[MS-RPCE]:
Remote Procedure Call Protocol Extensions

Revision Summary

Date	Revision History	Revision Class	Comments
10/22/2006	0.01		MCPP Milestone 1 Initial Availability
01/19/2007	1.0		MCPP Milestone 1
03/02/2007	1.1		Monthly release
04/03/2007	1.2		Monthly release
05/11/2007	1.3		Monthly release

Date	Revision History	Revision Class	Comments
06/01/2007	1.3.1	Editorial	Revised and edited the technical content.
07/03/2007	1.3.2	Editorial	Revised and edited the technical content.
07/20/2007	1.3.3	Editorial	Revised and edited the technical content.
08/10/2007	2.0	Major	Added new content.
09/28/2007	2.0.1	Editorial	Revised and edited the technical content.
10/23/2007	2.1	Minor	Added new content.
11/30/2007	2.1.1	Editorial	Revised and edited the technical content.
01/25/2008	2.1.2	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

This document specifies a set of extensions to the DCE Remote Procedure Call (RPC) 1.1 Specification, as specified in [C706]. This specification assumes that the reader has familiarity with the concepts and requirements specified in [C706]. Concepts and requirements specified in [C706] are not repeated in this specification, except where required to specify how they are extended.

1.1 Glossary

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

64-Bit Network Data Representation (NDR64)
Application Configuration File (ACF)
Authentication Level
Authentication Service (AS)
Authentication Type
Connection-Oriented RPC
Connectionless RPC
Conversation Callback
Correlation
Dynamic Endpoint
Endpoint
Endpoint Mapper
Interface
Interface Definition Language (IDL)
Listening State
Marshal
Microsoft Interface Definition Language (MIDL)
Named Pipe
NetBIOS
Network Data Representation (NDR)
Object UUID
Opnum
Protocol Data Unit (PDU)
Protocol Identifier
Protocol Tower
Remote Procedure Call (RPC)
RPC Client
RPC Protocol Sequence
RPC Server
RPC Transfer Syntax
RPC Transport
Security Provider
Strict NDR/NDR64 Data Consistency Check
Universally Unique Identifier (UUID) or Globally Unique Identifier (GUID)
Unmarshal
Well-Known Endpoint

The following terms are specific to this document:

Activity: Used as specified in [C706] section 9.5, "Connectionless Protocol".

Client Address Space (CAS): Used as specified in [C706] section 9.5, "Connectionless Protocol".

De-Serialize or De-Serialization: See **Unmarshal**.

Opaque: Opaque means the client does not use the data. It is data (or more often, a handle) for use on the server on behalf of the client. Opaque data is sent to the client and returned to the server and used to assecc data or state information needed to process client calls/requests.

Protocol Variant: Protocol version that is distinct and non-interoperable from other protocol versions when all versions are from the same group of related protocols.

Serialize or Serialization: See **Marshal**.

Stub: Used as specified in [C706] section 2.1.2.2. A **stub** that is used on the client is called a "client **stub**", and a **stub** that is used on the server is called a "server **stub**".

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.

[AT] Sidhu, G., Andrews, R., and Oppenheimer, A., "Inside AppleTalk, Second Edition", New York: Addison Wesley, 1990, ISBN: 0201550210, http://developer.apple.com/MacOs/opentransport/docs/dev/Inside_AppleTalk.pdf

If you have any trouble finding [AT], please check here.

[C311] The Open Group, "DCE 1.1: Authentication and Security Services--Document Number C311", October 1997, http://www.opengroup.org/onlinepubs/9668899/

[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-EERR] Microsoft Corporation, "ExtendedError Remote Data Structure", January 2007.

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

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

[MS-KILE] Microsoft Corporation, "Kerberos Protocol Extensions", January 2007.

[MS-NLMP] Microsoft Corporation, "NT LAN Manager (NTLM) Authentication Protocol Specification", June 2007.

[MS-NRPC] Microsoft Corporation, "Netlogon Remote Protocol Specification", March 2007.

[MS-RPCH] Microsoft Corporation, "Remote Procedure Call Over HTTP Protocol Specification", January 2007.

[MS-SECO] Microsoft Corporation, "Windows Security Overview", January 2007.

[MS-SMB] Microsoft Corporation, "Server Message Block (SMB) Protocol Specification", July 2007.

[MS-SPNG] Microsoft Corporation, "Simple and Protected Generic Security Service Application Program Interface Negotiation Mechanism (SPNEGO) Protocol Extensions", January 2007.

[NETBEUI] IBM Corporation, "LAN Technical Reference: 802.2 and NetBIOS APIs", 1986, http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/BK8P7001/CCONTENTS

If you have any trouble finding [NETBEUI], please check here.

[NETWARE] Novell Corporation, "NDK: NetWare Server Protocol Libraries for C Documentation", June 2006, http://developer.novell.com/documentation/nwprotlb/index.html?prot_enu/data/hixj68nq

If you have any trouble finding [NETWARE], please check here.

[RFC81.3] French, C. and Salz, R., "DCE Assigned Values", RFC 81.3, December 1998, http://www.opengroup.org/tech/rfc/rfc81.3.html

[RFC1001] Network Working Group, "Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods", RFC 1001, March 1987, http://www.ietf.org/rfc/rfc1001.txt

[RFC1002] Network Working Group, "Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Detailed Specifications", RFC 1002, March 1987, http://www.ietf.org/rfc/rfc1002.txt

[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

[RFC2743] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000, http://www.ietf.org/rfc/rfc2743.txt

[RFC4122] Leach, P., Mealling, M., and Salz, R., "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005, http://www.ietf.org/rfc/rfc4122.txt

1.2.2 Informative References

[IPX] Microsoft Corporation, "Internetwork Packet Exchange (IPX)", http://msdn2.microsoft.com/en-us/library/ms817906.aspx

[MSDN-MIDL] Microsoft Corporation, "Microsoft Interface Definition Language (MIDL)", http://msdn2.microsoft.com/en-us/library/ms950375.aspx

[MSDN-PIPENAME] Microsoft Corporation, "Pipe Names", http://msdn2.microsoft.com/en-us/library/aa365783.aspx

[MSDN-RPC] Microsoft Corporation, "Remote Procedure Call", http://msdn2.microsoft.com/en-us/library/aa378651.aspx

[MSFT-RPCIFRESTRICTION] Microsoft Corporation, "RPC Interface Restriction", http://technet2.microsoft.com/windowsserver/en/library/8836be57-597b-4cda-bcf1-eb124ae5d49a1033.mspx?mfr=true.

[PIPE] Microsoft Corporation, "Named Pipes", http://msdn2.microsoft.com/en-us/library/aa365590.aspx

1.3 Protocol Overview (Synopsis)

This specification defines a set of extensions to the DCE Remote Procedure Call (RPC) 1.1 Specification, as specified in [C706]. These extensions add new capabilities to the DCE RPC 1.1 Specification, allow for more secure implementations to be built, and, in some cases, place additional restrictions on the DCE RPC Specification.

This specification builds on and relies heavily on the DCE RPC 1.1 Specification, as specified in [C706]. See [C706] for details on the context in which each of these extensions is specified.

The extensions are grouped into the following categories:

- Support for additional **RPC transports**, specified in section 2.1.
- Extensions to the **endpoint mapper interface** designed to improve security, specified in section 2.2.1.2.
- Extensions to the remote management interface designed to improve security, specified in section 2.2.1.3.
- Extensions to improve diagnosis of errors returned from a remote node, specified in section 2.2.2.9 and in [MS-EERR].
- An additional **RPC transfer syntax** (**NDR64**) to allow for better performance on 64-bit systems, specified in section 2.2.5.
- An additional set of **Network Data Representation (NDR)** data consistency checks and **Interface Definition Language (IDL)**/**application configuration file (ACF)** attributes to allow for more secure processing on both the **RPC client** and **RPC server**, specified in section 3.1.1.5.2.
- An additional set of message protection conventions to allow for better and more efficient protection of messages transmitted on the network, specified in sections 2.2.2.11, 2.2.2.12, and 2.2.2.13.
- Additional capability negotiation mechanisms between clients and servers for backward compatibility, specified in sections 2.2.2.14, 2.2.2.15, and 3.3.1.5.3.
- Extensions to facilitate building more efficient client and server implementations, specified in sections 2.2.2.10 and 3.3.1.5.4.
- Miscellaneous extensions and clarifications of the DCE Remote Procedure Call (RPC) 1.1 Specification.

1.4 Relationship to Other Protocols

This document specifies a set of extensions built on the DCE Remote Procedure Call (RPC) 1.1 Specification, as specified in [C706].

The extensions that require message authentication and security rely on the following protocols: Kerberos (as specified in [MS-KILE]), SPNEGO (as specified in [MS-SPNG]), NTLM (as specified in [MS-NLMP]), and Netlogon (as specified in [MS-NRPC]). These extensions use the security protocols, using the protocol primitives as specified in [RFC2743].

The ExtendedError Remote Data Structure is built on top of these extensions and provides extended error information to an RPC client.

The Remote Procedure Call Over HTTP Protocol is built below these extensions and enables the DCE Remote Procedure Call (RPC) 1.1 Specification, as specified in [C706], with these extensions to be routed over an HTTP transport in a way that is friendly to firewalls and provides additional security. Details on the Remote Procedure Call Over HTTP Protocol are as specified in [MS-RPCH] and are not part of this document.

These extensions define mapping of the DCE RPC 1.1 Specification over SMB (Server Message Block), TCP, UDP, SPX (Sequenced Packet Exchange), IPX (Internetwork Packet Exchange), NetBIOS over IPX, NetBIOS over TCP, NetBIOS over NetBEUI, and AppleTalk as RPC transports.

The following diagram illustrates the layering of these extensions over various RPC transports.

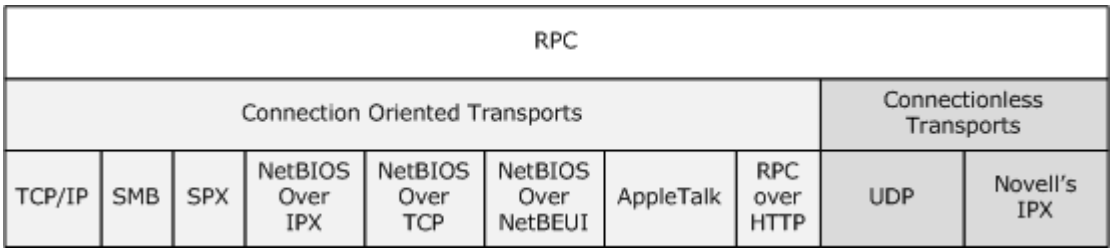


Figure 1: RPC Extensions Transports.

Protocols that require a secure request-reply message exchange can use an implementation of these extensions. Examples of protocols that use an implementation of these extensions include: Directory Services Setup Remote Protocol, Distributed Link Tracking: Central Manager Protocol, and Print System Asynchronous Notification Protocol.

1.5 Prerequisites/Preconditions

These extensions presume that the RPC transport on the server is in a **listening state**. The exact definition of a listening state for a given RPC transport is dependent on the RPC transport being used. For details, see the documentation for the respective RPC transport. These extensions also presume that the client and server **stubs** for each **remote procedure call** being executed are available to the implementation on the RPC client and RPC server, respectively.

The extensions do not impose other preconditions of their own, but they do inherit the preconditions required by the underlying RPC transport and **security provider** being used for a given RPC exchange.

1.6 Applicability Statement

The extensions specified herein do not change the basic applicability of the DCE RPC 1.1 Specification, as specified in [C706], but some extensions, as specified in section 1.3, improve security. The DCE RPC 1.1 Specification and the Remote Procedure Call Protocol are meta-protocols used to build application-level protocols. With its full set of extensions, the DCE RPC 1.1 Specification can be used in a wide range of scenarios.

1.7 Versioning and Capability Negotiation

- **Supported Transports:** These remote procedure call extensions can be implemented on top of various RPC transports, as specified in section 2.1. Higher-level protocols on the client should either discover the RPC transport supported by the server, or know it in advance. Higher-level protocols on the client may also determine if a server supports a given RPC transport by sending a message on the RPC transport. If the server supports the RPC transport, the communication

succeeds. If the server does not support the RPC transport, the RPC transport either returns a transport-dependent error or returns no reply, depending on the transport. For details on client behavior in the case of no reply, see sections 3.2.2 and 3.3.2. If the transport returns an error, an implementation-specific error is returned to the application or the higher-level protocols.

- **Protocol Versions:** These remote procedure call extensions do not introduce new **protocol variants**. The pre-existing protocol variants are specified throughout this document. Remote procedure call extensions constrain the DCE RPC 1.1 Specification, as specified in [C706], to only support protocol version 5.0 for **connection-oriented RPC**, protocol version 4.0 for **connectionless RPC**, and protocol version 2.0 for the NDR transfer syntax **universal unique identifier (UUID)**. The DCE RPC 1.1 Specification uses and extends the transfer syntax negotiation mechanism, as specified in section 3.3.1.5.7 and in specification [C706] Chapter 12. Version negotiation is performed separately for each RPC interface, as specified in [C706] Chapter 12.
- **Security and Authentication Methods:** Remote procedure call extensions use a model with a pluggable security provider module for the actual security and authentication work. Higher-level protocols on the client SHOULD discover the security provider supported by the server, or know them in advance. Higher-level protocols on the client can negotiate the use of RPC security providers by sending a message by using a given RPC security provider. If the server supports the RPC security provider, as specified in sections 3.2.3.1.1, 3.3.3.1, 3.2.3.5.4, and 3.3.3.5.3, the communication succeeds. If the server does not support the RPC security provider, the server returns an error, as specified in section 3.3.3.5.3 for connection-oriented RPC protocols, or as specified in section 3.2.3.5.4 for connectionless RPC protocols.
- **Capability Negotiation:** For the capability negotiation specified in sections 2.2.2.3 and 2.2.3.3, this protocol uses unused bits in the RPC **protocol data unit (PDU)** header, as specified in sections 2.2.2.3 and 2.2.3.3. This protocol also uses the bind time feature negotiation mechanism, as specified in section 3.3.1.5.3.

1.8 Vendor-Extensible Fields

In addition to the error codes specified in [C706], these extensions use Win32 error codes. These values are taken from the Windows error number space specified in [MS-ERREF]. Implementers MUST reuse those values with their indicated meanings. Choosing any other value runs the risk of collisions.

1.9 Standards Assignments

These extensions do not introduce any standards assignments other than what is specified in [C706] and [RFC81.3].

2 Messages

The following sections specify how connection-oriented and connectionless RPC messages are transported and the message syntax for each.

2.1 Transport

[C706] specifies two protocol variants within connection-oriented RPC and connectionless RPC. This specification maintains, as specified in [C706], categorization for the descriptions of the RPC protocol variants.

These extensions update the **protocol identifiers** that are specified in [C706] Appendix I, Protocol Identifiers. [C706] specifies that the protocol identifier can be one of three types:

1. An octet string derived from an interface UUID combined with a version number.
2. An octet string derived from OSI object identifiers (OIDs).
3. Single octet identifiers that are registered by the Open Software Foundation for commonly used protocols.

The extensions specified in this document mandate that the third type MUST be used for all communications.

Unless explicitly stated otherwise, the protocol identifier (used by each protocol sequence as specified in sections 2.1.1 and 2.1.2) is as specified in the table in [C706] Appendix I, Protocol Identifiers.

The **RPC protocol sequence** strings for the RPC transports defined by these extensions are specified in the following table.<1>

RPC Transport	RPC Protocol Sequence String
SMB	ncacn_np (see section 2.1.1.2)
TCP/IP (both IPv4 and IPv6)	ncacn_ip_tcp (see section 2.1.1.1)
UDP	ncadg_ip_udp (see section 2.1.2.1)
SPX	ncacn_spx (see section 2.1.1.3)
IPX	ncadg_ipx (see section 2.1.2.2)
NetBIOS over IPX	ncacn_nb_ipx (see section 2.1.1.4)
NetBIOS over TCP	ncacn_nb_tcp (see section 2.1.1.5)
NetBIOS over NetBEUI	ncacn_nb_nb (see section 2.1.1.6)
AppleTalk	ncacn_at_dsp (see section 2.1.1.7)
RPC over HTTP	ncacn_http (see section 2.1.1.8)

2.1.1 Connection-Oriented RPC Transports

All connection-oriented RPC protocols specified in this document are carried by transport protocols that SHOULD satisfy the following requirements:

- The transport protocol allows a client to establish a connection with a server given a network address, **endpoint**, and, optionally, one or more network options.
- Each transport protocol connection is a duplex communication session that supports reliable, in order, at most once delivery semantics.
- Each connection can be established and can be terminated. Once established, a connection allows sending and receiving of unlimited amounts of data. Optionally, a transport can detect if the other party to a connection has dropped off the connection and SHOULD indicate this to RPC runtime. The details of how and when this is handled are specified in sections 3.3.2.2.1 and 3.3.2.7.1.

In sections 2.1.1.1 through 2.1.1.8, for each transport protocol that supports these extensions, this document specifies how the transport protocol fulfills the requirements above, and any other relevant transport-specific details.

2.1.1.1 TCP/IP (NCACN_IP_TCP)

This protocol sequence specifies RPC directly over TCP/IP. There are no intermediate protocols between TCP/IP and RPC.

When extensions that are not specified in sections 2.1.1 through 2.1.2 are enabled over the TCP transport protocol, the network address MUST be an IPv4 or IPv6 address.<2>

The endpoint MUST be a 16-bit unsigned integer port number. The network address, of the server, and the endpoint are not transmitted over the network by these extensions, but are used by lower-layer protocols to setup the connection.

RPC over TCP/IP MUST use endpoint mapper **well-known endpoint** 135, as specified in [C706] Appendix H, Endpoint Mapper Well-Known Ports.

2.1.1.2 SMB (NCACN_NP)

This protocol sequence specifies RPC directly over SMB. There are no intermediate protocols between RPC and SMB.

When these extensions are enabled over the SMB transport protocol, the network address used by the client MUST be an IPv4 or IPv6 address or a server name <3>. The endpoint MUST be a **Named Pipe** name. The network address and endpoint are not transmitted on the network by these extensions but are used by lower-layer protocols to set up the connection.

RPC over SMB MUST use an endpoint mapper well-known endpoint of \pipe\epmapper.

RPC over SMB MUST use a protocol identifier of 0x0F instead of 0x10, as specified in [C706] Appendix I, Protocol Identifiers.<4>

The tower encoding for RPC over SMB MUST be the same as what is specified in [C706] Appendix L, Protocol Tower Encoding, for ncacn_ip_tcp. The port address MUST be the endpoint encoded into a NULL-terminated string in ASCII character format. The length of the string MUST be less than 0xFFFF bytes. For changes in how these extensions interpret the tower encoding (as specified in [C706] Appendix L, Protocol Tower Encoding), see section 3.1.3.5.3.

All PDUs sent over SMB MUST be sent as Named Pipe writes, and PDUs to be received MUST be received as named pipe reads, as specified in [MS-SMB]. However, in the case of synchronous RPCs, an implementation of these extensions MAY require the Server Message Block (SMB) Protocol implementation to execute a transaction encompassing the write of the last request PDU and the read of the first response PDU on the client. The last request PDU MUST be a bind, an alter_context,

or the last fragment of a request. The first response PDU MUST be a bind_ack or bind_nack, an alter_context_response, or the first fragment of a response. The transaction over a write and read is as specified in [MS-SMB] section 13.2.4.19.<5>

2.1.1.3 SPX (NCACN_SPX)

This protocol sequence specifies RPC directly over SPX. There are no intermediate protocols between RPC and SPX.<6>

When extensions that are not specified in sections 2.1.1 through 2.1.2 are enabled over the SPX transport protocol, the network address MUST be either a Netware machine name or a network and node number. For more information, see [IPX], IPX Addressing.

The endpoint MUST be a 16-bit unsigned integer port number. The network address, of the server, and the endpoint are not transmitted over the network by these extensions, but are used by lower-layer protocols to setup the connection.

RPC over SPX MUST use an endpoint mapper well-known endpoint of 34280.

2.1.1.4 NetBIOS over IPX (NCACN_NB_IPX)

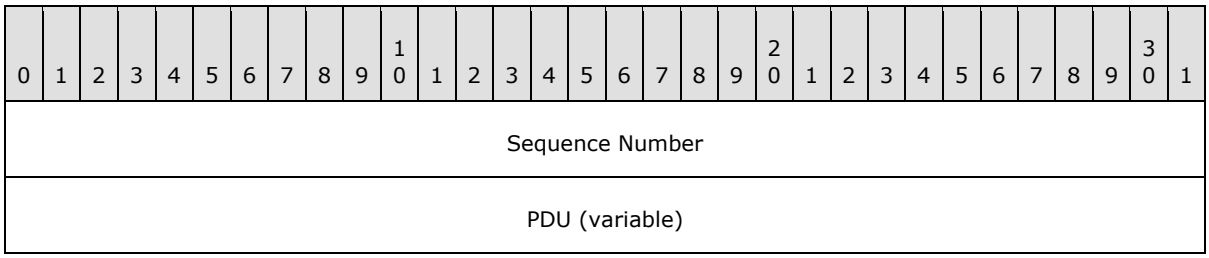
This protocol sequence specifies RPC directly over NetBIOS over IPX. There are no intermediate protocols between RPC and NetBIOS over IPX. These extensions define three **NetBIOS** mappings for RPC. The mappings are the same at the RPC level but use a different NetBIOS transport. Some implementations may offer higher layer protocols the opportunity to choose the NetBIOS transport to be used. This section covers the mapping of RPC to NetBIOS over IPX.<7><8>

When these extensions are enabled over the NetBIOS over IPX session service, as specified in [MS-SMB], the network address MUST be a NetBIOS machine name, as specified in [MS-SMB].

The endpoint MUST be an 8-bit unsigned integer socket number. The network address and endpoint are not transmitted on the network by these extensions but are used by lower-layer protocols to set up the connection.

RPC over NetBIOS over IPX MUST use an endpoint mapper well-known endpoint of 135. RPC over NetBIOS over IPX MUST use a protocol identifier of 0x12 instead of the value of 0x11, as specified in [C706] Appendix I, Protocol Identifiers.<9>

When communicating between a client and a server by using NetBIOS over IPX, each RPC PDU MUST be prefixed with a 4-octet sequence number encoded with little-endian byte ordering, as defined in this diagram.



The sequence numbers SHOULD start at 0 and increase monotonically, wrapping if it exceeds 2^{32}-1, for each sent PDU on a given NetBIOS connection. The server SHOULD ignore the sequence number values.

2.1.1.5 NetBIOS over TCP (NCACN_NB_TCP)

This protocol sequence specifies RPC directly over NetBIOS over TCP. There are no intermediate protocols between RPC and NetBIOS over TCP. These extensions define three NetBIOS mappings for RPC. The mappings are the same at the RPC level but use a different NetBIOS transport. Some implementations may offer higher layer protocols the opportunity to choose the NetBIOS transport to be used. This section covers the mapping of RPC to NetBIOS over TCP.<10><11><12>

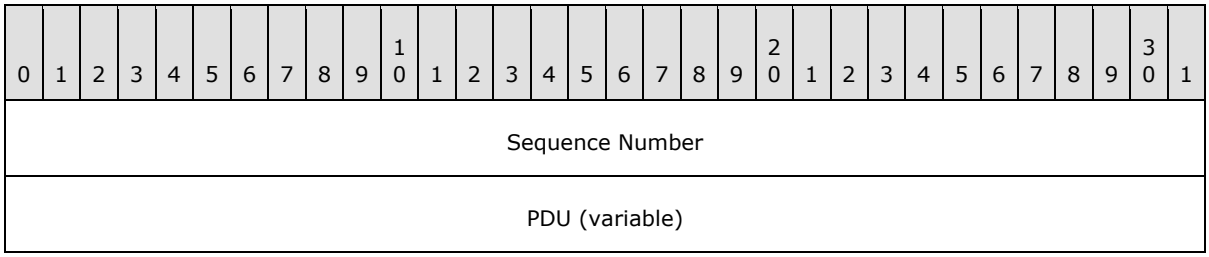
When these extensions are enabled over the NetBIOS over TCP session service, as specified in [RFC1001] and [RFC1002], the network address MUST be a NetBIOS machine name, as specified in [RFC1001] and [RFC1002].

The endpoint MUST be an 8-bit unsigned integer port number. The network address and endpoint are not transmitted on the network by these extensions, but are used by lower-layer protocols to set up the connection.

RPC over NetBIOS over TCP MUST use an endpoint mapper well-known endpoint of 135.

RPC over NetBIOS over TCP MUST use a protocol identifier of 0x12 instead of the value of 0x11, as specified in [C706] Appendix I, Protocol Identifiers.

When communicating between a client and a server by using NetBIOS over TCP, each RPC PDU MUST be prefixed with a 4-octet sequence number encoded with little-endian byte ordering, as defined in this diagram.



The sequence numbers SHOULD start at 0 and increase monotonically, wrapping if it exceeds 2^{32}-1, for each sent PDU on a given NetBIOS connection. The server SHOULD ignore the sequence number values.

2.1.1.6 NetBIOS over NetBEUI (NCACN_NB_NB)

This protocol sequence specifies RPC directly over NetBIOS over NetBEUI. There are no intermediate protocols between RPC and NetBIOS over NetBEUI. These extensions define 3 NetBIOS mappings for RPC. The mappings are the same at the RPC level but use a different NetBIOS transport. Some implementations may offer higher layer protocols the opportunity to choose the NetBIOS transport to be used. This section covers the mapping of RPC to NetBIOS over NetBEUI.<13><14>

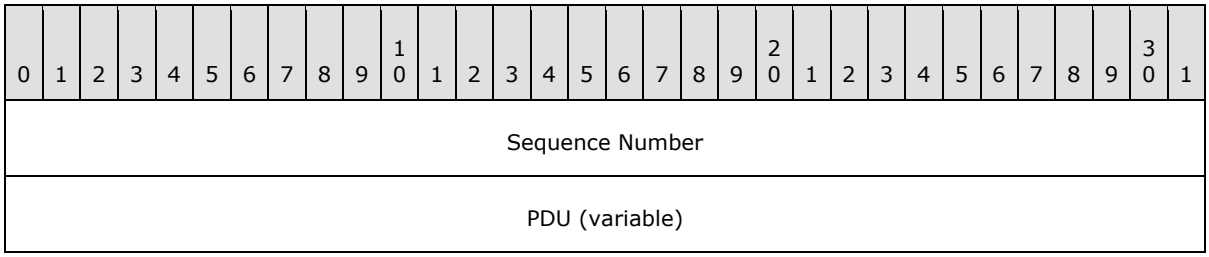
When these extensions are enabled over the NetBIOS over NetBEUI session service, as specified in [NETBEUI], the network address MUST be a NetBIOS machine name, as specified in [NETBEUI].

The endpoint MUST be an 8-bit unsigned integer port number. The network address and endpoint are not transmitted on the network by these extensions but are used by lower-layer protocols to set up the connection.

RPC over NetBIOS over NetBEUI MUST use an endpoint mapper well-known endpoint of 135.

RPC over NetBIOS over NetBEUI MUST use a protocol identifier of 0x12 instead of the value of 0x11, as specified in [C706] Appendix I, Protocol Identifiers.

When communicating between a client and a server by using NetBIOS over NetBEUI, each RPC PDU MUST be prefixed with a 4-octet sequence number encoded with little-endian byte ordering, as defined in the following diagram.



The sequence numbers SHOULD start at 0 and increase monotonically, wrapping if it exceeds 2^{32}-1, for each sent PDU on a given NetBIOS connection. The server SHOULD ignore the sequence number values.

2.1.1.7 AppleTalk (NCACN_AT_DSP)

This protocol sequence specifies RPC directly over AppleTalk. There are no intermediate protocols between RPC and AppleTalk.<15>

RPC over AppleTalk MUST use a well-known endpoint of endpoint mapper. When extensions that are not specified in sections 2.1.1 through 2.1.2 are enabled over the AppleTalk Data Stream Protocol (as specified in [AT]), the network address MUST be an AppleTalk name or in the format machinename@zonename. If no zone is provided, the protocol MUST default to the client's zone. The endpoint MUST be an ADSP socket number, as specified in [AT] section 12. The network address and endpoint are not transmitted on the network by these extensions but are used by lower-layer protocols to set up the connection.

2.1.1.8 RPC over HTTP (ncacn_http)

This protocol sequence specifies RPC over HTTP. The Remote Procedure Call Over HTTP Protocol, which is specified in [MS-RPCH], is the intermediate protocol between RPC and HTTP. RPC over HTTP v1 deviates from the requirements specified in section 2.1.1 (as specified in [MS-RPCH] in section 1.6).

Transport details are as specified in [MS-RPCH] section 2.1.

2.1.2 Connectionless RPC Transports

Earlier versions of [C706] refer to the CL_CANCEL packet as a QUIT packet and to a CANCEL_ACK packet as a QUACK packet. The latter forms are used in this document.<16>

2.1.2.1 UDP (NCADG_IP_UDP)

This protocol sequence specifies RPC directly over UDP. There are no intermediate protocols between RPC and UDP.<17>

When these extensions are enabled over the UDP transport protocol, the network address MUST be a DNS name or an IP address. The endpoint MUST be a UDP port number. The network address and

endpoint are not transmitted on the network by these extensions but are used by UDP or any lower-layer protocols to communicate with the server.

RPC over UDP MUST use endpoint mapper well-known endpoint 135, as specified in [C706] Appendix H, Endpoint Mapper Well-Known Ports. It MUST use protocol identifier 0x08, as specified in [C706] Appendix I, Protocol Identifiers.

2.1.2.2 Internetwork Packet Exchange (IPX) (NCADG_IPX)

This protocol sequence specifies RPC directly over IPX. There are no intermediate protocols between RPC and IPX.<18><19>

When these extensions are enabled over the IPX datagram service, the network address MUST be either a Netware machine name or a network and node number. For more information, see [IPX].

The endpoint MUST be a 16-bit unsigned integer socket number. The network address and endpoint are not transmitted on the network by these extensions but are used by lower-layer protocols to communicate with the server.

RPC over IPX MUST use an endpoint mapper well-known endpoint of 34280. It MUST use protocol identifier 0x14, as specified in [C706] Appendix I, Protocol Identifiers.

2.2 Message Syntax

For all non-IDL packet definitions in this section, this specification uses both [C706] definition style and a packet diagram to facilitate understanding of how the [C706] specification is extended. In all non-IDL packet definitions in this section, bit ordering rules are the same as what is specified in [C706], unless explicitly stated otherwise.

2.2.1 Connection-Oriented and Connectionless RPC Messages

This section defines the messages that are common to connection-oriented RPC and connectionless RPC protocol variants. The messages that are specific to connection-oriented RPC and connectionless RPC are specified in their respective sections, 2.2.2 and 2.2.3.

2.2.1.1 Common Types and Constants

2.2.1.1.1 RPC_IF_ID Type

This extension introduces a new type defined as:

```
typedef struct {
  UUID Uuid;
  unsigned short VersMajor;
  unsigned short VersMinor;
} RPC_IF_ID;
```

Use, meaning, and the layout of these fields are the same as the **rpc_if_id_t** type, as specified in [C706] Appendix N, IDL Data Type Declarations.

2.2.1.1.2 Extended Error Information Signature Value

The value for the UUID signature field that specifies the presence of extended error information in a **bind_nak** PDU MUST be 90740320-fad0-11d3-82d7-009027b130ab. The **bind_nak** PDU is as specified in [C706] section 12.6.4, Connection-Oriented PDU Definitions, and is as extended as specified in section 2.2.2.9.

