# [MC-COMQC]: Component Object Model Plus (COM+) Queued Components Protocol Specification



## Revision Summary

| Date | Revision History | Revision Class | Comments |
|---|---|---|---|
| 08/10/2007 | 0.1 | Major | Initial Availability |
| 09/28/2007 | 0.2 | Minor | Updated the technical content. |
| 10/23/2007 | 0.2.1 | Editorial | Revised and edited the technical content. |
| 11/30/2007 | 0.3 | Minor | Updates to MS-MQMQ external references. |

| Date | Revision History | Revision Class | Comments |
|---|---|---|---|
| 01/25/2008 | 0.3.1 | Editorial | Revised and edited the technical content. |

# Table of Contents

# 1 Introduction

The Component Object Model Plus (COM+) Queued Components Protocol (COMQC) is a protocol for persisting method calls made on COM+ objects in such a way that they can later be played back and executed. The protocol consists of a binary format used to store the information needed to achieve that goal.

## 1.1 Glossary

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

**Active Directory (AD)**
**ASCII**
**Client**
**CLSID**
**Component Object Model (COM)**
**Globally Unique Identifier (GUID)**
**Handle**
**Interface**
**Interface Definition Language (IDL)**
**Interface Identifier (IID)**
**Little-Endian**
**Marshal**
**Microsoft Interface Definition Language (MIDL)**
**Network Data Representation (NDR)**
**Object**
**Object Reference**
**Opnum**
**Security Context**
**UTF-16**

The following terms are specific to this document:

**Binary Large Object (BLOB):** A collection of binary data stored in a single continuous block.

**Message:** A data structure that represents a unit of data transfer between distributed applications. A message has message properties, which may include message header properties, a **message body** property, and message trailer properties.

**Message Body:** A distinguished **message** property that represents the application payload.

**Message Queue:** A data structure that contains an ordered list of zero or more messages. A message queue has a head and a tail and supports a first-in-first-out (FIFO) access pattern. Messages are appended to the tail through a write operation (Send) that appends the message. Messages are consumed from the head through a destructive read operation (Receive) that deletes the message. A message at the head may also be read through a non-destructive read operation (Peek).

**Message Queuing:** A communications service that provides asynchronous and reliable message passing between distributed **client** applications. In message queuing, clients send messages to **message queues** and consume messages from **message queues**. The **message queues** provide persistence of the messages, which enables the sending and receiving client applications to operate asynchronously from each other.

**MSMQ:** The **message queuing** protocol as defined in [MS-MQMQ].

**Partition:** A container for a specific configuration of a COM+ **object** class.

**Partition Identifier:** A **GUID** that identifies a **partition**.

**Proxy Object:** A local **object** that acts as an intermediary between an application and a remote **object**. The purpose of the proxy object is to monitor the life span of the remote **object** and to forward calls to the remote **object**.

**Workgroup Mode:** The operation of **message queuing** without an **Active Directory** connection. For more information, see [MS-MQMA].

**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

[MS-COM] Microsoft Corporation, "Component Object Model Plus (COM+) Protocol Specification", March 2007.

[MS-DCOM] Microsoft Corporation, "Distributed Component Object Model (DCOM) Remote Protocol Specification", March 2007.

[MS-DTCO] Microsoft Corporation, "MSDTC Connection Manager: OleTx Transaction Protocol Specification", July 2007.

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

[MS-MQMP] Microsoft Corporation, "Message Queuing (MSMQ): Queue Manager Client Protocol Specification", August 2007.

[MS-MQMQ] Microsoft Corporation, "Message Queuing (MSMQ): Data Structures", August 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

[RFC4234] Crocker, D., Ed. and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005, http://www.ietf.org/rfc/rfc4234.txt


### 1.2.2 Informative References


[MS-MQMA] Microsoft Corporation, "Message Queuing (MSMQ): Architecture Protocol Specification", August 2007.

[MSDN-IPersistStream] Microsoft Corporation, "IPersistStream", http://msdn2.microsoft.com/en-us/library/ms690091.aspx

## 1.3 Protocol Overview (Synopsis)

The COM+ Queued Components Protocol enables a client to asynchronously invoke methods on a server in scenarios of limited or intermittent connectivity. It does so by writing all the necessary state into a self-contained **BLOB**. That BLOB can be parsed and the method calls be replayed at a later point in time when it is possible to transmit the data to the recipient. As the BLOB is self-contained, the original creator of the BLOB is not required to be alive or connected at that point for the operation to succeed.

To transmit the BLOB without regard for connectivity, the protocol relies on a **message queuing** transport. Figure 1 shows the layering of the protocol stack.

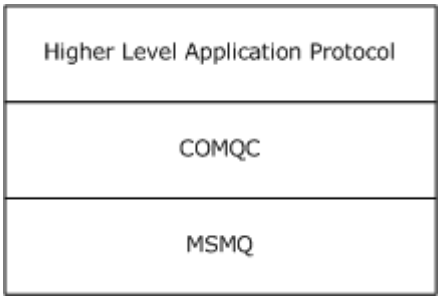



**Figure 1: Protocol layering**

The COM+ Queued Components Protocol (COMQC) is asynchronous and one-way with information flowing exclusively from the client to the server.

### 1.3.1 Server Role

The server is responsible for receiving COMQC **messages** and dispatching the recorded method calls.

Each COMQC server is associated with a single COM+ application using COMQC, and services messages for all server objects in that COMQC application. Each COMQC server has its own **message queue**. There can be multiple COMQC servers per machine.

### 1.3.2 Client Role

The **client** is responsible for recording method calls, packaging them up into a COMQC message and transmitting the message to the message queuing infrastructure.

A COMQC client queues calls to a single COMQC server object, and hence connects to a single message queue. A higher-layer application protocol can maintain multiple COMQC client that queue calls to different server objects, but for the purpose of this specification they are independent clients unrelated to each other. That is, even if a client application uses COMQC to queue calls to multiple server objects in the same COM+ application (and hence to the same message queue and the same COM+ Queued Components Protocol server), this involves a separate conceptual COMQC client per server object.

## 1.4 Relationship to Other Protocols

The COM+ Queued Components Protocol (COMQC) relies on DCE 1.1: Remote Procedure Call [C706] and the OLE Automation Protocol [MS-OAUT] to marshal the parameters of the recorded method calls. It further relies on [MS-MQMP] and [MS-MQMQ] as transport. The COMQC is limited to calls made on COM+ Protocol [MS-COM] objects.

There are no protocols that rely on the COM+ Queued Components Protocol.

## 1.5 Prerequisites/Preconditions

The COM+ Queued Components Protocol (COMQC) assumes that the message queuing infrastructure is already running, and that the client application is aware of the name of the message queue.

The server **object** must be installed on the COMQC server before a COMQC client attempts to invoke it.

## 1.6 Applicability Statement

The COM+ Queued Components Protocol (COMQC) is useful and appropriate for performing asynchronous method calls when the client and the server do not have constant connectivity and a message queuing mechanism is available.

It is not applicable to use with method calls that have output parameters or return values that the client application needs to get.

