[MS-MQQP]:
Message Queuing (MSMQ):
Queue Manager to Queue Manager Protocol Specification

Revision Summary

Date	Revision History	Revision Class	Comments
05/11/2007	0.1		MCPP Milestone 4 Initial Availability
08/10/2007	1.0	Major	Updated and revised the technical content.
09/28/2007	1.0.1	Editorial	Revised and edited the technical content.
10/23/2007	1.0.2	Editorial	Revised and edited the technical content.

Date	Revision History	Revision Class	Comments
11/30/2007	1.0.3	Editorial	Revised and edited the technical content.
01/25/2008	1.0.4	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

This document specifies the Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol. The Queue Manager to Queue Manager Protocol is an **RPC**-based protocol used by the **queue manager** and runtime library to read and **purge messages** from a **remote queue**.

1.1 Glossary

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

Authentication Level
Authentication Service
Dynamic Endpoint
Endpoint
Globally Unique Identifier (GUID)
Interface Definition Language (IDL)
Microsoft Interface Definition Language (MIDL)
Network Data Representation (NDR)
Opnum
Remote Procedure Call (RPC)
RPC Protocol Sequence
RPC Transfer Syntax
RPC Transport
Security Provider
Universally Unique Identifier (UUID)
Well-Known Endpoint

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

Cursor
Dependent Client
Message
Message Body
Message Header
Message Property
Message Queuing
Message Trailer
MSMQ
Queue
Queue Manager
Remote Queue
Remote Read

The following terms are specific to this document:

Purge: In the context of a **queue**, to delete all **messages** from the **queue**.

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

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

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

[MS-MQMQ] Microsoft Corporation, "Message Queuing (MSMQ): Data Structures", August 2007.

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

[MS-MQRR] Microsoft Corporation, "Message Queuing (MSMQ): Queue Manager Remote Read Protocol Specification", June 2007.

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

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

1.2.2 Informative References

[MSDN-MQEIC] Microsoft Corporation, "Message Queuing Error and Information Codes", http://msdn2.microsoft.com/en-gb/library/ms700106.aspx

1.3 Protocol Overview (Synopsis)

Message queuing is a communications service that provides asynchronous and reliable message passing between client applications, including those client applications running on different hosts. In message queuing, clients send messages to a **queue** and consume application messages from a queue. The queue provides persistence of the messages, enabling them to survive across application restarts, and allowing the sending and receiving client applications to operate asynchronously from each other.

Queues are typically hosted by a communications service called a queue manager. By hosting the queue manager in a separate service from the client applications, applications can communicate by exchanging messages via a queue hosted by the queue manager, even if the client applications never execute at the same time.

The queue manager may need to perform operations on a remote queue. When this scenario occurs, a protocol is required to insert messages into the remote queue, and another protocol is required to

consume messages from the remote queue. The Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol provides a protocol for consuming messages from a remote queue.

The Queue Manager to Queue Manager Protocol is used only to read messages from a queue or to purge messages from the queue. Reading a message also implies deleting the message after it is read, as specified in Queue Operations (section 1.3.3).

1.3.1 Messages

Each message that is exchanged in a message queuing system typically has a set of **message properties** that contain metadata about the message, and a distinguished property, called a **message body**, that contains the application payload.

Message properties that are serialized in front of the message body are referred to as **message headers**, and message properties that are serialized after the message body property are referred to as **message trailers**.

Messages that are carried by this protocol are treated as payload. The format and structure of the application messages are opaque to the protocol.

The protocol also requires that each message have a lookup identifier that is unique in the queue. This identifier is not part of the message, but is instead assigned by the server.

1.3.2 Queues

A queue is a logical data structure containing an ordered first-in-first-out (FIFO) list of zero or more messages.

This protocol provides a mechanism to open a queue. Opening provides an opportunity to check for the existence of the queue and to perform authorization checks. The protocol provides for the return of an RPC context handle that is used by the client to specify the queue to operate on in subsequent requests. The use of an RPC context handle provides a mechanism to ensure that server state is cleaned up if the connection between the client and server is lost.

When opening a queue, the client can specify an access mode that determines the operations (Peek, Receive, CancelReceive, and Purge) for which the returned handle can subsequently be used. The client can specify a sharing mode that either allows other clients to access the queue concurrently, or ensures that the client has exclusive access to the queue. The exclusive access sharing mode can be used to avoid race conditions caused by other clients operating on the queue at the same time. This sharing mode is specified when opening a remote queue, as specified in [MS-MQMP] section 3.1.4.2.

1.3.3 Queue Operations

The protocol provides mechanisms for the following operations against an open queue.

A message can be consumed from an open queue through a destructive read operation referred to as Receive. This operation atomically reads the message and removes it from the queue. Since this operation removes a message from a queue, losing a network connection during this operation could result in permanent loss of the message.

To guard against this situation, the protocol provides a mechanism for the client to either positively or negatively acknowledge receipt of the message. On receipt of positive acknowledgment from the client, the server can remove the message from the queue. While the server is awaiting acknowledgment from the client, access to the message by other clients is prevented.

A message can be read from an open queue through a nondestructive read operation referred to as "Peek". This operation reads the message, but does not remove it from the queue.

For both Receive and Peek operations, the client can limit the amount of the message body payload returned. This enables efficient use of network resources when the client requires only a portion of the message body, or when the client needs just the message properties.

All the messages can be removed from a queue through a Purge mechanism. The messages removed through this mechanism are not returned to the client.

A client can inform the server that it has no need of a message via a CancelReceive operation. The server can use this indication to inform the sender that the client did not consume the message. How a server implements this notification functionality is not addressed in this specification.

When a client does a destructive read, the message is not deleted from the queue until the client acknowledges receiving the message via an EndReceive operation.

1.3.4 Access Patterns

Messages in a queue can be consumed in a first-in-first-out (FIFO) access pattern. Because messages in a queue are ordered, there is a head, representing the front of the queue, and a tail, representing the end of the queue.

The protocol provides mechanisms to Peek or Receive the first message or the last message in the queue.

The protocol also allows the client to specify exactly which message to Peek or Receive, regardless of its position in the queue, through a unique lookup identifier assigned to each message by the server. A message can also be specified relative to the message identified by the lookup identifier, that is, the message immediately preceding or following the message identified by the lookup identifier.

Finally, the protocol provides a mechanism, referred to as a **cursor**, for sequential forward access through the queue. A cursor logically represents a current pointer that lies between the head and tail of the queue. A cursor can be specified to the Peek or Receive operation, which Peek or Receive the message at the current pointer represented by the cursor. The cursor's current pointer can be used through a modified Peek operation called PeekNext, to do a Peek on the next message in the queue without moving the cursor's current position. A Receive operation intrinsically moves the cursor forward.

Because cursors are stateful, the protocol provides mechanisms to close a cursor opened as specified in [MS-MQMP] section 3.1.4.4. Because a cursor represents a position within a queue, the protocol logically relates the cursor to the context handle associated with an open queue. The protocol places no limit on the number of concurrent cursors associated with a queue context handle.

1.4 Relationship to Other Protocols

This protocol is dependent upon RPC for its transport. This protocol uses RPC, as specified in section 2.1.

Some methods in this RPC interface operate on parameters that are obtained from the qmcomm RPC interface, as specified in [MS-MQMP] section 3.1, and as referenced in section 3 of this document. The server implementation MUST access internal state from the [MS-MQMP] RPC interface, specifically for Queue and Cursor handles. The [MS-MQMP] RPC interface creates Queue and Cursor handles that are required to be passed to the methods in this interface.

This protocol has been deprecated by the RemoteRead RPC interface, as specified in [MS-MQRR].

1.5 Prerequisites/Preconditions

The Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol is an RPC interface, and as a result, has prerequisites, as specified in [MS-RPCE], that are common to RPC interfaces.

It is assumed that the protocol client has obtained the name of a remote computer that supports this protocol before this protocol is invoked. This specification does not mandate how a client acquires this information.

This protocol uses authentication through RPC. The client must be in possession of valid credentials recognized by the server. The server must be started and fully initialized before the protocol can start.

1.6 Applicability Statement

This protocol provides functionality related to consumption of messages from a queue hosted at a queue manager running on a remote computer.<1> It does not provide functionality related to inserting messages into a queue.

The server side of the Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol is applicable for implementation by a queue manager providing message queuing communication services to clients. The client side of this protocol is applicable for implementation by client libraries providing message queuing services to applications, or by a client queue manager delegating requests on behalf of client applications.

1.7 Versioning and Capability Negotiation

Supported Transports: This protocol uses the RPC over TCP/IP protocol sequence, as specified in section 2.1. However, it supports a mechanism for explicitly negotiating the RPC **endpoint** to be used. For more information, see RemoteQMGetQMQMServerPort.

Protocol Versions: This protocol uses a single version of the RPC interface, but that interface has been extended by adding the following methods at the end:<2>

- **RemoteQmGetVersion (section 3.1.4.9)**
- **RemoteQMStartReceive2 (section 3.1.4.10)**
- **RemoteQMStartReceiveByLookupId (section 3.1.4.11)**

1.8 Vendor-Extensible Fields

This protocol uses HRESULTs, as specified in [MS-DTYP] section 1.2.2.16. 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.

1.9 Standards Assignments

Parameter	Value	Reference
RPC Interface **UUID**	{1088a980-eae5-11d0-8d9b-00a02453c337}	As specified in [C706].
Interface Version	1.0	As specified in [C706].

2 Messages

The following sections specify how Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol messages are transported and the common data types for this protocol.

2.1 Transport

This protocol SHOULD use the following **RPC protocol sequence**: RPC over TCP/IP (ncacn_ip_tcp), as defined in [MS-RPCE].<3> This protocol MAY use the RPC over SPX (ncacn_spx) protocol sequence if TCP/IP is unavailable.<4>

This protocol SHOULD use RPC **dynamic endpoints**, as specified in [C706], Part 4. This protocol MAY use an RPC static endpoint, as specified in RemoteQMGetQMQMServerPort, section 3.1.4.8 <5>

This protocol allows any user to establish a connection to the RPC server. The Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol depends on the qmcomm interface, as specified in [MS-MQMP], to use the underlying RPC protocol to retrieve the identity of the invoking client, as specified in [MS-RPCE], section 3.3.3.4.3. The qmcomm server uses this identity to perform method-specific access checks as specified in [MS-MQMP], section 3.1.4.

2.2 Common Data Types

This protocol MUST indicate to the RPC runtime that it is to support both the **NDR** and NDR64 transfer syntaxes and MUST provide a negotiation mechanism for determining which transfer syntax will be used, as specified in [MS-RPCE] section 3.

In addition to the RPC base types and definitions, as specified in [C706] and [MS-RPCE], additional data types are defined below.