2.2.1.1.3 UUID Format

Implementations of these extensions MUST NOT enforce the restrictions on the UUID format, as specified in [C706] Appendix A, Universal Unique Identifier. A UUID MUST be treated as an **opaque** 128-bit number. Implementations MAY choose any algorithm to generate a UUID as long as the generated UUIDs are unique in space and time, as specified in [C706] Appendix A.<20>

2.2.1.1.4 Mapping of a Context Handle

For an active context handle, implementations of these extensions SHOULD treat all the fields of the **ndr_context_handle**, as specified in [C706] Appendix N, Basic Type Declarations, as a single opaque token. There MUST be a 1:1 relationship between this token and the context handle on the server.<21>

2.2.1.1.5 version_t

The **version_t** structure specifies the major and minor version numbers of the run-time protocols supported by the server, as specified in [C706].

```
typedef struct _version_t {
  unsigned char  major;
  unsigned char  minor;
} version_t,
 *Pversion_t;
```

2.2.1.1.6 p_rt_versions_supported_t

The **p_rt_versions_supported_t** structure contains a list of the run-time protocol versions supported by the server, as specified in [C706].

```
typedef struct _p_rt_versions_supported_t {
  unsigned char  n_protocols;
  version_t p_protocols[-1];
} p_rt_versions_supported_t,
 *Pp_rt_versions_supported_t;
```

n_protocols: The number of protocols.

p_protocols: An array of **version_t** structures specifying major and minor protocol versions.

2.2.1.1.7 Security Providers

These extensions do not require support for the dce_c_rpc_authn_protocol_krb5 security provider, as specified in [C706] section 13. All of the requirements specified in [C706] section 13 are removed by these extensions.<22>

These extensions specify the following values for the security provider.

Name	Value	Security Provider
RPC_C_AUTHN_GSS_NEGOTIATE	0x09	Security Provider Simple and Protected GSS-API Negotiation Mechanism (SPNEGO)
RPC_C_AUTHN_WINNT	0x0A	NT LAN Manager (NTLM)
RPC_C_AUTHN_GSS_KERBEROS	0x10	Kerberos
RPC_C_AUTHN_NETLOGON	0x44	Netlogon

2.2.1.1.8 Authentication Levels

These extensions specify the following values for the **authentication levels**.

Name	Value	Meaning
RPC_C_AUTHN_LEVEL_CONNECT	0x02	Authenticates the credentials of the client and server.
RPC_C_AUTHN_LEVEL_CALL	0x03	Same as RPC_C_AUTHN_LEVEL_CONNECT, but also prevents replay attacks.
RPC_C_AUTHN_LEVEL_PKT	0x04	Same as RPC_C_AUTHN_LEVEL_CALL.
RPC_C_AUTHN_LEVEL_INTEGRITY	0x05	Same as RPC_C_AUTHN_LEVEL_PKT, but also verifies that none of the data transferred between the client and server has been modified.
RPC_C_AUTHN_LEVEL_PRIVACY	0x06	Same as RPC_C_AUTHN_LEVEL_INTEGRITY, but also ensures that the data transferred can only be seen unencrypted by the client and the server.

If the higher level application or protocol requests an authentication level that the implementation does not support, it MUST upgrade the request to the next highest supported level. RPC_C_AUTHN_LEVEL_PRIVACY MUST be supported.

On the client side, if the higher level protocol requests RPC_C_AUTHN_LEVEL_CALL, the implementation MUST upgrade it to RPC_C_AUTHN_LEVEL_PKT. Similarly, on the server side, if the auth_level field of the sec_trailer structure as specified in sections 2.2.2.11 and 2.2.3.4 is RPC_C_AUTHN_LEVEL_CALL, the implementation MUST process it in the same manner as a packet with auth_level RPC_C_AUTHN_LEVEL_PKT.

2.2.1.1.9 Impersonation Level

For secure calls, the higher-level layer protocols often specify the impersonation level. Various impersonation levels, listed in the following table, allow the higher layer protocols to control the capabilities of the client's identity that are available to the server. While building the security context, the client implementation passes this to the security provider on the first call to the

implementation-specific equivalent of the abstract GSS_Init_sec_context call, as specified in [RFC2743].

Client implementations of this extension MUST support the following impersonation levels. Note that the impersonation level does not itself appear in any RPC message and, hence, the numeric values of the following constants are implementation-specific. However, the values affect the token returned by the implementation-specific equivalent of the abstract GSS_Init_sec_context_call, as specified in [RFC2743].

Value	Meaning
RPC_C_IMPL_LEVEL_IDENTITY	The server can obtain information about the security context of the client but cannot impersonate the client's security context. The client MUST pass the GSS_C_IDENTITY_FLAG (defined in [RFC4757] section 7.1, which extends the [RFC2743]) to the implementation-specific equivalent of the abstract GSS_Init_sec_context_call.
RPC_C_IMPL_LEVEL_IMPERSONATE	The server can impersonate the client's security context on the server system but cannot make requests to remote machines using the client security context. This is the default behavior, as specified in [RFC2743].
RPC_C_IMPL_LEVEL_DELEGATE	The server can impersonate the client's security context on the server system and can make requests to remote machines using the client's security context. The client MUST pass the implementation-specific equivalent of the deleg_req_flag, as specified in [RFC2743] section 2.2.1.

If the higher level protocol does not specify an impersonation level, RPC_C_IMPL_LEVEL_IMPERSONATE MUST be used as the default.

2.2.1.1.10 Transport-Layer Impersonation Level

Some RPC transports have the capability to send the identity of the client with the request to the server. For details, on how this information is used by the RPC transport, see the documentation for the RPC transport.

Client implementations of these extensions MUST support the following impersonation levels. These impersonation levels allow protocols above RPC to control which capabilities of the client's identity are made available to the server. If the higher level protocol does not provide any value for this impersonation level, implementation of these extensions MUST allow the underlying RPC transport to choose the default value.

Currently the only RPC transport listed in section 2.1 that has this capability, sending the impersonation level to the server, is SMB (ncacn_np). See the specification of Impersonation level in [MS-SMB] section 2.2.8 for how this information is used by SMB.

Value	Meaning
SECURITY_ANONYMOUS	The server cannot obtain identification information about the client, and it cannot impersonate the client.
SECURITY_IDENTIFICATION	The server can obtain information about the security context of the client, but cannot impersonate the client's security context.

Value	Meaning
SECURITY_IMPERSONATION	The server can impersonate the client's security context on the server system, but cannot make requests to remote machines using the client security context.
SECURITY_DELEGATION	The server can impersonate the client's security context on the server system and can make requests to remote machines using the client's security context.

Note These Transport-Layer impersonation levels are separate from the ones specified in section 2.2.1.1.9 in the sense that they are specified separately by an application. Although the security meanings are the same (except that an anonymous level is not supported in section 2.2.1.1.9), the security is applied at different layers; for example, by the transport provider vs. the security provider chosen by the application.

2.2.1.2 Endpoint Mapper Interface Extensions

An endpoint mapper interface is specified in [C706] Appendix O . These extensions update the definition in [C706], as specified in the following sections. All parts of the definition that are not mentioned in the following sections MUST be the same as what is specified in [C706].

2.2.1.2.1 EPT_S_CANT_PERFORM_OP

This extension defines this constant to be equivalent to 0x6D8. It signifies general failure to perform the requested operation (method call) on the endpoint mapper interface.

2.2.1.2.2 twr_t Type

This extension redefines the **twr_t type**, as specified in [C706] Appendix L, **Protocol Tower** Encoding, by adding a range attribute to the **tower_length** field. The redefined type is specified as follows:<23>

```
typedef struct {
  [range(0,2000)] unsigned long tower_length;
  [size_is(tower_length)] BYTE tower_octet_string[];
} twr_t,
 *twr_p_t;
```

The purpose and use of this structure remains unchanged with an exception related to processing, as specified in section 3.1.3.5.3.

2.2.1.2.3 error_status Type

This type is used to communicate method-specific error status to a caller. The **error_status** type is defined as:

This type is declared as follows:

```
typedef unsigned int error_status;
```

2.2.1.2.4 ept_lookup Method

These extensions redefine the **ept_lookup** method, as specified in [C706] Appendix O, Endpoint Mapper Interface Definition, by:

- Adding the *ptr* attribute to the *object* and *Ifid* parameters.
- Adding the **range** attribute to the *max_ents* parameter.
- Removing the **[idempotent]** method attribute.

The redefined method is specified as follows:

```
void ept_lookup(
  [in] handle_t hEpMapper,
  [in] unsigned long inquiry_type,
  [in, ptr] UUID* object,
  [in, ptr] RPC_IF_ID* Ifid,
  [in] unsigned long vers_option,
  [in, out] ept_lookup_handle_t* entry_handle,
  [in, range(0,500)] unsigned long max_ents,
  [out] unsigned long* num_ents,
  [out, length_is(*num_ents), size_is(max_ents)]
    ept_entry_t entries[],
  [out] error_status* status
);
```

hEpMapper: An RPC binding handle as specified in [C706] section 2.

inquiry_type: The type of inquiry to perform. It MUST be one of the following values specified below.

Value	Meaning
RPC_C_EP_ALL_ELTS 0x00000000	Return all elements from the endpoint map. The Ifid, vers_option and object parameters MUST be ignored.
RPC_C_EP_MATCH_BY_IF 0x00000001	Return endpoint map elements that contain the interface identifier specified by the Ifid and vers_option values.
RPC_C_EP_MATCH_BY_OBJ 0x00000002	Return endpoint map elements that contain the **object UUID** specified by object.
RPC_C_EP_MATCH_BY_BOTH 0x00000003	Return endpoint map elements that contain the interface identifier and object UUID specified by Ifid, vers_option and object.

object: Optionally specifies an object UUID. A value of NULL indicates that no object UUID is specified.

Ifid: Optionally specifies an interface UUID. A value of NULL indicates that no interface UUID is specified.

vers_option: The interface version constraint. MUST be one of the values specified below:

Value	Meaning
RPC_C_VERS_ALL 0x00000001	Return endpoint map elements that contain the specified interface UUID, regardless of the version numbers.
RPC_C_VERS_COMPATIBLE 0x00000002	Return the endpoint map elements that contain the same major versions of the specified interface UUID and a minor version greater than or equal to the minor version of the specified UUID.
RPC_C_VERS_EXACT 0x00000003	Return endpoint map elements that contain the specified version of the specified interface UUID.
RPC_C_VERS_MAJOR_ONLY 0x00000004	Return endpoint map elements that contain the same version of the specified interface UUID and ignores the minor version.
RPC_C_VERS_UPTO 0x00000005	Return endpoint map elements that contain a version of the specified interface UUID less than or equal to the specified major and minor version.

entry_handle: On the first call, the client MUST set this to NULL. On successful completion of this method, returns a context handle that the client MUST use on subsequent calls to this method. In between calls if the client wishes to terminate the search, it MUST close the context handle by calling ept_lookup_handle_free.

max_ents: The maximum number of elements to be returned.

num_ents: The actual number of elements being returned.

entries: The elements which satisfy the specified search criteria.

status: MUST be a Win32 error code as specified in [MS-ERREF], 0x16c9a0cd or 0x16c9a0d6. All values, other than the ones specified below MUST be treated as a failure.

Value	Meaning
0x00000000	The method call returned at least one element that matched the search criteria.
0x16c9a0d6	There are no elements which satisfy the specified search criteria.

This method has no return values.

Everything else about this method remains as specified in [C706] Appendix O Endpoint Mapper Interface Definition.<24>

2.2.1.2.5 ept_map Method

These extensions redefine the ept_map method, as specified in [C706] Appendix O, Endpoint Mapper Interface definition, by:

- Adding the **ptr** attribute to the *obj* and *map_tower* parameters.
- Adding the range attribute to the *max_towers* parameter.
- Removing the **[idempotent]** method attribute.

The resulting method definition is specified as follows:

```
void ept_map(
  [in] handle_t hEpMapper,
  [in, ptr] UUID* obj,
  [in, ptr] twr_p_t map_tower,
  [in, out] ept_lookup_handle_t* entry_handle,
  [in, range(0,500)] unsigned long max_towers,
  [out] unsigned long* num_towers,
  [out, ptr, size_is(max_towers), length_is(*num_towers)]
    twr_p_t* ITowers,
  [out] error_status* status
);
```

hEpMapper: An RPC binding handle as specified in [C706] section 2.

obj: Optionally specifies an object UUID. A value of NULL indicates that no object UUID is specified.

map_tower: The tower encoding to search for, as specified in [C706] Appendix L.

entry_handle: On the first call, the client MUST set this to NULL. On successful completion of this method, returns a context handle that the client MUST use on subsequent calls to this method. In-between calls, if the client wants to terminate the search, it MUST close the context handle by calling ept_lookup_handle_free.

max_towers: The maximum number of elements to be returned.

num_towers: The actual number of elements being returned.

ITowers: The tower encoding, as specified in [C706] Appendix L, of the elements found in the endpoint map.

status: MUST be a Win32 error code, as specified in [MS-ERREF], 0x16c9a0cd or 0x16c9a0d6. All values besides the following ones MUST be treated as failure.

Value	Meaning
0x00000000	The method call returned at least one element that matched the search criteria.
0x16c9a0d6	There are no elements that satisfy the specified search criteria.

This method has no return values.

Everything else about this method remains as specified in [C706] Appendix O, Endpoint Mapper Interface Definition. For more details see section 2.3.3.3 in [C706]. Note, this redefintion has no wire impact and therefore it is interoperable with the [C706] implementation, as long as the max_towers value is less than 501.<25>

2.2.1.2.6 ept_insert Method

These extensions do not require support for the ept_insert method. These extensions do not provide an alternative method.<26>

2.2.1.2.7 ept_delete Method

These extensions remove support for the ept_delete method. A client implementation SHOULD NOT invoke this method.<27>

2.2.1.2.8 ept_lookup_handle_free Method

The syntax of **ept_lookup_handle_free** method is as specified in [C706] Appendix O, but [C706] Appendix O does not describe the meaning of the arguements. As such, the meaning of the arguments is given below.

```
void ept_lookup_handle_free(
  [in] handle_t hEpMapper,
  [in, out] ept_lookup_handle_t* entry_handle,
  [out] error_status* status
);
```

hEpMapper: An RPC binding handle as specified in [C706] section 2.

entry_handle: The context handle to free, which was received from a previous call to either ept_lookup or ept_map.

status: On return, this MUST be set to 0x00000000.

This method has no return values.

2.2.1.2.9 ept_inq_object Method

These extensions remove support for the ept_inq_object method. A client implementation SHOULD NOT invoke this method. These extensions do not provide an alternative method.<28>

2.2.1.2.10 ept_mgmt_delete Method

These extensions remove support for the ept_mgmt_delete method. A client implementation SHOULD NOT invoke this method. These extensions do not provide an alternative method.<29>

2.2.1.2.11 ept_lookup_handle_t Type

The **ept_lookup_handle_t** type defines an opaque pointer that is used to represent a context handle, as specified in [C706]. It is returned from the server to the client.

This type is declared as follows:

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

2.2.1.3 Management Interface Extensions

As specified in [C706] Appendix Q, Remote Management Interface defines a management interface. These extensions update the definition specified in [C706], as specified in the following sections. All

parts of the definition that are not mentioned in the following sections MUST be the same as specified in [C706].

2.2.1.3.1 rpc_if_id_vector_p_t Type

These extensions redefine the **rpc_if_id_vector_p_t** type, as specified in [C706] Appendix N, IDL Type Data Declarations, by changing the type of the **IfId** field from **rpc_if_id_p_t** to **RPC_IF_ID**. This change does not affect the compatibility with the type defined in [C706]

The redefined structure is specified as follows:

```
typedef struct {
  unsigned long Count;
  [size_is(Count)] RPC_IF_ID* IfId[];
} rpc_if_id_vector_t,
 *rpc_if_id_vector_p_t;
```

2.2.1.3.2 StatisticsCount Type

These extensions introduce a new type defined as:<30>

This type is declared as follows:

```
typedef [range(0,50)] unsigned long StatisticsCount;
```

It is used as the count of statistics elements for various methods.

2.2.1.3.3 rpc_mgmt_inq_stats Method

These extensions redefine the **rpc_mgmt_inq_stats** method, as specified in [C706] Appendix Q, Remote Management Interface, by changing the type of the *count* parameter from unsigned long to StatisticsCount. StatisticsCount, as defined in section 2.2.1.3.1, has range attribute which affects compatibility with the definition in [C706], as specified in section 3.3.1.3. The redefined method is specified as follows:<31>

```
void rpc_mgmt_inq_stats(
  [in] handle_t  binding_handle,
  [in, out] StatisticsCount* count,
  [out, size_is(*count)] unsigned long statistics[],
  [out] error_status_t* status
);
```

This method has no return values.

Everything else about this method remains as specified in [C706] Appendix Q Remote Management Interface.

2.2.1.3.4 rpc_mgmt_inq_princ_name Method

These extensions redefine the **rpc_mgmt_inq_princ_name** method, as specified in [C706] Appendix Q, Remote Management Interface, by adding a range attribute to the *princ_name_size* parameter. This change affects compatibility with the definition in [C706].

The redefined method is specified as follows:<32>

```
void rpc_mgmt_inq_princ_name(
  [in] handle_t binding_handle,
  [in] unsigned long authn_proto,
  [in, range(0, 4096)] unsigned long princ_name_size,
  [out, string, size_is(princ_name_size)]
    char princ_name[],
  [out] error_status_t* status
);
```

This method has no return values.

Everything else about this method remains as specified in [C706] Appendix Q Remote Management Interface.

2.2.2 Connection-Oriented RPC Messages

2.2.2.1 PDU Segments

A PDU can be viewed as having several different segments. These segments are:

- PDU **Header:** This is the header section of the PDU, as specified in [C706] section 12.6, Connection-Oriented RPC PDUs.
- PDU **Body:** This is the body section of the PDU, as specified in [C706] section 12.6, Connection-Oriented RPC PDUs. It also includes the padding octets specified in section 2.2.2.11.
- **sec_trailer Structure:** The structure specified in section 2.2.2.11.
- **Authentication Token:** The authentication token BLOB of the PDU, as specified in section 2.2.2.12.

Figure 2: PDU Structure

2.2.2.2 PFC_MAYBE Flag

This flag MAY appear in the **pfc_flags** field in the PDU header. Implementations of these extensions MAY<33> ignore this flag. This flag is specified in [C706] section 12.6, Connection-Oriented RPC PDUs.

2.2.2.3 PFC_SUPPORT_HEADER_SIGN Flag

These extensions define a new PDU flag for the **pfc_flags** in the common header fields that are specified in [C706] section 12.6 Connection-Oriented RPC PDUs, with the numeric value of 0x04. This has the same numeric value as the existing PFC_PENDING_CANCEL flag.

The PDU type MUST be examined to determine how to interpret this flag. (The PDU types are specified in [C706] section 12.6, Connection-Oriented RPC PDUs.) For PDU types bind, bind_ack, alter_context, and alter_context_resp, this flag MUST be interpreted as PFC_SUPPORT_HEADER_SIGN. For the remaining PDU types, this flag MUST be interpreted as PFC_PENDING_CANCEL.

2.2.2.4 negotiate_ack Member of p_cont_def_result_t Enumerator

These extensions specify a new member negotiate_ack to the p_cont_def_result_t enumeration specified in [C706] section 12.6, Connection-Oriented RPC PDUs, with the numeric value of 3. The enumeration SHOULD be:

```
typedef  short enum {
acceptance, user_rejection, provider_rejection, negotiate_ack
         } p_cont_def_result_t;
```

For details on how this is used, see section 3.3.1.1.

2.2.2.5 New Reasons for Bind Rejection

These extensions add two new reasons for rejection in the bind_nak packet that is specified in [C706], section 12.6, Connection-Oriented RPC PDUs. The reasons are defined as follows.

Reject reason	Value	Meaning
authentication_type_not_recognized	0x08	**Authentication type** requested by client is not recognized by server.
invalid_checksum	0x09	This rejection code is used for security-related errors not covered by other rejection codes.

2.2.2.6 alloc_hint Interpretation

These extensions impose additional restrictions on the **alloc_hint** field specified in [C706] section 12.6, Connection-Oriented RPC PDUs. Implementations MUST allow for 0 to be specified, as specified in [C706]; implementations SHOULD reject calls when the **alloc_hint** is non-zero but exceeds the combined stub data length of all fragments from a fragmented request or response.

If **alloc_hint** is set to a non-zero value and a request or a response is fragmented into multiple PDUs, implementations of these extensions SHOULD set the **alloc_hint** field in every PDU to be the combined stub data length of all remaining fragment PDUs.

An implementation that does not follow these rules might not be able to interoperate successfully with an implementation of these extensions.

2.2.2.7 RPC_SYNTAX_IDENTIFIER

This type is equivalent in syntax and semantics to the p_syntax_id_t type, as specified in [C706] section 12.6, Connection-Oriented RPC PDUs.

2.2.2.8 rpc_fault Packet

As specified in [C706] section 12.6, Connection-Oriented RPC PDUs allows for stub data to be present in rpc_fault PDUs. Clients implementing these extensions MUST ignore any stub data in an rpc_fault PDU, and servers MUST NOT generate stub data in an rpc_fault PDU. [C706] also prescribes that if the status in the rpc_fault PDU is 0, the actual error is in the stub data. These extensions always retrieve the actual error from the status field in the rpc_fault PDU. A server implementation MUST NOT send any of the error codes specified in section 3.3.3.5.

An implementation that does not follow these rules might not be able to interoperate successfully with an implementation of these extensions.

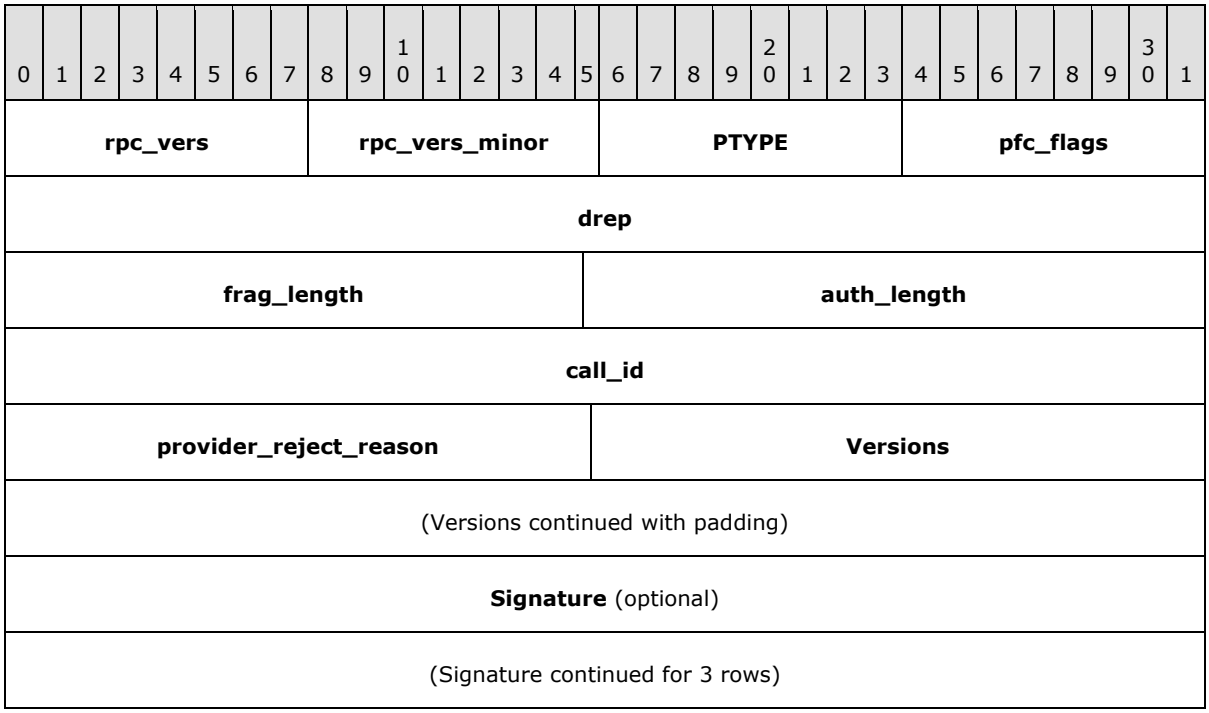
As specified in [C706] section 12.6 Connection-Oriented RPC PDUs sets aside a reserved field. These extensions specify the least significant bit of the reserved field to be a flag indicating the presence of RPC extended error information. Details on RPC extended error information are specified in [MS-EERR]. If RPC extended error information is present, it is specified as a variable length BLOB, and its length MUST be calculated as alloc_hint - 0x20.

2.2.2.9 bind_nak Packet

These extensions update the **bind_nak** packet, as specified in [C706] section 12.6.4.5, to have the following definition:

```
typedef struct {
  unsigned char rpc_vers;
  unsigned char rpc_vers_minor;
  unsigned char PTYPE;
  unsigned char pfc_flags;
```

```
    unsigned char drep[4];
    unsigned short frag_length;
    unsigned short auth_length;
    unsigned long call_id;
    unsigned short provider_reject_reason;
    p_rt_versions_supported_t  versions;
    UUID  Signature;
  } bind_nak;
```



These extensions add the Signature field at the end as an optional field. The presence or absence of the Signature field MUST be determined as follows:

Assume that the client calculates the length of the PDU until the Signature field as L.

- If the frag_length field is greater than or equal to L plus the size of the Signature field, the client SHOULD assume that the Signature field is present.
- Otherwise, the client SHOULD assume that the Signature field is not present.

The signature field MUST be interpreted as a UUID.

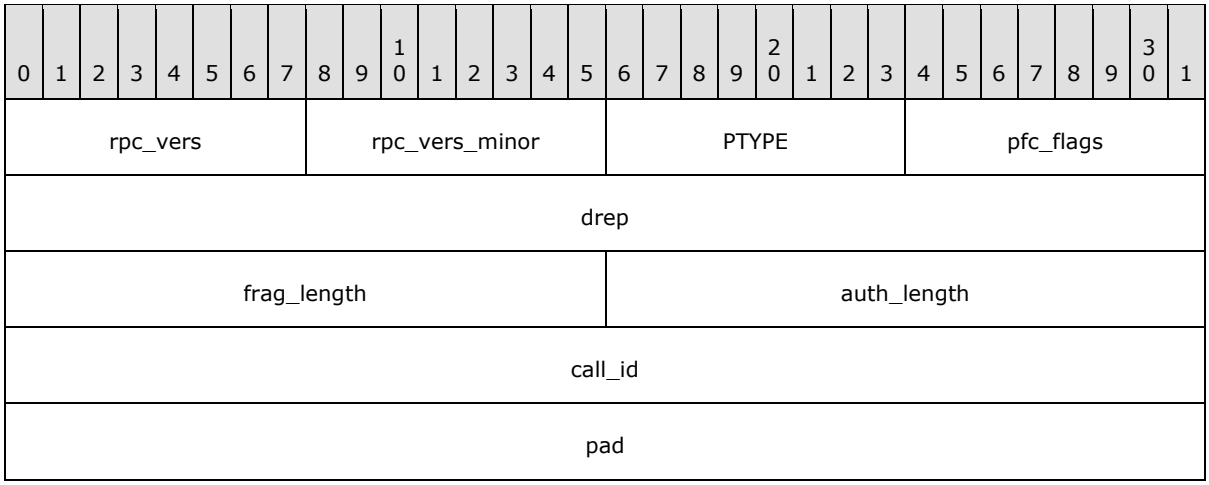
If the Signature field is equal to the extended error information signature value, as specified in section 2.2.1.1.2, the client MUST assume that the **bind_nak** PDU contains RPC extended error information appended as a BLOB, as specified in [MS-EERR], immediately following the Signature field that continues until the end of the PDU. If RPC extended error information is present, the length of the BLOB containing it MUST be calculated as frag_length – 0x28.

Clients MAY ignore the RPC extended error information BLOB. Clients that interpret the BLOB MUST do so as specified in [MS-EERR].<34>

If the Signature field is not equal to the extended error information Signature value, as specified in section 2.2.1.1.2, the client SHOULD ignore the Signature field and all information that follows it in this PDU.

2.2.2.10 rpc_auth_3 PDU

These extensions specify a new PDU type: **rpc_auth_3**. It is defined as:



rpc_vers (1 byte): As specified by **rpc_vers** field in rpc_bind PDU in [C706] section 12.6.

rpc_vers_minor (1 byte): As specified by **rpc_vers_minor** field in rpc_bind PDU in [C706] section 12.6.

PTYPE (1 byte): MUST be set to 0x10

pfc_flags (1 byte): As specified by **pfc_flags** field in rpc_bind PDU in [C706] section 12.6.

drep (4 bytes): As specified by **drep** field in rpc_bind PDU in [C706] section 12.6.

frag_length (2 bytes): As specified by **frag_length** field in rpc_bind PDU in [C706] section 12.6.

auth_length (2 bytes): As specified by **auth_length** field in rpc_bind PDU in [C706] section 12.6. It MUST be greater than 0 for this PDU type.

call_id (4 bytes): As specified by **call_id** field in rpc_bind PDU in [C706] section 12.6.

pad (4 bytes): "It can be set to any arbitrary value when set and it MUST be ignored on receipt. The pad field MUST be immediately followed by a sec_trailer structure whose layout, location and alignment are as specified in section 2.2.2.11.

All the rules for processing PDUs specified in [C706] section 12.6, Connection-Oriented RPC PDUs, including but not limited to data representation, **pfc_flags**, and protocol version numbers, MUST be applied to this PDU as well. For more information, see section 3.3.1.5.2.

2.2.2.11 sec_trailer Structure

These extensions define the sec_trailer structure to have syntax equivalent to the **auth_verifier_co_t** structure as specified in [C706] section 12.6, Connection-Oriented RPC PDUs.

The two structures have the same layout when sent on the network, but they name their fields differently and, in some cases, interpret their fields differently.

The sec_trailer structure and associated authentication information MUST be present in bind, bind_ack, alter_context, alter_context_resp, request, and response packets when a higher-level protocol instructs the RPC runtime to authenticate to the other side of an RPC message exchange.

If a higher-level protocol instructs the RPC runtime to authenticate to the other side of a message exchange, the **auth_length** field of all the previously mentioned PDUs MUST NOT be set to a zero value to indicate the presence of authentication information, including sec_trailer.

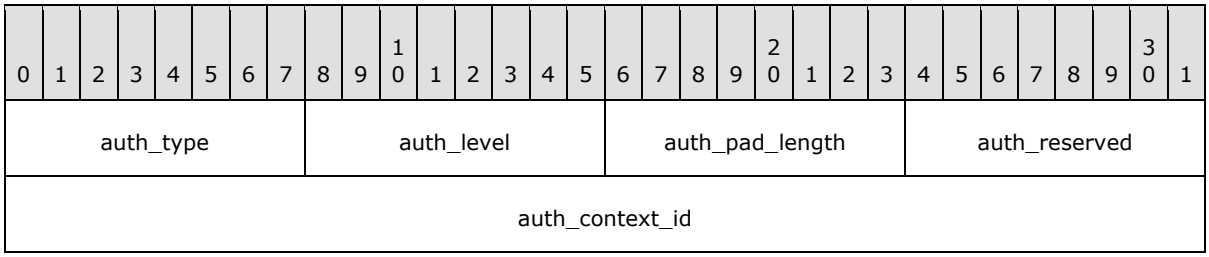
If the **auth_length** field is 0, an implementation MUST assume that the sec_trailer and associated authentication information is not present in the PDU.

For request and response PDUs, where the request and response PDUs are part of a fragmented request or response and authentication is requested, the sec_trailer structure MUST be present in every fragment of the request or response.

The sec_trailer structure MUST be placed at the end of the PDU, including past stub data, when present. The sec_trailer structure MUST be 4-byte aligned with respect to the beginning of the PDU. Padding octets MUST be used to align the sec_trailer structure if its natural beginning is not already 4-byte aligned.

All PDUs that carry sec_trailer information share certain common fields: **frag_length** and **auth_length**. The beginning of the sec_trailer structure for each PDU MUST be calculated to start from offset (**frag_length**–**auth_length**– 8) from the beginning of the PDU.

The structure is defined as:



auth_type (1 byte): MUST contain an authentication type. For information on how this is used, see sections 3.1.1.1.1, 3.3.1.5.2, and 3.3.3.1.1. If a request or response is fragmented, all PDUs from that request or response MUST have the same **auth_type**.

auth_level (1 byte): MUST contain one of the authentication levels as specified in section 2.2.1.1.8. The value serves a dual purpose. The first purpose is to specify what security protection is applied to what segment of the PDU, as specified in section 3.3.1.5.2. The second purpose is to serve as a parameter to the security provider that it SHOULD use to determine how to provide protection for the respective PDU segment. For information on how security providers use that, see the documentation for the respective security provider. If a request or response is fragmented, all PDUs from that request or response MUST have the same **auth_level**.

auth_pad_length (1 byte): The number of padding octets, as specified earlier in this section.

auth_reserved (1 byte): SHOULD be 0 on store, and SHOULD be ignored on read.

auth_context_id (4 bytes): Numeric identifier that uniquely identifies the security context that MUST be used for this PDU within the context of the current RPC connection. For information on security contexts, see section 3.3.1.5.4. An implementation MUST examine the **drep** field from the RPC PDU header to determine if this field is little-endian or big-endian, as specified in [C706] section 14.2.5 "Integers and Enumerated Types". If a request or response is fragmented, all PDUs from that request or response MUST have the same **auth_context_id**.

