.org/rfc/rfc3629.txt

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

### 1.2.2 Informative References

This protocol relies on no informative references.

## 1.3 Protocol Overview (Synopsis)

**IManagedObject** 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) when reentering the CLR.

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 and 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-grain 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 further used to match that 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 that can be turned into a proxy to the managed object, as specified by the .NET Remoting: Binary Format Data Structure (as specified in [MS-NRBF]).

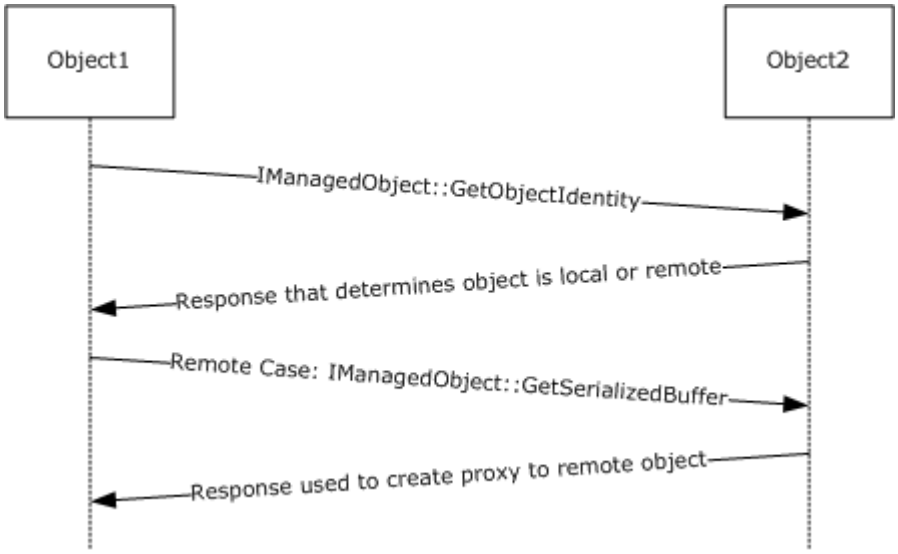


**Figure 1: IManagedObject request-response**

### 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

**IManagedObject** uses the Distributed Component Object Model (DCOM) Remote Protocol, as specified in [MS-DCOM].

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

Along with DCOM, this protocol also uses the OLE Automation Protocol (as specified in [MS-OAUT]) as part of the process of using the **BSTR** and **VARIANT** types from the **IDispatch** interface.

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

## 1.5 Prerequisites/Preconditions

This protocol depends on the Distributed Component Object Model (DCOM) Remote Protocol, as specified in [MS-DCOM]. The CLR MUST be installed on the client machine.

The **IRemoteDispatch** and **IServicedComponentInfo** interfaces also depend on datatypes defined in the OLE Automation Protocol, as specified in [MS-OAUT].

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 **IManagedObject** or **IRemoteDispatch** interface. The protocol also assumes that the client has the server object .NET 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 as 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 .NET 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 **2.3.2**.

This protocol uses HRESULT values as defined in [MS-DTYP] section **2.2.18**. Vendors can define their own HRESULT values, provided they set the C bit (0x20000000) for each vendor-defined value, indicating 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. Vendors SHOULD 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<br>{C3FCC19E-A970-11D2-8B5A-00A0C9B7C9C4} | The GUID associated with the **IManagedObject** interface. |
| IRemoteDispatch<br>{6619a740-8154-43be-a186-0319578e02db} | The GUID associated with the **IRemoteDispatch** interface. |
| IServedComponentInfo<br>8165B19E-8D3A-4d0b-80C8-97DE310DB583 | The GUID associated with the **IServicedComponentInfo** interface. |

# 2 Messages

The following sections specify how IManagedObject Interface Protocol messages are transported and common IManagedObject Interface Protocol data types.

## 2.1 Transport

This protocol uses **RPC dynamic endpoints** (as specified in [C706] Part 4) and DCOM (as specified in [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 definition 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 application initiates the conversation with the server by performing DCOM **activation** (as specified in [MS-DCOM] section 3.1.4.1.1) 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 Server Details

A CLR-managed object that has been exposed to COM will expose the COM interface **IManagedObject**. This interface is used to determine that 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 then back to managed, and to maintain 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 a proxy to 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

There are no protocol-specific timers.

### 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.<1>

Methods in RPC Opnum Order

| Method | Description |
|---|---|
| GetSerializedBuffer | Returns a binary-formatted representation of a proxy to the managed object, as specified in [MS-NRBF] section 2.3.<br>Opnum: 3 |
| GetObjectIdentity | Used to determine if a COM object is a managed object that belongs to this CLR instance and process subdivision.<br>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 proxy to the remote 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<br>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 if 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:** GUID that MUST indicate the CLR instance in which this object was created.

**AppDomainID:** Optional field. Implementation-specific, opaque, process-unique identifiers MUST denote the process subdivision in which this object resides.

**pCCW:** Optional field. Implementation-specific, opaque value 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<br>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**, as specified in [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**.<2>

Methods in RPC Opnum Order

| Method | Description |
|---|---|
| **RemoteDispatchAutoDone** | Invokes a call on the server object and deactivates it when the call completes.<br>Opnum: 7 |
| **RemoteDispatchNotAutoDone** | Invokes a call on the server object without deactivating it when the call completes.<br>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:** A UTF-8-encoded string (as specified in [RFC3629]) of binary data that contains the method input parameters. The binary data MUST be marshaled as specified in [MS-NRTP] section 3.1.5.1.1.