## 1.7 Versioning and Capability Negotiation

The COM+ Queued Components Protocol does not perform versioning or capability negotiation.

## 1.8 Vendor-Extensible Fields

The COM+ Queued Components Protocol defines no vendor-extensible fields.

## 1.9 Standards Assignments

This protocol has no standards assignments.

The following is a table of well-known **GUIDs** in the COM+ Queued Components Protocol (COMQC), as generated by the mechanism specified in [C706] section A.2.5.

| Parameter | Value |
|---|---|
| Message Signature | {1664BCFB-1751-11d2-B58E-00E0290E6C31} |
| Header Signature | {71bbdb83-fc41-11d0-b7640080c7ec3fc1} |
| Moniker GUID | {ecabafc6-7f19-11d2-978e-0000f8757e2a} |

# 2 Messages

## 2.1 Transport

For the remainder of section 2 and section 3, "message" refers to a COM+ Queued Components Protocol (COMQC) message unless otherwise specified.

COMQC uses MSMQ Queue Manager Client Protocol [MS-MQMP], and MSMQ: Data Structures [MS-MQMQ] as transport.

The message defined in section 2.2 MUST be stored in the body of the **MSMQ** message as a BLOB.

The PROPID_M_EXTENSION property of the MSMQ message (as specified in [MS-MQMQ] section 2.6.6.35) MUST be set to GUID {1664BCFB-1751-11d2-B58E-00E0290E6C31}.

When opening a message queue, a COMQC client MUST specify the MSMQ parameters MQ_SEND_ACCESS (as specified in [MS-MQMP] section **3.1.4.17**) and MQ_DENY_NONE. When opening a message queue, a COMQC server MUST specify MQ_RECEIVE_ACCESS and MQ_DENY_NONE. Both transacted and non-transacted message queues MUST be supported.

When running MSMQ in **workgroup mode**, PROPID_M_AUTH_LEVEL (as specified in [MS-MQMQ]section 2.6.6.24) MUST be set to MQMSG_AUTH_LEVEL_NONE and PROPID_M_SENDERID_TYPE (as specified in [MS-MQMQ] section 2.6.6.22) to MQMSG_SENDERID_TYPE_NONE. Other than that, a COMQC client SHOULD allow all MSMQ message-level parameters to be configurable. A COMQC server MUST support all MSMQ message-level parameters.

When running MSMQ in workgroup mode, the message queue name SHOULD be set to "<computer name>\PRIVATE$\<COM+ application name>", where <computer name> is the lower-case, non-fully-qualified domain name of the computer running the COMQC server and <COM+ application name> is the display name of the COM+ application hosting the objects that are exposed via COMQC.

Otherwise, the message queue name SHOULD be set to "<computer name>\<COM+ application name>".

The entire communication between client and server MUST be contained in a single message. The server MUST NOT send a reply message and the client MUST NOT expect one.

## 2.2 Message Syntax

A message contains the state of one or more method calls to be made on the server as well as general state associated with the request. Messages are self-contained and MUST NOT rely on any other message to be processed.

Every message MUST contain at least one method call.

A message is divided into headers. Figure 2 defines the ordering of the individual headers. The headers themselves are defined in section 2.2.1 to section 2.2.6.

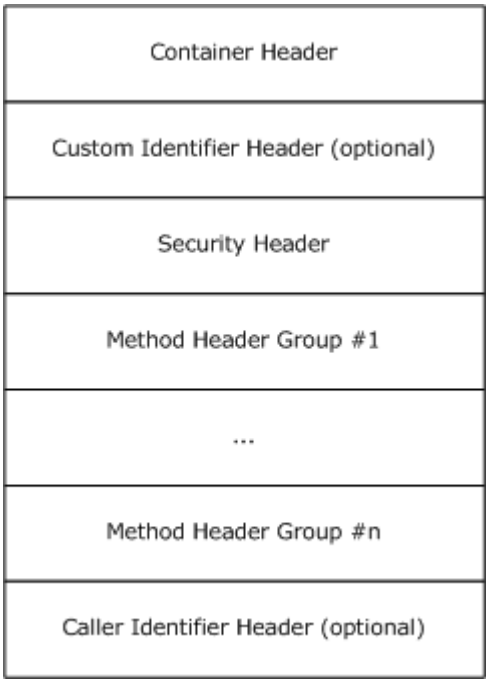



**Figure 2: Header ordering**

A Method Header group contains all state of a single method call to be made on the server. Every message MUST contain one or more Method Header groups. The default Method Header group is defined in Figure 3.

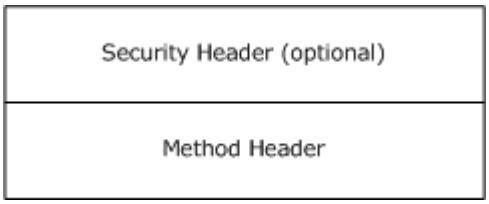



**Figure 3: Default Method Header group**

If the security properties of the method call as defined by the Security ORPC Extension, defined in [MS-COM] section 2.2.3.2, are identical to the security properties of the method call appearing immediately before it in the message, the security header SHOULD be omitted. Note that the first Method Header group contains no security header as it is already present as part of the overall layout.

If the method call is made on the same **interface** as the immediately preceding method call, then the optimization defined by figure 4 SHOULD be used.

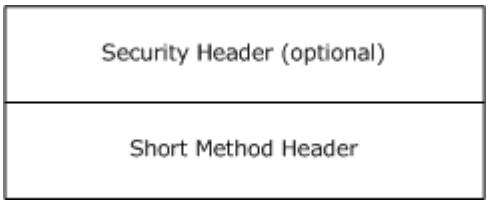


**Figure 4: Short method optimization**

If the security properties do not match the **security context** of the immediately preceding method call but do match the security context of a different previous method call, the optimization defined by figure 5 SHOULD be used.

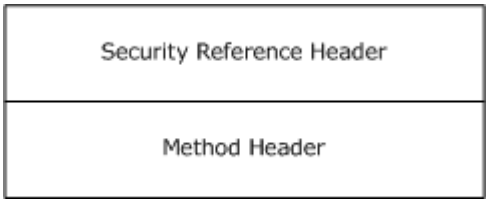



**Figure 5: Security reference optimization**

If the security properties do not match the security properties of the immediately preceding method call but does match the security properties of a different previous method call, and if the call is made on the same interface as the immediately preceding method call, optimization defined by figure 6 SHOULD be used.