Immediately after the sec_trailer structure, there MUST be a BLOB carrying the authentication information produced by the security provider. This BLOB is called authentication token, and MUST be of size **auth_length**. The size MUST also be equal to the length from the first octet immediately after the sec_trailer structure all the way to the end of the fragment; the two values MUST be the same. For more information on what the authentication token contains, see section 2.2.2.12.

A client or a server that (during composing of a PDU) has allocated more space for the authentication token than the security provider fills in SHOULD fill in the rest of the allocated space with zero octets. These zero octets are still considered to belong to the authentication token part of the PDU.<35>

2.2.2.12 Authentication Tokens

These extensions require the conceptual model specified in [RFC2743] for all interactions with all security providers. An implementation instructs the [GSS]-compatible security providers to operate in a DCE-compatible manner by setting the DCE Style protocol variable. The following table details what PDU type MUST carry (in its auth_ token segment) the output of what [GSS] call during processing, as specified in section 3.3.1.5.2.2.

RPC PDU name	GSS call producing auth_value
Bind	First call to GSS_Init_sec_context, as specified in [RFC2743] section 2.2.1.
bind_ack	First call to GSS_Accept_sec_context, as specified in [RFC2743] section 2.2.2.
alter_context, rpc_auth_3	Second and subsequent calls to GSS_Init_sec_context, as specified in [RFC2743] section 2.2.1.
alter_context_resp	Second and subsequent calls to GSS_Accept_sec_context, as specified in [RFC2743] section 2.2.2.
Request	If the auth_level (as specified in section 2.2.2.11) is RPC_C_AUTHN_LEVEL_PRIVACY, call to GSS_WrapEx; else call to GSS_GetMICEx. See section 3.3.1.5.2.2 for details.
Response	If the auth_level (as specified in section 2.2.2.11) is RPC_C_AUTHN_LEVEL_PRIVACY, call to GSS_UnwrapEx; else call to GSS_VerifyMICEx. See section 3.3.1.5.2.2 for details.

2.2.2.13 Verification Trailer

Within exchanges in which the security provider in use does not provide integrity protection, as specified in [C706] section 13.2.5, these extensions specify an additional provision for providing integrity protection for certain portions of PDUs. The verification trailer encompasses several data structures. The data structures MUST only appear in a request PDU, and MUST be placed in the PDU immediately after the stub data, but before the authentication padding octets. Therefore, for security purposes, the verification trailer is considered part of the PDU body. For a fragmented request, only the last PDU of the request MUST have a verification trailer. As a general rule,

implementations SHOULD add the verification trailer on request PDUs that have portions of the PDU that cannot be protected by the security provider while in transit on the network.<36>

The following diagram shows a PDU body within a PDU structure, with stub data, verification trailer and authentication padding octets.

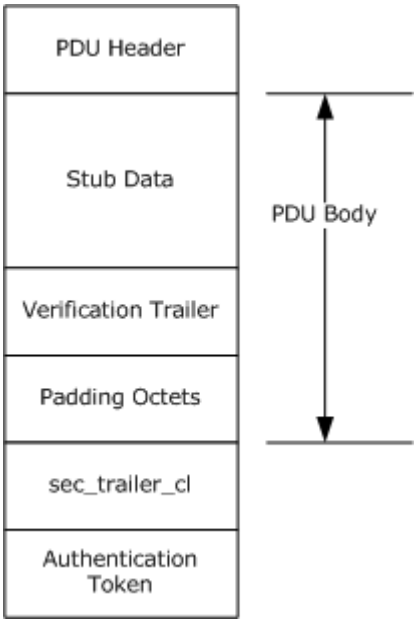


Figure 3: PDU structure with verification trailer

Client implementations MAY send a small number of extra octets of undefined content after the stub data. To maximize interoperability, server implementations SHOULD NOT assume that the verification trailer immediately follows the stub data, but instead SHOULD search for a sequence of octets that matches the value of the signature, as specified in section 2.2.2.13.1, starting immediately after the end of the stub data and continuing until the end of the PDU.<37>

The verification trailer consists of a header and a body. The header MUST always contain an instance of the **rpc_sec_verification_trailer** structure that is specified in section 2.2.2.13.1. The beginning of the header MUST be 4-byte aligned with respect to the beginning of the PDU. If the stub data does not end on a 4-byte aligned boundary, padding octets MUST be added after the stub data. The padding bytes SHOULD be set to 0.

The verification trailer header MUST be immediately followed by the verification trailer body. The verification trailer body MUST consist of, at most, one instance from each of several data structures called verification trailer commands that are specified in sections 2.2.2.13.2, 2.2.2.13.3, and 2.2.2.13.4.

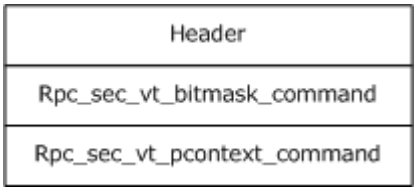
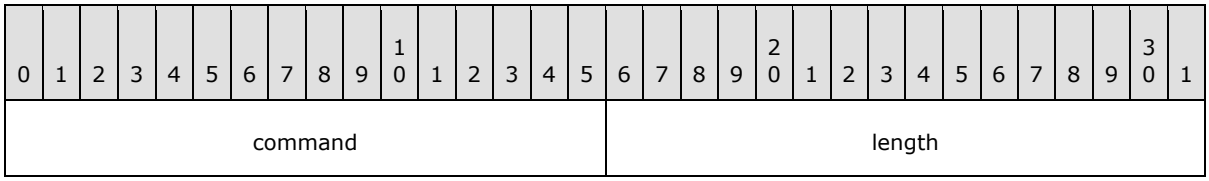


Figure 4: Verification trailer header and commands

The verification trailer commands may come in any order after the header. If more than one command is present, the next command MUST be placed immediately after the previous one. Each command MUST start with a common command header defined as the following.




```
typedef struct {
  USHORT command;
  USHORT length;
} SEC_VT;
```

command: The commands MUST be encoded by using little-endian encoding for all fields.

Valid combinations are defined immediately after the table:

Value	Meaning
SEC_VT_COMMAND_BITMASK_1 0x0001	This is an **rpc_sec_vt_bitmask** command, as specified in section 2.2.2.13.2.
SEC_VT_COMMAND_PCONTEXT 0x0002	This is an **rpc_sec_vt_pcontext** command, as specified in section 2.2.2.13.4.
SEC_VT_COMMAND_HEADER2 0x0003	This is an **rpc_sec_vt_header2** command, as specified in section 2.2.2.13.3.
SEC_VT_COMMAND_END 0x4000	This flag MUST be present in the last command in the verification trailer body.
SEC_VT_MUST_PROCESS_COMMAND 0x8000	Indicates that the server MUST process this command. If the server does not support the command, it MUST reject the request.

Least significant bits 0 through 13 (including 0 and 13) are used to hold the command type, and MUST be considered a single field. Bits 14 and 15 are used to indicate command processing rules. If a server does not understand a command, it MUST ignore it unless the SEC_VT_MUST_PROCESS_COMMAND bit is set. If the server does not understand the command and the SEC_VT_MUST_PROCESS_COMMAND bit is set, it MUST treat the request as invalid, as if **unmarshaling** failure occurred, as specified in section 3.1.3.5.2, except that a status code of 5 SHOULD be used instead of the status code specified in section 3.1.3.5.2. Any combination of a value for the command type (bits 0 through 13) and command processing rules (bits 14 and 15) is valid.

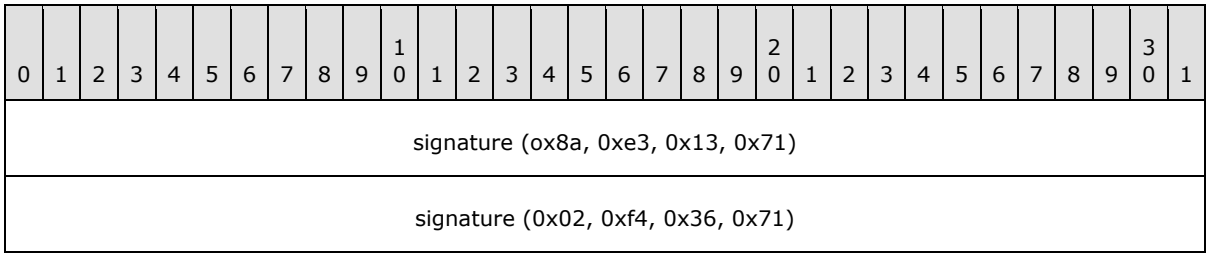
length: The length field is in octets, MUST be a multiple of 4, and MUST NOT include the length of the command header. For fixed-size commands, the length field MUST be equal to the length of the fixed size command.

2.2.2.13.1 rpc_sec_verification_trailer

The definition for this structure is:

```
typedef struct {
  unsigned char signature[8];
} rpc_sec_verification_trailer;
```

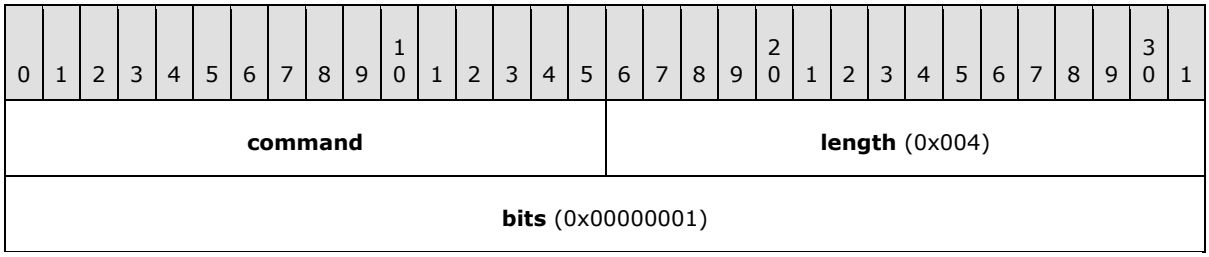
Whenever the verification trailer is present, the signature field MUST contain the following series of octets {0x8a, 0xe3, 0x13, 0x71, 0x02, 0xf4, 0x36, 0x71}. These values have no special protocol significance and only serve as a signature for this structure.



Windows sends the verification trailer header whenever it needs to send a verification trailer body. For details on when a verification trailer body is sent, see the verification trailer commands that follow.

2.2.2.13.2 rpc_sec_vt_bitmask

This command is defined as:




```
typedef struct {
  USHORT command;
  USHORT length;
  ULONG bits;
} rpc_sec_vt_bitmask;
```

command: Least significant bits 0 through 13 MUST be SEC_VT_COMMAND_BITMASK_1. Bits 14 and 15 are as specified in section 2.2.2.13.

Note SEC_VT_COMMAND_BITMASK_1 has a value of 0x0001.

length: MUST be 0x0004.

bits: The bits field is a bitmask. A server MUST ignore bits it does not understand. Currently, there is only one bit defined: CLIENT_SUPPORT_HEADER_SIGNING (bitmask of 0x00000001). If this bit is set, the PFC_SUPPORT_HEADER_SIGN bit, as specified in section 2.2.2.3, MUST

be present in the PDU header for the bind PDU on this connection. For information on how this is used, see section 3.3.1.5.2.2.<38>

2.2.2.13.3 rpc_sec_vt_header2

This command is defined as:<39>

```
typedef struct {
  USHORT command;
  USHORT length;
  unsigned char PTYPE;
  unsigned char Reserved1;
  unsigned short Reserved2;
  unsigned char drep[4];
  unsigned long call_id;
  USHORT p_context_id_t;
  unsigned SHORT opnum;
} rpc_sec_vt_header2;
```

command: Least-significant bits 0 through 13 MUST be SEC_VT_COMMAND_HEADER2 (0x0003). Bits 14 and 15 are as specified in section 2.2.2.13.

length: MUST be 0x0010.

PTYPE: MUST be the same as the **PTYPE** field in the request PDU header.

Reserved1: MUST be set to 0 on send, and MUST be ignored on receipt.

Reserved2: MUST be set to 0 on send, and MUST be ignored on receipt.

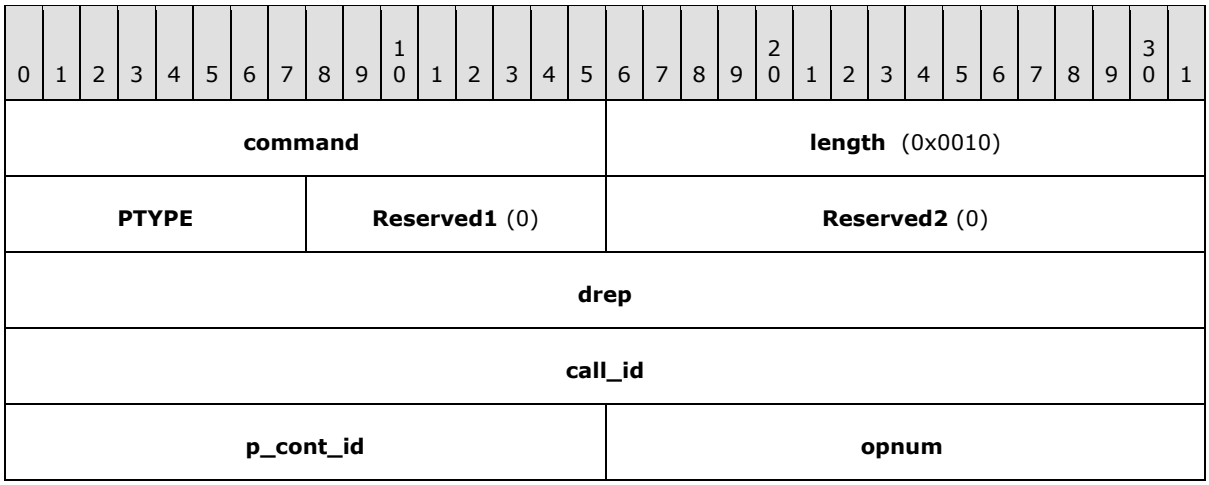
drep: MUST be the same as the **drep** field in the request PDU header.

call_id: MUST be the same as the **call_id** field in the request PDU header.

p_context_id_t: MUST be the same as the **p_cont_id** field in the request PDU header.

opnum: MUST be the same as the **opnum** field in the request PDU header.

The following table shows the format of the **rpc_sec_vt_header2**:



2.2.2.13.4 rpc_sec_vt_pcontext

This command is defined as:<40>

```
typedef struct {
  USHORT command;
  USHORT length;
  RPC_SYNTAX_IDENTIFIER InterfaceId;
  RPC_SYNTAX_IDENTIFIER TransferSyntax;
} rpc_sec_vt_pcontext;
```

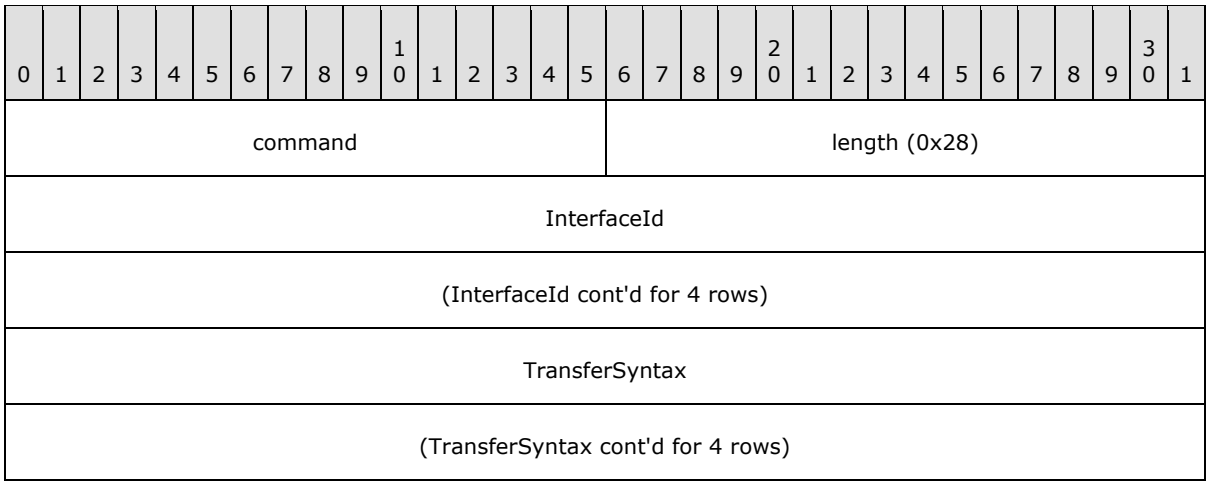
command: Least-significant bits 0 through 13 MUST be 0x0002. Bits 14 and 15 are as specified in section 2.2.2.13.

length: MUST be set to 0x28.

InterfaceId: The interface identifier for the presentation context of the request PDU in which this verification trailer appears. This MUST match the chosen abstract_syntax field from the bind or alter_context PDU where the presentation context was negotiated. For information on how a presentation context is negotiated, see section 3.3.1.5.7.

TransferSyntax: The transfer syntax identifier for the presentation context of the request PDU in which this verification trailer appears. This MUST match the chosen transfer_syntax from the bind or alter_context PDU where the presentation context was negotiated. For information on how a presentation context is negotiated, see section 3.3.1.5.7.

The following table shows the format of the **rpc_sec_vt_pcontext**:



2.2.2.14 BindTimeFeatureNegotiationBitmask

The bind time feature negotiation bitmask is an array of eight octets, each of which is interpreted as a bitmask. The format of the structure is:

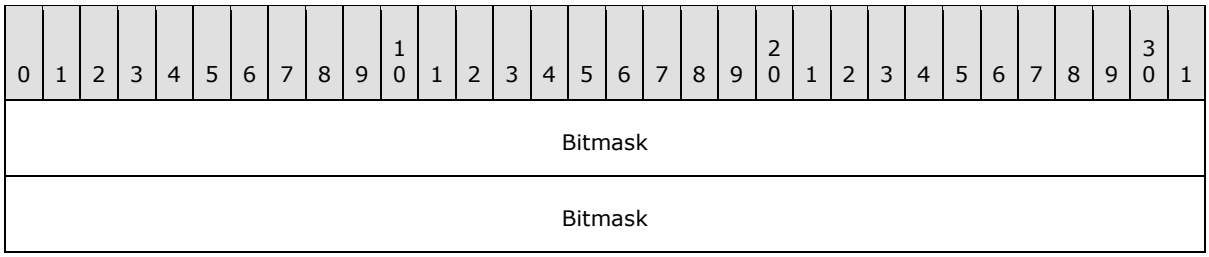
```
typedef struct {
  unsigned char Bitmask[8];
} BindTimeFeatureNegotiationBitmask;
```

Bitmask: Currently, only the two least-significant bits in the first element of the array are defined by the following table.

The rest SHOULD be reserved for future extensibility. For information on how this structure and the bits inside it are used, see section 3.3.1.5.3.

Value	Meaning
SecurityContextMultiplexingSupported 0x01	Client supports security context multiplexing, as specified in section 3.3.1.5.4.
KeepConnectionOnOrphanSupported 0x10	Client supports keeping the connection open after sending the orphaned PDU, as specified in section 3.3.1.5.11.

The following table shows the format of the **BindTimeFeatureNegotiationBitmask**:



2.2.2.15 BindTimeFeatureNegotiationResponseBitmask

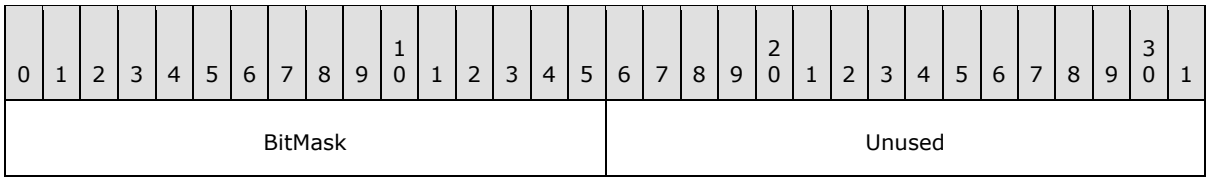
The bind time feature negotiation response bitmask is an array of two octets, each of which is interpreted as a bitmask. The format of the structure is:

```
typedef struct {
  unsigned char Bitmask[2];
} BindTimeFeatureNegotiationResponseBitmask;
```

Bitmask: Currently, only the two least-significant bits in the first element of the array are defined by the following table. The rest SHOULD be reserved for future extensibility. For information on how this structure and the bits inside it are used, see section 3.3.1.5.3.

Value	Meaning
SecurityContextMultiplexingSupported 0x01	Server supports security context multiplexing, as specified in section 3.3.1.5.4.
KeepConnectionOnOrphanSupported 0x10	Server supports keeping the connection open after sending the orphaned PDU, as specified in section 3.3.1.5.11.

The following table shows the format of the **BindTimeFeatureNegotiationResponseBitmask**:



2.2.3 Connectionless RPC Messages

The format of each PDU is as specified in [C706] section 12, RPC PDU Encodings.<41>

2.2.3.1 PDU Segments

A PDU can be viewed as having several different segments. These segments are:

- PDU Header: This is the header section of the PDU, as specified in [C706] section 12, RPC PDU Encodings.
- PDU Body: This is the body section of the PDU, as specified in [C706] section 12 RPC PDU Encodings.

- **sec_trailer_cl Structure**: The structure specified in section 2.2.2.11.
- Authentication Token: The authentication token BLOB of the PDU, as specified in section 2.2.3.5.

Figure 5: PDU Structure

2.2.3.2 Fault Packet

A fault PDU MUST NOT contain any of the error codes specified in section 3.2.3.5.

2.2.3.3 PF2_UNRELATED Flag

These extensions extend the PDU format by defining the **reserved_04** bit of the second set of PDU flags (flags2), as specified in [C706] section 12 RPC PDU Encodings, as PF2_UNRELATED. This flag has meaning only in a REQUEST packet.<42>

The server SHOULD set the PF2_UNRELATED flag in all conv_who_are_you2 and conv_who_are_you_auth requests to indicate to the client that the server can correctly interpret client requests with the flag set.

The client MUST set the PF2_UNRELATED flag in a REQUEST packet if the packet should not cancel the **activity's** previous call sequence numbers. For usage information, see section 3.

2.2.3.4 sec_trailer Structure

When a PDU header's auth_proto field is nonzero, [C706] section 12.3, Alignment and section 13.3.4, Authentication Verifier Encodings specify that the stub data of the packet is padded to the next 8-byte boundary, and MUST be followed by an auth_trailer_cl_t structure. These extensions divide the auth_trailer_cl_t type into a fixed-length security header and a variable length token following the security header. For information on the authentication token, including determination of its length, see section 2.2.3.5.

For request and response PDUs, where the request and response PDUs are part of a fragmented request or response and authentication is requested, the **sec_trailer_cl** structure is present in every fragment of the request or response.

```
typedef struct {
  unsigned char auth_level;
```

```
  unsigned char key_vers_num;
} sec_trailer_cl;
```

auth_level: This field MUST be one of the authentication levels specified in section 2.2.1.1.8. The values serve a dual purpose. The first purpose is to specify how security must be applied to the PDU, as specified in section 3.3.1.5.2. The second purpose is to serve as a parameter to the security provider that it SHOULD use to determine how to provide protection for the PDU. For details on how security providers use that, see the documentation for the respective security provider. If a request or response is fragmented, all PDUs from that request or response MUST have the same **auth_level**.

key_vers_num: This field is a numeric identifier that identifies the security context within the activity that MUST be used for this PDU.

Immediately after the **sec_trailer_cl** structure, there MUST be a sequence of padding bytes followed by a BLOB carrying the authentication information produced by the security provider. This BLOB is called the authentication token.

If the **auth_level** is **rpc_c_authn_level_pkt_privacy**, the number of padding bytes is calculated as follows:

Number of padding bytes = (EBLR4 - 2) - EBLR

Where

- EBLR4 : Encryption block length of the security context rounded to the next higher multiple of 4.
- EBL : Encryption block length of the security context.

For other **auth_level** values, the number of padding bytes is two.

2.2.3.5 Authentication Tokens

The token length is not transmitted explicitly. A recipient infers the length of the token by subtracting the combined length of the connectionless RPC header, stub data, **sec_trailer_cl**, and padding bytes from the length of the received packet, as reported by the underlying transport.

A client or a server (that, during processing, has allocated more space for the authentication token than the security provider fills in) SHOULD fill in the rest of the allocated space with zero octets. These zero octets are still considered to belong to the authentication token part of the PDU.<43> <44>

RPC PDU	**GSS call producing** auth_value
Conv_who_are_you_auth's in_data parameter	First call to GSS_Accept_sec_context, as specified in [RFC2743] section 2.2.2.
Conv_who_are_you_auth's out_data parameter	Second call to GSS_Init_sec_context, as specified in [RFC2743] section 2.2.1. If the data cannot be returned in a single PDU, the server queries the remainder with calls to conv_who_are_you_auth_more().
Request PDU	If the auth_level (as specified in section 2.2.3.4) is RPC_C_AUTHN_LEVEL_PRIVACY, call to GSS_Wrap (as specified in [RFC2743] section 2.3.3); else call to GSS_GetMIC (as specified in

RPC PDU	**GSS call producing** auth_value
	[RFC2743] section 2.3.1).
Response PDU	If the auth_level (as specified in section 2.2.3.4) is RPC_C_AUTHN_LEVEL_PRIVACY, call to GSS_Unwrap (as specified in [RFC2743] section 2.3.4); else call to GSS_VerifyMIC (as specified in [RFC2743] section 2.3.2).

2.2.4 IDL Syntax Extensions

Extensions specified in section 2.2.4.1 through 2.2.4.10 affect the syntax of the message, while extensions specified in sections 2.2.4.11 through 2.2.4.14 affect the processing of the message without directly changing the messages.

2.2.4.1 New Primitive Types

2.2.4.1.1 wchar_t

wchar_t designates a wide character type. It is treated as unsigned short by using the rules for unsigned short, as specified in [C706] section 14.2.5, Integer and Enumerated Types.

A **string** attribute can be applied to a pointer or array of type wchar_t. This indicates a string of wide characters, as specified in [C706] section 14.3.4, Strings. The terminator for wide character string is two octets of zero (0).

2.2.4.1.2 __int3264

In NDR transfer syntax,__int3264 MUST be represented as 4 octets in the octet stream by using the same format as long integer.

In 64-Bit Network Data Representation (NDR64) transfer syntax, __int3264 MUST be treated as hyper, as specified in [C706] section 14.2.5, Integer and Enumerated Types.

2.2.4.1.3 __int8,__int16, __int32,__int64

Sized integer types are supported in these extensions. Applications can declare 8-, 16-, 32-, or 64-bit integer variables by using the __intn type specifier, where n is 8, 16, 32, or 64. __int8, __int16, __int32, __int64 MUST be synonymous to small, short, long, and hyper, respectively, as specified in [C706] section 14.2.5, Integer and Enumerated Types.

2.2.4.2 Callback

These extensions allow static callback functions to be declared in the client side of a distributed application. This functionality provides a way for the server to make an RPC method call to the client. During a callback, the original client that initiates the call is defined as a callback server.

Callback routines are declared by using a **callback** keyword in an IDL file.

These extensions use operation numbers (**opnum**s) to inform a callback server of the operation it should invoke. Callback operations and non-callback operations use overlapping ranges of opnums starting at zero to identify the operation by using the following rules: Operation numbers for callback operations MUST be generated consecutively, counting callback operations only, beginning with 0 (zero), in the order in which callback operations appear in the IDL source. Callback operations MUST be excluded in calculating the operation numbers for non-callback operations. If an operation

is invoked in the context of a callback (for information on handling callbacks, see section 3.3.1.5.10), an implementation of this extension MUST use the callback opnum range for invoking the method. If an operation is not invoked in the context of a callback, an implementation of this extension MUST use the opnum range, as specified in [C706] section 5.2.1.

2.2.4.3 Array of Context Handles

These extensions extend the use of context handles (as specified in [C706] section 4.2.16.6, The **context_handle** Attribute) by allowing arrays of context handles.

Context handles MUST be parameters, as specified in [C706] section 4.2.16.6, The **context_handle** Attribute. They are valid as an array element, but MUST NOT be structure or union members and MUST NOT be the base type of a pipe.<45>

2.2.4.4 Array of Strings

As specified in [C706] section 14.3.5, array of strings are treated uniquely by requiring a common string length. These extensions override this base specification as follows: An array of strings MUST be represented as an ordered sequence of representations of the array elements.

2.2.4.5 ms_union

As an extension to the NDR definition of union alignment (as specified in [C706] section 14.3.8, Unions), these extensions dictate the alignment of a union (by itself or in an array) to be the largest alignment of all the arms when ms_union is enabled. ms_union MUST be ignored in NDR64 transfer syntax.

ms_union is enabled for a union by applying the **ms_union** type attribute to that union in its IDL file, or for all unions in the IDL file, by using an implementation-specific compiler option.<46>

2.2.4.6 v1_enum

Enumerated types MUST be treated as signed 32-bit integers when the **v1_enum** attribute is applied. v1_enum MUST be ignored in NDR64 transfer syntax.

v1_enum can be enabled by specifying v1_enum when defining Enumerated Types in **MIDL**.

2.2.4.7 Expression in Conformant, Varying, and Union Description

In these extensions, **first_is**, **last_is**, **length_is**, **size_is**, **max_is**, and union switch attributes SHOULD accept C-language expressions that evaluate to an integer that represents the runtime value of each specific attribute.<47>

For more information, see the example in section 4.7.

2.2.4.8 Unencapsulated Union

These extensions extend the specification for marshaling unions to allow *[in]* or *[in,out]* parameters to be used as the discriminant for [out] or [in,out] unencapsulated unions. As specified in [C706] section 14.3.8, the discriminant of an unencapsulated union MUST be **marshaled** both as the parameter specified in the switch_is construct and as the first part of the union representation. This custom-marshaling is extended as follows: The discriminant of the unencapsulated union MUST be marshaled as the parameter specified in the switch_is construct in the input or output octet stream(s) specified by the directional attribute(s) of the parameter. In addition, the discriminant MUST be marshaled as the first part of the union representation as specified in [C706] section

14.3.8, part 4, in the input or output octet stream(s) specified by the directional attribute(s) of the union.

2.2.4.9 pointer_default

With these extensions, the pointer_default attribute, as specified in [C706] section 4.2.4, Interface Header, is not required. Its default value MUST be pointer_default (unique) when the attribute is absent.

2.2.4.10 NDR Transfer Syntax Identifier

[C706] Appendix I, Protocol Identifiers, specifies the NDR transfer syntax identifier. These extensions augment the version number of the same NDR transfer syntax UUID to be 2.0, as specified below.

UUID	Version	Comments
8a885d04-1ceb-11c9-9fe8-08002b104860	02	Version 2.0 data representation protocol

2.2.4.11 byte_count

These extensions allow a higher-level protocol to specify the memory size in bytes of a given parameter as the value of another parameter. This MUST be specified by the byte_count parameter<48> attribute in an application configuration file (ACF), which the implementation MUST interpret as invoking this extension.

```
[function-attribute-list ] function-name(
    [byte_count(length-variable-name)] parameter-name,
    ...);
```

2.2.4.12 Range

The Range attribute is only applicable in **strict NDR/NDR64 data consistency checking**, as specified in section 3.1.1.5.3.

2.2.4.12.1 Range attribute to limit the Scope of Integral Values and the Number of Elements in Pipe Chunks

The range is specified by the [range] attribute accepted by [MIDL].

```
[range(low-val, high-val)] type-specifier declarator.
```

low-val and high-val are integer constant expressions as specified in [C706] "P 14.01" in section 4.4.1 "Grammar Synopsis".