The following table summarizes the types defined in this specification.

Data type name	Description
PCTX_RRSESSION_HANDLE_TYPE	A context handle representing an open queue.
PCTX_REMOTEREAD_HANDLE_TYPE	A context handle representing a read session.
REMOTEREADACK	An enumeration which represents an acknowledgment (ACK) or a negative acknowledgment (NACK).
REMOTEREADDESC	A structure used for receiving messages from a queue.
REMOTEREADDESC2	A structure containing the **REMOTEREADDESC** structure and defining an additional element for tracking transaction-related information.

2.2.1 Data Types

2.2.1.1 PCTX_RRSESSION_HANDLE_TYPE

The **PCTX_RRSESSION_HANDLE_TYPE** is a data type that defines an RPC context handle corresponding to an open queue handle. A client MUST call RemoteQMOpenQueue to create a **PCTX_RRSESSION_HANDLE_TYPE** and RemoteQMCloseQueue to delete a **PCTX_RRSESSION_HANDLE_TYPE**.

This type is declared as follows:

```
typedef [context_handle] void* PCTX_RRSESSION_HANDLE_TYPE;
```

2.2.1.2 PCTX_REMOTEREAD_HANDLE_TYPE

The **PCTX_REMOTEREAD_HANDLE_TYPE** is a data type that defines an RPC context handle corresponding to an open read session. A client MUST call RemoteQMStartReceive (Opnum 0), **RemoteQMStartReceive2 (Opnum 9)**, or **RemoteQMStartReceiveByLookupId (Opnum 10)** to create a **PCTX_REMOTEREAD_HANDLE_TYPE** context handle and call **RemoteQMEndReceive (Opnum 1)** to delete the **PCTX_REMOTEREAD_HANDLE_TYPE** handle.

This type is declared as follows:

```
typedef [context_handle] void* PCTX_REMOTEREAD_HANDLE_TYPE;
```

2.2.2 Structures

2.2.2.1 REMOTEREADACK

The **REMOTEREADACK** enumeration represents an acknowledgment (ACK) or a negative acknowledgment (NACK), indicating a successfully or an unsuccessfully delivered packet, respectively. This value is set by the server.

```
typedef  enum _REMOTEREADACK
{
  RR_UNKNOWN = 0x0000,
  RR_NACK = 0x0001,
  RR_ACK = 0x0002
} REMOTEREADACK;
```

RR_UNKNOWN: No acknowledgment.

RR_NACK: Negative acknowledgment for a packet.

RR_ACK: Acknowledgment for a packet.

2.2.2.2 REMOTEREADDESC

This structure is used to encapsulate the information necessary to perform operations **RemoteQMStartReceive**, RemoteQMStartReceive2, and RemoteQMStartReceiveByLookupId.

```
typedef struct _REMOTEREADDESC {
  DWORD hRemoteQueue;
  DWORD hCursor;
  DWORD ulAction;
```

```
  DWORD ulTimeout;
  [range(0, 4325376)] DWORD dwSize;
  DWORD dwpQueue;
  DWORD dwRequestID;
  DWORD Reserved;
  DWORD dwArriveTime;
  REMOTEREADACK eAckNack;
  [unique, size_is(dwSize), length_is(dwSize)]
    byte* lpBuffer;
} REMOTEREADDESC;
```

hRemoteQueue: A handle to the queue as obtained from the *phQueue* parameter of the **qmcomm:R_QMOpenRemoteQueue** method, as specified in [MS-MQMP] section 3.1.4.2. This value is set by the client.

hCursor: If nonzero, specifies a handle to a cursor that MUST have been obtained from a prior call to the **qmcomm:R_QMCreateRemoteCursor** method, as specified in [MS-MQMP] section 3.1.4.4. This value is set by the client.

ulAction: The following table describes possible actions. The Peek and Receive operations both enable access to the contents of a message. This value is set by the client.

Value	Type / Meaning
MQ_ACTION_RECEIVE 0x00000000	Type = Receive Reads and removes a message from the current cursor location if **hCursor** is nonzero or from the front of the queue if **hCursor** is set to zero.
MQ_ACTION_PEEK_CURRENT 0x80000000	Type = Peek Reads a message from the current cursor location if **hCursor** is nonzero or from the front of the queue if **hCursor** is set to zero, but does not remove it from the queue.
MQ_ACTION_PEEK_NEXT 0x80000001	Type = Peek Reads a message following the message at the current cursor location, but does not remove it from the queue.
MQ_LOOKUP_PEEK_CURRENT 0x40000010	Type = Peek Reads the message specified by a lookup identifier, but does not remove it from the queue.
MQ_LOOKUP_PEEK_NEXT 0x40000011	Type = Peek Reads the message following the message specified by a lookup identifier, but does not remove it from the queue.
MQ_LOOKUP_PEEK_PREV 0x40000012	Type = Peek Reads the message preceding the message specified by a lookup identifier, but does not remove it from the queue.
MQ_LOOKUP_PEEK_FIRST 0x40000014	Type = Peek Reads the first message in the queue, but does not remove it from the queue

Value	Type / Meaning
MQ_LOOKUP_PEEK_LAST 0x40000018	Type = Peek Reads the last message in the queue, but does not remove it from the queue.
MQ_LOOKUP_RECEIVE_CURRENT 0x40000020	Type = Receive Reads the message specified by a lookup identifier and removes it from the queue.
MQ_LOOKUP_RECEIVE_NEXT 0x40000021	Type = Receive Reads the message following the message specified by a lookup identifier and removes it from the queue.
MQ_LOOKUP_RECEIVE_PREV 0x40000022	Type = Receive Reads the message preceding the message specified by a lookup identifier and removes it from the queue.
MQ_LOOKUP_RECEIVE_FIRST 0x40000024	Type = Receive Reads the first message in the queue and removes it from the queue.
MQ_LOOKUP_RECEIVE_LAST 0x40000028	Type = Receive Reads the last message in the queue and removes it from the queue.

ulTimeout: Specifies a timeout in milliseconds for the server to wait for a message to become available in the queue. This value is set by the client. To specify an infinite timeout, the client MUST set this field to 0xFFFFFFFF.

dwSize: Specifies the maximum size, in bytes, of **lpBuffer**. The valid range is 0 to 4325376. This value is set by the server.

dwpQueue: A **DWORD** pointer to the queue as obtained from the **qmcomm:R_QMOpenRemoteQueue** method, as specified in [MS-MQMP] section 3.1.4.2. This value is set by the client.

dwRequestID: MUST be set by the client to a unique correlation identifier for the receive request. This value is set by the client.

Reserved: The client MUST set this parameter to 0x00000000. The server SHOULD set this value to 0x00000001.<6>

Value	Meaning
0x00000000	Returned by client.
0x00000001	Returned by server.

dwArriveTime: The server MUST set this value to the time that the message was added to the queue. MUST be expressed as the number of seconds elapsed since midnight 00:00:00.0, January 1, 1970 Coordinated Universal Time (UTC).

eAckNack: This field represents an acknowledgment (ACK) or a negative acknowledgment (NACK), indicating a successfully or an unsuccessfully delivered packet, respectively. This value is set by the client. The server MUST NOT use this value for processing.

Value	Meaning
RR_UNKNOWN 0x0000	No acknowledgment.
RR_NACK 0x0001	Negative acknowledgment for a packet.
RR_ACK 0x0002	Acknowledgment for a packet.

lpBuffer: This field represents a pointer to the message. This field is unique, and its size is **dwSize**. This value is set by the server.

2.2.2.3 REMOTEREADDESC2

This structure is used by RemoteQMStartReceive2 and RemoteQMStartReceiveByLookupId to encapsulate the parameters necessary for execution of these operations.

```
typedef struct _REMOTEREADDESC2 {
  REMOTEREADDESC* pRemoteReadDesc;
  double SequentialId;
} REMOTEREADDESC2;
```

pRemoteReadDesc: A pointer to a **REMOTEREADDESC** structure, as specified in section 2.2.2.2.

SequentialId: This field is set to the value of the **TransactionHeader.TxSequenceNumber** field of the UserMessage packet that delivered the message to the queue manager. For more information, see [MS-MQQB] section 2.2.8.5. This value is set by the server.

3 Protocol Details

The following sections specify details of the Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol including the abstract data model, interface method syntax, and message processing rules.

3.1 qm2qm 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 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.

Servers MUST maintain the following data elements:

Queue Table: A table of queues deployed at the server, keyed by name. Each entry MUST contain:

- The name of the queue.
- The current queue state, as specified in the Queue State Diagram (section 3.1.1.1).
- The queue shared open count.
- A table of queue properties, as specified in [MS-MQMQ] section 2.3.1.
- A message list.

Message List: A linked list of messages in a queue, in ascending order by arrival time within message priority. Each entry MUST contain:

- A server-assigned lookup identifier for the message, unique across all messages in the queue.
- A message locked flag indicating if the message is locked by a pending request.

OpenSessionHandle Table: A table of OpenSessionHandles. Each entry MUST contain:

- OpenSessionHandle.
- QueueContextHandle.
- Reference to queue table entry.

Cursor Table: A table of cursor handles. Each entry MUST contain:

- A cursor handle.
- A ReadSessionHandle
- A QueueContextHandle.
- The current position of the cursor within the queue.

- The current cursor state, as specified by the states in Cursor State Diagram (section 3.1.1.2).

Pending Request Table: Represents a table of pending requests. Each entry MUST contain:

- ReadSessionHandle
- The type of pending request set to one of the following values:
  - ReceiveWaitForMessage
  - PeekWaitForMessage
  - ReceiveWaitForEndOrCancel
- A request ID, supplied by the client.
- The QueueContextHandle specified with the request.
- The expiration time of the request.
- The LookupId of a message associated with the request, if a LookupId was associated with the request.
- The cursor handle associated with the request, if a cursor was associated with the request.

Note QueueContextHandle is obtained when the client calls the qmcomm:R_QMOpenRemoteQueue method, as specified in [MS-MQMP] section 3.1.4.2. This value is passed to the Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol methods.

3.1.1.1 Queue State Diagram

The following diagram specifies the state transitions for a queue. In particular, it specifies the queue state transitions resulting from Open and Close events, and how the sharing mode parameter specified on Open affects these state transitions.