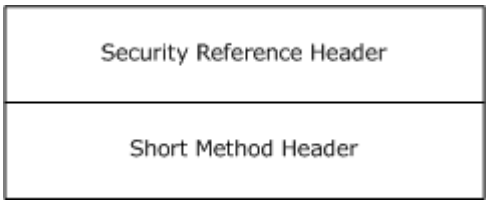
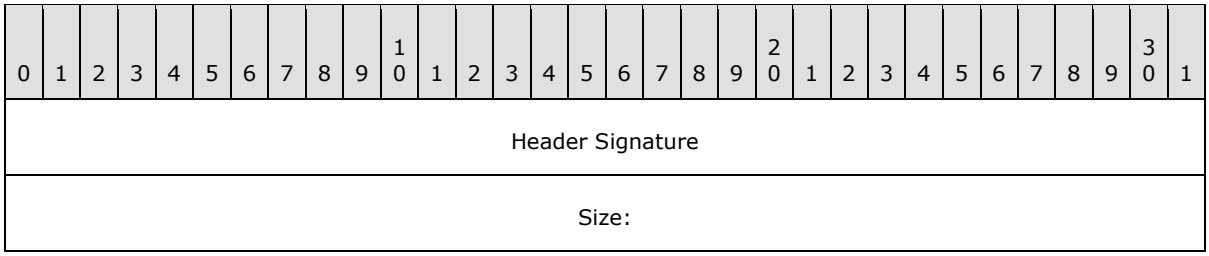



**Figure 6: Security reference and short header optimization**

Unless otherwise specified, all fields MUST be specified in **little-endian** format. Unless otherwise specified, all headers and all custom data MUST be aligned on 8-byte boundaries, meaning that each header MUST be padded at the end so that its length is a multiple of 8.

### 2.2.1 Common Header

The common header specifies the size and type of a header. Every header MUST begin with the common header. The common header MUST NOT appear by itself.



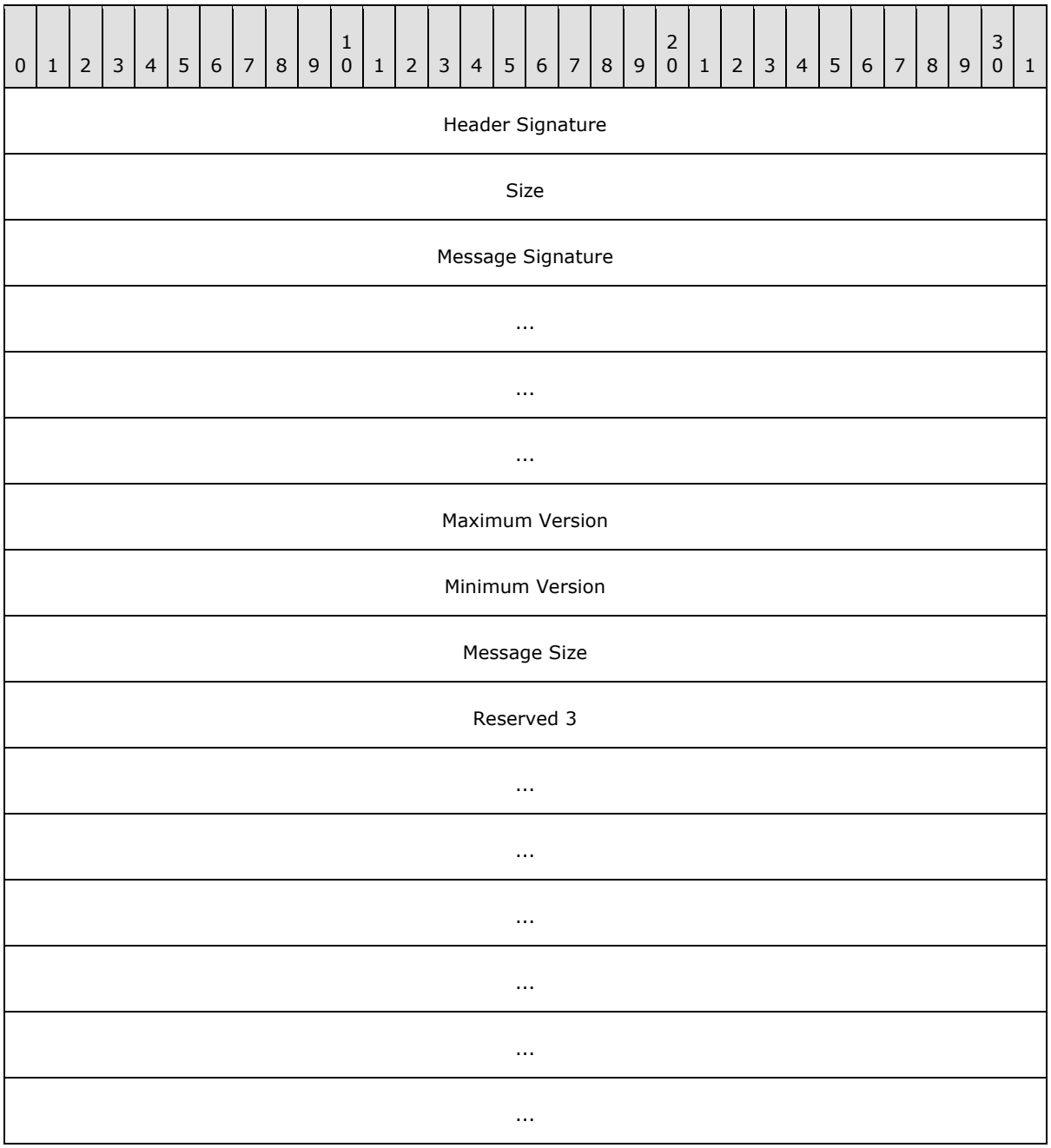

**Header Signature (4 bytes):** A predefined value that identifies the specific header. (The value is given in the section that defines each specific header.)

**Size: (4 bytes):** Size in bytes of the header, plus the size of header-specific variable length data fields if present. Some, but not all, headers have such fields following the header itself, and their size is included in this number. The value MUST be a multiple of 8. Adding the size to the starting offset of a header gives the starting offset of the next header.

All headers defined in section 2.2.2 to section 2.2.6 contain the common header. Unless otherwise specified, "Header Signature" refers to the Header Signature field of the common header that is embedded in the other headers.

### 2.2.2 Container Header

The container header contains general information regarding the message. Every message MUST have exactly one container header and it MUST be the first header in the message.



| |
|---|
| ... |
| Call Target Identifier Size |
| Reserved 4 |
| ... |
| Call Target Identifier (variable) |
| ... |
| Padding (variable) |
| ... |

**Header Signature (4 bytes):** MUST be set to 0x52444843 ("CHDR" in **ASCII**).

**Size (4 bytes):** Size in bytes of the container header.

**Message Signature (16 bytes):** GUID that MUST be set to {71bbdb83-fc41-11d0-b7640080c7ec3fc1}

**Maximum Version (4 bytes):** Maximum version of the header. MUST be set to 0x00000001.

**Minimum Version (4 bytes):** Minimum version of the header. MUST be set to 0x00000001.

**Message Size (4 bytes):** Size in bytes of the entire message.

**Reserved 3 (32 bytes):** Reserved. MUST be set to 0 and ignored on receipt.

**Call Target Identifier Size (4 bytes):** Size of the Call Target Identifier field plus padding. MUST be a multiple of 8.