2.2.4.12.2 range Attribute to Limit the Range of Maximum Count of Conformant Array and String Length

Microsoft Interface Definition Language (MIDL) extends the productions of Interface Definition Language (IDL) syntax with the range definition below:

```
[range(low-val, high-val), <conf_range_attr>] type-specifier
  declarator
conf_range_attr::=size_is<var_attr_list>|
max_is<var_attr_list>|
string
```

low-val and high-val are integer constant expressions as specified in [C706] "P 14.01" section 4.4.1 "Grammar Synopsis".

2.2.4.13 strict_context_handle

A strict context handle is activated by a strict_context_handle attribute in interface definition block in an application configuration file (ACF) file. This attribute is only applicable in strict NDR/NDR64 data consistency checking extension specified in section 3.1.1.5.3.

2.2.4.14 type_strict_context_handle

Type strict context handle is activated by specifying the type_strict_context_handle attribute in an interface definition block in an application configuration file (ACF). This attribute is only applicable in target level 6.0 of strict NDR/NDR64 data consistency checking, as specified in section 3.1.1.5.3.

2.2.4.15 disable_consistency_check

The Pointer attribute [disable_consistency_check] disables the check specified in section 3.1.1.5.3.3.1.2. This attribute is only applicable in the strict NDR/NDR64 data consistency checking extension specified in section 3.1.1.5.3.3.

2.2.5 64-Bit Network Data Representation

The 64-Bit Network Data Representation transfer syntax is a set of modifications to the NDR transfer syntax, as specified in [C706] Chapter 14, Transfer Syntax NDR.<49>

All PDUs encoded with the NDR64 transfer syntax MUST use a value of 0x10 for the data representation format label, as specified in [C706] section 14.1. This value indicates little-endian integer and floating-pointer byte order, IEEE floating-point format representation, and ASCII character format, as specified in [C706] section 14.1.

2.2.5.1 NDR64 Transfer Syntax Identifier

UUID	Version	Comments
71710533-BEBA-4937-8319-B5DBEF9CCC36	01	NDR64 data representation protocol

2.2.5.2 NDR64 Simple Data Types

NDR64 supports all simple types defined by NDR (as specified in [C706] section 14.2, NDR Primitive Types) with the same alignment requirements except for enumerated types, which MUST be represented as signed long integers (4 octets) in NDR64.

2.2.5.3 NDR64 Constructed Data Types

NDR64 supports constructed data types defined for NDR (as specified in [C706] section 14.3, NDR Constructed Types) with some exceptions. The following sections specify differences between the NDR64 data representation and the NDR data representation.

2.2.5.3.1 Representation Conventions

To be consistent with what is specified in [C706], diagrams describing data representation in NDR/NDR64 extensions follow representation conventions as specified in [C706] section 14.2.1.

2.2.5.3.2 Arrays

2.2.5.3.2.1 Conformant Arrays

NDR64 represents a conformant array as an ordered sequence of representations of the array elements preceded by an unsigned 64-bit integer. The 64-bit integer MUST specify the number of array elements transmitted, including empty elements.<50>

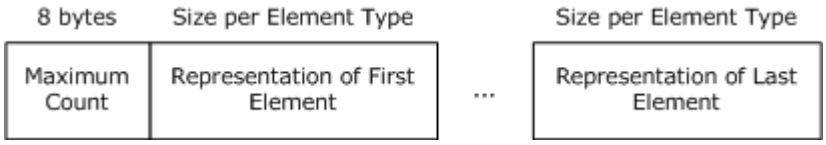


Figure 6: Conformant arrays

2.2.5.3.2.2 Varying Arrays

NDR64 represents a varying array as an ordered sequence of representations of the array elements preceded by two unsigned 64-bit integers. The first 64-bit integer MUST specify the offset from the first index of the array to the first index of the actual subset being passed. The second 64-bit integer MUST specify the actual number of elements being passed.<51>

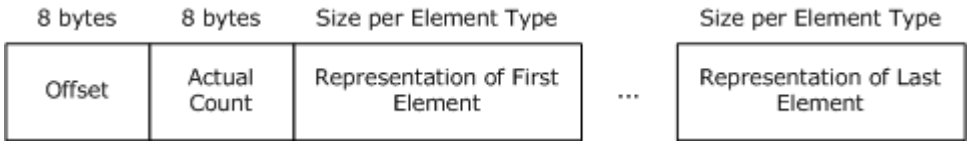


Figure 7: Varying arrays

2.2.5.3.2.3 Conformant Varying Arrays

NDR64 represents a conformant varying array as an ordered sequence of representations of the array elements preceded by three unsigned 64-bit integers. The first 64-bit integer MUST specify the maximum number of elements in the array. The second 64-bit integer MUST specify the offset from the first index of the array to the first index of the actual subset being passed. The third 64-bit integer MUST specify the actual number of elements being passed. The 64-bit integers that indicate the offset and the actual count MUST always be present, even if the maximum count is 0 (zero).<52>

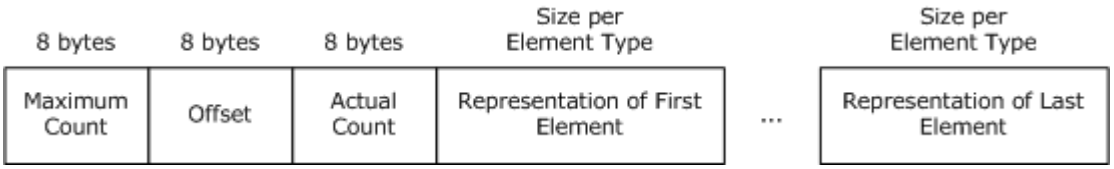


Figure 8: Conformant varying arrays

2.2.5.3.2.4 Multidimensional Arrays

NDR64 follows the same NDR representation for multidimensional arrays, as specified in [C706] sections 14.3.3.6 through 14.3.3.9, except for the maximum count, offset, and actual count. In NDR64, these MUST be specified as 64-bit unsigned integers rather than 32-bit long integers.

2.2.5.3.3 Strings

In NDR64, the elements in a string MUST be characters, wide characters (16-bit characters specified by wchar_t), octets, or structures, all of whose elements are octets.

2.2.5.3.3.1 Varying Strings

NDR64 represents a varying string as an ordered sequence of representations of the string elements preceded by two unsigned 64-bit integers. The first 64-bit integer MUST specify the offset from the first index of the string to the first index of the actual subset being passed. The second 64-bit integer MUST specify the actual number of elements being passed, including the terminator.

The first 64-bit integer (offset) MUST be 0 (zero).

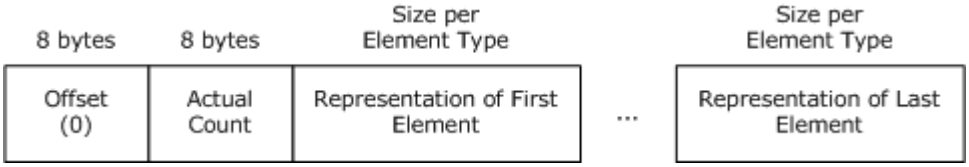


Figure 9: Varying strings

2.2.5.3.3.2 Conformant Varying Strings

NDR64 represents a conformant varying string as an ordered sequence of representations of the string elements preceded by three unsigned 64-bit integers. The first 64-bit integer MUST specify the maximum number of elements in the string, including the terminator. The second 64-bit integer MUST specify the offset from the first index of the string to the first index of the actual subset being passed. The third 64-bit integer MUST specify the actual number of elements being passed, including the terminator.

The second 64-bit integer (offset), MUST be 0 (zero).

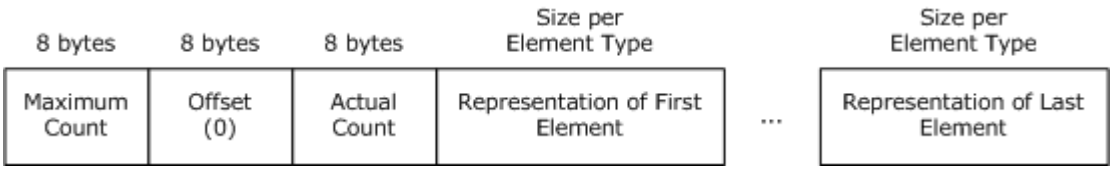


Figure 10: Conformant varying strings

2.2.5.3.4 Structures

2.2.5.3.4.1 Structure with Trailing Gap

NDR64 represents a structure as an ordered sequence of representations of the structure members. The trailing gap from the last non-conformant and non-varying field to the alignment of the structure MUST be represented as a trailing **pad**. The size of the structure MUST be a multiple of its alignment.

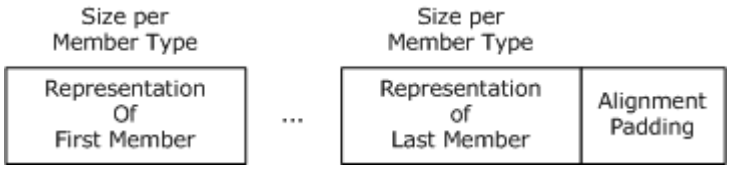


Figure 11: Structure with trailing gap

For more information, see the example in section 4.8.

2.2.5.3.4.2 Structure Containing a Conformant Array

In the NDR64 representation of a structure that contains a conformant array, the unsigned 64-bit long integers that specify maximum element counts for the dimensions of the array MUST appear at the beginning of the structure; and the array elements MUST appear in place at the end of the structure. The following diagram shows the representation of a structure containing a unidimensional conformant array.

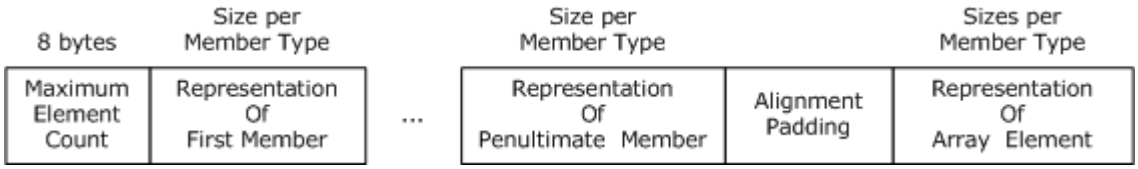


Figure 12: Structure containing a unidimensional conformant array

2.2.5.3.4.3 Structure Containing a Conformant Varying Array

In the NDR64 representation of a structure that contains a conformant varying array, the 64-bit maximum counts for dimensions of the array MUST appear at the beginning of the structure. The 64-bit offsets and the 64-bit actual counts MUST remain in place at the end of the structure immediately preceding the array elements. The following diagram shows the representation of a structure containing a unidimensional conformant varying array.

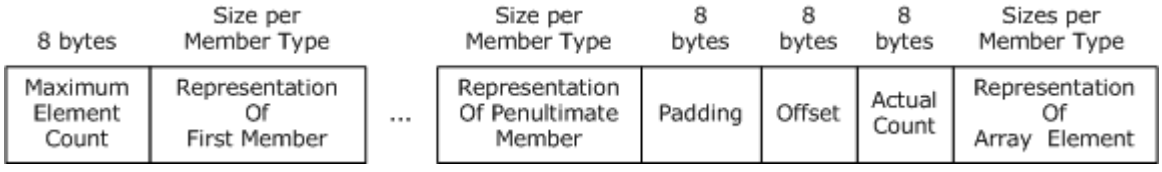


Figure 13: Structure containing a unidimensional conformant varying array

2.2.5.3.4.4 Unions

NDR64 represents a union as a representation of the tag followed by a representation of the selected member. Unions are aligned according to the largest of the union arms. The selected member is aligned to the largest alignment of all the arms.

Discriminant value of 0xffffffff is reserved, and MUST NOT be used in NDR64.

2.2.5.3.4.5 Pipes

In NDR64, a pipe element can be of any NDR primitive or constructed type except pipes, pointers, conformant or varying arrays, or both conformant and varying arrays, and structures that contain conformant or varying arrays, or both conformant and varying arrays.

NDR64 represents a pipe as a sequence of chunks. Each chunk is represented as an ordered sequence of representations of the elements in the chunk. The sequence MUST be preceded by a 64-bit unsigned integer that specifies the number of elements in the chunk, and MUST be followed by a 64-bit unsigned integer that specifies the arithmetic negate of the value of the number of elements in the chunk, treated as a signed 64-bit integer. The final chunk MUST contain no elements, and MUST consist only of two unsigned 64-bit integers with the value 0 (zero).<53>

Figure 14: A pipe as a sequence of chunks

2.2.5.3.5 Pointers

A pointer representation MUST be 8 bytes. Pointer representations MUST be aligned on 8-byte boundaries in the octet stream.

2.2.5.3.5.1 Embedded Reference Pointers

An embedded reference pointer MUST be represented in two parts: an 8-octet value in place that MUST NOT be NULL and a possibly deferred representation of the referent. The algorithm for deferral of referent is as specified by NDR in [C706] section 14.3.12.3, Algorithm for Deferral of Referents. NDR64 MUST NOT implement the special case specified by NDR for arrays of reference pointers, and the 8-octet non-NULL value MUST always be transmitted in place.

2.2.6 Type Serialization Version 1

Type serialization version 1 is a set of extensions to the IDL/+ pickle, as specified in [C311] Part 2, IDL/NDR Pickle. Implementations of these extensions allow marshaling/unmarshaling according to the NDR transfer syntax of application-specified types by using an application-provided octet stream.

Type serialization version 1 can use either a little-endian or big-endian integer and floating-pointer byte order, but MUST use the IEEE floating-point format representation and ASCII character format.

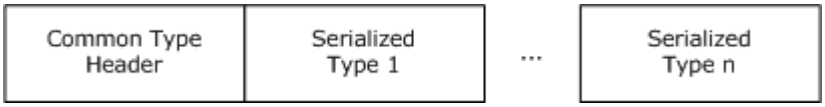


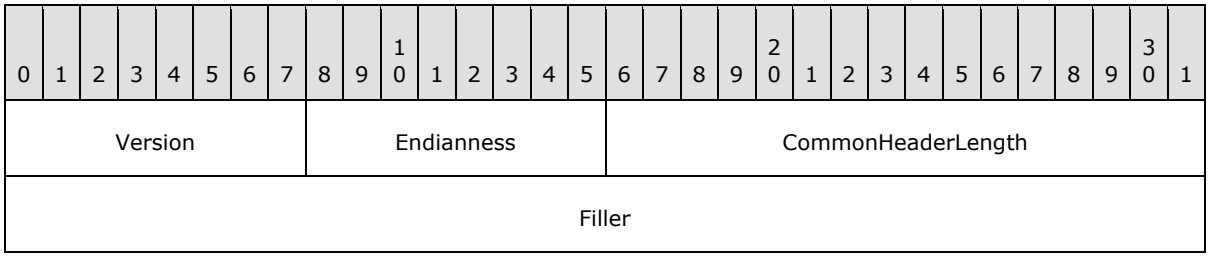
Figure 15: Type serialization version 1

Multiple top-level data types can be **serialized** into the same type serialization stream in the same way multiple parameters in a procedure are marshaling into an octet stream. A top-level data type is the data type an application provides to the implementation of these extensions to be serialized or **de-serialized**. A top-level data type MUST be either an NDR-constructed type or a primitive type. Each top-level data type is serialized/de-serialized as a whole, according to the rules that follow.

2.2.6.1 Common Type Header for the Serialization Stream

One common type header is created per serialization octet stream. The common header applies to all of the typed data in the octet stream. This common type header MUST be presented by using little-endian format in the octet stream. The first byte of the common type header MUST be equal to 1 to indicate this level of type serialization.

The common type header alignment MUST be aligned on an 8-byte boundary.



Version (1 byte): Must be set to 1 to indicate type serialization version 1.

Endianness (1 byte): Specifies the endianness of types serialized in the octet stream as follows.<54>

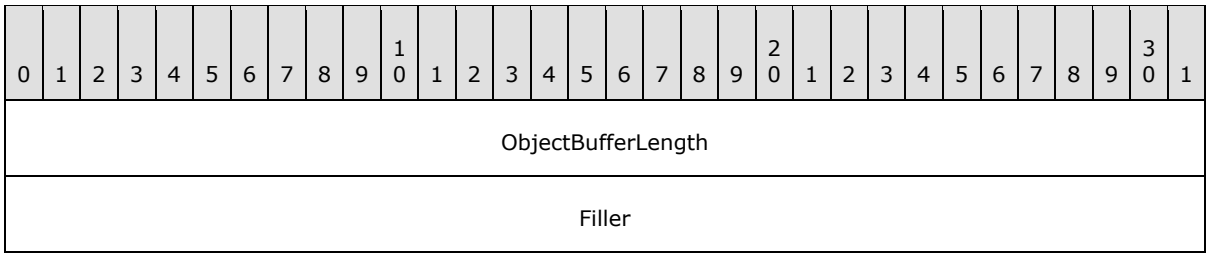
Value	Meaning
0x10	Little-endian
0x00	Big-endian

CommonHeaderLength (2 bytes): The length in bytes of this common type header. MUST be set to 8.

Filler (4 bytes): Reserved field. MUST be set to 0xcccccccc on marshaling, and SHOULD be ignored during unmarshaling.

2.2.6.2 Private Header for Constructed Type

A top-level NDR constructed type MUST be preceded by a private header, as specified below.



ObjectBufferLength (4 bytes): Indicates the length of a serialized top-level type in the octet stream. It MUST include the padding length and exclude the header itself.

Filler (4 bytes): Reserved field. MUST be set to 0 (zero) during marshaling, and SHOULD be ignored during unmarshaling.

The private type header MUST be aligned on an 8-byte boundary in the octet stream. If the length of the serialized top-level constructed type in the octet stream is not a multiple of 8 octets, the data MUST be padded at the end to ensure its total length is an integral multiple of 8 bytes in length.

Like a parameter in a procedure, the top-level constructed type MUST be represented in NDR format in the octet stream following the private header.

2.2.6.3 Primitive Type Serialization

For any top-level NDR primitive type, there MUST NOT be any private header preceding the actual type. The type MUST be aligned on an 8-byte boundary. If the size of the primitive type is not an integral multiple of 8 bytes, the data MUST be padded at the end to ensure its total length is an integral multiple of 8 bytes.

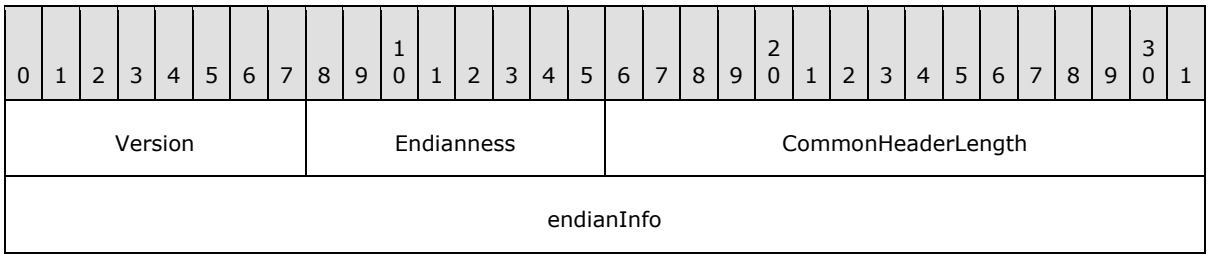
2.2.7 Type Serialization Version 2

Version 2 of type serialization is a set of modifications to type serialization version 1, as specified in section 2.2.6. Implementations of these extensions allow marshaling/unmarshaling of application-specified data types by using an application-provided serialization stream, according to either NDR or NDR64 transfer syntax.

Type serialization version 2 MUST use little-endian integer and floating-pointer byte order, IEEE floating-point format representation, and ASCII character format. The first byte in the octet stream MUST be 2 to indicate this level of type serialization.

2.2.7.1 Common Type Header

One common type header is created per serialization octet stream. The common header applies to all of the typed data in the octet stream. The common type header MUST be aligned on a 16-byte boundary.



Reserved
...
...
...
TransferSyntax
...
...
...
...
InterfaceID
...
...
...
...

Version (1 byte): MUST be set to 2 to indicate type serialization version 2.

Endianness (1 byte): MUST be set to little-endian (0x10).

CommonHeaderLength (2 bytes): Indicates the length in bytes of the common header. MUST be 0x40.

endianInfo (4 bytes): Reserved field. MUST be set to 0xcccccccc during marshaling, and SHOULD be ignored during unmarshaling.

Reserved (16 bytes): Reserved fields. MUST be set to 0xcccccccc during marshaling, and SHOULD be ignored during unmarshaling.

TransferSyntax (20 bytes): RPC transfer syntax identifier used to encode data in the octet stream. It MUST use RPC_SYNTAX_IDENTIFIER format, as specified in section 2.2.2.7. It MUST be either the NDR transfer syntax identifier or the NDR64 transfer syntax identifier.

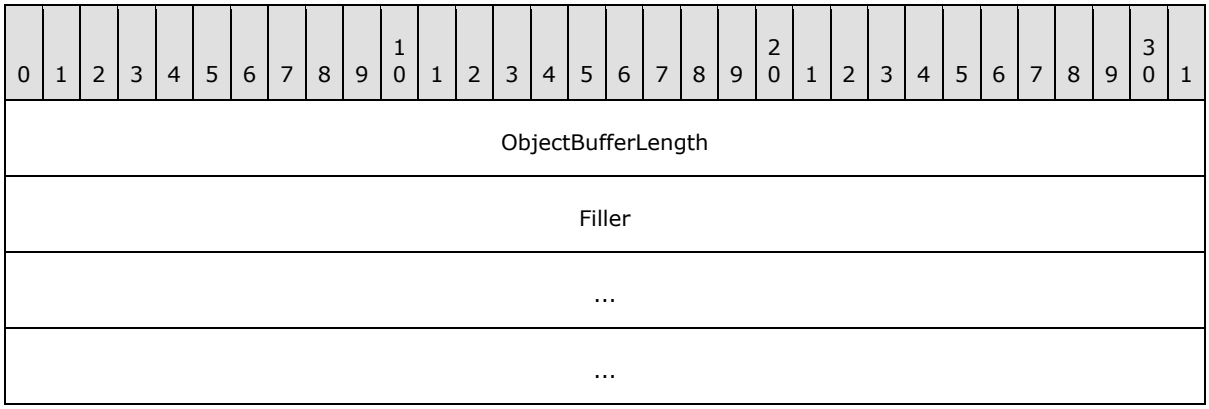
InterfaceID (20 bytes): Interface identifier, as specified in the IDL file. It MUST use the interface identifier format, as specified in [C706] section 3.1.9, Interface Identifier. Implementations MAY ignore the value of this field.<55>

Similar to Type Serialization Version 1 (section 2.2.6), multiple top-level data types can be serialized into the same type serialization stream, in the same way multiple parameters in a procedure are marshaled into an octet stream. All top-level data types in the same octet stream MUST be serialized by using the same transfer syntax as specified in the Common Type Header.

2.2.7.2 Private Header

In type serialization version 2, the private header MUST precede all top-level data types in the octet stream.

The private type header MUST be aligned on a 16-byte boundary. If the length of the serialized top-level data type in the octet stream is not a multiple of 16 octets, the data must be padded at the end to ensure its total length is an integral multiple of 16 octets in length.



ObjectBufferLength (4 bytes): Indicates the length of a serialized top-level data type in the octet stream. It MUST include the padding length and exclude the header itself.

Filler (12 bytes): Reserved field. MUST be set to 0 (zero).

3 Protocol Details

Remote procedure call (RPC) extensions preserve the DCE Remote Procedure Call (RPC) 1.1 Specification ([C706]) model of operation between an initiator (or client) and a responder (or server). RPC has two protocol variants: connection-oriented and connectionless. The following sections first specify protocol details that are common between connectionless RPC and connection-oriented RPC protocol variants, and then specify details particular to each.

3.1 Connectionless and Connection-Oriented RPC Protocol Details

This section defines the protocol details that are common between connectionless RPC and connection-oriented RPC protocol variants.

3.1.1 Common Details

This section defines the protocol details that are common between the client and server roles.

3.1.1.1 Abstract Data Model

3.1.1.1.1 Security Context

A data element is maintained by the security provider for each logical stream of data in RPC.

A security context is created by the security provider but is used by the RPC runtime and higher-level protocols.

A security context has an expiry time associated with it. The security context SHOULD NOT be used after its expiry time.

3.1.1.2 Timers

There are no timers that are common between connectionless RPC and connection-oriented RPC protocol variants.

3.1.1.3 Initialization

There is no initialization that is common between connectionless RPC and connection-oriented RPC protocol variants.

3.1.1.4 Higher-Layer Triggered Events

3.1.1.4.1 Causal Ordering

These extensions allow for higher-level protocols to issue method calls that are said to be causally ordered. If any two method calls N and N+1 are specified to be causally ordered on the client, these extensions MUST ensure that N is dispatched before N+1 on the server. On the client, the exact way in which method calls are specified to be causally ordered is implementation-specific. On the server, the exact way in which dispatch of N is determined to be complete so that N+1 can be dispatched is also implementation-specific.

3.1.1.5 Message Processing Events and Sequencing Rules

3.1.1.5.1 Processing Extensions Details

3.1.1.5.1.1 Extension in NDR Transfer Syntax

Section 2.2.4 specifies the Interface Definition Language (IDL) extensions that affect the syntax and processing of the messages.

3.1.1.5.1.1.1 __int3264

__int3264 is represented in the octet stream as 4 octets in Network Data Representation (NDR) transfer syntax. On 32-bit platforms, it is represented as a 4-byte integer in memory. On 64-bit platforms, it is represented as an 8-byte integer, and the higher 4 bytes MUST be truncated on the sender side during marshaling, and MUST be extended appropriately (signed or unsigned) on the receiving side during unmarshaling.

3.1.1.5.1.1.2 Binding Handle Extension

[C706] section 4.3.5 specifies requirements for binding handle usage. In the Remote Procedure Call Protocol, a binding handle MAY appear anywhere in a method's list of parameters.<56>

3.1.1.5.2 Indicating Octet Stream as Invalid

The remote procedure call (RPC) runtime MUST indicate to higher-layer protocols on the client about invalid octet streams, including different data consistency check failures, as specified in section 3.1.2.4.1. On the server side, RPC runtime MUST handle an invalid octet stream, as specified in section 3.1.3.5.2.

3.1.1.5.3 Strict NDR/NDR64 Data Consistency Check

These extensions update the DCE Remote Procedure Call (RPC) 1.1 Specification ([C706]) by specifying that, during unmarshaling, invalid octet streams SHOULD be rejected by enforcing a set of rules referred to as strict data consistency checks. All the consistency check rules specified in the following sections are also applicable to Network Data Representation (NDR) 64 transfer syntax. This is often referred to as robust check in Windows documentation and implementation.

The consistency checks are grouped into categories called target levels. The two target levels are: Target level 5.0 (as specified in section 3.1.1.5.3.2) and target level 6.0 (as specified in section 3.1.1.5.3.3). Target level 6.0 is a strict superset of target level 5.0.

A consistency check is the act of ascertaining a certain relation between two or more values in the octet stream inside an implementation of these extensions. If the relation is true, the consistency check MUST be regarded as passing. If the relation is not true, the consistency check MUST be regarded as failing. The set of consistency check rules follow, and **correlation** validation is the most important one.

3.1.1.5.3.1 Correlation Validation

Correlation validation is performed between two fields or two parameters during unmarshaling. The fields/parameters that can be correlated are defined using the productions specified in [C706] section 4.4.1. In the productions that specify Interface Definition Language (IDL) syntax, in production 67-69, the field with a specific <field_attribute> is defined as being correlated to the

argument specified by <Identifier>. The argument identified by <Identifier> is defined as dictating the correlation.

The correlation validation process MUST validate the consistency between the two correlated values in the octet stream according to the rules that follow. Correlation validation MUST be regarded as succeeding if the two values are evaluated to be equal to each other; otherwise, the validation MUST be regarded as failing. There are several basic types of correlation validation:

- Conformance correlation validation: Succeeds if the maximum count is equal to the evaluation result for the correlated argument where the correlated argument is determined via the production rules above.
- Varying correlation validation: Succeeds if the actual count is equal to the evaluation result for the correlated argument where the correlated argument is determined via the production rules above.
- Offset correlation validation: Succeeds if the offset count is equal to the evaluation result for the correlated argument where the correlated argument is determined via the production rules above.
- Union correlation validation: Succeeds if the union tag is equal to the evaluation result for the correlated argument where the correlated argument is determined via the production rules above.

In these extensions, an expression is allowed in conformance, varying, or union, as specified in section 2.2.4.7. Correlation validation SHOULD check the correlation between the correlated values after the expression is evaluated, and MUST succeed if the correlated values are equal after evaluating the expression.

For correlation validation usage, see the example in section 4.6.

3.1.1.5.3.2 Target Level 5.0

This section specifies target level 5.0 strict NDR/NDR64 data consistency check correlation validation checks. <57>

3.1.1.5.3.2.1 Correlation Validation Checks

These extensions clarify the interpretation as specified in [C706] for the cases that follow with regard to different correlation validation scenarios.

3.1.1.5.3.2.1.1 Maximum Count of a Conformant Array or Conformant Varying Array Is Dictated by Another Parameter or Field of a Structure

This target level implementation of these extensions SHOULD validate the conformance correlation between the maximum count of the conformant array and the parameter or field dictating the conformance. If the conformant correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.2 Maximum Count of a Conformant Structure or Conformant Varying Structure Is Dictated by a Field of the Structure

This target level implementation of these extensions SHOULD validate the conformance correlation between the maximum count of the conformant array and the field dictating the conformance. If the conformance correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.3 Maximum Count of a Conformant Array or Conformant Varying Array Is a Constant Defined in IDL File

This target level implementation of these extensions SHOULD validate the conformance correlation between the maximum count of the conformant array and the constant. If the conformance correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.4 Maximum Count of a Conformant Structure or Conformant Varying Structure Is a Constant

This target level implementation of these extensions SHOULD validate the conformance correlation between the maximum count of the conformant array and the constant. If the conformant correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.5 first_is of a Varying Array or Conformant Varying Array Is Specified by Another Parameter or Field of a Structure

This target level implementation of these extensions SHOULD validate the offset correlation between the offset of the varying array and the parameter or field dictating the offset. If the offset correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.6 first_is of a Conformant Varying Structure Is Specified by a Field in the Structure

This target level implementation of these extensions SHOULD validate the offset correlation between the offset of the varying array and the field dictating the offset. If the offset correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.7 first_is of a Varying Array, Conformant Varying Array, or Conformant Varying Structure Is Not Present in IDL

This target-level implementation of these extensions SHOULD validate that the offset of the varying array equals 0 (zero). If the offset value is not 0 (zero), the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.8 Actual Count of a Varying Array or Conformant Varying Array Is Dictated by Another Parameter or Field of a Structure

This target level implementation of these extensions SHOULD validate the varying correlation between the actual count of the varying array and the parameter or field dictating the actual count. If the varying correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.9 Actual Count of a Conformant Varying Structure Is Dictated by a Field in the Structure

This target level implementation of these extensions SHOULD validate the varying correlation between the actual count of the varying array and the field dictating the actual count. If the varying correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.10 Maximum Count of a Conformant and Varying String Is Dictated by Another Parameter or Field of a Structure

This target level implementation of these extensions SHOULD validate conformance correlation between the maximum count of the conformant and varying string against the parameter or field dictating the conformance, and it SHOULD also validate that the offset of the string is equal to 0 (zero). If either validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.11 Union Validation

Similar to conformant validation, this target-level implementation of these extensions SHOULD validate the discriminant of the union against the representation of the union tag, as specified in [C706] section 14.3.8, Unions. If the union correlation validation fails, the implementation MUST indicate the octet stream as invalid.

3.1.1.5.3.2.1.12 General Conformant Varying Validation

In all conformant varying cases, the maximum count MUST be equal or greater than the sum of actual count and offset. If this validation fails, the implementation MUST treat the octet stream as invalid.

3.1.1.5.3.2.2 Additional Limitations

These extensions add the following limitations to those as specified in [C706].

3.1.1.5.3.2.2.1 Limiting Maximum Count and Octet Stream Length

These extensions specify that a conformant array or conformant and varying string SHOULD have, at most, 2^{31}-1 elements in each dimension.<58>

3.1.1.5.3.2.2.2 strict_context_handle

A context handle created by a method belonging to one interface SHOULD NOT be accepted by a method belonging to another interface when a strict_context_handle consistency check is activated. For more information on syntax details, see section 2.2.4.13.