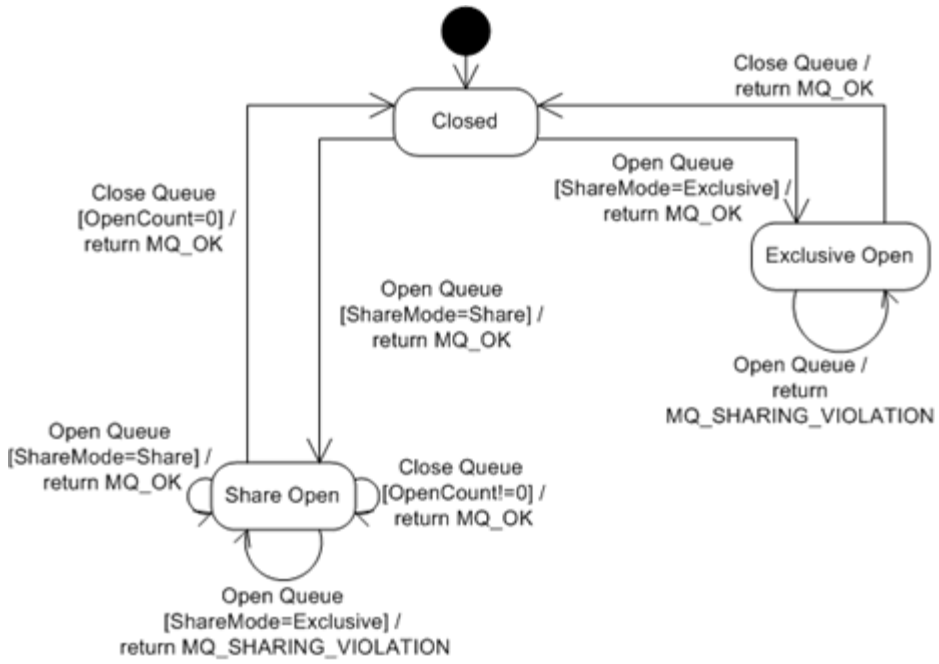

Figure 1: State transitions for Open and Close events

3.1.1.2 Cursor State Diagram

The following diagram specifies the state transitions for a cursor. A cursor represents a position in a queue. The diagram specifies how cursor state transitions are affected by PeekCurrent, PeekNext, Receive, Message Added, and Message Deleted events, as well as by the presence or absence of messages in the queue.

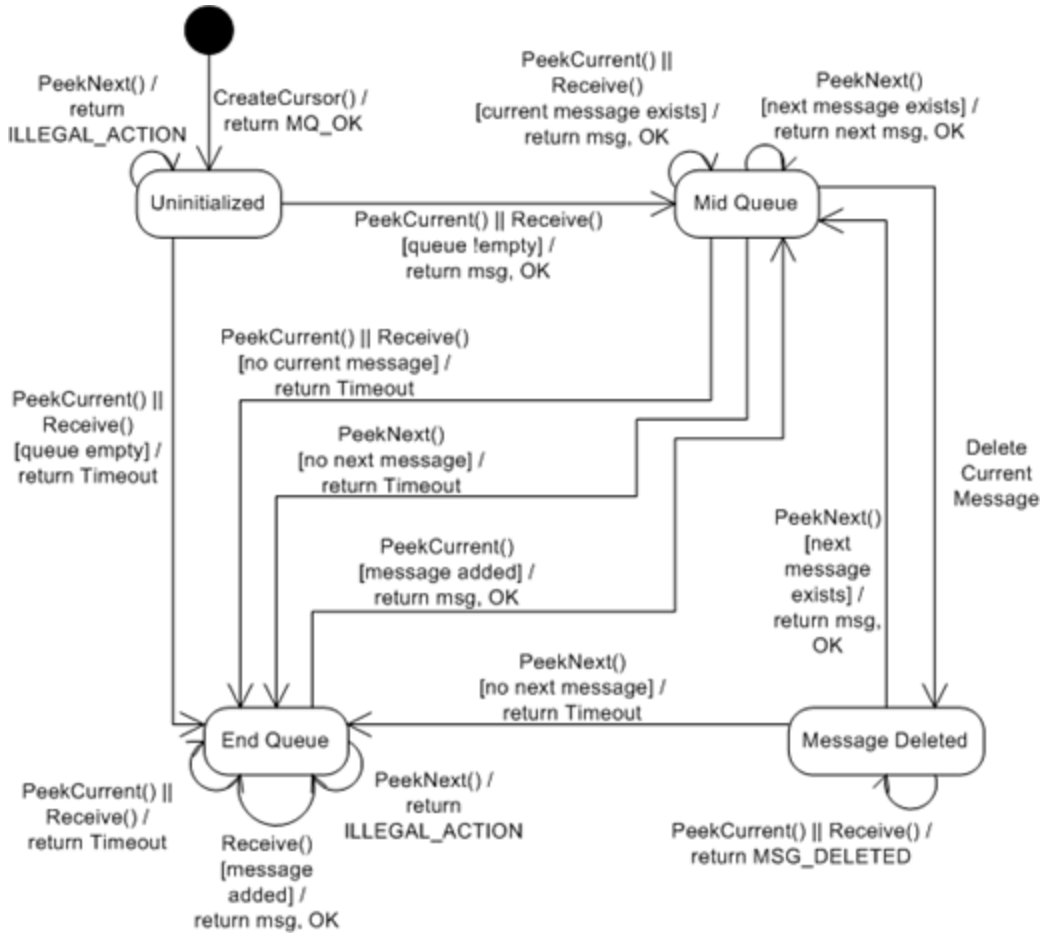

Figure 2: Cursor state transitions

3.1.2 Timers

Beyond protocol timers used internally by RPC to implement resiliency to network outages (for more information, see [MS-RPCE]), the server MUST maintain the following timers:

Call Timer: The server MUST maintain a per-call timer for each call to **RemoteQMStartReceive** or RemoteQMStartReceive2 in which the REMOTEREADDESC.ulTimeout parameter is nonzero. The timer MUST be set to the REMOTEREADDESC.ulTimeout parameter that is specified on the call.

3.1.3 Initialization

The client MUST create an RPC connection to the remote computer, using the details specified in section 2.1.

The client MUST call the RemoteQMGetQMQMServerPort method. This method returns an RPC endpoint port number for the RPC over TCP/IP protocol sequence as defined in [MS-RPCE] as specified by the client. The client MUST create an RPC binding handle to the remote computer using the returned RPC port. All other method calls on the server interface MUST use the resulting binding handle.

3.1.4 Message Processing Events and Sequencing Rules

This protocol MUST indicate to the RPC runtime that it is to perform a strict NDR data consistency check at target level 6.0, as specified in [MS-RPCE] section 3.

The qm2qm interface includes the following methods:

Methods in RPC Opnum Order

Method	Description
RemoteQMStartReceive	Initiates a Receive or Peek request on the queue. Opnum: 0
RemoteQMEndReceive	Finishes a Receive request. Opnum: 1
RemoteQMOpenQueue	Opens a queue. Opnum: 2
RemoteQMCloseQueue	Closes a queue. Opnum: 3
RemoteQMCloseCursor	Closes a cursor. Opnum: 4
RemoteQMCancelReceive	Cancels a pending Receive request. Opnum: 5
RemoteQMPurgeQueue	Deletes all messages in a queue. Opnum: 6
RemoteQMGetQMQMServerPort	Returns an RPC endpoint port number to use in subsequent calls on the interface. Opnum: 7
RemoteQmGetVersion	Returns the server version. Opnum: 8
RemoteQMStartReceive2	Initiates a Receive or Peek request on the queue by using a sequential ID. Opnum: 9
RemoteQMStartReceiveByLookupId	Initiates a Receive or Peek request on the queue by using a lookup ID. Opnum: 10

3.1.4.1 RemoteQMStartReceive (Opnum 0)

The **RemoteQMStartReceive** method peeks or receives a message from an open queue.

If **RemoteQMStartReceive** is invoked with a Peek action type, as specified in the *ulAction* member of the *lpRemoteReadDesc* parameter, the operation completes when RemoteQMStartReceive returns.

If **RemoteQMStartReceive** is invoked with a receive action type, as specified in the *ulAction* member of the *lpRemoteReadDesc* parameter, the client MUST pair each call to **RemoteQMStartReceive** with a call to **RemoteQMEndReceive** to complete the operation, or to **RemoteQMCancelReceive** to cancel the operation.

For each call to **RemoteQMCancelReceive**, the *dwRequestID* parameter MUST match the *dwRequestID* member of the *lpRemoteReadDesc* parameter in a previous call to **RemoteQMStartReceive**.

If the client specifies a nonzero value for the **ulTimeout** member of the *lpRemoteReadDesc* parameter, and a message is not available in the queue at the time of the call, the server waits up to the specified timeout for a message to become available in the queue before responding to the call. The client can call **RemoteQMCancelReceive** with a matching REMOTEREADDESC.dwRequestID to cancel the pending **RemoteQMStartReceive** request.

Before calling this method, the client MUST have already called **RemoteQMOpenQueue**.

```
HRESULT RemoteQMStartReceive(
  [in] handle_t hBind,
  [out] PCTX_REMOTEREAD_HANDLE_TYPE* pphContext,
  [in, out] REMOTEREADDESC* lpRemoteReadDesc
);
```

hBind: An RPC binding handle parameter, as specified in [MS-RPCE] section 2, that MUST be specified.

pphContext: The server MUST return a value for this handle. This handle will be used by the client in subsequent calls to **RemoteQMCancelReceive** and **RemoteQMEndReceive**.

lpRemoteReadDesc: A pointer to an instance of a **REMOTEREADDESC** structure. This value MUST NOT be set to NULL.

In addition, the *ulAction* member of the *lpRemoteReadDesc* parameter MUST be set by the client to one of the following values. In the table below, *ulAction* and *hCursor* are semantically equivalent to *lpRemoteReadDesc.ulAction* and *lpRemoteReadDesc.hCursor*, respectively, and the '.' operator is the same as '->'.

Value of *ulAction*	Meaning
MQ_ACTION_RECEIVE 0x00000000	If *hCursor* is nonzero, read and remove the message for the current cursor location and advance the cursor to the next position. If *hCursor* is zero, read and remove the message from the front of the queue.
MQ_ACTION_PEEK_CURRENT 0x80000000	If *hCursor* is nonzero, read the message at the current cursor location but do not remove it from the queue. If *hCursor* is zero, read the message at the front of the queue but do not remove it from the queue.
MQ_ACTION_PEEK_NEXT 0x80000001	Read the message following the message at the current cursor location but do not remove it. The *hCursor* parameter MUST be set to a nonzero cursor handle that is obtained from the **qmcomm:R_QMCreateRemoteCursor** method, as specified in [MS-MQMP] section 3.1.4.4.

If the *hCursor* member of *lpRemoteReadDesc* is nonzero, the handle MUST NOT have been closed through a prior call to **RemoteQMCloseCursor**.