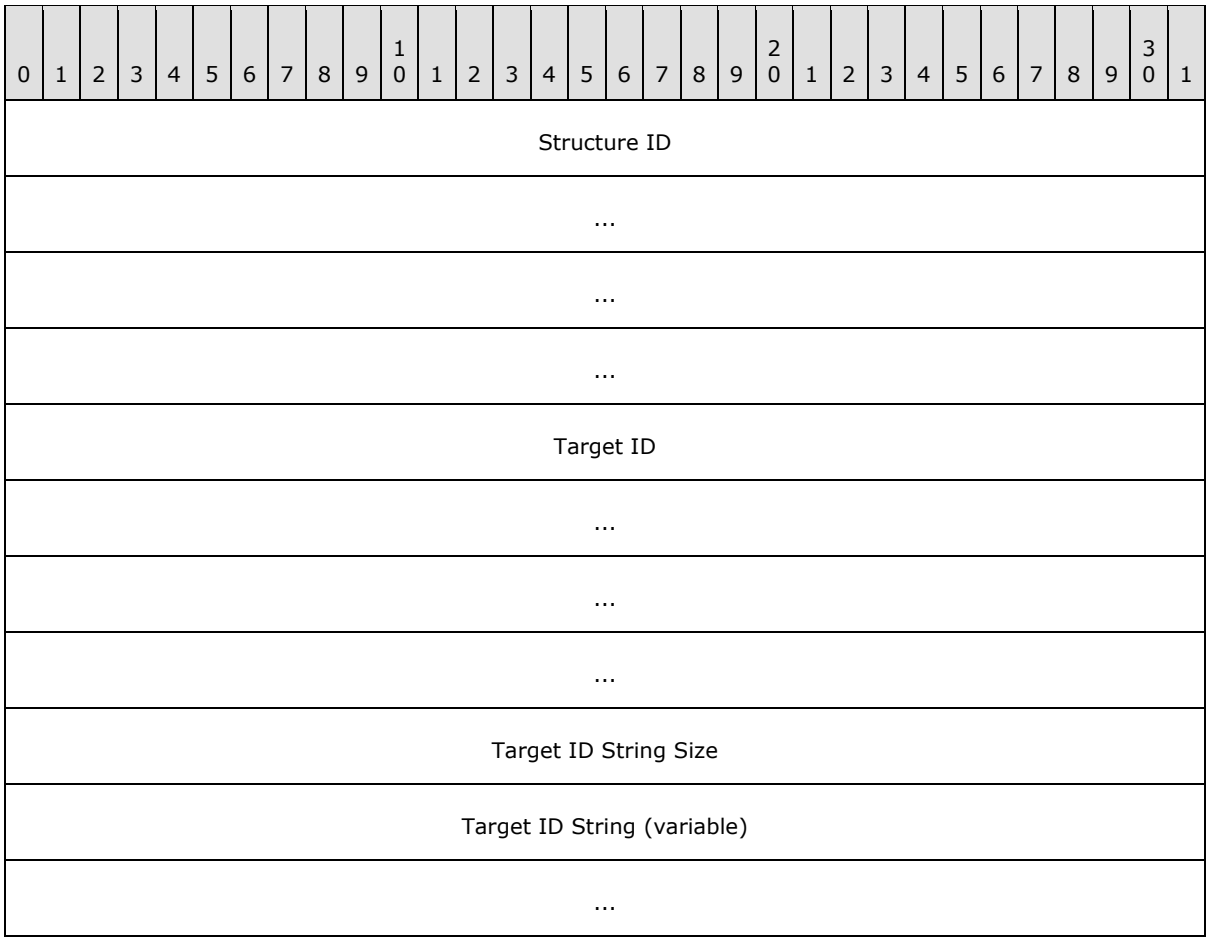
**Reserved 4 (8 bytes):** Reserved. MUST be set to 0 and ignored on receipt.

**Call Target Identifier (variable):** The target of the call, as defined in section 2.2.2.1.

**Padding (variable):** Enough space to pad the message to an 8-byte boundary. MUST be set to 0 and ignored on receipt.

### 2.2.2.1 Call Target Identifier

The call target identifier is part of the container header and identifies the target of the call. The server MUST use the information stored here to determine where to dispatch the call.




**Structure ID (16 bytes):** GUID identifying the Call Target Identifier structure. MUST be set to {ecabafc6-7f19-11d2-978e-0000f8757e2a}.

**Target ID (16 bytes):** A **Component Object Model (COM)** **CLSID** (see [MS-DCOM] section **2.2.1.4**) identifying the target of the call.

**Target ID String Size (4 bytes):** Size of the Target ID String field in bytes.

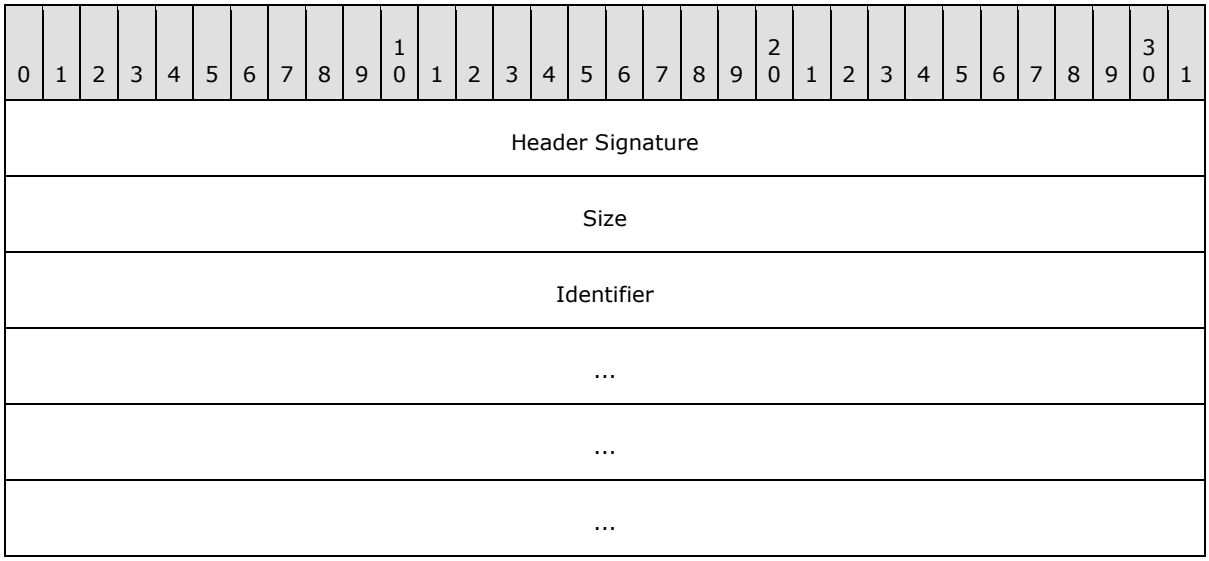
**Target ID String (variable):** String buffer identifying the target of the call. It SHOULD be set to the string representation of the Target ID, and on receipt the value MUST be ignored after validating that the string conforms to the syntax. The string representation MUST conform to the following ABNF (as specified in [RFC4234]) syntax:

```
GUID = [ UUID / "{" UUID "}" ]
```

where UUID is defined by [C706] section 3.1.17. The string MUST be encoded in UTF-16 and MUST be NULL terminated.