3.1.1.5.3.2.2.3 Rejecting Insufficient Octet Stream

An octet stream MUST contain sufficient data to unmarshal all the required parameters. Implementation of these extensions SHOULD indicate the octet stream as invalid if there is insufficient data.

3.1.1.5.3.2.2.4 range Attribute to Limit the Scope of Integral Values and the Number of Elements in Pipe Chunks

In target level 5.0 of strict NDR/NDR64 data consistency checking, implementation of these extensions can limit the allowed scope for integral types and pipes. If the integral data value is out of the specified range scope, the implementation SHOULD indicate the octet stream as invalid.

Implementation of these extensions can also limit the acceptable range of elements in a pipe chunk. The implementation SHOULD indicate the octet stream as invalid if the number of elements in a pipe chunk is out of the specific range scope. For syntax information, see section 2.2.4.12.1.

3.1.1.5.3.2.2.5 auto_handle Deprecation

Implementation of this level of the extensions SHOULD NOT accept the auto_handle attribute if specified on an interface.<59>

3.1.1.5.3.2.2.6 Ignoring Alignment Gap

The content of alignment gaps, either within a structure or before an item in the octet stream, SHOULD be ignored.

3.1.1.5.3.3 Target Level 6.0

This section specifies target level 6.0 strict NDR/NDR64 data consistency check limitations. <60>

3.1.1.5.3.3.1 Additional Limitations

3.1.1.5.3.3.1.1 type_strict_context_handle

An implementation of these extensions at this target level can activate the type strict context handle. When it is activated, the implementation SHOULD reject the use of context handles as an argument if the argument type on the method being called is different from the argument type on the method that created the context handle.

Context handles defined with unique type names are treated as being of different types for the purpose of the type_strict_context_handle check. For example, the following two context handles are two different types:

```
typedef [context_handle] void * PCTXT1;
typedef [context_handle] void * PCTXT2;
```

For syntax information, see section 2.2.4.14.

3.1.1.5.3.3.1.2 Unique or Full Pointer to Conformant Array Consistency Check

A conformant array or conformant and varying string correlated with another parameter or field can be referred by a unique pointer or full pointer. While it is allowed to have a non-zero correlated value with a NULL pointer (as specified in [C706] section 14.3.10), implementations of these extensions SHOULD indicate the octet stream as invalid if the following conditions are met:

- Correlated value evaluates to be non-zero.
- The unique or full pointer referring the conformant array or conformant and varying string is NULL (0).
- The conformant array or conformant and varying string does not have the disable_consistency_check attribute as specified in section 2.2.4.15.<61>

3.1.1.5.3.3.1.3 range Attribute to Limit the Range of Maximum Count of Conformant Array and String Length

In target level 6.0 of strict NDR/NDR64 data consistency check, in addition to the target level 5.0 range checks, implementations of these extensions can also limit the acceptable range for conformant and string. Implementations can indicate the acceptable value range for the maximum count of the conformant array when a range is applied to the conformance. The implementation

SHOULD indicate the octet stream as invalid if the maximum count of a conformant array is not in the specified acceptable range.

When a range is applied to a conformant and varying string without correlation, it indicates the acceptable length, including the NULL terminator, of the string. The implementation SHOULD indicate the octet stream as invalid if the string length, including terminator, is outside the acceptable range. For syntax information, see section 2.2.4.12.2.

3.1.1.5.4 Restriction on Remote Anonymous Calls

For security reasons, an implementation of these extensions MAY choose to reject remote anonymous calls.<62>

3.1.1.6 Timer Events

There are no timer events that are common between connectionless RPC and connection-oriented RPC protocol variants.

3.1.1.7 Other Local Events

There are no other local events that are common between connectionless RPC and connection-oriented RPC protocol variants.

3.1.2 Client Details

3.1.2.1 Abstract Data Model

There is no abstract data model that is common between clients for connectionless RPC and connection-oriented RPC protocol variants.

3.1.2.2 Timers

There are no timers that are common between clients for connectionless RPC and connection-oriented RPC protocol variants.

3.1.2.3 Initialization

There is no initialization that is common between clients for connectionless RPC and connection-oriented RPC protocol variants.

3.1.2.3.1 Higher-Layer Triggered Events

There are no higher-layer triggered events that are common between clients for connectionless RPC and connection-oriented RPC protocol variants.

3.1.2.4 Message Processing Events and Sequencing Rules

3.1.2.4.1 Indicating Invalid Octet Stream on Client

Implementations of these extensions MUST notify higher layers of invalid octet streams, including data consistency check failures, in an implementation-specific way. This may be through returning a status code, throwing an exception, or in some other implementation-specific way that is not defined by this specification. Details on Win32 error codes are specified in [MS-ERREF].

3.1.2.5 Other Local Events

3.1.2.5.1 Client Conformant Validation Processing for Response Data

In target level 5.0 of strict NDR/NDR64 data consistency check, as specified in section 3.1.1.5.3.2, implementations of these extensions SHOULD perform the following correlation validation in the client stub if the remote procedure call (RPC) runtime writes into client-provided memory during unmarshaling.

3.1.2.5.1.1 Maximum Count of a Conformant Array Is Dictated by Another Parameter or Field of a Structure

This target level of implementation for these extensions MUST:

- Capture the evaluation result of the parameter dictating the conformance before unmarshaling for later use during unmarshaling.
- Indicate the octet stream as invalid during unmarshaling if the maximum count of the conformant array of the response data exceeds the evaluation result of the parameter dictating the conformance that was previously captured.

3.1.2.5.1.2 Offset and/or Actual Count of a Conformant Array Is Dictated by Another Parameter or Field of a Structure

This target level of implementation for these extensions MUST:

- Capture the evaluation result of the parameter dictating the conformance before unmarshaling for later use during unmarshaling.
- Indicate the octet stream as invalid during unmarshaling if the sum of offset and actual count of the conformant varying array of the response data exceeds the evaluation result of the parameter dictating the conformance that was previously captured.

3.1.2.5.1.3 Maximum Count of a Conformant and Varying String Is Dictated by Another Parameter

This target level of implementation for these extensions MUST:

- Capture the evaluation result of the parameter dictating the conformance before unmarshaling for later use during unmarshaling.
- Indicate the octet stream as invalid during unmarshaling if the string length, including terminator, of the response data exceeds the evaluation result of the parameter dictating the conformance that was previously captured.

3.1.2.5.1.4 Maximum Count of Conformant Varying String Is Not Dictated by Other Parameters or Fields

This target level of implementation for these extensions MUST:

- Capture the string length, including terminator, before unmarshaling for later use during unmarshaling.

- Indicate the octet stream as invalid during unmarshaling if the string length, including terminator, of the response data exceeds the evaluation result of the parameter dictating the conformance that was previously captured.

3.1.2.5.1.5 Conformant Structure

This target level of implementation for these extensions MUST:

- Capture the evaluation result of the field dictating the conformance before unmarshaling for later use during unmarshaling.
- Indicate the octet stream as invalid during unmarshaling if the maximum count of the conformant structure of the response data exceeds the evaluation result of the field dictating the conformance that was previously captured.

3.1.2.5.1.6 Conformant Varying Structure

This target level of implementation for these extensions MUST:

- Capture the evaluation result of the field dictating the conformance before unmarshaling for later use during unmarshaling.
- Indicate the octet stream as invalid during unmarshaling if the sum of offset and actual count of the conformant varying structure from the response data exceeds the evaluation result of the field dictating the conformance that was previously captured.<63>

3.1.3 Server Details

3.1.3.1 Abstract Data Model

There is no abstract data model that is common between servers for connectionless RPC and connection-oriented RPC protocol variants.

3.1.3.2 Timers

There are no timers that are common between servers for connectionless RPC and connection-oriented RPC protocol variants.

3.1.3.3 Initialization

3.1.3.3.1 Delay Use of Protocol Sequences on the Endpoint Mapper

On a system that supports a given protocol sequence, these extensions explicitly allow an endpoint mapper instance to delay listening on that protocol sequence until at least one server using **dynamic endpoints** on the system is listening on that protocol sequence.

Even though a system is fully capable of using a protocol sequence, it MAY choose not to listen on a particular protocol sequence when no server is using it. Therefore, a client implementation of these extensions MUST NOT assume that a system that is not listening on a particular protocol sequence is necessarily incapable of supporting that protocol sequence.<64>

3.1.3.4 Higher-Layer Triggered Events

There are no higher-layer triggered events that are common between servers for connectionless RPC and connection-oriented RPC protocol variants.

3.1.3.5 Message Processing Events and Sequencing Rules

3.1.3.5.1 Server Stub Memory Allocation Limit

See the Windows behavior tag.<65>

3.1.3.5.2 Indicating Invalid Octet Stream in Server

When the RPC runtime determines that a network octet stream is invalid, it MUST indicate the failure to the client. The form of the indication is dependent on whether the RPC protocol variant used is connection-oriented or connectionless. For information about how a connection-oriented protocol variant returns a server unmarshaling failure to the client, see section 3.3.3.4.1. For information about how a connectionless protocol variant returns a server unmarshaling failure to the client, see section 3.2.3.5.1. In either case, the status code returned MUST be 0x6f7.

Details about Win32 error codes are specified in [MS-ERREF].

3.1.3.5.3 Interpretation of Tower Encodings

These extensions change some details on how the tower encodings, as specified in [C706] Appendix L, are interpreted. All provisions specified in [C706] that are not specifically overridden here are assumed to be the same as specified in [C706].

- Implementations of these extensions MUST ignore the network address portion of the tower. Therefore, the endpoint mapper MUST only accept interface registration of interfaces that are running locally on the machine.
- As specified, [C706] allows for any number of floors in the tower encoding. Implementations of these extensions SHOULD reject towers with more than six floors.
- [C706] specifies that the floor count, LHS byte count, and RHS byte count fields in the tower floor must be encoded by using little-endian encoding. Implementations of these extensions MUST handle both little-endian and big-endian encodings for these fields.

3.1.3.6 Timer Events

There are no timer events that are common between servers for connectionless RPC and connection-oriented RPC protocol variants.

3.1.3.7 Other Local Events

There are no other local events that are common between servers for connectionless RPC and connection-oriented RPC protocol variants.

3.2 Connectionless RPC Protocol Details

See the Windows behavior tag.<66>

3.2.1 Common Details

3.2.1.1 Abstract Data Model

This section specifies 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.

3.2.1.1.1 State Machines

As specified in [C706] section 9.6, Connectionless Protocol Machines contains state machines for the client and server roles. These extensions replace the state machines, as specified in [C706], with the state machines specified in sections 3.2.2.1 and 3.2.3.1.<67>

3.2.1.2 Timers

There are no timers that are common between the client and server.

3.2.1.3 Initialization

There is no initialization that is common between the client and server.

3.2.1.4 Higher-Layer Triggered Events

3.2.1.4.1 Building and Using a Security Context

To make a secure call, a security context needs to be created before it can be used. The process of creation involves exchanging one or more messages between the client and server implementations of a security provider, and is also called building a security context. During the process of building a security context, a security provider may optionally exchange messages with an entity other than the client or server (for example, a KDC), but this exchange is not addressed in this document. The scope of a built security context is the connection. If a client wants to use a security context on a different connection, it MUST totally rebuild it for that different connection.

Upon receiving and processing an authentication token at any point in the authentication on either the client or server, the security provider MUST indicate to RPC runtime one of three abstract results from the processing: an error, a success, or a request for further security legs, as specified in [RFC2743]. If the security provider indicates an error, the RPC runtime takes recovery action that is dependent on the location of the error.

The process of building a security context MUST start on the client. The client retrieves an authentication token from the security provider by using an implementation-specific equivalent of the abstract GSS_Init_sec_context call, as specified in [RFC2743]. The client MUST choose a value for the **key_vers_num** field of the **sec_trailer_cl** structure such that it is unique within the scope of the given connection. The client then MUST use the token to sign or seal one or more request PDUs, and then sends them to the server. If any of these steps encounters a failure, the client RPC runtime MUST discard the security context and MUST NOT send any further PDUs on that connection. It SHOULD discard the activity unless it is expecting responses on a multiplexed activity, as specified in this section, in which case it SHOULD wait for all responses to arrive before discarding the activity.

When the server receives a PDU containing a non-zero **auth_proto** field, it checks the **key_vers_num** field of the PDUs **sec_trailer_cl** structure. If the server does not already have a security context matching the **key_vers_num**, it MUST:

- Locate a security provider matching the **auth_proto** field.
- Request that it create a new security context.

- Create a token through an implementation-specific equivalent of the abstract **GSS_Accept_sec_context** call, as specified in [RFC2743].

The server MUST send the token to the client by creating a binding handle to the client and calling **conv_who_are_you_auth** with the token in the **in_data** parameter. If any of these steps encounters an error, the server SHOULD send a fault or reject PDU, as appropriate, and discard the security context.

The client MUST provide the token to its security provider by using an implementation-specific equivalent of the abstract **GSS_Init_sec_context** call, as specified in [RFC2743], and MUST send the response token to the server in the **out_data** parameter of the **conv_who_are_you_auth**. If the response token is large enough to require calls to **conv_who_are_you_auth_more**, the client MUST preserve the token until it has returned all of the token to the server. If any of these steps encounters an error, the client SHOULD send a fault or reject PDU, as appropriate, and discard the security context.

The server MUST provide the response token to the security provider using an implementation-specific equivalent of the abstract GSS_Accept_sec_context call, as specified in [RFC2743]. If the security provider returns success from processing the authentication token, the security context is successfully created. If this step encounters an error, the server SHOULD send a fault or reject PDU to the client and discard the security context.

If the security provider indicates a request for further security legs, the server should send a nocall PDU to the client and discard the security context.

For information on client and server state machines, see sections 3.2.2.1 and 3.2.3.1.

Once negotiated, a security context SHOULD be maintained by both client and server implementations for the lifetime of the connection it is negotiated on, unless the security provider indicates that the context has expired.

3.2.1.4.1.1 Using a Security Context

After a security context is built, the security context can be used by the RPC runtime and higher-level protocols to perform authorization decisions. Besides using the security context for authorization decisions, the RPC runtime can also use the security context to create a logical stream of data that is protected from tampering and information disclosure on the network.

The amount of protection applied depends on the authentication level for the security context requested by the client when the security context is created. The authentication level is applied in two dimensions.

- In the first dimension, the authentication level controls what capabilities the RPC runtime MUST request from the security provider when the security context is being built, as detailed in the first table below. It is possible for a security provider to not be able to provide a certain capability. In this case, the lack of the capability MUST be considered by the RPC runtime as equivalent to the security provider returning an error, and is handled as specified in the previous section.
- In the second dimension, the authentication level controls how the RPC runtime MUST perform protocol data unit (PDU) protection for the different PDU segments using the security context as detailed in the second table below.

The following table specifies the abstract capability that the RPC runtime MUST request from the security provider when the security context is being created. The capabilities below are further specified in [RFC2743] section 1.2.1.2. The capabilities requested at each level include the ones requested at the previous level.

Authentication level	Capability requested
RPC_C_AUTHN_LEVEL_CONNECT	None
RPC_C_AUTHN_LEVEL_CALL	Replay Detect
RPC_C_AUTHN_LEVEL_PKT	Replay Detect
RPC_C_AUTHN_LEVEL_INTEGRITY	Sequence Detect, Integrity
RPC_C_AUTHN_LEVEL_PRIVACY	Confidentiality

As specified above, once the security context is built, the RPC runtime MUST also use the authentication level to determine how the different PDU segments are protected.

Authentication level	PDU header	PDU body	Sec_Trailer
RPC_C_LEVEL_CONNECT	None	None	None
RPC_C_AUTHN_LEVEL_CALL	None	None	None
RPC_C_AUTHN_LEVEL_PKT	None	None	None
RPC_C_AUTHN_LEVEL_INTEGRITY	None	Integrity	None
RPC_C_AUTHN_LEVEL_PRIVACY	None	Confidentiality	None

In the table above, "None" means no protection, "Integrity" means an integrity check per [RFC2743] section 2.3.1 MUST be applied, and "Confidentiality" means that the segment MUST be encrypted (conf_req_flag is TRUE per [RFC2743] section 2.3.3).

This protocol does not specify whether the authentication token itself is protected from tampering by the security provider. It also does not specify how the security provider applies integrity or confidentiality protection to a PDU segment. The algorithms for doing so are specific to the security provider. For information about a security provider, see the documentation for that security provider.

3.2.1.4.2 Callbacks

Connectionless RPC protocols do not have support for application-level callback calls.

3.2.1.5 Message Processing Events and Sequencing Rules

3.2.1.5.1 Authentication

The marshaled stub data of a client's **conv_who_are_you_auth** response MUST fit into a single unfragmented RESPONSE packet to interoperate with a server running on Windows NT 4.0. This is not a requirement for a client to interoperate with a server running on Windows 2000.<68>

These extensions do not require support for the **Authentication Service** rpc_c_authn_dce_secret, as specified in [C706] section 13.1.2.2, Authentication Services. It supports authentication by using the (NTLM) Authentication Protocol and Kerberos Protocol, using authentication type constants as specified in section 2.2.1.1.7. The authentication tokens present in each PDU are specified in section 2.2.3.5.<69>

3.2.1.5.2 Overlapped Calls

These extensions extend the connectionless protocol, as specified in [C706], to allow multiple simultaneously active calls in a single activity. This reduces the overhead of asynchronous calls, which ordinarily require a separate activity and security context for each overlapping call. Use of the new feature requires that both the client and server support the extension.

the processing order for calls on the server is specified in [C706] section 6.1. That definition is preserved in these extensions. These extensions deviate from what is specified in [C706] by allowing the [in] and [out] buffers of multiple calls to overlap in transmission.

The server **conv_who_are_you2** and **conv_who_are_you_auth** **conversation callbacks** SHOULD set the **PF2_UNRELATED** bit; this indicates to the client that the server is capable of handling overlapped calls correctly.

After the client has successfully processed a conversation callback with the **PF2_UNRELATED** flag set, it MAY overlap calls on that activity if the application's call invocation allows it. If so, the client MUST set the **PF2_UNRELATED** flag in each REQUEST packet that is sent before a call with a lower sequence number has completed. This informs the server not to cancel or complete other active calls with lower sequence numbers.<70>

When the client has not successfully processed a conversation callback with the **PF2_UNRELATED** flag set, it MUST NOT overlap multiple calls of an activity. In particular, the client MUST NOT send a REQUEST for a call until all calls with lower sequence numbers have entered **STATE_ACK_PENDING**, **STATE_COMPLETE**, or **STATE_FAULT**. The client MUST NOT set the **PF2_UNRELATED** flag in any REQUEST packet.

The client and server MUST NOT set the **PF2_UNRELATED** flag in the header of any other packet type.

3.2.1.6 Timer Events

There are no common timers between the client and server.

3.2.1.7 Other Local Events

There are no other local events that are common between the client and server.

3.2.2 Client Details

3.2.2.1 Abstract Data Model

This section specifies 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.

3.2.2.1.1 Client Address Space or Association

Definitions of the **client address space (CAS)** identifier are specified in [C706] section 9.5.4.

Like a connection-oriented association, the CAS holds data relevant to the client's view of a particular RPC server:

- Server's transport

- Server's host name or address
- Server's endpoint, or the transport's endpoint mapper endpoint if the server endpoint is unknown

The client also caches several parameters of the server instance to improve the speed and latency of future calls:

- The optimum size of the server fragment window
- Its maximum protocol data unit (PDU) length
- A collection of connections
- A flag that indicates if the server supports the **PF2_UNRELATED** flag

3.2.2.1.2 Activity ID or Connection

For each activity, the client maintains the following:

- Activity universal unique identifier (UUID)
- Sequence number, as specified in [C706] section 12.5.2.11
- Security provider
- Authentication level

3.2.2.1.3 Call

The call is a data element that encapsulates the state associated with a client call. The client call is specified by a state machine with the following states.

STATE	Description
STATE_SEND_FRAGS	The client is sending fragments of the call's [in] parameters to the server. This is the call's initial state.
STATE_DISPATCHED	The server has called the server application stub.
STATE_RECEIVE_FRAGS	The server is sending fragments of the call's [out] parameters to the client.
STATE_ACK_PENDING	[out] parameters are received, and the call is waiting to send an ACK packet.
STATE_COMPLETE	The call completed successfully. This is a terminal state.
STATE_FAULT	The call failed. This is a terminal state.

The call maintains several state elements:

- A flag F_CANCELED that is true when the client application cancels the call.
- A counter CANCEL_EVENT_ID that identifies a particular cancellation attempt.
- A list of data fragments sent and received.
- A 32-bit unsigned integer STATUS that is the status code for the call.

The following diagram illustrates the state transitions.

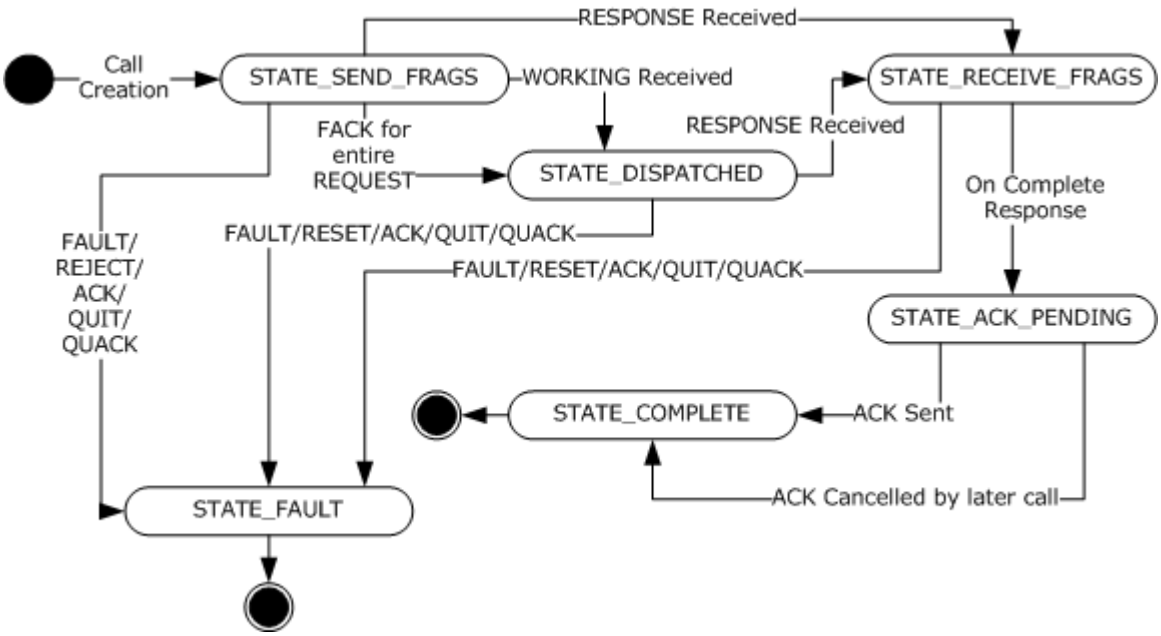


Figure 16: State transitions

Note The above conceptual data can be implemented by using a variety of techniques. An implementation is at liberty to implement such data in any way it pleases.

3.2.2.2 Timers

3.2.2.2.1 Packet Retransmission Timer

This timer is started when the client call transmits a REQUEST, FACK, PING, or QUIT packet. The timer is canceled when the client accepts a packet by using the rules specified in section 3.1.1.4.1. If the timer expires, the previously transmitted packet SHOULD be considered lost, and the client SHOULD send a new packet. If the call's **F_CANCELED** flag is set, a QUIT packet is sent; otherwise, the packet type depends on the call state:<71>

- STATE_SEND_FRAGS -> REQUEST
- STATE_DISPATCHED-> PING
- STATE_RECEIVE_FRAGS ->FACK

3.2.2.2.2 Cancel Timeout Timer

This timer MUST be started when the client call's **F_CANCELED** flag is set by an external entity. The timer MUST be canceled when the client receives a QUACK packet whose event ID matches the current cancel event ID. If the timer expires, the call MUST move into **STATE_FAULT**.<72>

3.2.2.2.3 Delayed Ack Timer

This timer MUST be started when the call enters **STATE_ACK_PENDING**. It MUST be canceled when the client initiates another call by using the same connection. If the timer expires, the client MUST send an ACK packet to the server and enter **STATE_COMPLETE**.<73>

3.2.2.2.4 Context-Handle Keep-Alive Timer

This timer is per-activity, not per-call. It SHOULD be started with an interval of 20 seconds when the client receives a server context handle from any call belonging to the activity, as long as the timer is not already started. It SHOULD be canceled when the client no longer holds any server context handles belonging to the activity. If the timer expires, the client SHOULD choose one connection from the activity and make a call to convc_indy specifying the activity's UUID as *the cas_uuid* parameter.<74>

3.2.2.3 Initialization

A client is initialized when a higher-level protocol supplies to the client-side implementation of the RPC runtime sufficient information to start making RPCs. The following distinct steps may be taken by the higher-level protocol.

3.2.2.3.1 Create a Binding Handle

Information about creating a binding handle is specified in [C706] section 2.3.

3.2.2.3.2 Specify Security Settings

If a higher-level protocol requires security for its remote procedure method calls, it MUST supply to the client-side implementation of the remote procedure call (RPC) runtime information on:

- What security provider it wants to use.
- What authentication level it wants to use.
- Optionally what impersonation level it wants to use.
- Any other security provider-specific information necessary for the security provider to function.<75>

3.2.2.4 Higher-Layer Triggered Events

3.2.2.4.1 Make an RPC Method Call

3.2.2.4.1.1 Find a CAS (Association)

The client MUST find or create a client address space (CAS) compatible with the binding handle's host, protocol sequence, and endpoint. A client SHOULD choose an existing CAS if a compatible one exists.

If a new CAS is created, and the server is using a dynamic endpoint, the CAS initially points to the dynamic endpoint for the RPC protocol sequence being used. Otherwise, the CAS refers to the server's well-known endpoint.

3.2.2.4.1.2 Find an Activity

The client MUST choose an activity that is compatible with the binding handle's current security provider, identity, authentication level and impersonation level.

The client SHOULD use an existing activity if possible. Selection depends on the causal ordering of the current call with respect to outstanding asynchronous calls in the same association:

- If the current call should be causally ordered with respect to a previous call, the client MUST choose the same activity. If the server does not support the PF2_UNRELATED flag, the client cannot begin the call until its predecessor has completed.
- If the current call does not need to be causally ordered, the client SHOULD choose an existing activity satisfying one of the following two conditions:
  - No call is active.
  - A call is active and is in ACK_PENDING state.

If no such activity is available, the client MUST create a new one. Its sequence number SHOULD be initialized to zero.

3.2.2.4.1.3 Find a Security Context

The client SHOULD use the activity's current security context unless the context has expired. Each security context MUST have a unique **key_vers_num**, which is specified in the sec_trailer structure, to allow the server to identify which one is used for a given PDU.

3.2.2.4.1.4 Create a Call

If the activity has a prior call in STATE_COMPLETED or STATE_ACK_PENDING, the activity's sequence number MUST be incremented, and the previous call's delayed-ack timer MUST be canceled.

A new call MUST be created by using the current activity ID and sequence number. The new call MUST be placed in STATE_SEND_FRAGS, F_CANCELED MUST be set to false, CANCEL_EVENT_ID MUST be set to zero, and the send and receive fragment windows MUST be empty. The client MUST send one or more request fragments.

3.2.2.4.2 Cancel Requested

The call's F_CANCELED flag is set.

3.2.2.5 Message Processing Events and Sequencing Rules

The packet semantics are the same as what is specified in [C706], sections 6 and 12.5.

The packet type MUST be one of the connectionless packet types specified in [C706]; otherwise, the packet is discarded. Incoming packets MUST be processed while the call is in STATE_SEND_FRAGS, STATE_DISPATCHED, or STATE_RECEIVE_FRAGS; in other states, they MUST be discarded.

For a non-REQUEST packet, the activity ID and the sequence number in the packet MUST match those of the call itself. The **auth_proto** field MUST match the call's authentication type, and, if nonzero, the packet must pass verification by the security provider. Otherwise, the packet MUST be discarded silently.

A packet that has not been discarded by one of the preceding rules MUST cancel the call packet retransmission timer, as specified in section 3.1.1.1. If the server uses a dynamic endpoint, and the CAS points to the endpoint mapper endpoint for the protocol, the CAS and the call SHOULD be updated to point to the server endpoint that sent the packet. For more information, see the protocol example in section 4.5.

The following sections define handling of specific packet types.

3.2.2.5.1 REQUEST

A REQUEST packet MUST have auth-type equal to zero, and its interface ID MUST match the conversation manager interface as specified in [C706], Appendix P. The packet's Header.Flags.frag bit MUST be zero. Otherwise, the packet MUST be discarded.

If the packet is accepted, it is processed as specified in [C706], Appendix P.

3.2.2.5.2 PING

A PING MUST be discarded unless it refers to a conversation manager callback in progress. In this case, the client MAY respond with a WORKING packet.<76>

3.2.2.5.3 RESPONSE

All outbound fragments MUST be marked as received. If the call is in STATE_SEND_FRAGS or STATE_DISPATCHED, the state MUST change to STATE_RECEIVE_FRAGS. The inbound fragment window MUST be updated to include the response fragment, and a FACK MUST be sent unless the packet's Header.Flags.Nofack flag is set. If all inbound fragments are received, RPC MUST deliver the data to the client application, and the call MUST enter STATE_ACK_PENDING.

3.2.2.5.4 FAULT

The call state MUST change to STATE_FAULT with STATUS set to the status code in the fault packet.

3.2.2.5.5 WORKING

All outbound fragments MUST be marked as received. If the call is in STATE_SEND_FRAGS, the state MUST change to STATE_DISPATCHED.

3.2.2.5.6 NOCALL

The outbound fragment window SHOULD be updated, and the client SHOULD send a burst of REQUEST fragments.<77>

3.2.2.5.7 REJECT

The call state MUST change to STATE_FAULT with STATUS set to the status code in the packet.

3.2.2.5.8 ACK

The call state SHOULD change to STATE_FAULT with STATUS set to 0x6c0.

3.2.2.5.9 QUIT

The call state SHOULD change to STATE_FAULT with STATUS set to 0x6c0.

3.2.2.5.10 FACK

The outbound fragment window MUST be updated, and the client SHOULD send a burst of REQUEST fragments. If the server has received all request fragments, the call state SHOULD change to STATE_DISPATCHED.

3.2.2.5.11 QUACK

If the F_CANCELED flag is false, the packet MUST be discarded. If the packet has body data, and it is less than 9 bytes, or the body version is not zero, or the packet's event ID does not match the call's CANCEL_EVENT_ID, the packet MUST be discarded. Otherwise, the call state is changed to STATE_FAULT, and STATUS SHOULD be set to 0x6c0.

3.2.2.6 Timer Events

For information on timers, see section 3.2.2.2.

3.2.2.7 Other Local Events

3.2.3 Server Details

3.2.3.1 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.

3.2.3.1.1 Table of Security Providers

A server implementation MUST maintain an abstraction of a table of security providers indexed by authentication type. The table must have fields for authentication type and principal name.

Higher-level protocols indicate to the RPC runtime when to add rows to the table, when to delete rows from the table, and when to modify fields in the table by using implementation-specific APIs.

Many PDUs that arrive at a server have a field that selects an authentication type (also called security provider). These extensions MUST use the authentication type in the PDU as a selector in the table of security providers to route the PDU for processing to the correct security provider.

3.2.3.1.2 Table of Activity IDs

The server maintains a table of activities indexed by the activity UUID. For each activity, it maintains the following:

A lowest-allowed-sequence counter.

A CAS UUID that is an index into the client address space (CAS) table.

If the activity is secure, a table indexed by the security context ID (this is the same as the value of the **auth_context_id** field in the sec_trailer data structure, as specified in section 2.2.2.11). A new row is added to the table when a new security context is built. Removal of rows is implementation-dependent.

Once created, a call is active until it reaches STATE_COMPLETE. The activity's lowest-allowed-sequence is the sequence of the lowest active call. Received packets with lower sequence numbers are discarded.