The dwRequestID member of the *lpRemoteReadDesc* parameter MUST be used in a subsequent call to **RemoteQMEndReceive (Opnum 1)** or **RemoteQMCancelReceive (Opnum 5)** to correlate that call with the call to **RemoteQMStartReceive (Opnum 0)**. The value MUST NOT be used in another **RemoteQMStartReceive (Opnum 0)** call on the same QueueContextHandle until a call to either **RemoteQMEndReceive** or **RemoteQMCancelReceive** with the same dwRequestID value has been completed.

Return Values: The method MUST return ERROR_SUCCESS (0x00000000) on success; otherwise, it MUST return an implementation-specific nonzero value.

Exceptions Thrown: None except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

While processing this method, the server MUST look up the dwpQueue member of the *lpRemoteReadDesc* parameter in the OpenSessionHandle table as QueueContextHandle.

If the message is found:

- Find the message in the queuemessage list as specified in the *ulAction* member of the *lpRemoteReadDesc* parameter description:
  - If the *hCursor* parameter is nonzero and the *ulAction* member of the *lpRemoteReadDesc* parameter is MQ_ACTION_RECEIVE or MQ_ACTION_PEEK_CURRENT, the message is at the current cursor location.
  - If the *ulAction* member of the *lpRemoteReadDesc* parameter is MQ_ACTION_PEEK_NEXT, the message is the first available message found starting immediately following the message at the current cursor location and walking toward the end of the queue.
  - If the *hCursor* parameter is zero and the *ulAction* member of the *lpRemoteReadDesc* parameter is MQ_ACTION_RECEIVE or MQ_ACTION_PEEK_CURRENT, the message is the first available message found starting at the front of the queue and "walking" toward the end of the queue.

If the message is not found:

- If the ulTimeout member of the *lpRemoteReadDesc* parameter is 0x00000000:
  - Return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0008).
- If the ulTimeout member of the *lpRemoteReadDesc* parameter is not 0x00000000:
  - Add an entry to the pending request table with the following:
    - If the *ulAction* type, as defined in the table under the *ulAction* parameter, is Receive, set the entry type to ReceiveWaitForMessage; otherwise, set the entry type to PeekWaitForMessage.
    - Set the request ID to the dwRequestID member of the *lpRemoteReadDesc* parameter.
    - Set the QueueContextHandle to the dwpQueue member of the *lpRemoteReadDesc* parameter.
    - Create a new ReadSessionHandle.

- Set the ReadSessionHandle to pphContext.
- Set the Request timeout to the ulTimeout member of the *lpRemoteReadDesc* parameter.
- Set the cursor handle to the *hCursor* member of the *lpRemoteReadDesc* parameter.

- Create a Call Timer which waits for the message for the number of milliseconds specified in the ulTimeout member of the *lpRemoteReadDesc* parameter.
- If the message is not found before the timeout expires:
  - Return MQ_ERROR_IO_TIMEOUT (0xC00E001B).

Fill the lpBuffer member of the *lpRemoteReadDesc* parameter with the message.

If the value of the *ulAction* member of the *lpRemoteReadDesc* parameter, as defined in the table under the *ulAction* parameter, is Receive, the server MUST do the following:

- Set the message locked flag to TRUE, indicating that the message is locked by the pending request.
- Add an entry to the pending request table and do the following:
  - Set the type to ReceiveWaitForEndOrCancel.
  - Set the request ID to the dwRequestID member of the *lpRemoteReadDesc* parameter.
  - Set the QueueContextHandle to the dwpQueue member of the *lpRemoteReadDesc* parameter.
  - Create a new ReadSessionHandle.
  - Set the ReadSessionHandle to pphContext.
  - Set the request timeout to 0x00000000.
  - Set the cursor handle to the *hCursor* member of the *lpRemoteReadDesc* parameter.

Else (the dwpQueue member of the *lpRemoteReadDesc* parameter is not found in the QueueContextHandle table), return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.2 RemoteQMEndReceive (Opnum 1)

The client MUST invoke the **RemoteQMEndReceive** method to advise the server that the message packet returned by the **RemoteQMStartReceive**, **RemoteQMStartReceive2**, or **RemoteQMStartReceiveByLookupId** method has been received.

The combination of the **RemoteQMStartReceive**, **RemoteQMStartReceive2**, or **RemoteQMStartReceiveByLookupId** method and the positive acknowledgment of the **RemoteQMEndReceive** method ensures that a message packet is not lost in transit from the server to the client due to a network outage during the call sequence.

Before calling this method, the following methods MUST be called:

- RemoteQMOpenQueue
- RemoteQMStartReceive, RemoteQMStartReceive2, or RemoteQMStartReceiveByLookupId.

```
HRESULT RemoteQMEndReceive(
```

```
  [in] handle_t hBind,
  [in, out] PCTX_REMOTEREAD_HANDLE_TYPE* pphContext,
  [in, range(1, 2)] DWORD dwAck
);
```

hBind: MUST be an RPC binding handle parameter for use by the server, as specified in [MS-RPCE] section 2.

pphContext: MUST be set by the client to a pointer to a context handle obtained from the **RemoteQMStartReceive** method.

dwAck: An acknowledgment (ACK) or negative acknowledgment (NACK) set by the client about the status of the message packet requested from **RemoteQMStartReceive**, **RemoteQMStartReceive2**, or **RemoteQMStartReceiveByLookupId**.

Value	Meaning
RR_NACK 0x00000001	The client acknowledges that the message packet was not delivered successfully. The server MUST keep the message in the queue and make it available for subsequent consumption.
RR_ACK 0x00000002	The client acknowledges that the message packet was delivered successfully. The server MUST remove the message from the queue and make it unavailable for subsequent consumption.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure HRESULTs identically.

MQ_OK (0x00000000)

Exceptions Thrown: No exceptions are thrown except those thrown by the underlying RPC protocol, [MS-RPCE].

When processing this call, the server MUST:

- Look up the {pphContext} in the pending requests table as ReadSessionHandle.
- If found, and the type is ReceiveWaitForEndOrCancel:
  - Find the message identified by the pending request entry.
  - If the supplied *dwAck* parameter value is RR_ACK (0x00000002), remove the message from the message list.
  - If the supplied *dwAck* parameter value is RR_NACK (0x00000001), set the message locked flag to FALSE, indicating that the message is no longer locked by the pending request.
  - Remove the entry from the pending request table.
- Else, return MQ_ERROR_OPERATION_CANCELLED (0xC00E0008).

3.1.4.3 RemoteQMOpenQueue (Opnum 2)

The **RemoteQMOpenQueue** method opens a queue in preparation for subsequent operations against it. This method assumes that the client has called **qmcomm:R_QMOpenRemoteQueue** to obtain a queue handle; for more information, see [MS-MQMP] section **3.1.4.2**. This method MUST be called prior to calling any of the following operations:

- **RemoteQMStartReceive**
- **RemoteQMEndReceive**
- **RemoteQMCloseQueue**
- **RemoteQMCloseCursor**
- **RemoteQMCancelReceive**
- **RemoteQMPurgeQueue**
- **RemoteQMGetQMQMServerPort**
- **RemoteQMStartReceive2**
- **RemoteQMStartReceiveByLookupId**

```
HRESULT RemoteQMOpenQueue(
  [in] handle_t hBind,
  [out] PCTX_RRSESSION_HANDLE_TYPE* phContext,
  [in] GUID* pLicGuid,
  [in, range(0, 16)] DWORD dwMQS,
  [in] DWORD hQueue,
  [in] DWORD pQueue,
  [in] DWORD dwpContext
);
```

hBind: MUST be set to an RPC binding handle, as specified in [MS-RPCE] section 2.

phContext: A pointer to a context handle that contains the information about the opened queue, which corresponds to the abstract data model's OpenSessionHandle. The server MUST set this value; it gets deleted on a call to **RemoteQMCloseQueue**.

pLicGuid: MUST be a pointer to a valid **GUID** which uniquely identifies the client.

dwMQS: MUST be set to 0x00000000 by client.

hQueue: A queue identifier returned by the **qmcomm:R_QMOpenRemoteQueue** method in the *pphContext* parameter, as specified in [MS-MQMP] section **3.1.4.2**.

pQueue: A **DWORD** that contains a pointer to a context value for the remote queue. This value MUST be set to the *hQueue* parameter value.

dwpContext: A **DWORD** that contains a pointer to a context value for the remote queue. This value MUST be set to the *pQueue* parameter value.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure HRESULTs identically.

MQ_OK (0x00000000)

Exceptions Thrown:

No exceptions are thrown except those thrown by the underlying RPC protocol (see [MS-RPCE]).

When processing this call, the server MUST do the following:

Lookup the queue in the queue table. If not found, throw MQ_ERROR_QUEUE_NOT_FOUND (0xC00E0003).

Add a new OpenSessionHandle table entry, and do the following:

- Create a new OpenSessionHandle
- Set the QueueContextHandle to *pQueue*.
- Add a reference to the queue entry.

Set *phContext* to the OpenSessionHandle value.

3.1.4.4 RemoteQMCloseQueue (Opnum 3)

The **RemoteQMCloseQueue** method closes a **PCTX_RRSESSION_HANDLE_TYPE** that was previously opened by using a call to the **RemoteQMOpenQueue** method. The client MUST call this method to reclaim resources on the server allocated by the **RemoteQMOpenQueue** method.

```
HRESULT RemoteQMCloseQueue(
  [in] handle_t hBind,
  [in, out] PCTX_RRSESSION_HANDLE_TYPE* pphContext
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

pphContext: MUST be set to the same **PCTX_RRSESSION_HANDLE_TYPE** returned by a previous call to **RemoteQMOpenQueue**.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

Exceptions Thrown: No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST:

- Look up *pphContext* in the OpenSessionHandle table.
- If found:
  - Find all entries in the pending request table that contain the QueueContextHandle and cancel the operations.

- Find all entries in the cursor table that are associated with the QueueContextHandle and execute the same steps as specified in **RemoteQMCloseCursor**.
- If the queue state in the associated queue entry is Exclusive Open:
  - Transition the queue state to Closed.
- Else:
  - Decrement the shared open count.
  - If the shared open count is zero, transition the queue state to Closed.
- Remove the entry from the OpenSessionHandle table.
- Set pphContext to NULL.

- Else (pphContext is not found in the OpenSessionHandle table):
  - Return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.5 RemoteQMCloseCursor (Opnum 4)

The **RemoteQMCloseCursor** method closes the handle for a previously created cursor. The client MUST call this method to reclaim resources on the server allocated by the **qmcomm:R_QMCreateRemoteCursor** method, as specified in [MS-MQMP] section **3.1.4.4**.