### 2.2.3 Partition Identifier Header

The partition identifier header contains the COM+ **partition identifier** (as defined in [MS-COM] section 1.3.6) that the server object resides in.




**Header Signature (4 bytes):** MUST be set to 0x54524150 ("PART").

**Size (4 bytes):** The size in bytes of the partition identifier header. MUST be set to 0x00000018.

**Identifier (16 bytes):** GUID identifying the **partition** that the server object resides in.

### 2.2.4 Security Header

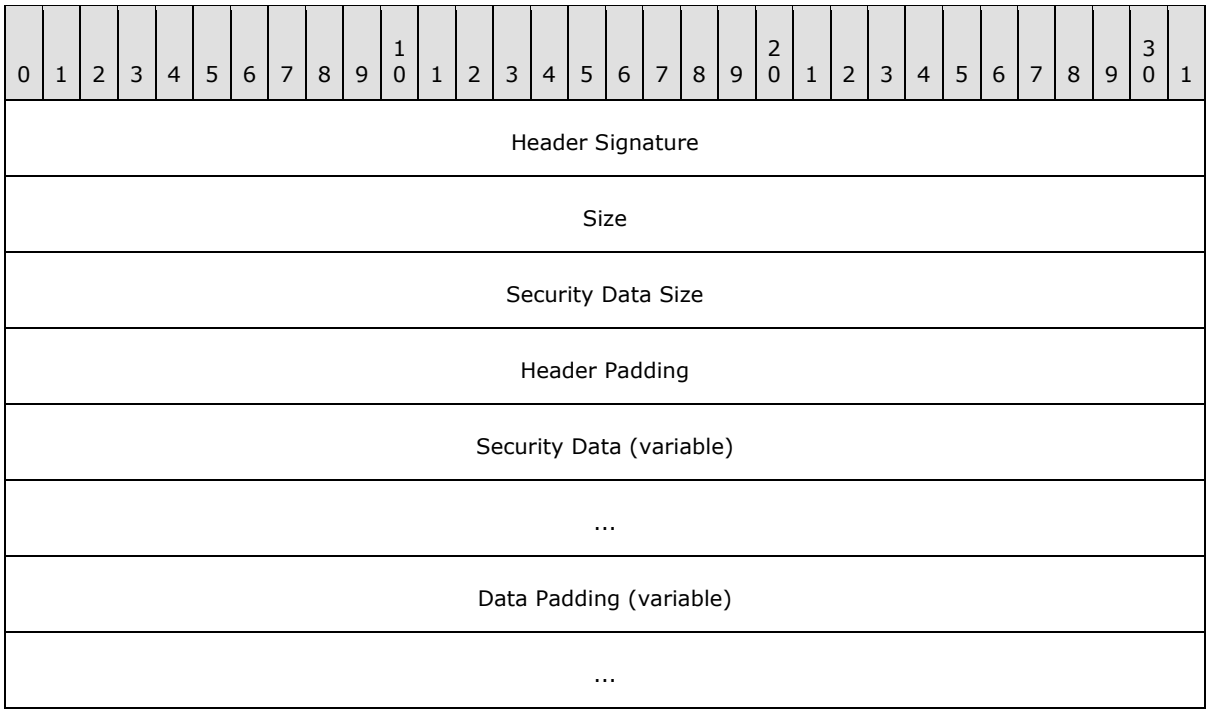
The security header is a wrapper for the Security ORPC Extension defined in [MS-COM] section 2.2.3.2. It is used to capture the current security context so that it can be recreated on the COMQC server when the stored calls are being dispatched.

The security properties stored in this header MUST be applied to all method calls following the security header in the message until another security header or security reference header (section 2.2.5) is encountered.

If the security properties of a method call are identical to the security properties of the method call stored in the message immediately prior to the current method call, a security header SHOULD NOT be included.

If the security properties are identical to the security context of any other previous method call, a security reference header SHOULD be included instead. Otherwise, a full security header MUST be included.

A security header MUST be included before the first method call.




**Header Signature (4 bytes):** MUST be set to 0x44434553 ("SECD").

**Size (4 bytes):** The size in bytes of the security header.

**Security Data Size (4 bytes):** Size in bytes of the Security Data field.

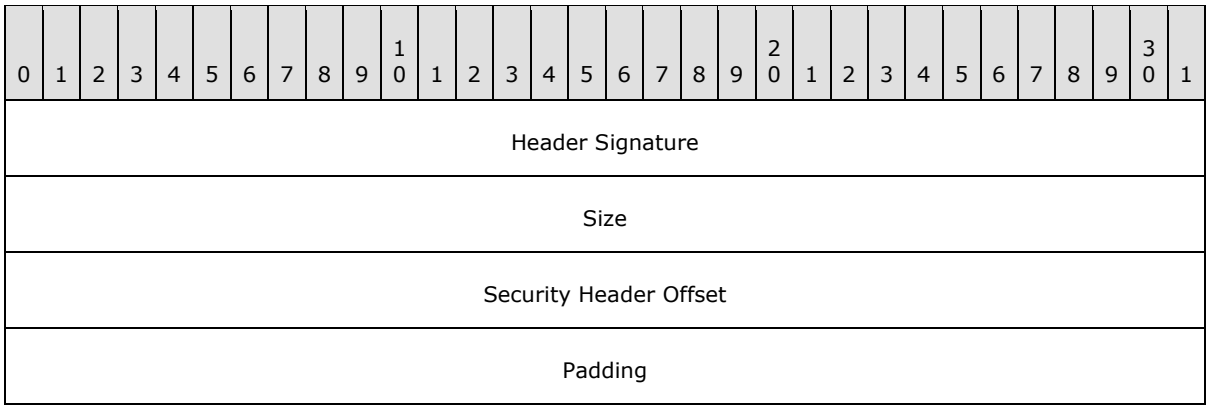
**Header Padding (4 bytes):** Aligns the Security Data field on an 8-byte boundary. MUST be set to 0 and ignored on receipt.

**Security Data (variable):** Security ORPC Extension as defined in [MS-COM] section 2.2.3.2.

**Data Padding (variable):** Enough space to extend the header to an 8-byte boundary. MUST be set to 0 and ignored on receipt.

### 2.2.5 Security Reference Header

A security reference header is used to refer to a previous security header. If the security properties of the current call match the security properties of the call stored immediately before it in the message a security reference header SHOULD NOT be included. If the security context matches any other previous call, a security reference header SHOULD be included instead of a security header.