When an activity is created, its CAS UUID is NULL; when a conv_who_are_you2 or conv_who_are_you_auth call for the activity completes successfully, the activity's CAS UUID is set to the returned value.

An incoming request PDU with a given security context ID MUST be routed to the security context retrieved from the table row with the same security context ID.<78>

3.2.3.1.3 Table of Client Address Spaces

The server maintains a table of client address spaces (CASs) indexed by CAS UUID, as specified in [C706] Appendix P. For each CAS, the server maintains a table of context handles associated with the client address space. Whenever a call on an activity instantiates a context handle, the context handle is added to the list for the activity's CAS.

The server deletes a CAS UUID and its associated context handles and activities if none of the CAS UUIDs activities receive any packets over a five-minute period. This follows TIMEOUT_IDLE, as specified in [C706] Appendix K.<79>

3.2.3.1.4 Call

The server maintains a set of active calls per activity.

The server call is defined by a state machine with the following states.

STATE	Description
STATE_INIT	The call has not received a packet. This is the initial state.
STATE_RECEIVE_FRAGS	The server has not received enough fragments to dispatch to the application stub.
STATE_WORKING	The server has dispatched the call to the application stub.
STATE_SEND_FRAGS	The server is sending the reply to the client.
STATE_COMPLETE	The call is no longer active. This is a terminal state.

Contrary to what is specified in [C706] Appendix P, Conversation Manager Interface, implementations of these extensions MUST NOT call conv_who_are_you. Instead, they MUST call conv_who_are_you2.

These extensions also MUST NOT call conv_are_you_there.

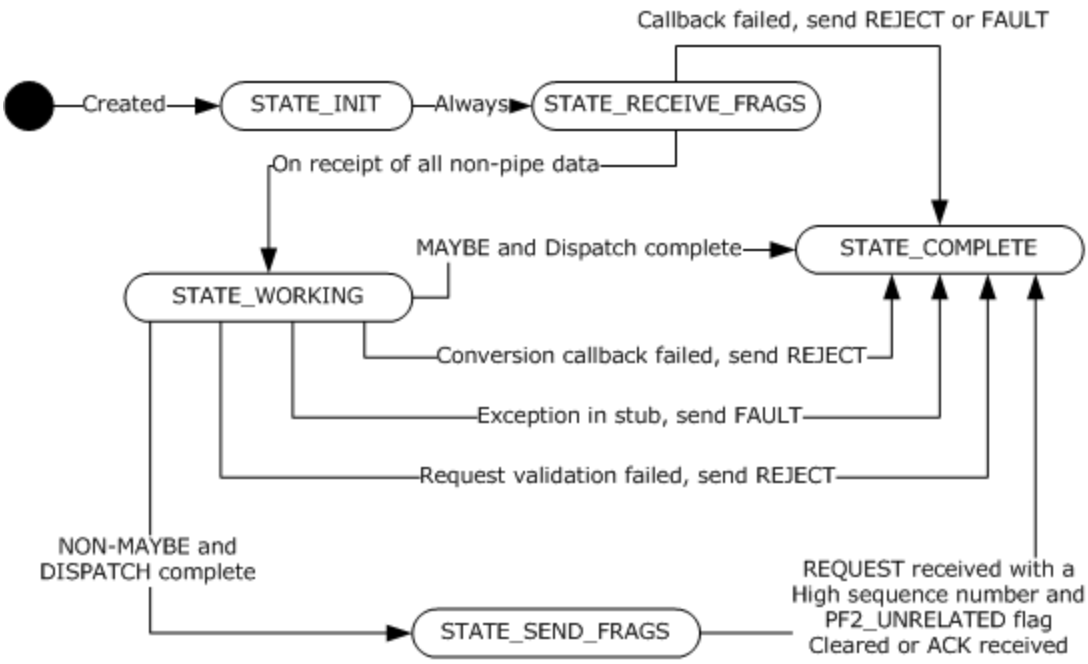


Figure 17: State diagram for server call.

Note The preceding conceptual data can be implemented by using a variety of techniques. Any data structure that stores the above conceptual data may be used in the implementation.

3.2.3.2 Timers

3.2.3.2.1 Call Fragment Retransmission Timer

This timer MUST be set when a burst of fragments (one or more) is sent to the client. It MUST be canceled when the fragments are acknowledged by the client explicitly via **FACK** or implicitly by **ACK** or a higher-sequence **REQUEST**. When the timer expires, the server SHOULD resend the burst of fragments.<80>

3.2.3.3 Initialization

These extensions make no changes to initialization other than what is specified in section 3.2.3.1.

3.2.3.4 Higher-Layer Triggered Events

3.2.3.4.1 Failure Semantics

A server protocol built on top of these extensions can encounter a failure while executing a method call. It may handle the failure at the application protocol layer, it may expose the failure to the RPC protocol layer, or it may choose application-specific handling not specified in this document.

If it handles the error at the application protocol layer, the interaction appears to be successful from the point of view of the RPC runtime. The [out] parameters are filled, and the RPC implementation on the server sends a response with the stub data (as specified in [C706] section 14.4, NDR Input and Output Streams). In this case, the [out] parameters SHOULD indicate the occurrence of an error, although the exact mechanism for doing so is left to the application protocol layer.

If the server implementation of the application protocol layer exposes the error to the RPC protocol layer, it SHOULD indicate to the RPC runtime (usually through calling an API) that the method call has failed, and, if so, it also SHOULD supply a single unsigned long number that indicates the failure code.

In this case, the server SHOULD send back to the client a fault (as specified in [C706] section 12.5.3.5, The Fault) where the status field of the fault is set to the failure code received from the application protocol layer. The call then enters **STATE_COMPLETE**.<81>

3.2.3.4.2 Retrieving Client Identity

During the authorization process, a higher-level protocol on the server often needs to retrieve the identity of the client making a given request. A server implementation MUST try to retrieve the client identity by executing the following steps in this order:

1. If the **auth_proto** field of the client request is nonzero, the server MUST find the security context indicated by the **key_vers_num** in the sec_trailer of the request, and MUST request that the security provider that created the security context retrieve the client identity. For details on how a security provider determines the client identity, see the documentation for the respective security provider.
2. If the **auth_proto** field of the client request is zero, the server MUST report this to the higher-level protocol in an implementation-specific way.

3.2.3.4.3 Authorization Policy

See the Windows behavior tag.<82>

3.2.3.4.4 Context Handle Generation

If a server stub needs to create a context handle, and the activity of the call has a NULL CAS UUID, the server SHOULD generate a conv_who_are_you2 conversation callback to determine the correct CAS UUID. If the conversation callback fails, the stub SHOULD raise an exception with the status code of the conversation callback.

3.2.3.5 Message Processing Events and Sequencing Rules

The packet semantics are as specified in [C706] sections 6 and 12.

3.2.3.5.1 Failure Semantics

If, during the processing of a method call on the server, the server encounters an error, it SHOULD send back to the client a fault PDU (as specified in [C706] section 12.5.3.5, The Fault PDU) where the status field of the fault PDU is set to a descriptive status code. If the server is unable to send a fault PDU, as specified here, it MUST ignore further packets with the same activity ID and sequence number.

Servers can send any status code in the status field of a fault PDU except the following status codes, which a server MUST NOT send to the client. These status codes have special significance in Windows environments, and their presence in the status field may be flagged as a protocol error by the client:

Status Codes that MUST NOT be sent by RPC servers
ERROR_SUCCESS (0x00000000)

Status Codes that MUST NOT be sent by RPC servers
STATUS_GUARD_PAGE_VIOLATION (0x80000001)
STATUS_DATATYPE_MISALIGNMENT (0x80000002)
STATUS_BREAKPOINT (0x80000003)
STATUS_ACCESS_VIOLATION (0xC0000005)
STATUS_IN_PAGE_ERROR (0xC0000006)
STATUS_ILLEGAL_INSTRUCTION (0xC000001D)
STATUS_PRIVILEGED_INSTRUCTION (0xC0000096)
STATUS_INSTRUCTION_MISALIGNMENT (0xC00000AA)
STATUS_STACK_OVERFLOW (0xC00000FD)
STATUS_POSSIBLE_DEADLOCK (0xC0000194)
STATUS_HANDLE_NOT_CLOSABLE (0xC0000235)
STATUS_STACK_BUFFER_OVERRUN (0xC0000409)
STATUS_ASSERTION_FAILURE (0xC0000420)

3.2.3.5.2 Sequencing in Case of Errors

If a fragmented request with multiple PDUs includes a PDU with an error, implementations of these extensions SHOULD return a fault PDU as soon as they have processed the PDU with the error. They SHOULD NOT wait to receive all PDUs of a fragmented request before sending the fault PDU.

3.2.3.5.3 Packet Processing

Received packets MUST have a valid RPC header, and the packet type MUST be one of the following: REQUEST, PING, FACK, QUIT, or ACK. Other packet types MUST be discarded.

If the PDU's activity ID matches an existing activity on the server, but the PDU's **dc_rpc_cl_pkt_hdr_t.auth_proto** or **sec_trailer_cl.auth_level** fields do not match those in the activity, then the server SHOULD ignore the packet.<83>

Handling of specific packet types follows.

3.2.3.5.4 REQUEST

When a message of type REQUEST is received, the server MUST execute the following steps:

Find or create an activity object for the activity ID in the header. If the activity's lowest-allowed-sequence number is higher than the packet sequence number, the server MUST discard the packet.<84>

If the packet's PF2_UNRELATED flag is false:

- If no active call exists with the packet sequence, the server MUST create a call with that sequence in STATE_INIT, and add it to the activity. The server MUST set the activity's lowest-allowed-sequence to the packet sequence.
- Process the packet according to the call state.

If the packet's PF2_UNRELATED flag is true:

- If the object contains no active call for this sequence, the server MUST create a new call in STATE_INIT, and add it to the activity.
- Process the packet according to the call state.

3.2.3.5.4.1 STATE_INIT

The server MUST clear the send and receive fragment windows. The server MUST set the state to STATE_RECEIVE_FRAGS and continue with processing for that state.

3.2.3.5.4.2 STATE_RECEIVE_FRAGS

- If all request fragments are received and a conversation callback is in progress, the server MUST reply with a WORKING packet. The server MUST discard the packet; no further processing is required.
- If the packet length is greater than the server's PDU limit, the server MUST reply with a FACK. The server MUST discard the packet; no further processing is required.

Otherwise, the server MUST:

1. Update the receive fragment window.
2. If a conversation callback is not in progress, check if a conversation callback is required. If the call is not secure, is non-idempotent, and has an unknown CAS UUID, begin a **conv_who_are_you2**. If the call is secure, and the server does not have a security context matching the key_vers_num in the packet's security trailer, begin a **conv_who_are_you_auth**. If the server has no credentials matching the packet's **auth_proto** field, fail the conversation callback with status 0x000006D3.
    - If the conversation callback fails, send a REJECT to the client, change the call state to STATE_COMPLETE, remove the call from the activity, and update the lowest-allowed-sequence of the activity.
3. If all receive fragments are present, or if the call uses DCE pipes, and the server has received all the [in] arguments that are not marked with the [pipe] attribute in the IDL file, set the call to STATE_WORKING and dispatch to the application stub.

When a call is dispatched:

1. If the call is secure, ask the security provider to verify or decrypt the received packets, as appropriate. If an error occurs, send a REJECT to the client, change the call state to STATE_COMPLETE, remove the call from the activity, and update the lowest-allowed-sequence of the activity. The call is finished.
2. Dispatch to the application stub. Any [out] context handles will be associated with the call's CAS. If no CAS is defined because this is an idempotent call, execute a conv_who_are_you2 call to get the call's CAS. If an error occurs, send a REJECT or FAULT, as appropriate, change the call state

to STATE_COMPLETE, remove the call from the activity, and update the lowest-allowed-sequence of the activity. The call is finished.

3. After the application stub completes successfully, check if a later call sequence has already been dispatched on this connection. If so, skip further processing of this sequence.

4. If the [maybe] flag is set, no reply is needed. Change the call state to STATE_COMPLETE, remove the call from the activity, and update the lowest-allowed-sequence of the activity. The call is finished.

5. Set the call to STATE_SEND_FRAGS, and send one or more response fragments to the client.

3.2.3.5.4.3 STATE_WORKING

If all request fragments are received, the server MUST reply with a WORKING packet. No further processing is required.

If the packet length is greater than the server's PDU limit, the server MUST reply with a FACK. No further processing is required.

If neither of the above conditions are true and the packet's **Header.Flags.Nofack** flag is not set, then the server MUST send a FACK. If the packet's **Header.Flags.Nofack** flag is set, then the server SHOULD NOT send a reply packet.

3.2.3.5.4.4 STATE_SEND_FRAGS

The server MUST send a burst of RESPONSE fragments.

3.2.3.5.5 PING

If the packet sequence is higher than all of the activity's active calls, the server MUST reply with a NOCALL without body data. Otherwise, if the activity contains no active call for the packet sequence, discard the packet.

Otherwise, the packet matches an active call. Since client packets may be duplicated and reordered in transit, the server MAY ignore the packet using implementation-specific criteria in order to avoid redundant responses.<85> If not, the server MUST check the call state, as specified in the sections that follow.

3.2.3.5.5.1 STATE_INIT

The server MUST reply with **NOCALL-with-body**.

3.2.3.5.5.2 STATE_RECEIVE_FRAGS

If all request fragments for the call have been received, the server MUST reply with WORKING. Otherwise, the server MUST reply with **NOCALL-with-body**.

3.2.3.5.5.3 STATE_WORKING

The server MUST reply with WORKING.

3.2.3.5.5.4 STATE_SEND_FRAGS

The server MUST send a burst of RESPONSE fragments.

3.2.3.5.6 FACK

If the call state is not STATE_SEND_FRAGS, then discard the packet. Otherwise, update the send fragment window and send a burst of RESPONSE fragments.

3.2.3.5.7 QUIT

If the cancel event ID is new, cancel the current call and send a QUACK.

If the event ID is the current one, reply with a QUACK.

If the event ID is older than the current one, discard the packet.

3.2.3.5.8 ACK

if the call state is not STATE_SEND_FRAGS, discard the packet. Otherwise, change the call state to STATE_COMPLETE, remove the call from activity, and update the lowest-allowed-sequence of the activity. The call is finished.

3.2.3.6 Timer Events

For more information on timers, see section 3.2.3.2.

3.2.3.7 Other Local Events

No local events are specified for implementations of connectionless RPC servers.

3.3 Connection-Oriented RPC Protocol Details

3.3.1 Common Details

This section defines the protocol details that are common between a connection-oriented RPC server and a connection-oriented RPC client.

3.3.1.1 Abstract Data Model

This section specifies a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The specified 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.

3.3.1.1.1 Association

An association is a set of RPC transport connections between a client process and a server endpoint. On the abstract level, the association can have any number of connections in it, although memory constraints and limitations of the RPC transport that establishes these connections mean that, in practice, the number of connections in an association is much more limited. All RPC transport connections in a given association are explicitly joined to an association, as specified in section 3.3.1.5.8. Both the client and server have an abstraction for association.

As specified, [C706] uses the phrase association group for what this specification refers to as an association.

3.3.1.1.2 Context Handle Scope

The operations on a context handle are as specified in [C706] section 5.1.6, Context Handle Rundown. This section clarifies the scope of the context handle as interpreted by these extensions. As specified in [C706] section 5.1.6, Context Handle Rundown, the context handle is created by the client sending a null context handle in a method call, and by the server returning a non-null context handle in the stub data in the response to the same method call. The RPC transport connection on which the request and response are transmitted belongs to an association, as specified in sections 3.3.1.1.1 and 3.3.1.1.3. The scope of a context handle is this association. If a request/response exchange on one association leads to the creation of a context handle, and this context handle is passed to a different association, the server SHOULD reject the request.

3.3.1.1.3 Connection

A connection is an RPC level abstraction that denotes the data structures associated with a given RPC transport connection. There is a 1:1 relationship between an RPC transport connection and an RPC connection. For example, the RPC runtime on both the client and server has a data structure for each TCP/IP connection if the RPC transport is TCP/IP. Each connection MUST belong to exactly one association. Once a connection is tied to an association, a connection cannot change the association that it belongs to.

As specified, [C706] uses the term association for what this document refers to as a connection.

Note The above conceptual data can be implemented by using a variety of techniques. An implementation is at liberty to implement such data in any way it pleases.

3.3.1.2 Timers

There are no timers that are common between a connection-oriented client and a connection-oriented server.

3.3.1.3 Initialization

There is no initialization that is common between a connection-oriented client and a connection-oriented server.

3.3.1.4 Higher-Layer Triggered Events

There are no higher-layer triggered events that are common between a connection-oriented client and a connection-oriented server.

3.3.1.5 Message Processing Events and Sequencing Rules

3.3.1.5.1 Protocol Version Number

These extensions constrain the protocol version numbers that are used in PDUs, as specified in [C706] section 12, RPC PDU Encodings. These extensions recognize only major version 5 and minor version 0. If a PDU with a different major or minor version is sent to a client or server, the client or server SHOULD return an error.<86>

3.3.1.5.2 Building and Using a Security Context

3.3.1.5.2.1 Building a Security Context

To make a secure call, a security context needs to be created before it can be used. The process of creation involves exchanging one or more messages between the client and server implementations of a security provider and is also called building a security context.

During the process of building a security context, a security provider may optionally exchange messages with an entity other than the client or server (for example, a KDC).

The scope of a built security context is the connection. If a client wants to use a security context on a different connection, it MUST totally rebuild it for that different connection.

To build a security context, an RPC client and an RPC server exchange a series of bind/bind_ack or alter_context/alter_context_resp PDUs with authentication information. The process MUST start on the client, as follows.

- If the client has already sent a bind PDU on the connection it wants to build the security context on, it MUST start the sequence of building a security context with an alter_context PDU.
- If the client has not already sent a bind PDU on that connection, it MUST start the sequence of building a security context with a bind PDU.

The process continues on the server as follows:

- If the server receives a bind PDU, it MUST respond with a bind_ack or bind_nak PDU.
- If a server receives an alter_context PDU, it MUST respond with an alter_context_resp PDU or, in the case of error, with a fault PDU.

In case of catastrophic errors, a server MAY send a fault PDU or just close the connection. For information on client and server state machines, see sections 3.3.2 and 3.3.3.

Once a client decides on the type of PDU, it MUST start the sequence by requesting the security provider for an authentication token using an implementation-specific equivalent of the abstract GSS_Init_sec_context call, as specified in [RFC2743]. This PDU MUST be sent to the server with authentication information added, as specified in section 2.2.2.11.

The client MUST choose a value for the **auth_context_id** of the sec_trailer structure such that it is unique within the scope of the given connection. Each message with an authentication token sent to the other party is also called a security leg. Thus, the first message from the client to the server is also called the first leg of the security context creation. The server MUST retrieve the authentication token and hand it off to the security provider indicated by the **auth_type** field.

The interaction between these extensions and the security provider on the server MUST happen through an implementation-specific equivalent of the abstract GSS_Accept_sec_context call, as specified in [RFC2743]. Upon receiving and processing an authentication token at any leg of the authentication on either the client or server, the security provider MUST indicate to RPC runtime one of three abstract results from the processing: an error, a success, or a request for further security legs, as specified in [RFC2743].

- If the security provider indicates an error, the RPC runtime MUST take recovery action depending on if this is the client or server.
  - If this is the client, the RPC runtime discards the security context and MUST NOT send any further PDUs on that connection. It SHOULD close the connection unless it is expecting

responses on a multiplexed connection, as specified in section 3.3.1.5.9, in which case it SHOULD wait for all responses to arrive before closing the connection. If it does not wait for all responses on a multiplexed connection, it MUST provide indication in an implementation-specific way to upper layers that the outstanding calls have failed.

  - If the security provider returns an error on the server, the server MUST respond with a bind_nak or a fault PDU, depending on the PDU that the client sent, as specified above. The server SHOULD also discard the security context in this case.

- If the security provider returns a success from processing the authentication token, the security context is successfully created. If the security provider returns a success on the client, the client is ready to use this security context. If the security provider on the server returns a success, the server MUST still respond with a bind_ack or alter_context_resp PDU, as specified above. In this case, it SHOULD return an empty (zero-length) authentication token to the client.

- If the security provider indicates to the RPC runtime a request for further security legs, it MUST always produce another authentication token along with the request for further security legs. In this case, the RPC runtime MUST send another leg of the security context creation by using that authentication token. If this happens on the client, the client MUST send an alter_context PDU. The p_context_elem structure of the alter_context PDU SHOULD be the same as the content of the PDU sent in the previous leg from the client. If this happens on the server, it MUST respond with a bind_ack or an alter_context_resp PDU, except when a security provider has an odd number of legs as specified in the following section, using the authentication token produced by the security provider.

If a client has prior knowledge of how many legs different security providers use, and it knows that a given security provider has an odd number of legs, the client SHOULD use an **rpc_auth_3 PDU** instead of an alter_context PDU for the last leg. The client MUST NOT use an **rpc_auth_3 PDU** unless it is certain that the security provider on the server will indicate to the RPC runtime success or error. The server MUST NOT respond to an **rpc_auth_3 PDU**. If the processing of the authentication token from an **rpc_auth_3 PDU** results in an error, the RPC runtime on the server SHOULD remember the error in the security context and return a fault PDU on the first request that uses this security context with the status field set to the error it remembered.

If a client is not sure how many legs a given security provider uses, it MUST assume that the number of legs is even.<87>

Once negotiated, a security context SHOULD be maintained by both the client and server implementations for the lifetime of the connection it was negotiated on.

3.3.1.5.2.2 Using a Security Context

After a security context is built, the security context can be used by the RPC runtime and higher-level protocols to perform authorization decisions. Besides using the security context for authorization decisions, the RPC runtime can also use the security context to create a logical stream of data that is protected from tampering and information disclosure on the network.

The amount of protection applied depends on the authentication level for the security context requested by the client when the security context is created. The authentication level is applied in two dimensions:

- In the first dimension, it controls what capabilities the RPC runtime MUST request from the security provider when the security context is being built, as detailed in the first table below. It is possible for a security provider to not be able to provide a certain capability. In this case, the lack of the capability MUST be considered by the RPC runtime as equivalent to the security provider returning an error, and MUST be handled as specified in the previous section.

- In the second dimension, the authentication level controls how the security provider runtime MUST perform PDU protection on the different PDU segments using the security context, as detailed in the second table below.

The following table specifies the abstract capability that the RPC runtime MUST request from the security provider when the security context is being created. The capabilities below are further specified in [RFC2743] section 1.2.1.2. The capabilities requested at each level include the ones requested at the previous level.

Authentication level	Capability requested
RPC_C_AUTHN_LEVEL_CONNECT	None
RPC_C_AUTHN_LEVEL_CALL	Replay Detect
RPC_C_AUTHN_LEVEL_PKT	Replay Detect
RPC_C_AUTHN_LEVEL_INTEGRITY	Sequence Detect, integrity
RPC_C_AUTHN_LEVEL_PRIVACY	Confidentiality

As specified above, once the security context is built, the RPC runtime MUST also use the authentication level to control how the security context is used to protect request and response PDUs sent to the other side.

One of the first decisions that need to be negotiated is if the security provider on each side supports what this document calls header signing. Header signing is an operation where a security provider can provide integrity protection to a segment of the PDU such that the integrity protection does not modify the content of that segment. The segments of the PDU are specified in section 2.2.2.1. The RPC runtime on the client determines in an implementation-specific way if the security provider on the client supports header signing. If it does, the first bind or alter_context PDU that the client sends on a connection that carries authentication information and whose authentication level is integrity or higher MUST have its PFC_SUPPORT_HEADER_SIGN bit set. The RPC runtime on the server also determines in an implementation-specific way if the security provider on the server supports header signing, and, if it does not, it MUST respond to the client with a PDU whose PFC_SUPPORT_HEADER_SIGN bit is cleared. If it does support header signing, it MUST respond to the client with a PDU whose PFC_SUPPORT_HEADER_SIGN bit is set.

Using this mechanism, the client and server agree if header signing should be done for this connection. Once agreed, the client and server apply protection to request and response PDUs in the same way.

If the client and server support header signing, the party that sends the PDU asks the security provider to apply the following protection to the different PDU segments.

Authentication level	PDU header	PDU body	sec_trailer
RPC_C_AUTHN_LEVEL_CONNECT	None	None	None
RPC_C_AUTHN_LEVEL_CALL	None	None	None
RPC_C_AUTHN_LEVEL_PKT	None	None	None
RPC_C_AUTHN_LEVEL_INTEGRITY	Integrity	Integrity	Integrity
RPC_C_AUTHN_LEVEL_PRIVACY	Integrity	Confidentiality	Integrity

If either the client or server does not support header signing, the RPC runtime on the sending side asks the security provider to apply the following protection to the different PDU segments.

Authentication level	PDU header	PDU body	sec_trailer
RPC_C_AUTHN_LEVEL_CONNECT	None	None	None
RPC_C_AUTHN_LEVEL_CALL	None	None	None
RPC_C_AUTHN_LEVEL_PKT	None	None	None
RPC_C_AUTHN_LEVEL_INTEGRITY	None	Integrity	None
RPC_C_AUTHN_LEVEL_PRIVACY	None	Confidentiality	None

In the tables above, "None" means no protection, "Integrity" means an integrity check per [RFC2743] section 2.3.1 MUST be applied, and "Confidentiality" means that the segment MUST be encrypted.

The PDU Header, PDU Body and sec_trailer MUST be passed in the input message, in this order, to GSS_WrapEx, GSS_UnwrapEx, GSS_GetMICEx and GSS_VerifyMICEx. For integrity protection the sign flag for that PDU segment MUST be set to TRUE, else it MUST be set to FALSE. For confidentiality protection, the conf_req_state flag for that PDU segment MUST be set to TRUE, else it MUST be set to FALSE.

The PDU Header, PDU Body and sec_trailer from the output message of GSS_WrapEx and GSS_VerifyMICEx MUST be sent to the other side (client or server) as part of the request or response PDU and the signature output MUST be sent to the other side (client or server) as the authentication token as specified in section 2.2.2.12.

If the authentication level is RPC_C_AUTHN_LEVEL_PRIVACY, the PDU Body will be encrypted. The PDU Body from the output message of GSS_UnwrapEx represents the plain text version of the PDU Body. The PDU Header and sec_trailer output from the output message SHOULD be ignored. Similarly the signature output SHOULD be ignored.

For further details on GSS_WrapEx, see [MS-NLMP] section 3.4.9 and [MS-KILE] section 3.4.5.4.

For details on GSS_UnwrapEx, see [MS-NLMP] section 3.4.10 and [MS-KILE] section 3.4.5.5.

For further details on GSS_GetMICEx, see [MS-NLMP] section 3.4.11 and [MS-KILE] section 3.4.5.6.

For further details on GSS_VerifyMICEx, see [MS-NLMP] section 3.4.12 and [MS-KILE] section 3.4.5.7.

If the authentication level is connect, the security provider MUST use for request and response PDUs an authentication token that is optional and that does not need to be transmitted to the other side.

This protocol does not specify whether the authentication token itself is protected from tampering by the security provider. It also does not specify how the security provider applies integrity or confidentiality protection to a PDU segment. The algorithms for doing so are specific to the security provider. For details about a security provider, see the documentation for that security provider.

3.3.1.5.3 Bind Time Feature Negotiation

These extensions introduce additional rules about how a bind PDU SHOULD be composed by the client and processed by the server, and how the response bind_ack PDU SHOULD be composed by the server and processed by the client. [C706] sections 12.6.4.3 and 12.6.4.4 specify a bind PDU

and a bind_ack PDU. When sending a bind PDU, a client MAY add an element in the **p_cont_elem** array that has the same value for the **abstract_syntax** field as the previous element in the **p_cont_elem** array, but that MUST have exactly one element in the **transfer_syntax** array; also, its **if_uuid** field MUST have the following prefix: 6CB71C2C-9812-4540 and a version number of 1.0. If a client does so, it is said to have indicated support for bind time feature negotiation. A client MUST have, at most, one element in the **p_cont_elem** array that has an **if_uuid** with that prefix in the **transfer_syntax** array. If a client has indicated support for bind time feature negotiation, the message processing rule in this section SHOULD be applied by the server implementation to all messages for this connection. If a client has not indicated support for bind time feature negotiation, the message processing rules in this section do not apply to this connection.

If a client has indicated support for bind time feature negotiation, the eight octets immediately after the prefix are interpreted as BindTimeFeatureNegotiationBitmask, as specified in section 2.2.2.14. If the **SecurityContextMultiplexingSupported** bit is set, this means the client supports security context multiplexing, as specified in section 3.3.1.5.4. If the KeepConnectionOnOrphan bit is set, this means the client supports keeping the connection open after an orphaned PDU is sent, as specified in section 3.3.1.5.11.

If the server supports bind time feature negotiation, it MUST reply with the result field in the p_result_t structure of the bind_ack PDU equal to negotiate_ack, and it MUST use the reason field of the p_result_t structure as a BindTimeFeatureNegotiationResponseBitmask structure. If a client has set the **SecurityContextMultiplexingSupported** bit in the BindTimeFeatureNegotiationResponseBitmask structure, and the server supports security context multiplexing, the server SHOULD set the **SecurityContextMultiplexingSupported** bit of the BindTimeFeatureNegotiationResponseBitmask structure.

If the server does not support security context multiplexing, it MUST clear the **SecurityContextMultiplexingSupported** bit of the BindTimeFeatureNegotiationResponseBitmask structure. If the **SecurityContextMultiplexingSupported** bit in the BindTimeFeatureNegotiationResponseBitmask structure is set, and if the client supports security context multiplexing, it MAY start using security context multiplexing on this connection, as specified in section 3.3.1.5.4.<88>

If a client has set the KeepConnectionOnOrphanSupported bit in the BindTimeFeatureNegotiationBitmask structure and the server supports keeping the connection open after an orphan PDU is received, the server SHOULD set the KeepConnectionOnOrphanSupported bit in the BindTimeFeatureNegotiationResponseBitmask structure.

If the server does not support keeping the connection open after an orphan PDU is received, it MUST clear the KeepConnectionOnOrphanSupported bit in the BindTimeFeatureNegotiationResponseBitmask. If the KeepConnectionOnOrphanSupported bit in the BindTimeFeatureNegotiationResponseBitmask is set and client supports keeping the connection open after an orphan PDU is sent, the client MAY start keeping the connection open after sending an orphan PDU on the connection specified in Keeping connections open after client sends an orphan PDU as specified in section 3.3.1.5.11.<89>

For future extensibility, these rules MUST be applied by the server and the client to all reserved bits in the BindTimeFeatureNegotiationResponseBitmask and BindTimeFeatureNegotiationResponseBitmask structures:

- If a client supports a given feature, it MUST set the bit (or set of bits) associated with this feature.
- If a bit (or set of bits) used to communicate that a client supports a given feature is not set, the server MUST assume that the client does not support this feature.

- If a server does support the feature, it MUST set the bits associated with that feature in the BindTimeFeatureNegotiationResponseBitmask bitmask.
- A server MUST clear all bits in the BindTimeFeatureNegotiationResponseBitmask bitmask that it interprets are reserved.<90>

3.3.1.5.4 Security Context Multiplexing

These extensions allow for a client implementation to use more than one security context per connection. A client implementation MUST NOT do security context multiplexing unless it has negotiated this capability with the server, as specified in section 3.3.1.5.3. When security context multiplexing has been negotiated, if a client needs to negotiate a new security context, it is allowed to do so on an existing connection subject to the constraints in the server state machine. These extensions also introduce some constraints and conventions along with this capability. If there is only one security context on a given connection, and this security context has the authentication level connect, a client and a server MAY choose not to send authentication information for that security context. In such a case, the server MUST treat request PDUs without authentication information as if they had Connect level authentication information, and all other security context attributes are picked from the only security context negotiated on the connection.<91>

A client MUST send authentication information for all request PDUs if the higher-level protocol on the client has asked for the connect authentication level, and there is more than one security context negotiated for the connection.

A client MUST NOT build more than 2,000 security contexts per connection, but it MAY choose to impose an even lower limit on the number of security contexts that can be built on a connection.<92>

3.3.1.5.5 Connection Affinities

All PDUs related to a given method call MUST be sent on the same connection. This includes request, response, fault, cancel, and orphaned PDUs.