```
HRESULT RemoteQMCloseCursor(
  [in] handle_t hBind,
  [in] DWORD hQueue,
  [in] DWORD hCursor
);
```

hBind: MUST be set to an RPC binding handle, as specified in [MS-RPCE] section 2.

hQueue: A queue handle value acquired from the **qmcomm:R_QMOpenRemoteQueue** method as specified in [MS-MQMP] section **3.1.4.2**.

hCursor: Specifies the cursor handle. MUST NOT be NULL on input. This value is originally obtained by the client from the **R_QMCreateRemoteCursor** method of [MS-MQMP].

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

Exceptions Thrown: No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST:

- Look up the {*hCursor*} in the cursor table.

If found:

- For each entry in the pending request table that contains the cursor handle value matching *hCursor* and QueueContextHandle matching *hQueue*, cancel the operation, as specified in RemoteQMCancelReceive.
- Remove the entry from the cursor table.

Else, return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.6 RemoteQMCancelReceive (Opnum 5)

The **RemoteQMCancelReceive** method cancels a pending call to RemoteQMStartReceive and provides a way for the client to cancel a blocked request.

Before calling this method, the following methods MUST be called:

- **RemoteQMOpenQueue**.
- **RemoteQMStartReceive**, **RemoteQMStartReceive2**, or **RemoteQMStartReceiveByLookupId**.

```
HRESULT RemoteQMCancelReceive(
  [in] handle_t hBind,
  [in] DWORD hQueue,
  [in] DWORD pQueue,
  [in] DWORD dwRequestID
);
```

hBind: MUST be set to an RPC binding handle, as specified in [MS-RPCE] section 2.

hQueue: A queue handle value acquired from the **qmcomm:R_QMOpenRemoteQueue** method, as specified in [MS-MQMP] section **3.1.4.2**.

pQueue: Contains a pointer to the open queue. MUST be equal to the hQueue parameter value.

dwRequestID: This value MUST be set to the value of the dwRequestID member of the REMOTEREADDESC or REMOTEREADDESC2 input parameters for **RemoteQMStartReceive** , **RemoteQMStartReceive2** or **RemoteQMStartReceiveByLookupId**.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

Exceptions Thrown: No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST:

- Look up the {dwRequestID} as request id in the pending requests table.

If the request type is ReceiveWaitForMessage or PeekWaitForMessage:

- If a call timer is associated with the call, cancel the timer.
- Respond to the pending RemoteQMStartReceive request with MQ_ERROR_OPERATION_CANCELLED (0xC00E0008).

- Remove the entry from the pending requests table.

Else, return MQ_ERROR_OPERATION_CANCELLED (0xC00E0008).

3.1.4.7 RemoteQMPurgeQueue (Opnum 6)

The **RemoteQMPurgeQueue** method removes all messages from the queue.

Before calling this method, the **RemoteQMOpenQueue** method MUST be called.

```
HRESULT RemoteQMPurgeQueue(
  [in] handle_t hBind,
  [in] DWORD hQueue
);
```

hBind: MUST specify an RPC binding handle parameter, as specified in [MS-RPCE] section 2.

hQueue: A queue handle value acquired from the *phQueue* parameter of the **qmcomm:R_QMOpenRemoteQueue** method as specified in [MS-MQMP] section **3.1.4.2**.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULTs** identically.

Exceptions Thrown: No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

When processing this call, the server MUST lookup *hQueue* in the QueueContextHandle table.

If found:

- Remove all messages from the message list associated with the queue. If the message has its locked flag set to TRUE, it MUST NOT be removed.

Else, return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.4.8 RemoteQMGetQMQMServerPort (Opnum 7)

The **RemoteQMGetQMQMServerPort** method returns an RPC port number (see [MS-RPCE]) for the requested combination of interface and protocol.

```
DWORD RemoteQMGetQMQMServerPort(
  [in] handle_t hBind,
  [in, range(0, 1)] DWORD dwPortType
);
```

hBind: MUST be set to an RPC binding handle, as specified in [MS-RPCE] section 2.

dwPortType: Specifies the interface for which a port value is to be returned. One of the following values MUST be specified; otherwise, this method MUST return 0x00000000 to indicate failure.

Value	Meaning
IP_HANDSHAKE 0x00000000	Requests that the server return the RPC port number for the qmcomm and qmcomm2 interfaces bound to TCP/IP. For more information on the qmcomm and qmcomm2 interfaces see [MS-MQMP].
IP_READ 0x00000001	Requests that the server return the RPC port number for the qm2qm interface bound to TCP/IP. For more information on the qm2qm interface see section 3.1.4.
IPX_HANDSHAKE 0x00000002	Requests that the server return the RPC port number for the qmcomm and qmcomm2 interfaces bound to SPX.<7> For more information on the qmcomm and qmcomm2 interfaces see [MS-MQMP].
IPX_READ 0x00000003	Requests that the server return the RPC port number for the qm2qm interface bound to SPX.<8> For more information on the qm2qm interface see section 3.1.4.

Return Values: On success, this method returns a nonzero IP port value for the RPC interface specified by the dwPortType parameter. If an invalid value is specified for dwPortType, or if the requested interface is otherwise unavailable, or if any other error is encountered, this method MUST return 0x00000000.

Exceptions Thrown: No exceptions are thrown beyond those thrown by the underlying RPC protocol (see [MS-RPCE]).

3.1.4.9 RemoteQmGetVersion (Opnum 8)

The **RemoteQmGetVersion** method retrieves the Message Queuing version of the server; this method is called before the **RemoteQMOpenQueue** method.<9>

```
void RemoteQmGetVersion(
  [in] handle_t hBind,
  [out] unsigned char* pMajor,
  [out] unsigned char* pMinor,
  [out] unsigned  short* pBuildNumber
);
```

hBind: MUST be set to an RPC binding handle, as specified in [MS-RPCE] section 2.

pMajor: A pointer to a character array. The server SHOULD <10> set this parameter to the current major build number. .

pMinor: A pointer to a character array. The server SHOULD <11> set this parameter to the current minor build number.

pBuildNumber: A pointer to a character array. The server SHOULD <12>set this parameter to the current build number.

Return Values: This method has no return values.

Exceptions Thrown: No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

Since the Remote Read Protocol, as specified in [MS-MQRR] deprecates the Queue Manager to Queue Manager Protocol, as mentioned in section 1.4, the client SHOULD<13> use the **RemoteQmGetVersion** method to determine which Protocol to use.

3.1.4.10 RemoteQMStartReceive2 (Opnum 9)

The **RemoteQMStartReceive2** method functions in the same way as **RemoteQMStartReceive**, with the exception that it returns a structure that contains the SequentialId of the message.<14>

```
HRESULT RemoteQMStartReceive2(
  [in] handle_t hBind,
  [out] PCTX_REMOTEREAD_HANDLE_TYPE* pphContext,
  [in, out] REMOTEREADDESC2* lpRemoteReadDesc2
);
```

hBind: MUST be set to an RPC binding handle, as specified in [MS-RPCE] section 2.

pphContext: The server MUST return a value for this handle. This handle will be used by the client in subsequent calls to **RemoteQMCancelReceive** and **RemoteQMEndReceive**.

lpRemoteReadDesc2: A pointer to an instance of a **REMOTEREADDESC2** structure. The members of the pRemoteReadDesc member of the lpRemoteReadDesc2 pointer should be assigned in the same manner as that described in **RemoteQMStartReceive**.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure **HRESULT**, and the client MUST treat all failure **HRESULT**s identically.

Exceptions Thrown: No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

The processing for this method is the same as in **RemoteQMStartReceive**, with the exception that it returns the SequentialId of the message in the SequentialId member of the lpRemoteReadDesc2 pointer.

3.1.4.11 RemoteQMStartReceiveByLookupId (Opnum 10)

The **RemoteQMStartReceiveByLookupId** method reads a message from the opened remote queue by using the lookup identifier.<15>