**Header Signature (4 bytes):** MUST be set to 0x52434553 ("SECR").

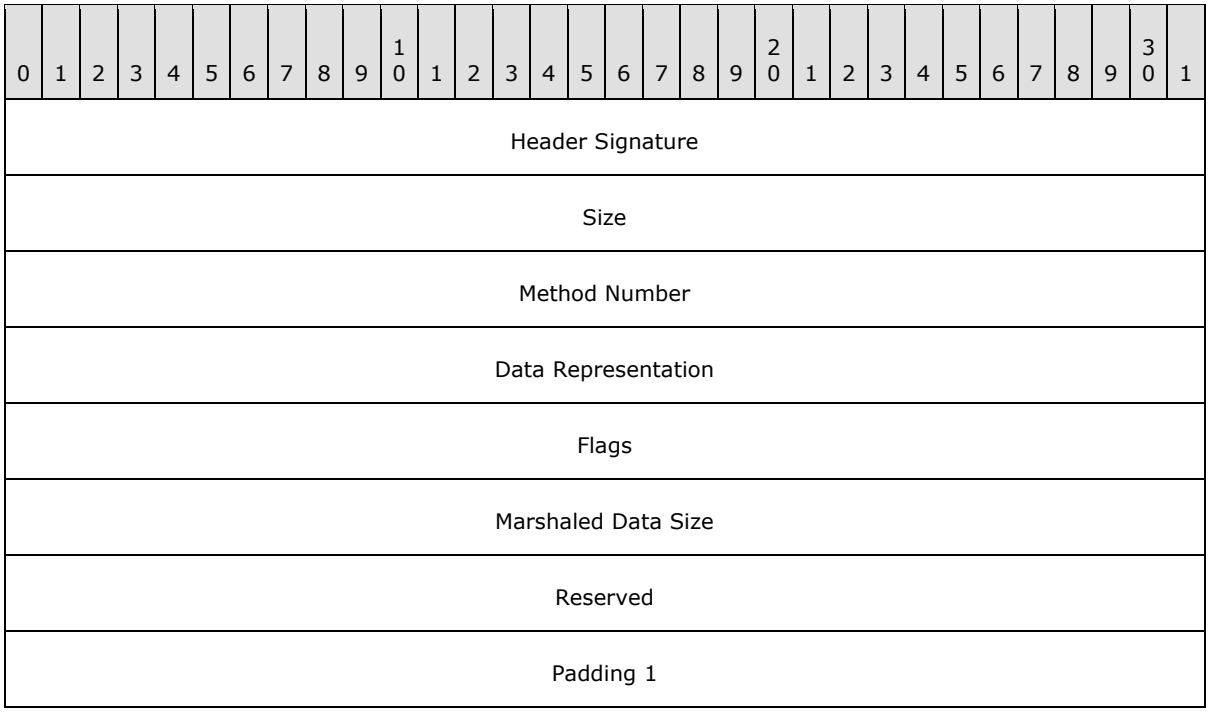
**Size (4 bytes):** The size in bytes of the security reference header. MUST be set to 0x00000010.

**Security Header Offset (4 bytes):** Offset, in bytes, into the message where the security header this header refers to is located.

**Padding (4 bytes):** Aligns the header on an 8 byte boundary. MUST be set to 0 and ignored on receipt.

### 2.2.6 Method Header

The method header encapsulates a method call. Each method call has exactly one method header. Each message MUST contain one or more method headers.



| Interface ID (optional) |
| --- |
| ... |
| ... |
| ... |
| Marshaled Data (variable) |
| ... |
| Padding 2 (variable) |
| ... |

**Header Signature (4 bytes):** MUST be set to 0x4854454D ("METH") if the **Interface ID** field is present. Otherwise, MUST be set to 0x48544D53 ("SMTH").

**Size (4 bytes):** The size in bytes of the method header.

**Method Number (4 bytes):** **Opnum** of the method to invoke as it appears on the **Interface Definition Language (IDL)** describing the interface. See [C706] section 4.

**Data Representation (4 bytes):** The data representation that was used to marshal the method parameters. MUST be set to 0x00000010.

**Flags (4 bytes):** MUST be set to 0x00001000.

**Marshaled Data Size (4 bytes):** Size in bytes of the Marshaled Data field.

**Reserved (4 bytes):** Reserved. MUST be set to 0x00000001.

**Padding 1 (4 bytes):** Padding bytes used to align the Interface ID on 8-byte boundary. MUST be set to 0 and ignored on receipt.

**Interface ID (optional) (16 bytes):** Interface ID (IID) of the interface being invoked. This field is optional. If the method call is made on the same interface as the immediately preceding method call, the Interface ID SHOULD be absent. Implementations of this specification MUST use the Interface ID specified in the most recent method header to invoke this call. Since no previous interface is specified at that point, the first method header in a message MUST contain the Interface ID field.

**Marshaled Data (variable):** Binary representation of the marshaled method parameters as defined in section 2.2.6.1.

**Padding 2 (variable):** Aligns the marshaled data on an 8-byte boundary. See section 2.2.6.1.4 for details.

#### 2.2.6.1 Marshaled Data

This field contains the binary representation of the marshaled method parameters. The binary format is based on either the **Network Data Representation (NDR)** format defined in [C706], or the dispatch format (defined in [MS-OAUT] section 3.1.4.4.2) with two exceptions defined in section 2.2.6.1.4 and section 2.2.6.1.5.

If the Interface ID in the method header is set to IID_IDispatch (as defined in [MS-OAUT] section **1.9**), the dispatch marshaling format MUST be used. Otherwise the NDR format MUST be used.

##### 2.2.6.1.1 NDR Marshaling

If the Interface ID specified in the method header is not IID_IDispatch, the marshaled data MUST conform to [C706] with the exception of what is defined in section 2.2.6.1.4 and section 2.2.6.1.5.

When using NDR marshaling, input/output and output ([in, out] and [out] in MIDL) parameters MUST NOT be supported. Input parameters ([in]) MUST be supported.

##### 2.2.6.1.2 Dispatch Marshaling

If the Interface ID specified in the method header is IID_IDispatch, the marshaled data MUST conform to [MS-OAUT] section 3.1.4.4.2, with the exception of what is defined in section 2.2.6.1.4 and section 2.2.6.1.5.

When using dispatch marshaling, [out] parameters MUST NOT be supported. [in] and [in, out] parameters MUST be supported. However, since COMQC is a one-way protocol, [in, out] parameters MUST NOT be filled with data returned from the server.

##### 2.2.6.1.3 Supported Types

All types that are OLE Automation–compliant MUST be supported, with the exception of **object references** as defined in [MS-DCOM] section 1.3.2. See [MS-OAUT] section 2.2.16.2 for a definition of OLE Automation–compliant types. Other types MUST NOT be supported. See section 2.2.6.1.5 regarding handling of object references.

##### 2.2.6.1.4 Padding of the Marshaled Data

Following the marshaled method parameters, the marshaled data MAY<1> contain an arbitrary amount of undefined padding that is not part of the marshaled method parameters as defined in section 2.2.6.1.1 and section 2.2.6.1.2.

Implementations of this specification MUST ignore that data when unmarshaling the method parameters on the server, and MUST NOT reject incoming messages that contain such data.