3.3.1.5.6 Primary and Secondary Endpoint Address

As specified, [C706] section 9.3.3.2, Primary and Secondary Endpoint Addresses, allows a server to have a primary and secondary endpoint address. These extensions recognize the syntactic rules associated with a primary and secondary endpoint address, but they discard all semantic meaning of a primary and secondary endpoint address. Servers that implement these extensions SHOULD return a secondary endpoint address that is the same as the primary endpoint address. Clients that implement these extensions SHOULD ignore the secondary endpoint address.

3.3.1.5.7 Presentation Context and Transfer Syntax Negotiation

These extensions extend and augment the message processing rules for presentation context and transfer syntax negotiation, as specified in [C706] section 12.6, Connection-Oriented RPC PDUs. The scope of a presentation context in these extensions is a connection.

The basic model for the negotiation process is that the client enumerates all transfer syntaxes it supports, and the server chooses one of them. A detailed description of the processing rules follows.

If a client supports multiple transfer syntaxes, the client SHOULD send multiple elements in the **p_cont_elem** array of the p_cont_elem_t structure, as specified in [C706] section 12, RPC PDUs Encodings. The **abstract_syntax** field in each element of the array SHOULD contain the same **if_uuid** and **if_version**, and the **transfer_syntax** array of each element SHOULD have 1 element

only. The **if_uuid** and **if_version** of the element in the **transfer_syntax** array MUST contain the **transfer_syntax** UUID and version number for the **transfer_syntax** the client is proposing.

The server responds with a bind_ack or alter_context_resp PDU depending on what PDU the client sent to it. The server SHOULD accept, at most, one of the transfer syntaxes. The response of the server is a p_result_list_t structure that MUST have the same number of elements as the p_cont_elem_t structure the client sent to it. Each array element in the p_result_list_t structure is interpreted to correspond to the array element in the p_cont_elem_t structure in the same position of the array. For example, the first array element in the p_result_list_t structure is interpreted to correspond to the first array element in the p_cont_elem_t structure. If the server does not recognize the **abstract_syntax** field in an array element in the p_cont_elem_t structure, it MUST set the result field in the p_result_list_t structure corresponding to that array element to **abstract_syntax_not_supported**. If the server recognizes the **abstract_syntax** field, the server MUST set the result field corresponding to the transfer syntax it prefers to use to the 'acceptance' value and the result field corresponding to all other transfer syntaxes to the 'provider_rejection' value. Both of these values are as specified in [C706] section 12.6, Connection-Oriented RPC PDUs.

The client SHOULD NOT interpret the rejection of a transfer syntax as indication that the server will not accept this transfer syntax at a future date, but instead SHOULD interpret the rejection as indication that the server prefers the transfer syntax it accepted over the other transfer syntaxes proposed by the client. A client is allowed to propose a rejected transfer syntax at a later time, but if it has a choice, the client SHOULD use the transfer syntax that the server accepted instead of trying to renegotiate a transfer syntax that was rejected earlier by the server.

Once negotiated, a presentation context SHOULD be maintained by both the client and server implementations for the lifetime of the connection it was negotiated on.

Servers SHOULD implement at least transfer syntax NDR, as defined in this document, to allow for a fallback transfer syntax if another transfer syntax cannot be negotiated.<93>

3.3.1.5.8 Adding a New RPC Transport Connection to an Association

The **assoc_group_id** field in the bind PDU is as specified in [C706] section 12.6.4.3. These extensions add some constraints to the protocol specified in [C706]. If a new connection tries to join an existing association by setting the **assoc_group_id** field to the value of an existing association, the server SHOULD establish from the RPC transport whether the connection comes from the same machine as the connection that created the association. If yes, it MUST allow the connection to join the association. If no, it SHOULD not allow the connection to join the association. Some RPC transports are not capable of determining this conclusively. In this case, the server SHOULD allow the connection to join the association.

3.3.1.5.9 Multiplexed Connections

A client can indicate to the server that it wants to do concurrent multiplexing on a connection. It does that by setting the PFC_CONC_MPX bit, as specified in [C706] section 12, RPC PDU Encodings. If the server also supports this capability, it responds with a PDU that also has the same bit set. Once concurrent multiplexing on a connection is negotiated, a client is allowed to send another request on a connection before it receives a response on a previous request, provided the server is in Context Negotiated or Dispatched state. A client still MUST send all request PDUs for a fragmented request before it can move on to the next. Each request on the connection MUST abide by the same rule.

If a client negotiates a connection that does not support concurrent multiplexing (also called an exclusive connection), a client MUST wait for all PDUs of a response to arrive before it can send a request PDU for the next call.<94>

3.3.1.5.10 Handling of Callbacks

Method calls declared as callbacks have some additional rules for handling on the network compared to calls without this attribute. A callback on the network is represented as a regular RPC except that the direction of the PDUs is reversed. The server sends one or more request PDUs, and the client responds with one or more response PDUs. The server MUST NOT send request PDUs while in any state other than the Dispatch state, and a client SHOULD NOT accept callbacks in any state other than the Wait For Response state. Callbacks are allowed to recourse to any level that the implementation is willing to support. That is, if a client gets callback, it MAY initiate another RPC method call by sending more request PDUs instead of replying to the previous request. This same rule applies for the server. For the server, if it sends one or more request PDUs during the Dispatch state, the call that is in the Dispatch state is called a nesting or outer call. The callback call that the request PDUs sent from the server is called nested or inner call. For the client, if it sends one or more request PDUs during the Wait For Response state, the call that is in the Wait For Response state is called a nesting or outer call. The callback call that the request PDUs sent from the client is called nested or inner call. Callback calls by definition use the presentation and security context of the nesting call, and MUST NOT send bind or alter_context PDUs.

The **call_id** field for all request and response PDUs of a nested callback MUST be the same as the **call_id** of the request/response PDUs of the nesting callback.

3.3.1.5.11 Keeping Connections Open After Client Sends an Orphaned PDU

A client implementation MUST NOT keep the connection open after sending the orphaned PDU unless it has negotiated this capability with the server, as specified in section 3.3.1.5.3.<95>

3.3.1.6 Timer Events

There are no timer events that are common between a connection-oriented client and a connection-oriented server.

3.3.1.7 Other Local Events

There are no other local events that are common between a connection-oriented client and a connection-oriented server.

3.3.2 Client Details

The following diagram defines the client state machine:

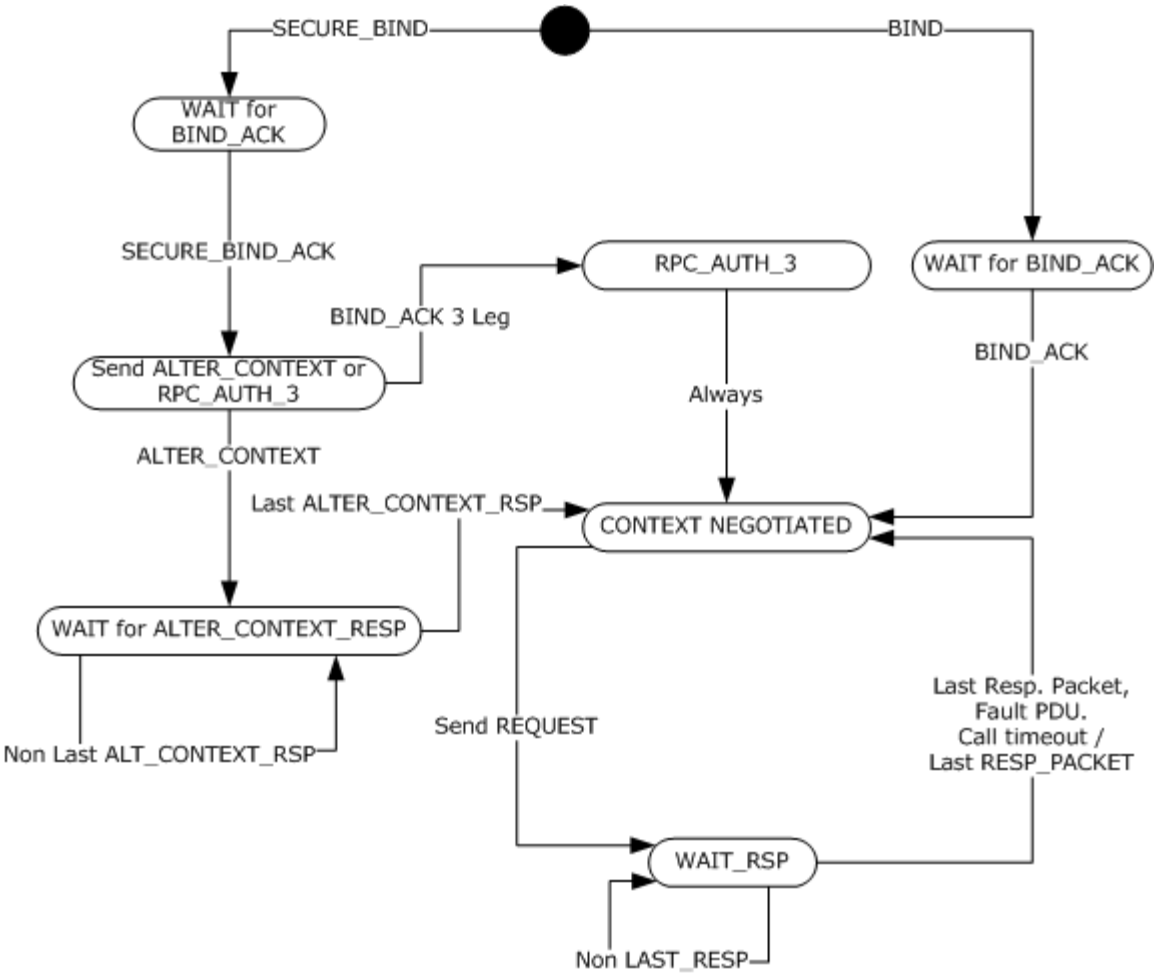


Figure 18: Client state machine

State	Description
WAIT for BIND_ACK	The client has sent the bind PDU, in case of an un-secure call.
Send ALTER_CONTEXT or RPC_AUTH_3	Send an ALTER_CONTEXT or RPC_AUTH_3
CONTEXT_NEGOTIATED	The client is ready to send the request PDU.
WAIT for ACK	The client has sent a bind PDU, in case of a secure call.
WAIT for ALTER_CONTEXT_RESPONSE	The client has sent an alter context PDU.
RPC_AUTH_3	The client has sent an rpc_auth_3 PDU.
WAIT_RSP	The client is waiting for a response PDU.

Notes on this state machine:

When a state does not show an error transition, these extensions handle the error from this state by closing the connection.

When concurrent multiplexing is used on a connection, as soon as an independent logical thread of execution makes a transition from CONTEXT NEGOTIATED to WAIT_RSP state, another independent logical thread of execution can make the transition from CONTEXT_NEGOTIATED to WAIT_RSP. Only one logical thread of execution is allowed to make this transition at a given time, but multiple logical threads of execution can be in the WAIT_RSP state. A client MUST NOT send any request PDU for request N+1 before it sends all request PDUs for request N.

If concurrent multiplexing on a connection is not enabled, a client MUST NOT send any request PDU for request N+1 before it receives all the response PDUs for request N.

3.3.2.1 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.

Note The above conceptual data can be implemented by using a variety of techniques. An implementation is at liberty to implement such data in any way it pleases.

3.3.2.2 Timers

3.3.2.2.1 Connection Timeout

Whenever a method call is pending, a client MAY instruct the RPC transport to monitor the state of the connection above and beyond the monitoring provided by default by the RPC transport, so that if the server crashes or loses network connectivity to the client, the client can take recovery action. A method is considered pending on the server from a client perspective if all fragments of request have been sent and no replies have started arriving. Depending on the protocol sequence for the method call, the establishment of the timer acts only as advice to the RPC runtime system.<96>

When this timer expires, the expiry is not noticed at the RPC protocol level but is noticed at the TCP/IP protocol level, and shows as a local event, as specified in section 3.3.2.7.1.

3.3.2.2.2 Call Timeout

The RPC runtime on the client allows a higher-level protocol to instruct it to set up a timer that expires if the send of a PDU has not completed within the prescribed time interval, or no response PDUs are received within the prescribed time interval after all request PDUs are sent. The time interval for this timer is supplied to the RPC runtime by a higher-level protocol. If a bind time out and a call time out are applicable during the same method call, the lower of the two time-out values MUST be used.

3.3.2.2.3 Idle Connection Cleanup Timeout

An idle connection is called a connection on which all the response PDUs to all outstanding calls have been received. To reduce load on the server, a client SHOULD close connections to the server that have been idle for some time. A client MUST NOT close all connections from a given association to a server if there are one or more active context handles on that association. A timer is started when a connection becomes idle and is stopped when a request PDU is sent on it. The exact value of this timer is implementation dependent.<97>

3.3.2.2.4 Bind Timeout

After sending a bind PDU, the RPC runtime on the client SHOULD set up a 15-minute timer. If both a call and a bind timeout are present, the client SHOULD pick the lower of the two. The timer is stopped when a bind_ack PDU is received.<98>

3.3.2.3 Initialization

A client is initialized when a higher-level protocol supplies to the client-side implementation of the RPC runtime sufficient information to start making RPCs. The distinct steps that follow may be taken by the higher-level protocol.

3.3.2.3.1 Create a Binding Handle

The information needed to create a binding handle is as specified in [C706] section 2, Introduction to the RPC API.

3.3.2.3.2 Specify Security Settings

If a higher-level protocol wants to use security for its remote procedure method calls, it MUST supply to the client-side implementation of the RPC runtime information on:<99>

- What security provider it wants to use.
- What authentication level it wants to use.
- Any other security provider-specific information necessary for the security provider to function.

3.3.2.4 Higher-Layer Triggered Events

3.3.2.4.1 Make a Remote Procedure Method Call

When a higher-level protocol on the client makes a remote procedure method call, the client makes a number of choices that determine what actions are triggered.

3.3.2.4.1.1 Resolve the Binding Handle

As a first step, a client MUST ensure that the binding handle is a fully bound binding handle, and, if not, it MUST resolve it. In this stage, these extensions conform to those specified in [C706] section 6.2.2, Endpoints, and the Endpoint Mapper. This document also refers to a fully bound binding handle as a resolved binding handle.

3.3.2.4.1.2 Find an Association and a Connection

Once a binding handle is fully bound, the client MUST find or create an association for this call. If an association cannot be found, the client MUST attempt to create a new one, as specified in [C706] section 9.3, Connection-Oriented Protocol. A client is free to use any association to the same server endpoint that it knows of, provided that the constraints as specified in section 3.3.1.1.2 are kept. Within an existing association, a client can choose to use an existing connection or create a new connection. A client is free to use any connection that meets the requirements specified in this document. For any two causally ordered calls N and N+1, a client MUST choose the same connection for N+1 that it chose for N.

3.3.2.4.1.3 Build Security/Presentation Context

A client cannot execute a remote procedure method call on a connection if there is no presentation context for the interface and transfer syntaxes used by the call. If such a presentation context already exists, the client can use it. If not, the client follows the steps specified in section 3.3.1.5.7 and in [C706] sections 9, 11, and 12 to create a presentation context.

If the remote procedure method call uses security, the client MUST attempt to find or create a security context for that call. The steps to create a security context are specified in section 3.3.1.5.2.<100>

The client SHOULD try to reuse existing presentation contexts and security contexts that are present on the connection. If the client needs to negotiate both a new presentation context and a new security context on the connection, the client also SHOULD do so with a single exchange of bind/bind_ack or alter_context/alter_context_resp, which MAY take multiple PDUs, where the PDUs carry both information necessary for building the security context and information necessary for building the presentation context.

3.3.2.5 Message Processing Events and Sequencing Rules

3.3.2.5.1 rpc_fault PDU Processing Rules

If a client receives an rpc_fault PDU where the status field is one of the error codes as specified in section 3.3.3.5.1, it SHOULD treat this as a protocol error, and SHOULD return an error code to the client application indicative of a protocol error.<101>

3.3.2.5.2 Handling Responses

If connection concurrent multiplexing is used, a client may receive response PDUs for many requests concurrently. The client MUST use the **call_id** field of the response PDU to determine what response belongs to what remote procedure method call.

An implementation of these extensions on the client SHOULD enforce a limit on the alloc_hint it receives in the response PDU to be no more than 2^{31}-1.

3.3.2.6 Timer Events

3.3.2.6.1 Call Timeout

If this timer expires, the call for which it was set up MUST be considered canceled. The definition of a canceled call is implementation-specific.

3.3.2.6.2 Idle Connection Cleanup Timeout

If this timer expires, the connection for which it was set up MUST be closed.

3.3.2.6.3 Bind Timeout

If this timer expires, the call for which it was set up MUST be considered canceled. The definition of a canceled call is implementation-specific.

3.3.2.6.4 Endpoint Mapper Requests Security Information

As specified, [C706] does not make it explicit what security information needs to be applied to requests from the client to the endpoint mapper to resolve the endpoint for an interface. These

extensions prescribe that clients MAY use security for making requests to the endpoint mapper. If they do, the authentication type SHOULD be the same as the authentication type for the partial binding handle that the client is trying to resolve and SHOULD have the integrity authentication level.

If a security provider uses an authentication type that is not specified here, and this authentication type requires other parameters for the authentication, an implementation SHOULD choose values for these parameters that maximize interoperability while making the endpoint mapper requests safe from tampering when in transit on the network.<102>

3.3.2.7 Other Local Events

3.3.2.7.1 Transport Connection Timeout

This event is triggered when the RPC transport indicates to RPC that the connection has timed out. Different RPC transports interpret this differently. For information on how a given RPC transport times out connections, see the documentation for the respective transport. When this event occurs on a connection, all security and presentation contexts are considered invalid. All calls that are in progress on this connection are considered failed, and an implementation-specific error is returned to the higher-layer protocol. A call is considered in progress on a connection if at least one PDU has been sent for that call, and not all PDUs from the server have been received for that call.

3.3.3 Server Details

The following diagram illustrates the state machine for an RPC connection:

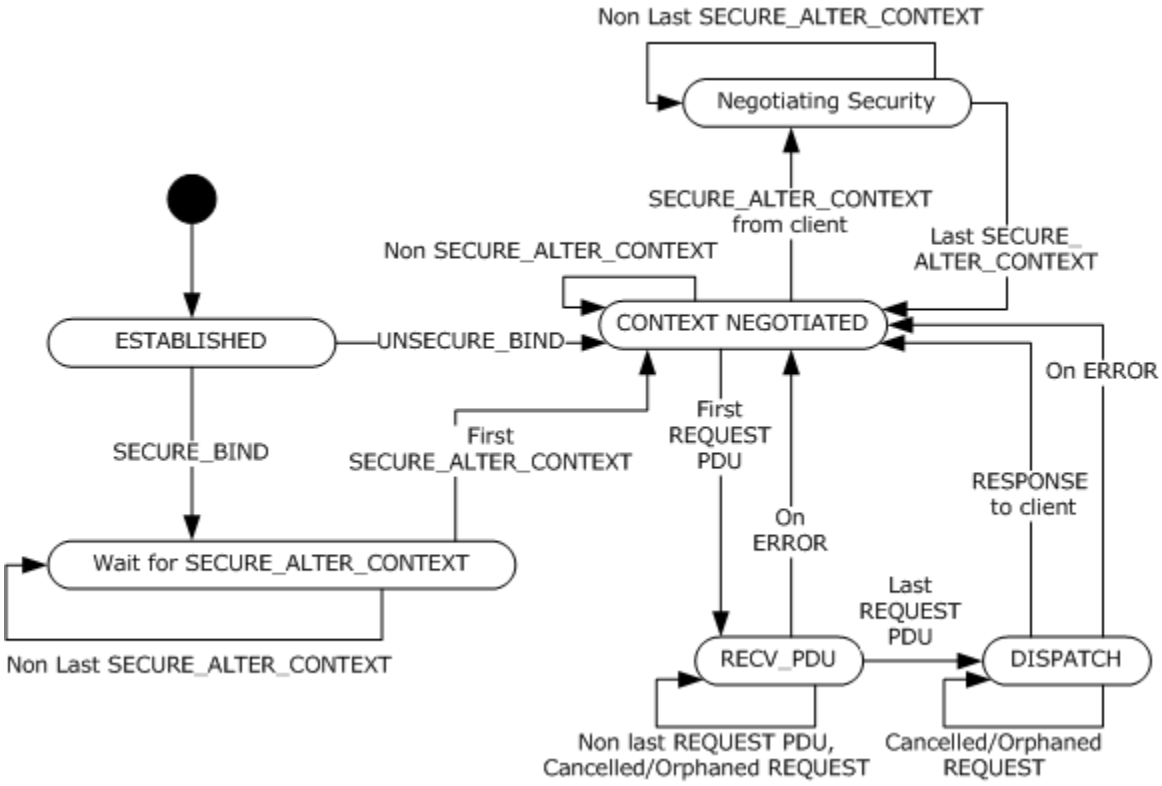


Figure 19: State machine for an RPC connection

Notes on this state machine:

When a state does not show an error transition, these extensions handle errors from this state by closing the connection or sending a bind_nak/fault PDU, as specified in sections 3.3.1.5.2, 3.3.1.5.7, and 3.3.3.5.7.

When concurrent multiplexing is used on a connection, as soon as an independent logical thread of execution makes a transition from RECV_PDU state to DISPATCH, another independent logical thread of execution can make the transition from CONTEXT_NEGOTIATED to RECV_PDU. Only one logical thread of execution is allowed to reside in the RECV_PDU state, but multiple logical threads of execution can be in the DISPATCH state. A client MUST NOT send any request PDU for request N+1 before it sends all request PDUs for request N.

If concurrent multiplexing on a connection is not enabled, a client MUST NOT send any request PDU for request N+1 before it receives all the response PDUs for request N.

3.3.3.1 Abstract Data Model

This section specifies 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.

3.3.3.1.1 Table of Security Providers

A server implementation MUST maintain an abstraction of a table of security providers indexed by authentication type. The table must have fields for authentication type and principal name.

Higher-level protocols indicate to the RPC runtime when to add rows to the table, when to delete rows from the table, and when to modify fields in the table by using implementation-specific APIs.

Many PDUs that arrive at a server have a field that selects an authentication type (also called security provider). These extensions MUST use the authentication type in the PDU as a selector in the table of security providers to route the PDU for processing to the correct security provider.

3.3.3.1.2 Table of Security Contexts

The server implementation MUST implement an abstraction of a table indexed by the security context ID (which is the same as the value of the **auth_context_id** field in the sec_trailer data structure). An incoming request PDU with a given security context ID MUST be routed to the security context retrieved from the table row with the same security context ID. A new row is added to the table when a new security context is built.

3.3.3.1.3 Table of Presentation Contexts

The server implementation MUST implement an abstraction of a table indexed by the presentation context ID (which is the same as the value of the **p_cont_id** field in the request PDU header, as specified in [C706] section 12.6.4.9, The Request PDU). An incoming request PDU with a given presentation context ID MUST be routed to the interface retrieved from the table row with the same presentation context ID. A new row is added to the table when a new presentation context is negotiated.

3.3.3.1.4 Current call_id

The server maintains a current call_id for each connection. The current call_id is the highest **call_id** that the server has processed on this connection.

3.3.3.1.5 Client Identity on the Server

When the client has supplied authentication information to the server, the server SHOULD export to higher-level protocols the capability to retrieve the client identity associated with a particular remote procedure method call. For information, see section 3.3.3.4.3.

Note that the above conceptual data can be implemented by using a variety of techniques. An implementation is at liberty to implement such data in any way it pleases.

3.3.3.2 Timers

3.3.3.2.1 Connection Timeout

A higher-level protocol on the server can instruct the RPC runtime to monitor the state of the connection above and beyond the monitoring provided by default by the RPC transport, so that if the client crashes or loses network connectivity to the server, the server can take recovery action. In the common case, the recovery action is a context handle rundown. Depending on the protocol sequence for the method call, the establishment of the timer acts only as advice to the RPC runtime system.<103>

When this timer expires, the expiry is not noticed at the RPC protocol level, but is noticed at the TCP/IP protocol level and shows as a local event, as specified in section 3.3.3.7.1.

3.3.3.3 Initialization

3.3.3.3.1 Server-Side Initialization

These extensions are initialized by performing the actions as specified in the following topics.

3.3.3.3.1.1 Registering a Protocol Sequence by a Higher-Level Protocol

A higher-level protocol MUST register a protocol sequence. Without an RPC transport to deliver the messages, these extensions cannot work.

3.3.3.3.1.2 Registering an Interface by a Higher-Level Protocol

A higher-level protocol MUST register an interface for these extensions to be useful. Even without registering an interface, these extensions can function, but they return errors, as specified in section 3.3.1.5.7, for all attempts to negotiate a presentation context, which means no RPCs can be made.

3.3.3.3.1.3 Registering a Security Provider by a Higher-Level Protocol

If receiving a secure call is expected, a higher-level protocol MUST indicate to the RPC runtime on the server that it is willing to accept calls that are secured by a given security provider. The higher-level protocol does this by registering with the RPC runtime on the server the information specified in section 3.3.3.1.1.<104>

3.3.3.3.1.4 Register a Dynamic Endpoint with Endpoint Mapper

If a server is using a dynamic endpoint, it SHOULD register the list of endpoints that are associated with the given interface UUID/version and object UUID with the local instance of the endpoint mapper. This is done in an implementation-specific way. These extensions do not allow registering on non-local instances of the endpoint mapper.

If a server uses a well-known endpoint, or uses a mechanism specified outside these extensions for discovery of dynamic endpoint, it may skip this step.

The endpoint mapper may not be listening on this RPC protocol sequence until this step. For information, see section 3.1.3.3.1.

3.3.3.3.1.5 Start Listening

A server MUST instruct its RPC transport to get into listening state. The definition of a listening state depends on the RPC transport being used. For details on a given RPC transport, see the documentation for that RPC transport.

3.3.3.4 Higher-Layer Triggered Events

3.3.3.4.1 Failure Semantics

A server protocol built on top of these extensions can encounter a failure while executing a method call. It has two options to handle the failure. It can handle the failure at the application protocol layer or at the RPC protocol layer.

If it handles the error at the application protocol layer, the interaction appears to be successful from an RPC point of view. The [out] parameters are filled, and the RPC implementation on the server sends a response PDU with the stub data, as specified in [C706] section 14.4, NDR Input and Output Streams. In this case, the [out] parameters SHOULD indicate the occurrence of an error, although the exact mechanism for doing so is left to the application protocol layer.

If the error is handled at the RPC protocol layer, the server implementation of the application protocol layer indicates to the RPC runtime (usually through calling an API) that the method call failed, and then supplies a single, unsigned long number that indicates the failure code. In this case, the server SHOULD send back to the client a fault PDU (as specified in [C706] section 12.6.4.7), The fault PDU, where the status field of the fault PDU is set to the failure code received from the application protocol layer.<105>

3.3.3.4.2 shutdown PDUs

Servers MAY send shutdown PDUs, as specified in [C706] section 12.6.4.11, when they need the client to terminate a connection and free up server resources.<106>

3.3.3.4.3 Retrieve the Client Identity and Authorization Information

During the authorization process, a higher-level protocol on the server often needs to retrieve the identity of the client that is making a given request. A server implementation MUST try to retrieve the client identity by executing the following steps in this order:

- If authentication information is associated with the client request, as specified in section 2.2.2.11, the server MUST find the security context indicated by the **auth_context_id** in the sec_trailer of the request, and MUST ask the security provider that created the security context

to retrieve the client identity. For details on how a security provider determines the client identity, see the documentation for the respective security provider.

- If no authentication information is associated with the client request, as specified in section 2.2.2.11, the server MUST ask the RPC transport on which the request was delivered to retrieve the client identity associated with this request. Some RPC transports have the capability to retrieve the client identity associated with the request and some RPC transports do not. If the RPC transport is not capable of retrieving the identity of the client, the server MUST report this to the higher-level protocol in an implementation-specific way. For details on how an RPC transport retrieves the identity of a client, see the documentation for the respective RPC transport.<107>

3.3.3.4.4 Authorization Policy

An administrator on the server machine can deploy authorization policies that restrict access to individual RPC servers or all RPC servers on that machine. In such cases, the server SHOULD send back to the client a fault PDU, as specified in [C706] section 12.6.4.7, The fault PDU, where the status field of the fault PDU is set to status code 5.

3.3.3.5 Message Processing Events and Sequencing Rules

3.3.3.5.1 Failure Semantics

If, during the processing of a method call on the server, the server encounters an error, it SHOULD send back to the client a fault PDU, as specified in [C706] section 12.6.4.7, The fault PDU, where the status field of the fault PDU is set to a descriptive status code. If the server is unable to send a fault PDU, as specified here, it MUST close the transport connection. The exact protocol primitive used for closing a transport connection depends on the RPC transport and is documented in the normative reference for that transport.

Servers can send any status code in the status field of a fault PDU except the following status codes, which a server MUST NOT send to the client. These status codes have special significance in Windows environments, and their presence in the status field MAY be flagged as a protocol error by the client.

Status codes that MUST NOT be sent by RPC servers
ERROR_SUCCESS (0x00000000)
STATUS_GUARD_PAGE_VIOLATION (0x80000001)
STATUS_DATATYPE_MISALIGNMENT (0x80000002)
STATUS_BREAKPOINT (0x80000003)
STATUS_ACCESS_VIOLATION (0xC0000005)
STATUS_IN_PAGE_ERROR (0xC0000006)
STATUS_ILLEGAL_INSTRUCTION (0xC000001D)
STATUS_PRIVILEGED_INSTRUCTION (0xC0000096)
STATUS_INSTRUCTION_MISALIGNMENT (0xC00000AA)
STATUS_STACK_OVERFLOW (0xC00000FD)

Status codes that MUST NOT be sent by RPC servers
STATUS_POSSIBLE_DEADLOCK (0xC0000194)
STATUS_HANDLE_NOT_CLOSABLE (0xC0000235)
STATUS_STACK_BUFFER_OVERRUN (0xC0000409)
STATUS_ASSERTION_FAILURE (0xC0000420)

3.3.3.5.2 call_id Field Must Increase Monotonically

The **call_id** field of any request that arrives on the server MUST monotonically increase. All PDUs of a fragmented request MUST have the same value in the **call_id** field. An implementation SHOULD reject PDUs that violate this rule, as specified in section 3.3.3.5.7.

3.3.3.5.3 Unknown Security Provider

If a bind or alter_context PDU arrives on the server with an **auth_type** field set to a security provider that is not present in the abstract table specified in section 3.3.3.1.1, an implementation of these extensions MUST return error **authentication_type_not_recognized** in the bind_nak or fault PDU.

3.3.3.5.4 Maximum Server Input Data Size

The combined length of the stub data for all fragments of a request SHOULD not exceed 4 MB. If it exceeds 4 MB, the server implementation SHOULD return a fault packet with the status field set to 5.<108>

3.3.3.5.5 Limits of Presentation Contexts Negotiated

A client MUST NOT negotiate more than 4,000 * NumberOfRegisteredInterfaces presentation contexts on the server where NumberOfRegisteredInterfaces is the number of registered interfaces in the server process.

3.3.3.5.6 Dropping Packets for Old Calls

If a server implementation receives a request PDU without the PFC_FIRST_FRAG flag, and the connection is not in Receiving PDU state, it SHOULD compare the **call_id** field from the PDU to the current **call_id** on the connection. If the **call_id** field is smaller by 150 or less, the server SHOULD ignore the packet. If the **call_id** field is smaller by more than 150, the server SHOULD treat this as a protocol error, as specified in section 3.3.3.7.<109>

3.3.3.5.7 Handling Protocol Errors

If a server implementation encounters a condition it interprets to be a protocol error as a result of processing a request PDU, it MUST send back to the client a fault PDU with the status field set to 0x1C01000B. This status value is specified in [C706] section N.2.

3.3.3.5.8 Sequencing in Case of Errors

In the case of a fragmented request with multiple PDU and an error found in a non-last PDU, implementations of these extensions SHOULD return a fault PDU as soon as they have processed the

PDU with the error. They SHOULD NOT wait to receive all PDUs of a fragmented request before sending the fault PDU.

3.3.3.6 Timer Events

For more information on timer events, see section 3.3.3.2.1.