**pRetVal:** A UTF-8-encoded string (as specified in [RFC3629]) of binary data that contains the method output parameters. The binary data MUST be marshaled as specified in [MS-NRTP] section 3.1.5.1.1.

**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, it 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)

This 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:** A UTF-8 encoded string (as specified in [RFC3629]) of binary data that contains the method input parameters. The binary data MUST be marshaled as specified in [MS-NRTP] section 3.1.5.1.1.

**pRetVal:** A UTF-8 encoded string (as specified in [RFC3629]) of binary data that contains the method output parameters. The binary data MUST be marshaled as specified in [MS-NRTP] section 3.1.5.1.1.

**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, it 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 3.2.1.5.8. 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.<br>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:** The bit 0x00000004 MUST be set to 1. Other bits MUST be ignored by the server.

**infoArray:** An array that represents the information returned by the server.

**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.

The server MUST verify that the infoMask bit 0x00000004 is set. It MUST set the infoMask to 0x00000004, and MUST return a **SAFEARRAY** (as specified in [MS-OAUT] section **2.2.27.10**) of type **VT_BSTR** with one element. This element MUST be a URI (as specified in [RFC3986]) specified as a **BSTR**, as specified in [MS-OAUT] section **2.2.20**. The URI MUST represent the server object identity.

### 3.1.5 Timer Events

There are no protocol-specific timer events.

### 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 proxy to the CLR-managed object underlying the COM object.

### 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. Otherwise, the client fetches a binary-formatted string representation (as specified in [MS-NRBF] section 2.3) to the underlying managed object by using **GetSerializedBuffer**, which can be used to create a proxy to the object.

### 3.2.5 Timer Events

There are no protocol-specific timers.

### 3.2.6 Other Local Events

There are no other local events for the client.

# 4 Protocol Examples

The following sections describe several operations as used in common scenarios to illustrate the function of the IManagedObject Interface Protocol.

## 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 proxy to connect to the object that resides in another CLR (which could be a completely different implementation). 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.

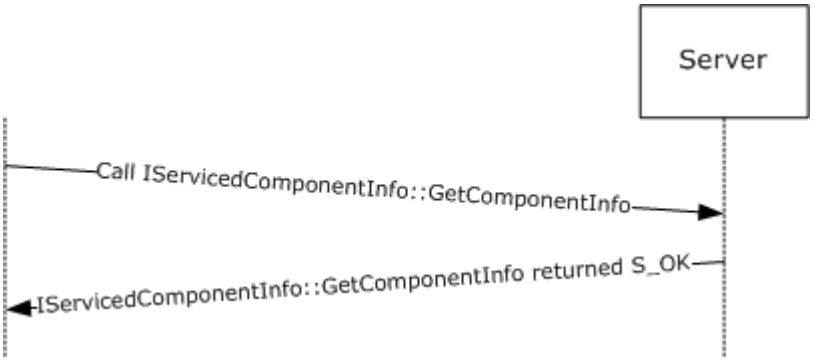


**Figure 2: Call sequence for determining server object identity**

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 with 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.


**Figure 3: Call sequence for dispatching a call with Deactivate**

1. The client calls IUnknown::QueryInterface to determine whether or not the object implements IRemoteDispatch. The object returns S_OK and a pointer to an IRemoteDispatch interface pointer.

2. The client marshals the method call it wishes to make on the object as specified in [MS-NRTP] section 3.1.5.1.1.

3. The client dispatches the call by invoking the IRemoteDispatch::RemoteDispatchAutoDone method on the object.

```
HRESULT
RemoteDispatchAutoDone(
    [in] BSTR s = {UTF-8 encoded payload representing the
                   method call},
    [out, retval] BSTR* pRetVal= {uninitialized pointer to
                                  receive the reply in a BSTR});
```

4. The server receives the method call, decodes the method request, and dispatches the method call.

5. After the method call completes, the server encodes the reply into pRetVal as specified in [MS-NRTP] section 3.1.5.1.1. It then deactivates the object instance and returns successfully.

```
HRESULT = S_OK
RemoteDispatchAutoDone(
    [in] BSTR s = {unchanged},
    [out, retval] BSTR* pRetVal= {UTF-8 encoded string containing
                                  the reply of the server});
```

6. The client calls IUnknown::Release on the interface pointer obtained in step 1. If this is the last outstanding reference on the object, the object is destroyed.

# 5 Security

The following sections specify security considerations for implementers of the interfaces in the IManagedObject Interface Protocol.

## 5.1 Security Considerations for Implementers

There are no security considerations for this protocol.

## 5.2 Index of Security Parameters

There are no protocol-specific security parameters.

# 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(x) 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: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows 2000
- Windows XP
- Windows Server 2003
- Windows Vista
- Windows Server 2008

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Windows behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that Windows does not follow the prescription.

<1> 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.

<2> Section 3.1.4.2: On the Windows platform, the **IRemoteDispatch** interface is only exposed by components inheriting from **System.EnterpriseServices.ServicedComponent**.

# 8 Index

## T



## V



## W