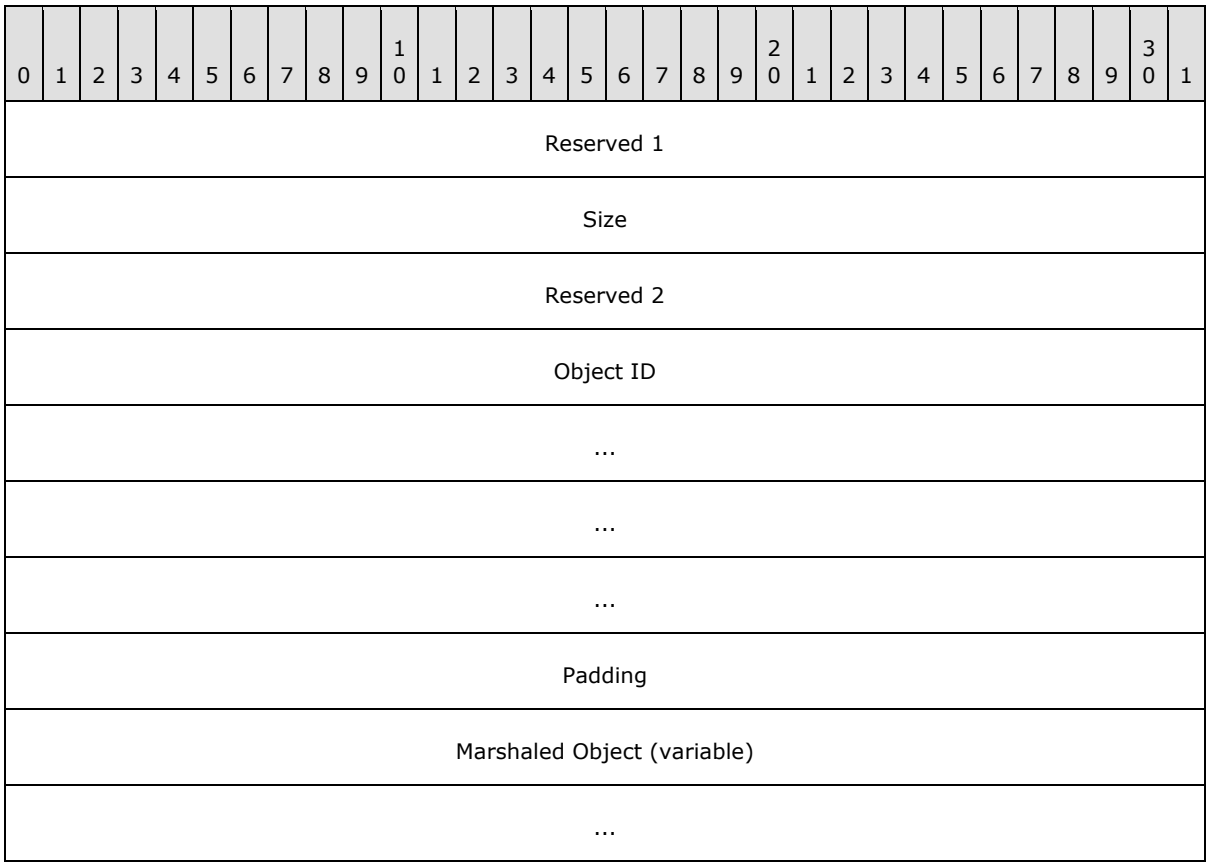
##### 2.2.6.1.5 Object References

Object references as defined in [MS-DCOM] section 1.3.2 MUST NOT be supported unless there exists a well-known binary representation of the object.<2> It is the responsibility of the implementation of this specification to obtain that binary representation.

When encountering an object reference while marshaling the method parameters into the marshaled data buffer, implementations of this specification MUST write the structure defined below into the buffer instead of writing the reference. If that is not possible, the marshaling MUST fail.

When unmarshaling, a COMQC server MUST replace the binary representation with an instance of the object specified by the binary representation before dispatching the call.

The Object References structure is defined below. The alignment of this structure is defined by the marshaling format used. This specification does not mandate any particular alignment.




**Reserved 1 (4 bytes):** MUST be set to 0 and ignored on receipt.

**Size (4 bytes):** Size of the header not counting the size of the marshaled object. That is, it counts everything up to and including the padding.

**Reserved 2 (4 bytes):** MUST be set to either 0x00000001 or 0x00000003 and MUST be ignored on receipt.

**Object ID (16 bytes):** COM **CLSID** identifying the object to be marshaled.

**Padding (4 bytes):** Aligns the Marshaled Object field on an 8-byte boundary. MUST be set to 0 and ignored on receipt.

**Marshaled Object (variable):** The binary representation of the object.

# 3 Protocol Details

This section specifies the two roles of the COM+ Queued Components Protocol (COMQC): the client role and the server role.

## 3.1 Server Details

### 3.1.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The organization is provided to explain how the protocol behaves. This document does not mandate that implementations adhere to this model provided that their external behavior is consistent with that specified in this document.

Servers maintain the following data elements:

Queue Handle: A **handle** that identifies the message queue and is used to retrieve messages from the message queue.

Object Table: A table that associates object GUIDs with the object they refer to.

### 3.1.2 Timers

This protocol does not define any timers.

### 3.1.3 Initialization

During initialization, the COMQC server MUST open a message queue on the already running MSMQ infrastructure and store the queue handle for future use. It MUST then initialize the objects on which the client makes calls or initialize a client activation mechanism (as specified in [MS-COM] sections 3.8.4 and 3.10.4) used to create the client upon receipt of a message.

### 3.1.4 Higher-Layer Triggered Events

This protocol does not define any higher-layer triggered events.

### 3.1.5 Message Processing Events and Sequencing Rules

When a message arrives, the server MUST verify that the PROPID_M_EXTENSION property of the MSMQ message is set to {1664BCFB-1751-11d2-B58E-00E0290E6C31}. Messages that do not have that value set MUST be rejected and ignored.

Next, the server MUST validate that the message conforms to the format defined in section 2.2. Messages that do not conform to that format MUST be rejected and ignored.

Otherwise, the server MUST search the object table for an object that matches the target ID of the received message as defined in section 2.2.2.1. If no matching object is found, the message MUST be rejected.

Otherwise, the server MUST attempt to use the security properties that are part of the received message in accordance with [MS-COM] section 3.2.2.2. If this fails, the message MUST be rejected.

The server MUST then play back the recorded method calls in the order in which they appear in the message. Playing back a method call means locally executing the method call as if it were invoked locally (without the involvement of COMQC).

Any communication following the receipt of a message MUST be treated as a new exchange independent of the previous one, where the server acts as a client and the client acts as a server with no knowledge of the previous exchange.

### 3.1.6 Timer Events

This protocol does not define any timer events.

### 3.1.7 Other Local Events

When stopping, the COMQC server MUST close the queue handle.

## 3.2 Client Details

The client composes a message in accordance with the format defined in section 2.2 and then transmits it via MSMQ.

After sending the message to the transport, the client side of the message exchange is complete.

COMQC does not specify how to deal with transmission failures. An implementation MAY communicate failures to the higher-layer application or protocol.