```
HRESULT RemoteQMStartReceiveByLookupId(
  [in] handle_t hBind,
  [in] double LookupId,
  [out] PCTX_REMOTEREAD_HANDLE_TYPE* pphContext,
  [in, out] REMOTEREADDESC2* lpRemoteReadDesc2
);
```

hBind: MUST be set to an RPC binding handle, as specified in [MS-RPCE] section 2.

LookupId: Lookup identifier of the message to be returned. **MSMQ** assigns a lookup identifier to a message when it is placed in a queue.

pphContext: The server MUST return a value for this handle, which is used by the client in subsequent calls to **RemoteQMEndReceive**.

lpRemoteReadDesc2: A REMOTEREADDESC2 instance that contains the remote description accompanied by a sequential ID. The members of the pRemoteReadDesc member of the *lpRemoteReadDesc2* parameter should be assigned in the same manner as that specified in **RemoteQMStartReceive** and 2.2.2.2.

lpRemoteReadDesc2.pRemoteReadDesc.ulAction MUST be set to one of the following values. In the table below, *ulAction*, *hCursor* and *ulTimeout* are semantically equivalent to *lpRemoteReadDesc2.pRemoteReadDesc.ulAction*, *lpRemoteReadDesc2.pRemoteReadDesc.hCursor* and *lpRemoteReadDesc2.pRemoteReadDesc.ulTimeout*, respectively, and the '.' operator is the same as '->'.

Value of *ulAction*	**Meaning**
MQ_LOOKUP_PEEK_CURRENT 0x40000010	Read the message that is specified by the *LookupId* parameter but do not remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_PEEK_NEXT 0x40000011	Read the message following the message that is specified by *LookupId* but do not remove it. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_PEEK_PREV 0x40000012	Read the message preceding the message that is specified by the *LookupId* parameter but do not remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_PEEK_FIRST 0x40000014	Read the first message in the queue but do not remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_PEEK_LAST 0x40000018	Read the last message in the queue but do not remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_CURRENT 0x40000020	Read the message that is specified by the *LookupId* parameter and remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.

Value of *ulAction*	Meaning
MQ_LOOKUP_RECEIVE_NEXT 0x40000021	Read the message following the message that is specified by the *LookupId* parameter and remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_PREV 0x40000022	Read the message preceding the message that is specified by the *LookupId* parameter and remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST NOT be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_FIRST 0x40000024	Read the first message in the queue and remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.
MQ_LOOKUP_RECEIVE_LAST 0x40000028	Read the last message in the queue and remove it from the queue. The *hCursor* parameter MUST be set to zero. The *LookupId* parameter MUST be set to 0. The *ulTimeout* parameter MUST be set to 0x00000000.

Return Values: On success, this method MUST return MQ_OK (0x00000000).

If an error occurs, the server MUST return a failure HRESULT, and the client MUST treat all failure HRESULTs identically.

Exceptions Thrown: No exceptions are thrown except those thrown by the underlying RPC protocol, as specified in [MS-RPCE].

Note In the following text, *dwpQueue*, *ulAction*, and *lpBuffer* are semantically equivalent to the fully qualified *lpRemoteReadDesc2.pRemoteReadDesc.member* notation, where the '.' operator is the same as '->'.

While processing this method, the server MUST:

- Lookup *dwpQueue* in the OpenSessionHandle table as QueueContextHandle.

If found, find the message in the queuemessage list as specified in the *ulAction* parameter description:

- If the *ulAction* parameter is MQ_LOOKUP_PEEK_CURRENT or MQ_LOOKUP_RECEIVE_CURRENT, the message is specified by the *LookupId* parameter.
- If the *ulAction* parameter is MQ_LOOKUP_PEEK_PREV or MQ_LOOKUP_RECEIVE_PREV, the message is the first available message found starting immediately preceding the message specified by the *LookupId* parameter and walking toward the front of the queue.
- If the *ulAction* parameter is MQ_LOOKUP_PEEK_NEXT or MQ_LOOKUP_RECEIVE_NEXT, the message is the first available message found starting immediately following the message specified by the *LookupId* parameter and walking toward the end of the queue.

- If the *ulAction* parameter is MQ_LOOKUP_PEEK_FIRST or MQ_LOOKUP_RECEIVE_FIRST, the message is the first available message found starting at the front of the queue and walking towards the end of the queue.
- If the *ulAction* parameter is MQ_LOOKUP_PEEK_LAST or MQ_LOOKUP_RECEIVE_LAST, the message is the first available message found starting at the end of the queue and working towards the front of the queue.

If the message is not found:

- Return MQ_ERROR_MESSAGE_NOT_FOUND (0xC00E0008).

Fill the lpBuffer with the message.

If the *ulAction* type, as defined in the table under the *ulAction* parameter, is Receive, the server MUST do the following:

- Set the message locked flag to TRUE, indicating that the message is locked by the pending request.

Else (*dwpQueue* is not found in the QueueContextHandle table), return MQ_ERROR_INVALID_HANDLE (0xC00E0007).

3.1.5 Timer Events

The server MUST maintain a timer associated with each invocation of **RemoteQMStartReceive2**, **RemoteQMStartReceive**, or **RemoteQMStartReceiveByLookupId** in which the *ulTimeout* parameter is nonzero. The length of time to wait is specified by the client in the *ulTimeout* parameter of the request. If the server is unable to satisfy the request within the specified timeout, it MUST return MQ_ERROR_IO_TIMEOUT (0xC00E001B).

3.1.5.1 Call Timer Expired

The server sets a per-call timer for each invocation of **RemoteQMStartReceive2**, **RemoteQMStartReceive**, or **RemoteQMStartReceiveByLookupId** in which the *ulTimeout* parameter is nonzero. If the server is unable to satisfy the pending request and the timer expires, the server receives this event.

While processing this event, the server MUST:

- Look up the entry in the pending request table associated with the ReadSessionHandles that are passed as parameters to the **RemoteQMStartReceive2**, **RemoteQMStartReceive**, or **RemoteQMStartReceiveByLookupId** methods.
- Cancel the call to **RemoteQMStartReceive2**, **RemoteQMStartReceive**, or **RemoteQMStartReceiveByLookupId**; return MQ_ERROR_IO_TIMEOUT (0xC00E001B) to the caller.
- Remove the entry from the pending request table.

3.1.6 Other Local Events

The following local events trigger operations on the server:

- **PCTX_RRSESSION_HANDLE_TYPE** rundown.
- **PCTX_REMOTEREAD_HANDLE_TYPE** rundown.

- Message added to the queue.
- Message deleted from the queue.

3.1.6.1 PCTX_RRSESSION_HANDLE_TYPE Rundown

This event occurs when a **PCTX_RRSESSION_HANDLE_TYPE** context handle has been established between a client and server through a call to **RemoteQMOpenQueue** and the connection between the client and server is severed before the context handle is closed via a call to **RemoteQMCloseQueue**.

The server MUST use the context handle supplied as an event argument to **RemoteQMCloseQueue** to look up the context handle in the OpenSessionHandle table and close the OpenSessionHandle, as specified in **RemoteQMCloseQueue**.

3.1.6.2 PCTX_REMOTEREAD_HANDLE_TYPE Rundown

This event occurs when **PCTX_REMOTEREAD_HANDLE_TYPE** context handle has been established between a client and server through a call to **RemoteQMStartReceive**, and the connection between the client and server is severed before the context handle is closed via a call to **RemoteQMEndReceive**.

The server MUST use the context handle supplied as an event argument to look up the context handle in the ReadSessionHandle table and close the ReadSessionHandle, as specified in **RemoteQMEndReceive**. The server MUST set the *dwAck* parameter to RR_NACK in this case.

3.1.6.3 Message Added to the Queue

This event is received when a new message is added to the end of the message list of a queue.

While processing this event, the server MUST:

- Set the arrived time of the new message to the current time.
- Find the first entry in the pending request list which has the type ReceiveWaitForMessage and which has a QueueContextHandle associated with the queue to which the message was added.

If found:

- Complete the **RemoteQMStartReceive2** or **RemoteQMStartReceive**, or the **RemoteQMStartReceiveByLookupId**
- Change the entry type to ReceiveWaitForEndOrCancel.

Else, for each entry in the pending request list with the type PeekWaitForMessage:

- Complete the **RemoteQMStartReceive2** or **RemoteQMStartReceive**, or the **RemoteQMStartReceiveByLookupId** request by returning the message to the caller.
- Remove the entry from the pending request list.

For each entry in the cursor table that has the cursor state End Queue:

- Transition the state to Mid Queue.

3.1.6.4 Message Deleted from the Queue

The event is received when a message is deleted from the message list of a queue.

While processing this event, the server MUST:

- For each entry in the cursor table that has a cursor location that points to the deleted message, change the state to Message Deleted.

3.2 qm2qm Client Details

3.2.1 Abstract Data Model

Clients MUST maintain the following data elements:

- A QueueContextHandle associated with a queue.
- An OpenSessionHandle returned when opening the queue and disposed of when closing the queue.
- A ReadSessionHandle returned by **RemoteQMStartReceive2**, **RemoteQMStartReceive**, or **RemoteQMStartReceiveByLookupId**.
- A table of cursor handles associated with a QueueContextHandle.

3.2.2 Timers

No protocol timers are required beyond those used internally by RPC to implement resiliency to network outages. For more information, see [MS-RPCE].

3.2.3 Initialization

The client MUST create an RPC connection to the remote computer, using the details specified in Transport (section 2.1).

3.2.4 Message Processing Events and Sequencing Rules

The operation of the protocol is initiated and subsequently driven by the following higher-layer triggered events.

A message queuing application:

- Opens a queue.
- Peeks or Receives a message.
- Cancels a pending Peek or Receive.
- Purges a queue.
- Uses a cursor to Peek or Receive messages.
- Closes a cursor.
- Closes a queue.

3.2.4.1 Opening a Queue

Opening a queue consists of the following sequence of operations.

The client MUST call **qmcomm:R_QMOpenRemoteQueue**, as specified in [MS-MQMP], to obtain a queue handle.

The client MUST construct an RPC binding handle to the server, as specified in [C706] section 2.3.

The client calls the **RemoteQMGetQMQMServerPort** method using the RPC handle from the previous step. This method returns the RPC endpoint port on which subsequent method calls to this interface are to be invoked.

The client constructs a new RPC binding handle to the server using the RPC endpoint port determined in the previous step and replaces it with the initial RPC binding handle to the server.

The client MUST call the **RemoteQMOpenQueue** method and MUST specify the following parameter values:

- The RPC binding handle constructed in previous steps.
- *pLicGuid* set to a GUID which uniquely identifies the client.
- *dwMQS* set to 0x00000000.
- *hQueue* set to 0x00000000.
- *pQueue* set to the *pphContext* parameter of the **qmcomm:R_QMOpenRemoteQueue** method as specified in [MS-MQMP] section **3.1.4.2**.
- *dwContext* set to the pQueue parameter.

The client MUST record the returned **PCTX_RRSESSION_HANDLE_TYPE**.

3.2.4.2 Peeking a Message

The client MUST specify the handle of the queue to be read from, the timeout parameter for the operation, and an action from the table in the description of the *ulAction* parameter in **RemoteQMStartReceive** with an action type of Peek.

The client MUST call the **RemoteQMStartReceive** method and MUST specify the following parameter values for the **REMOTEREADDESC** structure (lpRemoteReadDesc):

- *hRemoteQueue* set to the queue handle returned by **qmcomm:R_QMOpenRemoteQueue**, as specified in [MS-MQMP] section **3.1.4.2**.
- *hCursor* set to NULL.
- *ulAction* set to the action specified by the message queuing application.
- *ulTimeout* set to the timeout value.
- A *dwRequestID* value that uniquely identifies this call from all other pending calls to this protocol.

The client MUST reconstruct the message from the returned *lpBuffer* element of the **REMOTEREADDESC** structure (lpRemoteReadDesc).

The client MUST record the returned **PCTX_REMOTEREAD_HANDLE_TYPE**.

3.2.4.3 Receiving a Message

The client MUST specify the handle of the queue to be read from, the timeout parameter for the operation and an action from the table in the description of the *ulAction* parameter in the **RemoteQMStartReceive** method, as specified in section **3.1.4.1**, with an action type of Receive.

The client MUST call the **RemoteQMStartReceive** method and MUST specify the following parameter values for the **REMOTEREADDESC** structure (lpRemoteReadDesc):

- *hRemoteQueue* set to queue handle returned by **qmcomm:R_QMOpenRemoteQueue**, as specified in [MS-MQMP] section **3.1.4.2**.
- *hCursor* set to NULL.
- *ulAction* set to the action specified by the message queuing application.
- *ulTimeout* set to the timeout value.
- A *dwRequestID* value that uniquely identifies the call from all other pending calls to this protocol.

The client MUST:

- Reconstruct the message from the returned *lpBuffer* element of the **REMOTEREADDESC** structure (lpRemoteReadDesc).
- Record the returned **PCTX_REMOTEREAD_HANDLE_TYPE**.
- Advise the server that the message was received by calling the **RemoteQMEndReceive** method with the following parameter values:
  - *pphContext* set to the *pphContext* value returned from the call to **RemoteQMStartReceive**.
  - *dwAck* set to 0x00000002 (RR_ACK).

3.2.4.4 Purging a Queue

The client MUST call the **RemoteQMPurgeQueue** method with the *hQueue* parameter set to a queue handle returned from the **qmcomm:R_QMOpenRemoteQueue** method, as specified in [MS-MQMP] section **3.1.4.2**.

3.2.4.5 Peeking a Message by Using a Cursor

The client MUST specify the handle of the queue to be read from, the timeout parameter for the operation, the cursor handle, and an action from the table in the description of the *ulAction* parameter in **RemoteQMStartReceive** with an action type of Peek.

The client MUST call the **RemoteQMStartReceive** method and MUST specify the following parameter values for the **REMOTEREADDESC** structure (lpRemoteReadDesc):

- *hRemoteQueue* set to a queue handle returned by **qmcomm:R_QMOpenRemoteQueue**, as specified in [MS-MQMP].
- *hCursor* set to the cursor handle obtained from **qmcomm:R_QMCreateRemoteCursor**, as specified in [MS-MQMP].

- *ulAction* set to the action specified by the message queuing application.
- *ulTimeout* set to the timeout value.
- A *dwRequestID* value that uniquely identifies this call from all other pending calls to this protocol.

The client MUST reconstruct the message from the returned *lpBuffer* element of the **REMOTEREADDESC** structure (lpRemoteReadDesc).

The client MUST record the returned **PCTX_REMOTEREAD_HANDLE_TYPE**.

3.2.4.6 Receiving a Message by Using a Cursor

The client MUST specify the handle of the queue to be read from, the timeout parameter for the operation, the cursor handle, and an action from the table in the description of the *ulAction* parameter in the **RemoteQMStartReceive** method with an action type of Receive.

The client MUST call the **RemoteQMStartReceive** method and MUST specify the following parameter values for the **REMOTEREADDESC** structure (lpRemoteReadDesc):

- *hRemoteQueue* set to queue handle returned by **qmcomm:R_QMOpenRemoteQueue**, as specified in [MS-MQMP].
- *hCursor* set to the cursor handle obtained from **qmcomm:R_QMCreateRemoteCursor**, as specified in [MS-MQMP].
- *ulAction* set to the action specified by the message queuing application.
- *ulTimeout* set to the timeout value.
- A *dwRequestID* value that uniquely identifies the call from all other pending calls to this protocol.

The client MUST:

- Reconstruct the message from the returned *lpBuffer* element of the **REMOTEREADDESC** structure (lpRemoteReadDesc).
- Record the returned **PCTX_REMOTEREAD_HANDLE_TYPE**.
- Advise the server that the message was received by calling the **RemoteQMEndReceive** method with the following parameter values:
  - *pphContext* set to the *pphContext* value returned from in the call to **RemoteQMStartReceive**.
  - *dwAck* set to 0x00000002 (RR_ACK).

3.2.4.7 Canceling a Pending Peek or Receive

The client MUST specify the handle of the queue and the *dwRequestID* of the operation to be canceled. The client MUST call the **RemoteQMCancelReceive** method with the following:

- *hQueue* set to the queue handle returned by **qmcomm:R_QMOpenRemoteQueue**, as specified in [MS-MQMP].
- *pQueue* set to *hQueue*.

- *dwRequestID* set to the *dwRequestID* specified in the **RemoteQMStartReceive** call instance that is to be canceled.

3.2.4.8 Closing a Cursor

The client MUST specify the queue handle and the cursor handle to be closed.

The client MUST call the **RemoteQMCloseCursor** method with the following:

- *hQueue* set to the queue handle returned by **qmcomm:R_QMOpenRemoteQueue**, as specified in [MS-MQMP].
- *hCursor* set to the cursor handle.

The client MUST remove the cursor handle from its state.

3.2.4.9 Closing a Queue

The client MUST specify the PCTX_RRSESSION_HANDLE_TYPE to be closed. If there are any pending requests associated with the PCTX_RRSESSION_HANDLE_TYPE, the client MUST cancel them. If any open cursor handles are associated with the queue, the client MUST close them.

The client MUST call the **RemoteQMCloseQueue** method with the following:

- *pphContext* set to the PCTX_RRSESSION_HANDLE_TYPE returned by **RemoteQMOpenQueue**.

The client MUST remove the PCTX_RRSESSION_HANDLE_TYPE from its state.

3.2.5 Timer Events

There are no timer events.

3.2.6 Other Local Events

There are no local events.

4 Protocol Examples

The following sections describe several operations as used in common scenarios to illustrate the function of the Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol.

4.1 Receive Example

The following sequence diagram illustrates a client receiving a message from a queue at the server. The call to **RemoteQMStartReceive2**, **RemoteQMStartReceive**, or **RemoteQMStartReceiveByLookupId** includes a *ulAction* value of MQ_ACTION_RECEIVE (0x00000000), and a unique *dwRequestID* value chosen by the client.

In response, the server associates a pending request with the passed *dwRequestID*, which is used to correlate a subsequent call to **RemoteQMEndReceive** or **RemoteQMCancelReceive** with the same value for *dwRequestID*. In addition, the server returns the message.

Next, the client indicates that the message was successfully received by calling **RemoteQMEndReceive**, specifying RR_ACK (0x00000002) for *dwAck*.

The server completes the corresponding pending request created by the call to **RemoteQMStartReceive2**, **RemoteQMStartReceive**, or **RemoteQMStartReceiveByLookupId**, and because RR_ACK is specified, removes the message from the queue.

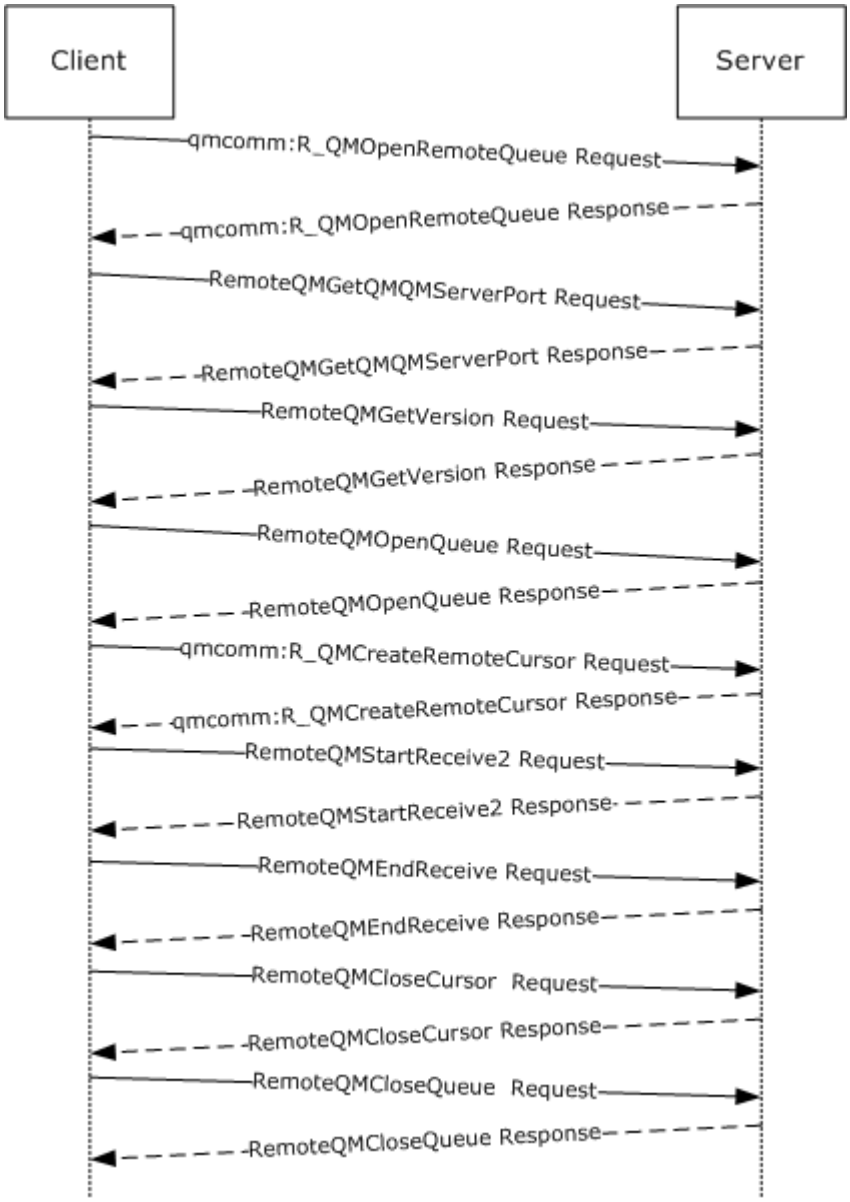

Figure 3: Client Receive

4.2 Purge Example

The following sequence diagram illustrates a scenario when the client purges a queue. In addition, the diagram shows how the static RPC endpoint port is acquired by the client to create an RPC binding handle.

The client begins the sequence by creating an RPC binding for the server. Next, the client calls the **RemoteQMGetQMQMServerPort** method, which returns an RPC endpoint port number with which the client creates a new binding. The client uses the new binding for all subsequent calls to the server.

Using the binding from the previous step, the client calls the **RemoteQMOpenQueue** method. On success, the server returns a new OpenSessionHandle.

Next, the client calls the **RemoteQMPurgeQueue** method. The server then removes all messages from the queue.

Finally, the client closes the OpenSessionHandle with a call to the **RemoteQMCloseQueue** method.

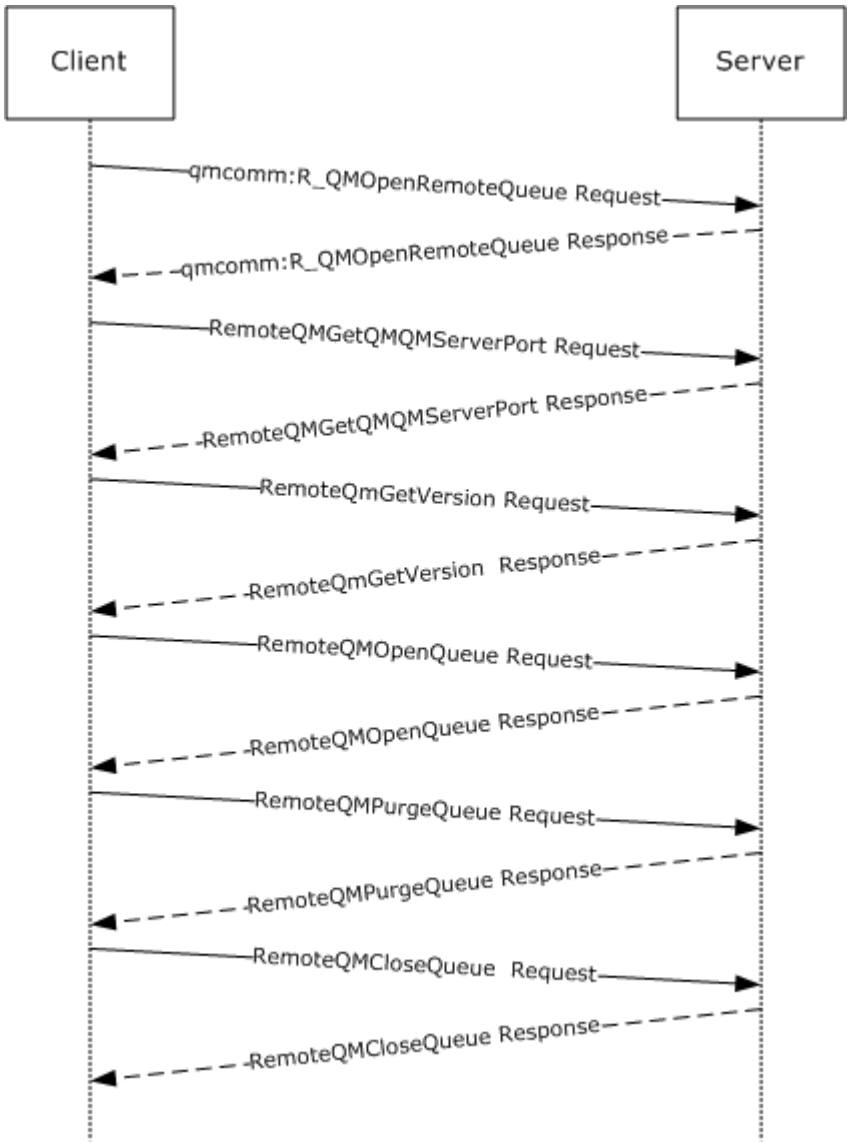

Figure 4: Purging a queue

5 Security

Clients MAY invoke methods of this interface at the "none" **authentication level**, depending on the network environment. Server implementations SHOULD be designed with careful consideration given to the security implications of accepting method calls from unauthenticated clients. Server implementations SHOULD reject methods invoked by unauthenticated clients by returning RPC_S_ACCESS_DENIED (0x00000005).

The Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol depends on security checks being done when the queue is opened by using the **qmcomm:R_QMOpenRemoteQueue** method, as specified in [MS-MQMP] section **3.1.4.2**.

The **RemoteQMGetQMQMServerPort** method is an exception to the above consideration, since clients MAY, depending on the network environment, invoke **RemoteQMGetQMQMServerPort** prior to configuring security for the RPC binding. For this reason, server implementations MUST NOT restrict access to the **RemoteQMGetQMQMServerPort** method.

5.1 Security Considerations for Implementers

None.

5.2 Index of Security Parameters

None.

6 Appendix A: Full IDL

For ease of implementation, the full **IDL** is provided below, where "ms-dtyp.idl" is the IDL found in [MS-DTYP] Appendix A and "ms-mqmq.idl" is the IDL found in [MS-MQMQ] Appendix A.