### 3.2.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

Clients maintain the following data elements:

- Proxy: Local **proxy object** to the server object that is used by the higher-layer application or protocol to invoke method calls on the server object.
- List of Pending Calls: A list containing all calls that the higher-layer protocol or application has made but that have not been sent to the server. Each entry contains all state needed to later write a method call to the message:
- Method Number: The Opnum of the method to call.
- Parameter Values: The parameter values to pass to the method.
- Security Context: The current security context data as defined by [MS-COM] section 2.2.3.2.

### 3.2.2 Timers

This protocol does not define any timers.

### 3.2.3 Initialization

This protocol does not define any initialization requirements.

### 3.2.4 Higher-Layer Triggered Events

#### 3.2.4.1 Application Requesting Interface

When a higher-layer protocol or application requests an interface to the server object, the COMQC client MUST attempt to create a local proxy object representing the server object and fail the call if it cannot. (The server is not notified that a proxy object has been created.)

#### 3.2.4.2 Application Making Method Call

When a higher-layer protocol or application performs a method call on the proxy object, the COMQC client MUST attempt to create a new entry in the list of pending calls and store the data describing the call, and fail the call if it cannot. The relevant data are the method parameters and the security context as defined in [MS-COM] section 2.2.3.2.

Alternatively, an implementation MAY immediately convert the method call into a message conforming to the format defined in section 2.2, and send it to the message queue without storing any internal state. (Such an implementation would not implement application signaling as defined in section 3.2.4.3.)

#### 3.2.4.3 Application Signaling that Method Calls Are Complete

When the higher-layer protocol or application signals that no more method calls are coming (for example by destroying the proxy object), the COMQC client MUST convert all entries in the list of pending calls into a message conforming to the format defined in section 2.2, open a connection to the server's message queue, send the message to the message queue and close the connection to the message queue. If the list of pending calls is empty, an MSMQ message MUST NOT be sent.

An implementation MAY provide feedback to the higher-layer protocol or application when a message queue transfer succeeds or fails, but if or how this is done is outside of the scope of this protocol.

### 3.2.5 Message Processing Events and Sequencing Rules

None.

### 3.2.6 Timer Events

This protocol does not define any timer events.

### 3.2.7 Other Local Events

This protocol does not define any other events.

# 4 Protocol Examples

This section describes an example of a common use of the COM+ Queued Components Protocol where a client application running on a roaming computer with intermittent access to a network wants to send data to a server application without regard for the current state of connectivity.

## 4.1 Client Creating and Sending a Message

A typical sequence of events for a COMQC client is as follows:

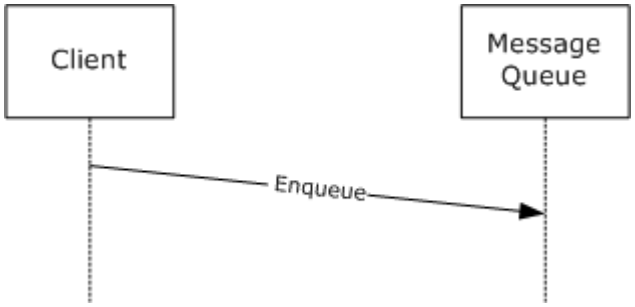



**Figure 7: Client-side exchange**

The client application requests an interface to the server from the COMQC client via the interface UUID and server name. The COMQC client creates a proxy object and an empty list of pending calls.

The client application invokes a specific method on the interface by supplying the method number and parameter values. The COMQC client creates a new entry in the list of pending calls and stores the method number and parameter values, along with the current security context data as defined by [MS-COM] section 2.2.3.2.

The client application invokes another method. The COMQC client repeats the previous steps.

The client application destroys the proxy object. At this point, the COMQC client creates a COMQC message (as defined in section 2.2) that contains the data that it received from the client application.

The COMQC client asynchronously transmits the COMQC message in the payload of an MSMQ message. This concludes the client-side portion of the protocol.

## 4.2 Server Retrieving and Processing a Message

A typical sequence of events for a COMQC server is as follows.

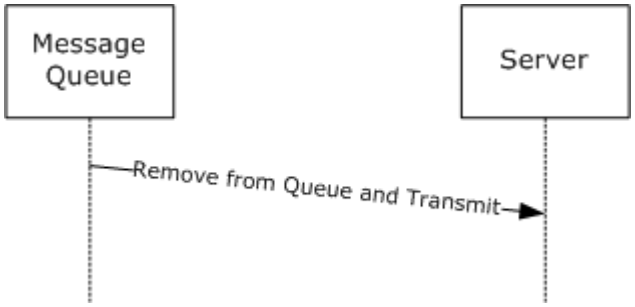


**Figure 8: Server-side exchange**

The COMQC server is notified by MSMQ that a message arrived. The server receives the message from MSMQ and validates it.

The server sets up the security context that was specified in the message.

The server unmarshals the parameters of the first method call, looks up the target ID in its object table to find the target object, and locally calls the specified method on the target object.

The server repeats the previous two steps for each method that was stored in the message.

After the call last has finished, all state relevant to the message is freed. This concludes the exchange.

# 5 Security

## 5.1 Security Considerations for Implementers

The COM+ Queued Components Protocol does not define any specific means by which the contents of a message have to be encrypted. COMQC messages, as defined in this specification, are transmitted in plain text. They are also not numbered, time stamped or otherwise identified. If implementers of this specification require properties such as confidentiality or nonrepudiation, they must use functionality provided by the underlying transport to achieve the desired result.

The protocol also does not define how to validate and set up the security properties that were transmitted as part of the message beyond what is defined in [MS-COM] section 2.2.3.2. It is the responsibility of the implementer of this protocol to build the security infrastructure since the protocol does not provide feedback to the client regarding how to communicate security violations.

Many fields of a COMQC message are of variable length. Implementers must exercise caution when accessing memory based on the size fields in the message because the protocol does not specify any validation fields such as checksums.

This protocol does not define ways to detect or prevent replays of messages. As a matter of fact, due to the use of this protocol in scenarios of limited or unreliable connectivity, multiple transmission attempts by the underlying transport may occur. This protocol itself does not define any retry mechanisms and relies on MSMQ to retransmit the messages in case of failures and to ensure the messages is delivered only once.

## 5.2 Index of Security Parameters

| Security parameter | Section |
|---|---|
| MQ_SEND_ACCESS | 2.1 |
| MQ_DENY_NONE | 2.1 |
| MQ_RECEIVE_ACCESS | 2.1 |
| PROPID_M_AUTH_LEVEL | 2.1 |
| MQMSG_AUTH_LEVEL_NONE | 2.1 |
| PROPID_M_SENDERID_TYPE | 2.1 |
| MQMSG_SENDERID_TYPE_NONE | 2.1 |

# 6 Appendix A: 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 2.2.6.1.4: The Windows COMQC client puts a varying amount of undefined data into the padding. No other guarantee regarding the content or length of the padding is given. The Windows COMQC server ignores the superfluous data.

<2> Section 2.2.6.1.5: The Windows implementation uses the IPersistStream [MSDN-IPersistStream] interface to obtain the binary representation of the object. If the object does not support IPersistStream, the call to be recorded will be rejected by the Windows COMQC client.

# 7 Index

## S



## T



## V



## W