```
import "ms-dtyp.idl";
import "ms-mqmq.idl";

[
    uuid(1088a980-eae5-11d0-8d9b-00a02453c337),
    version(1.0),
    pointer_default(unique)
]
interface qm2qm
{
    typedef [context_handle] void *PCTX_RRSESSION_HANDLE_TYPE;

    typedef [context_handle] void *PCTX_REMOTEREAD_HANDLE_TYPE;

    typedef enum _REMOTEREADACK {
        RR_UNKNOWN,
        RR_NACK,
        RR_ACK
    } REMOTEREADACK ;

    typedef struct _REMOTEREADDESC {
        DWORD  hRemoteQueue ;
        DWORD  hCursor ;
        DWORD  ulAction ;
        DWORD  ulTimeout ;
        [range (0, 4325376)] DWORD  dwSize ;
        DWORD  dwpQueue ;
        DWORD  dwRequestID;
        DWORD  Reserved;
        DWORD  dwArriveTime ;
        REMOTEREADACK eAckNack ;
        [unique, size_is(dwSize), length_is(dwSize)] byte *lpBuffer ;
    } REMOTEREADDESC ;


    HRESULT
    RemoteQMStartReceive(
        [in] handle_t  hBind,
        [out] PCTX_REMOTEREAD_HANDLE_TYPE *pphContext,
        [in, out] REMOTEREADDESC* lpRemoteReadDesc
    );

    HRESULT
    RemoteQMEndReceive(
        [in] handle_t  hBind,
        [in, out] PCTX_REMOTEREAD_HANDLE_TYPE *pphContext,
        [in, range(1, 2)] DWORD dwAck
    );

    HRESULT
    RemoteQMOpenQueue  (
```

```
    [in] handle_t  hBind,
    [out] PCTX_RRSESSION_HANDLE_TYPE *phContext,
    [in]  GUID *pLicGuid,
    [in, range(0, 16)] DWORD dwMQS,
    [in] DWORD hQueue,
    [in] DWORD pQueue,
    [in] DWORD dwpContext
);

HRESULT
RemoteQMCloseQueue  (
    [in] handle_t  hBind,
    [in, out] PCTX_RRSESSION_HANDLE_TYPE *pphContext
);

HRESULT
RemoteQMCloseCursor  (
    [in] handle_t hBind,
    [in] DWORD    hQueue,
    [in] DWORD    hCursor
);

HRESULT
RemoteQMCancelReceive  (
    [in] handle_t  hBind,
    [in] DWORD     hQueue,
    [in] DWORD     pQueue,
    [in] DWORD     dwRequestID
);

HRESULT
RemoteQMPurgeQueue  (
    [in] handle_t hBind,
    [in] DWORD    hQueue
);

DWORD
RemoteQMGetQMQMServerPort  (
    [in]              handle_t hBind,
    [in, range(0, 1)] DWORD    dwPortType
);

typedef struct _REMOTEREADDESC2 {
    REMOTEREADDESC * pRemoteReadDesc;
    double           SequentialId;
} REMOTEREADDESC2;

void
RemoteQmGetVersion(
    [in] handle_t  hBind,
    [out] unsigned char *  pMajor,
    [out] unsigned char *  pMinor,
    [out] unsigned short * pBuildNumber
);

HRESULT
RemoteQMStartReceive2(
    [in] handle_t  hBind,
```

```
        [out] PCTX_REMOTEREAD_HANDLE_TYPE *pphContext,
        [in, out] REMOTEREADDESC2* lpRemoteReadDesc2
    );

    HRESULT
    RemoteQMStartReceiveByLookupId(
        [in] handle_t  hBind,
        [in] double LookupId,
        [out] PCTX_REMOTEREAD_HANDLE_TYPE *pphContext,
        [in, out] REMOTEREADDESC2* lpRemoteReadDesc2
    );
}
```

7 Appendix B: Windows Behavior

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

- Windows Vista
- Windows Server 2003
- Windows XP
- Windows 2000
- Windows NT

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 1.6: This protocol is used only when the client is on Windows NT, Windows 2000, or Windows XP and the server is Windows NT, Windows 2000, Windows XP or Windows Vista. When client and server are Windows Server 2003, or Windows Vista, the RemoteRead protocol is used (see [MS-MQRR]). Additionally, for architectural reasons, if the originating MSMQ application is a **dependent client**, the supporting Queue Manager uses this protocol rather than RemoteRead [MS-MQRR], regardless of Windows client or server versions.

<2> Section 1.7: These methods are not implemented by Windows NT or Windows 2000. All other versions of Windows implement these methods.

<3> Section 2.1: The ncacn_spx protocol sequence is only supported by Windows NT and Windows 2000, and MAY only be supported if TCP/IP is unavailable. Support for IPX and the ncacn_spx protocol sequence is deprecated on Windows XP, Windows Server 2003 and Windows Vista. The ncacn_ip_tcp protocol sequence SHOULD be supported when TCP/IP is available.

<4> Section 2.1: The ncacn_spx protocol sequence is only supported by Windows NT and Windows 2000, and MAY only be supported if TCP/IP is unavailable. Support for IPX and the ncacn_spx protocol sequence is deprecated on Windows XP, Windows Server 2003 and Windows Vista. The ncacn_ip_tcp protocol sequence SHOULD be supported when TCP/IP is available.

<5> Section 2.1: Windows NT, Windows 2000 and Windows XP clients use RPC dynamic endpoints to obtain the initial RPC binding handle. The client makes the RemoteQMGetQMQMServerPort call as specified in section 3.1.4.8 with the initial binding handle and uses the returned value to obtain a new RPC binding handle to be used for all subsequent RPC method calls on the protocol.

<6> Section 2.2.2.2: If the server is Windows NT, Windows 2000, Windows Server 2003, or Windows XP, the server MUST return a value other than 0x00000000. On Windows Vista the server MUST return 0x00000000.

Value	Meaning
0x00000000	Returned by Windows Vista.
0x00000001 — 0xFFFFFFFF	Returned by Windows NT, Windows 2000, Windows Server 2003, or Windows XP.

<7> Section 3.1.4.8: RPC over SPX is only supported by Windows NT and Windows 2000. This value is not supported by Windows XP, Windows Server 2003, and Windows Vista , and the server MUST return 0x00000000 to indicate failure.

<8> Section 3.1.4.8: RPC over SPX is only supported by Windows NT and Windows 2000. This value is not supported by Windows XP, Windows Server 2003, and Windows Vista and the server MUST return 0x00000000 to indicate failure.

<9> Section 3.1.4.9: This method is not implemented by Windows NT or Windows 2000. All other versions of Windows implement this method.

<10> Section 3.1.4.9: The client MAY pass in NULL, in which case the server MUST NOT set this value. If the *pMajor* parameter is not NULL, then this parameter MUST be set by the server to 0x05 if running on Windows XP or Windows Server 2003 and 0x06 if running Windows Vista.

<11> Section 3.1.4.9: The client MAY pass in NULL, in which case the server MUST NOT set this value. If the *pMinor* parameter is not NULL, the server MUST set this parameter to 0x01 if running on Windows XP, to 0x02 if running on Windows Server 2003 and to 0x0 if running on Windows Vista.

<12> Section 3.1.4.9: The client MAY pass in NULL, in which case the server MUST NOT set this value. If the *pBuildNumber* parameter is not NULL, then the server MUST set this parameter to 1108 if running on Windows XP, to 1738 if running on Windows Server 2003 and to 6000 if running on Windows Vista.

<13> Section 3.1.4.9: The client MUST use The Queue Manager to Queue Manager protocol only when the client is on Windows NT, Windows 2000, or Windows XP and the server is Windows NT, Windows 2000, Windows XP or Windows Vista. When client and server are Windows Server 2003, or Windows Vista, the client MUST use the Remote Read Protocol, as specified in [MS-MQRR].

<14> Section 3.1.4.10: This method is not implemented by Windows NT or Windows 2000. All other versions of Windows implement this method.

<15> Section 3.1.4.11: This method is not implemented by Windows NT or Windows 2000. All other versions of Windows implement this method.

8 Index

R

S

T

V

W