3.3.3.7 Other Local Events

3.3.3.7.1 Transport Connection Shutdown

This event is triggered when the RPC transport indicates to RPC that a connection has timed out. Different RPC transports interpret this differently. For details on how a given RPC transport times out connections, see the documentation for the respective transport. When this event occurs on a connection, all security contexts and presentation contexts are considered invalid. If the connection is the last connection for an association, the context handles belonging to that association are run down.

4 Protocol Examples

The following sections describe protocol examples for both connection-oriented RPC and connectionless RPC scenarios.

4.1 Packet Sequence for Secure, Connection-Oriented RPC Using Kerberos as Security Provider

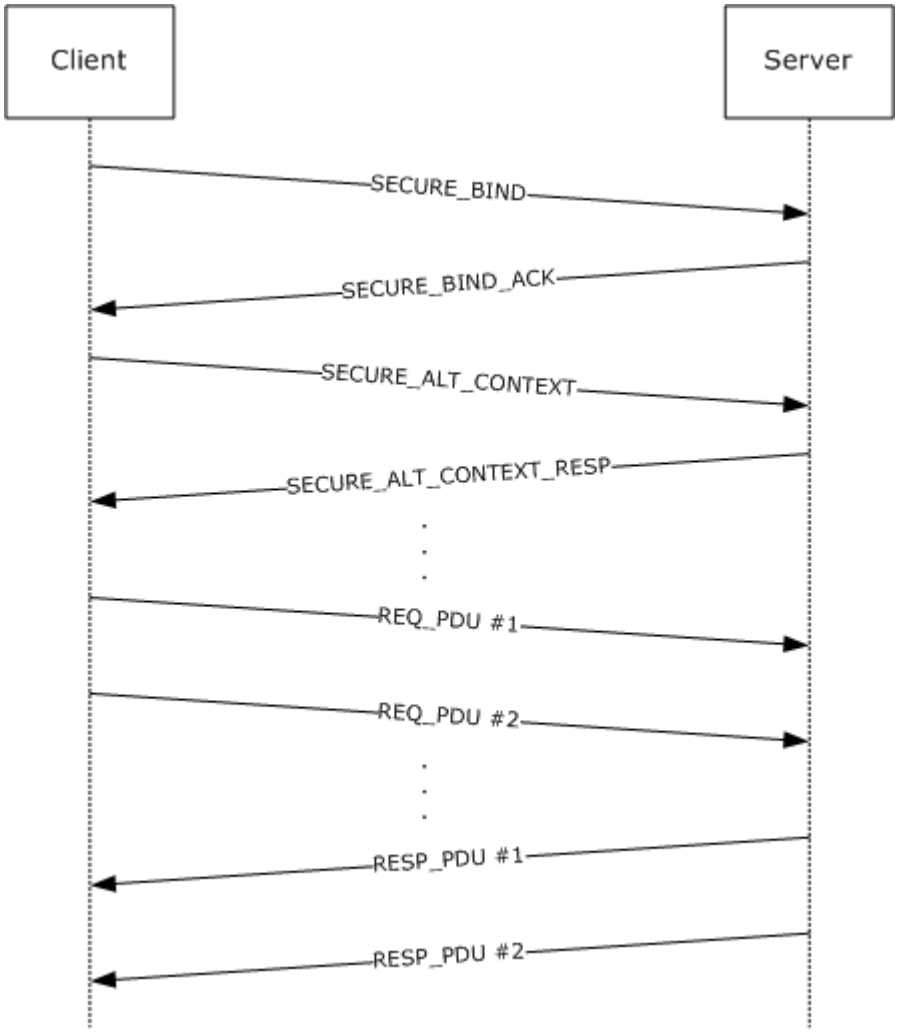


Figure 20: Packet sequence

This example shows a packet sequence for a secure, connection-oriented RPC using Kerberos as the security provider. Individual packet exchanges are specified in detail.

SECURE_BIND: RPC bind PDU with sec_trailer and auth_token. Auth_token is generated by calling the implementation equivalent of the abstract GSS_Init_sec_context call. Upon receiving this, the server calls the implementation equivalent of the abstract GSS_Init_sec_context call, which returns an auth_token and continue status in this example. Assume that:

The client chooses the **auth_context_id** field in the sec_trailer sent with this PDU to be 1.

The client uses the RPC_C_AUTHN_LEVEL_PRIVACY authentication level, and the Authentication Service (AS) is Kerberos.

The client sets the PFC_SUPPORT_HEADER_SIGN flag in the PDU header.

SECURE_BIND_ACK: RPC bind_ack PDU with sec_trailer and auth_token. **PFC_SUPPORT_HEADER_SIGN** flag in the PDU header is also set in this example. Auth_token is generated by the server in the previous step. Upon receiving that PDU, the client calls the implementation equivalent of the abstract GSS_Init_sec_context call, which returns an auth_token and continue status in this example.

SECURE_ALT_CONTEXT: An alter_context PDU with the auth_token obtained in the previous step. Upon receiving this PDU, the server calls the implementation equivalent of the abstract GSS_Init_sec_context call, which returns an auth_token and continue status in this example.

SECURE_ALT_CONTEXT_RESP: An alter_context_resp PDU with sec_trailer and auth_token. Auth_token is generated by the server in the previous step. Upon receiving that PDU, the client calls the implementation equivalent of the abstract GSS_Init_sec_context call, which returns an auth_token and success status in this example. The client knows the security context is ready to be used.

REQ_PDU #1: Client marshals the application data and prepares a stream of octets with the marshaled stub data. In this example, assume that the stream is larger than one PDU and fits into two PDUs. The client sends a request PDU that contains a header, a message body with as much stub data as it can fit in this PDU, sec_trailer with the **auth_context_id** field set to 1, and auth_token generated by the implementation-specific equivalent of the abstract GSS_WrapEx. The message body is sealed, and the header is signed by the GSS_WrapEx. Upon receiving this PDU, the server calls the implementation-specific equivalent of the abstract GSS_UnwrapEx call to verify that the packet has not been tampered with.

REQ_PDU #2: Request PDU that contains a header, a message body with remaining stub data, sec_trailer with the **auth_context_id** field set to 1, and auth_token generated by the implementation-specific equivalent of the abstract GSS_WrapEx call. The message body is sealed, and the header is signed by the GSS_WrapEx. Upon receiving this PDU, the server calls the implementation-specific equivalent of the abstract GSS_UnwrapEx call to verify that the packet has not been tampered with. The server has the full octet stream with the verified stub data and unmarshals the data, invokes the server routine for this method, and waits for it to finish execution. Once this completes, it proceeds to the next step.

RESP_PDU #1: Server marshals the application data into an octet stream with the marshaled stub data. Assume that the marshaled stub data does not fit into a single PDU. The server sends a response PDU that contains a header, a message body with as much stub data as it can fit into this PDU, sec_trailer with the **auth_context_id** field set to 1, and auth_token generated by the implementation-specific equivalent of the abstract GSS_WrapEx. The message body is sealed, and the header is signed by the GSS_WrapEx. Upon receiving this PDU, the client calls the implementation-specific equivalent of the abstract GSS_UnwrapEx call to verify that the packet has not been tampered with.

REQ_PDU #2: Response PDU that contains a header, a message body with remaining stub data, sec_trailer with the **auth_context_id** field set to 1, and auth_token generated by the implementation-specific equivalent of the abstract GSS_WrapEx call. The message body is sealed, and the header is signed by the GSS_WrapEx. Upon receiving this PDU, the client calls the implementation-specific equivalent of the abstract GSS_UnwrapEx call to verify that the packet has not been tampered with. Then it unmarshals the application data from the octet stream in the stub data and returns the data to the client application.

4.2 Packet Sequence for Secure, Connection-Oriented RPC Using NT LAN Manager (NTLM) as Security Provider

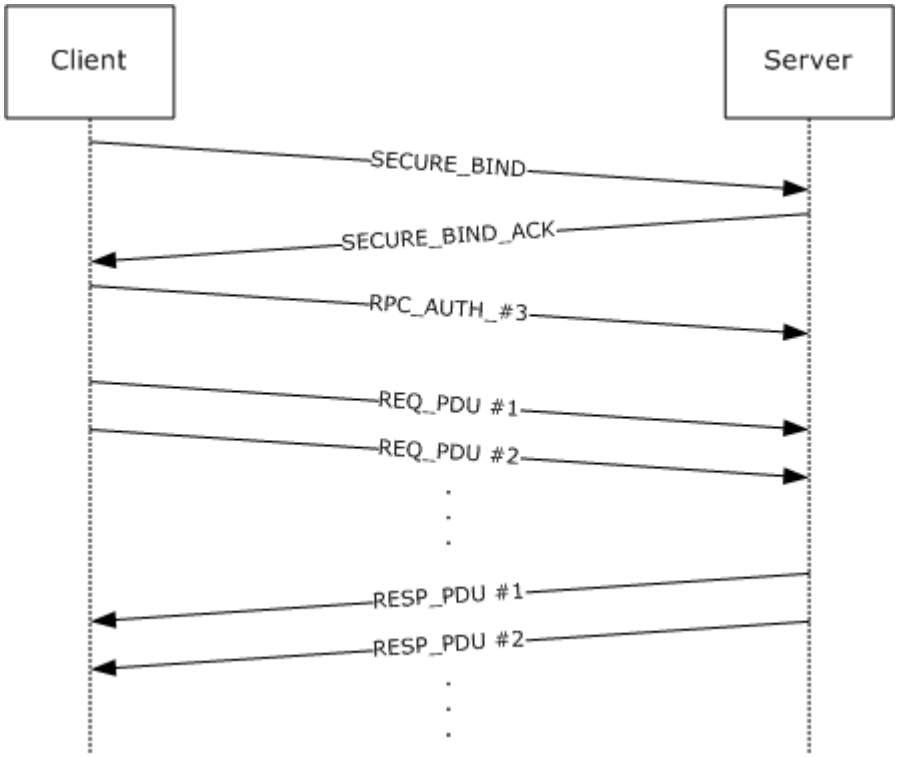


Figure 21: Packet exchange sequence

This example shows a packet exchange sequence for a secure, connection-oriented RPC using NT LAN Manager (NTLM) as the security provider. Individual packets are specified in detail.

SECURE_BIND: RPC bind PDU with sec_trailer and auth_token. Auth_token is generated by calling the implementation equivalent of the abstract GSS_Init_sec_context call. Upon receiving that, the server calls the implementation equivalent of the abstract GSS_Init_sec_context call, which returns an auth_token and continue status in this example. Assume that:

- The client chooses the **auth_context_id** field in the sec_trailer sent with this PDU to be 1.
- The client uses the RPC_C_AUTHN_LEVEL_PRIVACY authentication level and the Authentication Service (AS) NTLM.
- The client sets the **PFC_SUPPORT_HEADER_SIGN** flag in the PDU header.

SECURE_BIND_ACK: RPC bind_ack PDU with sec_trailer and auth_token. The PFC_SUPPORT_HEADER_SIGN flag in the PDU header is also set in this example. Auth_token is generated by the server in the previous step. Upon receiving that PDU, the client calls the implementation equivalent of the abstract GSS_Init_sec_context call, which returns an auth_token and continue status in this example.

RPC_AUTH_3: The client knows that this is NTLM that uses three legs. It sends an **rpc_auth_3 PDU** with the auth_token obtained in the previous step. Upon receiving this PDU, the server calls

the implementation equivalent of the abstract GSS_Init_sec_context call, which returns success status in this example.

REQ_PDU #1: The client marshals the application data and prepares a stream of octets with the marshaled stub data. In this example, assume that the stream is larger than one PDU and fits into two PDUs. The client sends a request PDU that contains a header, a message body with as much stub data as it can fit in this PDU, sec_trailer with the **auth_context_id** field set to 1, and auth_token generated by the implementation-specific equivalent of the abstract GSS_WrapEx. The message body is sealed, and the header is signed by the GSS_WrapEx. Upon receiving this PDU, the server calls the implementation-specific equivalent of the abstract GSS_UnwrapEx call to verify that the packet has not been tampered with.

REQ_PDU #2: Request PDU that contains a header, a message body with remaining stub data, sec_trailer with the **auth_context_id** field set to 1, and auth_token generated by the implementation-specific equivalent of the abstract GSS_WrapEx call. The message body is sealed, and the header is signed by the GSS_WrapEx. Upon receiving this PDU, the server calls the implementation-specific equivalent of the abstract GSS_UnwrapEx call to verify that the packet has not been tampered with. The server has the full octet stream with the verified stub data and unmarshals the data, invokes the server routine for this method, and waits for it to finish execution. Once this completes, it proceeds to the next step.

RESP_PDU #1: Server marshals the application data into an octet stream with the marshaled stub data. Assume that the marshaled stub data does not fit into a single PDU. The server sends a response PDU that contains a header, a message body with as much stub data as it can fit into this PDU, sec_trailer with the **auth_context_id** field set to 1, and auth_token generated by the implementation-specific equivalent of the abstract GSS_WrapEx. The message body is sealed, and the header is signed by the GSS_WrapEx. Upon receiving this PDU, the client calls the implementation-specific equivalent of the abstract GSS_UnwrapEx call to verify that the packet has not been tampered with.

REQ_PDU #2: Response PDU that contains a header, a message body with remaining stub data, sec_trailer with the **auth_context_id** field set to 1, and auth_token generated by the implementation-specific equivalent of the abstract GSS_WrapEx call. The message body is sealed, and the header is signed by the GSS_Wrap. Upon receiving this PDU, the client calls the implementation-specific equivalent of the abstract GSS_UnwrapEx call to verify that the packet has not been tampered with. Then it unmarshals the application data from the octet stream in the stub data and returns them to the client application.

4.3 Packet Sequence of the First Non-Idempotent RPCs of a Connectionless Activity

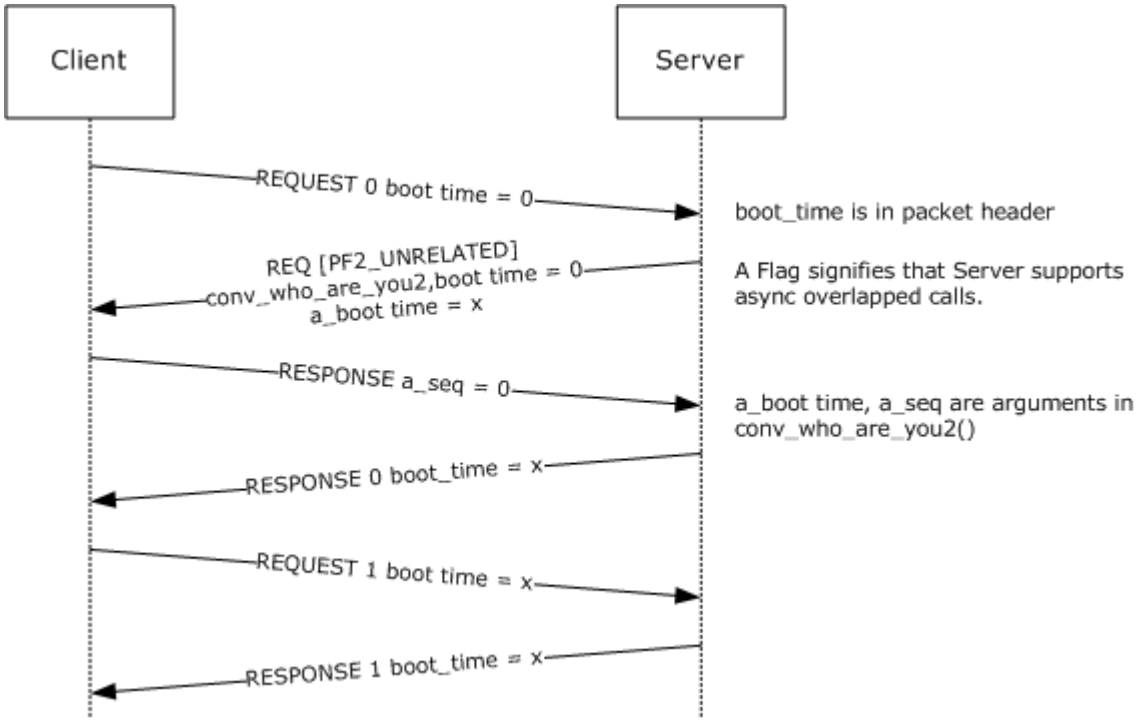


Figure 22: Packet exchange

This example shows the packet exchange when a connectionless client makes two sequential non-idempotent RPCs to a server that the client process has not previously contacted. Individual packets are defined here.

REQUEST 0: A request PDU for the client's first call. The activity ID in the header is the newly formed client activity ID, and the sequence number is zero. Because the client does not know the server's boot time, the boot time in the packet header is zero.

REQ [PF2_UNRELATED]: A conv_who_are_you2 request. This is a REQUEST PDU. The activity ID is a newly generated GUID, and the sequence number is zero because this is the first call from the server process to the client process.

The *actuid* parameter of the request contains the activity ID of **REQUEST 0**. The boot time parameter of the request contains the server's non-zero boot time.

The PF2_UNRELATED flag is set because the server supports this feature.

RESPONSE: A conv_who_are_you2 response. This is a RESPONSE PDU. The activity ID and sequence number in the RPC header match the ones in **REQ [PF2_UNRELATED]**.

The *seq* parameter of the response contains zero because the lowest currently active call sequence of *actuid* is zero.

The *cas_uuid* parameter of the response contains the client's non-NULL CAS UUID.

RESPONSE 0: A response PDU for the client's first call. The activity ID and sequence number match those in the **REQUEST 0**.

The boot time in the packet header is the server's non-zero boot time.

REQUEST 1: A request PDU for the client's second call. The activity ID is the same as in **REQUEST 0**; sequence number is one; boot time is the same as boot_time from **REQ [PF2_UNRELATED]**.

RESPONSE 1: A response PDU for the client's second call. The activity ID and sequence number match those in **REQUEST 1**.

The boot time in the packet header is the server's non-zero boot time.

4.4 Connectionless RPCs with and Without a Delayed ACK

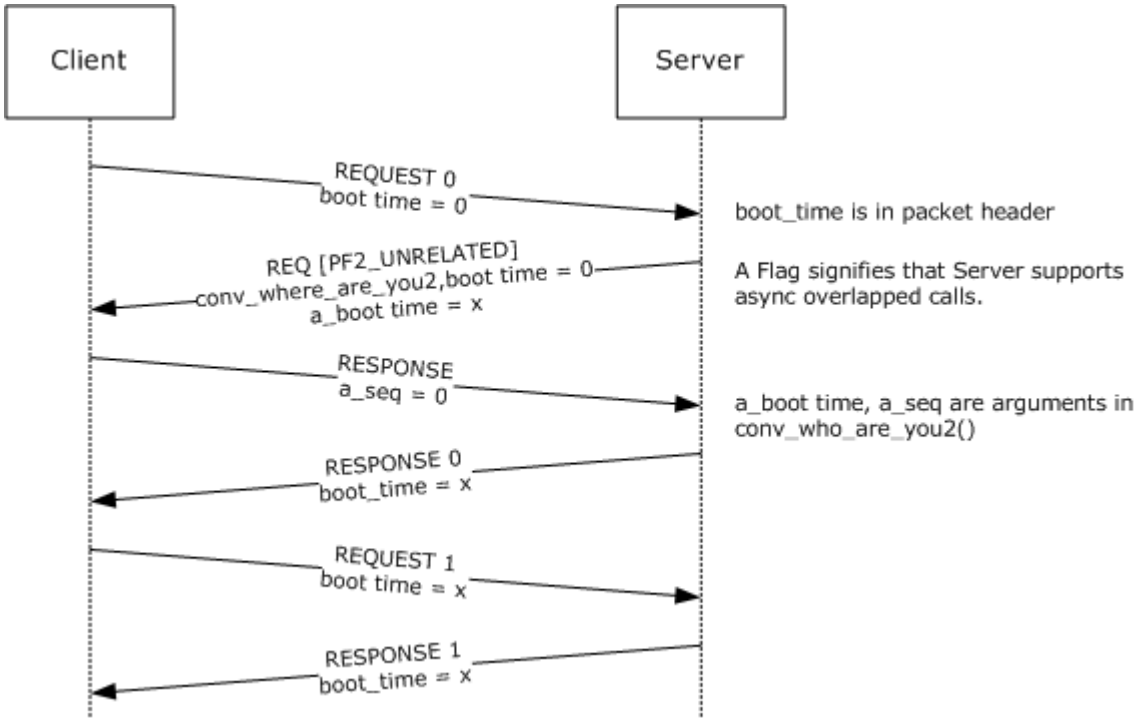


Figure 23: Client sending an ACK packet

This example illustrates the client sending an ACK packet.

There is no ACK sent between the RPCs with sequence numbers 0 and 1 because less than 2 seconds elapse between RPC 0's last response PDU and RPC 1's first request PDU.

The corresponding delay between sequence 1 and sequence 2 is larger than 2 seconds, so 2 seconds after the last response PDU of call 1, the client sends an ACK packet whose sequence number is 1. This does not affect the PDUs for call sequence 2.

4.5 Connectionless Client Communicating with a Dynamic Server Endpoint

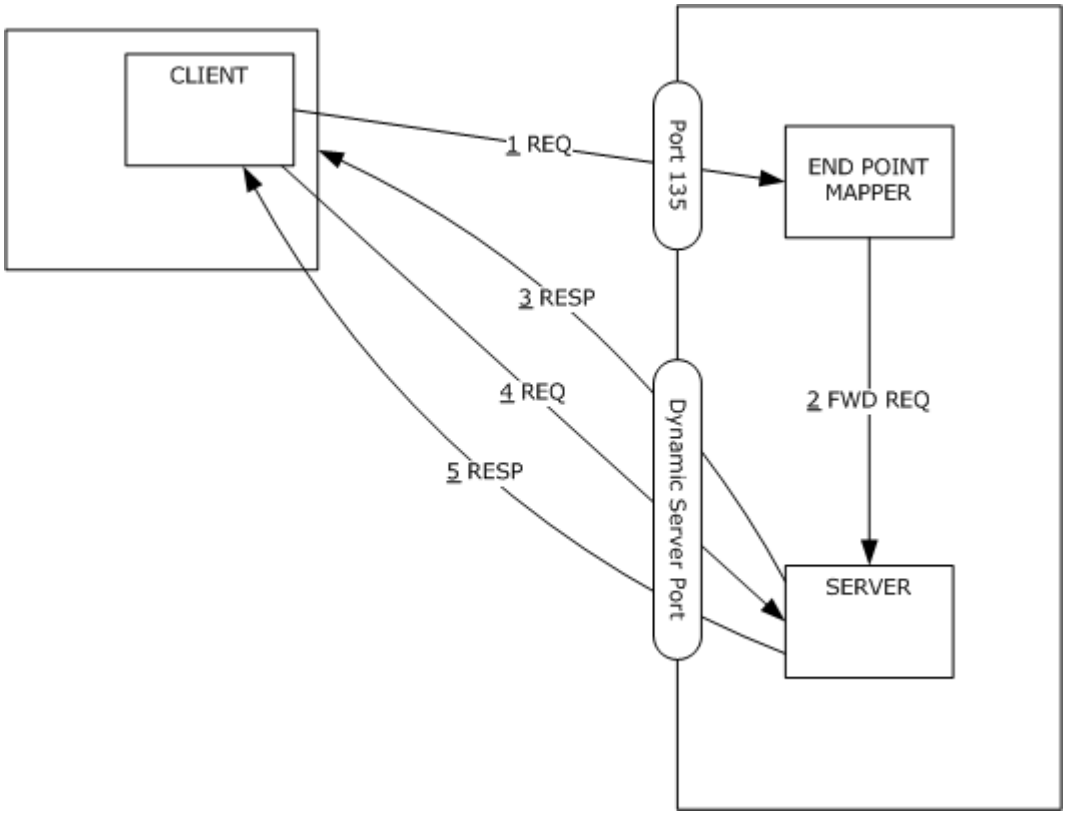


Figure 24: Connectionless client issuing a sequence of calls

This example illustrates a connectionless client issuing a sequence of calls to a server that uses a dynamic UDP endpoint.

Initially, the client has no knowledge of the server's actual endpoint, so the first request PDU (packet #1) is sent to the endpoint mapper port for UDP, which is port 135.

The endpoint mapper forwards the request and the client endpoint to the RPC server in an implementation-dependent way (packet #2).

The server processes the request, as usual, and replies directly to the client endpoint (packet #3).

While processing the received PDU, the client updates its internal structures so that further PDUs belonging to this activity are sent directly to the correct server port. After this point, the client communicates directly with the RPC server (request #4 and reply #5).

4.6 Correlation Example

```
void CorrelatedMethod([in] long Size,
    [in,size_is(Size)] short * pArray );
```

In this method, the value of Size dictates the size of conformant array pArray in octet stream. Parameter Size is correlated to parameter pArray.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Value of Size																															
Maximum Count = Size																															
Representation of first element																Representation of second element															
continued																continued															
continued																Representation of the n-th element, where n=Size															

The maximum count of the conformant array is the value of Size. In the example, correlation validation succeeds if the value of Size is equal to the maximum count of the conformant array referred by pArray.

4.7 UNICODE_STRING Representation

The following structure uses expression in both conformance and varying description:

```
typedef _UNICODE_STRING {
unsigned short MaximumLength;
unsigned short Length;
[size_is(MaximumLength/2),length_is(Length/2)]
   unsigned short * pString;
} UNICODE_STRING;
```

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
MaximumLength																Length															
Pointer Identifier																															
Maximum Count = MaximumLength/2																															
Offset (0)																															
Actual Count = Length/2																															
Representation of first element																continued															
continued																continued															
continued																Representation of last element															

In this example, in the conformant varying array referred by pString, the maximum count is what the expression (MaximumLength/2) evaluates to, and the actual count is what the expression (Length/2) evaluates to.

In the target level 5.0 data consistency check, the implementation validates that the maximum count of the conformant varying array referred by pString is equal to the evaluation result of the expression (MaximumLength/2), and the actual count is equal to the evaluation result of the expression (Length/2).

4.8 Example of Structure with Trailing Gap in NDR64

This example shows a structure with a trailing gap in NDR64.

```
typedef struct _StructWithPad
{
long l;
short s;
} StructWithPad;
```

The size of the structure in the octet stream MUST contain a 2-byte trailing gap to make its size 8, a multiple of the structure's alignment, 4.

5 Security

The following sections describe security considerations for implementers of Remote Procedure Call Protocol Extensions.

5.1 Security Considerations for Implementers

5.1.1 Authentication Levels

Implementations SHOULD create programming interfaces and corresponding documentation for accessing functionality offered by these extensions in a way that encourages higher-level protocols to use authentication levels of RPC_C_AUTHN_LEVEL_INTEGRITY and RPC_C_AUTHN_LEVEL_PRIVACY. Lower authentication levels provide weak security only. If integrity or confidentiality protection is requested, it should be provided either by the security provider or by using the verification trailer, as specified in section 2.2.2.13.

5.1.2 Preferred Security Providers

Implementations SHOULD create programming interfaces and corresponding documentation for accessing functionality offered by these extensions in a way that encourages higher-level protocols to not use NT LAN Manager (NTLM) as the security provider. Simple and Protected [GSS]-API Negotiation Mechanism and Kerberos offer stronger security.

5.1.3 Impersonation Levels

Implementers should create programming interfaces and corresponding documentation for accessing functionality offered by these extensions in a way that encourages higher-level protocols to use the principle of least privilege. As the default impersonation level is RPC_C_IMPL_LEVEL_IMPERSONATE, higher level protocols should use RPC_C__IMPL_LEVEL_IDENTIFY if possible.

5.2 Index of Security Parameters

Security Parameter	Section
Security Provider	2.2.1.1.7
Authentication Level	2.2.1.1.8
Impersonation Level	2.2.1.1.9

6 Appendix A: Full Remote Procedure Call (RPC) Extensions IDL

For ease of implementation, the full RPC extensions IDL interface is provided.

```
typedef struct _GUID
{
    unsigned long  Data1;
    unsigned short Data2;
    unsigned short Data3;
    byte           Data4[8];
} GUID;

typedef GUID UUID;

typedef struct _FILETIME
{
    unsigned long dwLowDateTime;
    unsigned long dwHighDateTime;
} FILETIME;

typedef struct _LARGE_INTEGER {
    __int64 QuadPart;
} LARGE_INTEGER;

typedef __int64 LONGLONG;
typedef unsigned __int64 ULONGLONG;
typedef long NTSTATUS;

typedef unsigned long DWORD;
```

7 Appendix B: Windows Behavior

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

- Windows Server 2008
- 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 2.1: Protocol towers based on Banyan Vines, DECnet, and MSMQ are deprecated and are not supported in Windows XP and later versions. Except for those, all protocol towers that Microsoft supports or previously supported in Windows 2000 and later versions are specified in this document and its normative references.

<2> Section 2.1.1.1: In Windows releases earlier than Windows XP, IPv6 addresses are not supported.

<3> Section 2.1.1.2: In Windows NT, Windows 2000 IPv6 addresses are not supported.

<4> Section 2.1.1.2: This protocol identifier was implemented by legacy versions of Windows for historical reasons and is preserved by current versions for backward compatibility.

<5> Section 2.1.1.2: Windows always asks the SMB Protocol implementation to execute a transaction over the named pipe encompassing the write of the last PDU and the read of the first PDU on the client for synchronous RPCs.

<6> Section 2.1.1.3: Starting with Windows Vista, Windows no longer supports this protocol sequence.

<7> Section 2.1.1.4: This protocol identifier was implemented by legacy versions of Windows for historical reasons and is preserved by current versions for backward compatibility.

<8> Section 2.1.1.4: Windows implementations of NetBIOS require processes to listen on a specific network interface device, and they have no provisions for routing messages between network interfaces that are not directly attached to the same link. For a Windows RPC client and RPC server to communicate, the server MUST be listening on a network interface that the client can reach.

<9> Section 2.1.1.4: Windows XP and later releases no longer support this protocol sequence.

<10> Section 2.1.1.5: This protocol identifier was implemented by legacy versions of Windows for historical reasons and is preserved by current versions for backward compatibility.

<11> Section 2.1.1.5: Windows implementations of NetBIOS require processes to listen on a specific network interface device, and they have no provisions for routing messages between network interfaces that are not directly attached to the same link. For a Windows RPC client and RPC

server to communicate, the server MUST be listening on a network interface that the client can reach.

<12> Section 2.1.1.5: Windows XP and later releases no longer support this protocol sequence.

<13> Section 2.1.1.6: Windows implementations of NetBIOS require processes to listen on a specific network interface device, and they have no provisions for routing messages between network interfaces that are not directly attached to the same link. For a Windows RPC client and RPC server to communicate, the server MUST be listening on a network interface that the client can reach.

<14> Section 2.1.1.6: Windows XP and later releases no longer support this protocol sequence.

<15> Section 2.1.1.7: Windows XP and later releases no longer supports this protocol sequence.

<16> Section 2.1.2: Starting with Windows Vista, Windows no longer supports connectionless RPC exchanges or connectionless RPC transports.

<17> Section 2.1.2.1: When a connectionless RPC server or RPC client runs over UDP on Windows NT 4.0, the maximum size of a PDU is 1,024 bytes. Details on PDU length and fragmentation of request and response buffers is as specified in [C706] section 12.5.1, Connectionless PDU Structure. When a connectionless RPC server or RPC client runs over UDP on Windows 2000 and later versions, the maximum size of a PDU is 4,096 bytes. Details on PDU length and fragmentation of request and response buffers is as specified in [C706] section 12.5.3, Connectionless PDU Definitions.

<18> Section 2.1.2.2: When connectionless RPC exchange occurs over IPX on Windows NT 4.0, the maximum size of a PDU is 1,024 bytes. For details of PDU length and fragmentation of request and response buffers, see [C706] section 12.5.1, Connectionless PDU Structure. When connectionless RPC exchange occurs over IPX on Windows 2000 and later versions, the maximum size of a PDU is 1,464 bytes. For details of PDU length and fragmentation of request and response buffers, see [C706] section 12.5.3, Connectionless PDU Definitions.

<19> Section 2.1.2.2: Starting with Windows XP, Windows no longer supports this protocol sequence.

<20> Section 2.2.1.1.3: Windows uses the algorithm, as specified in [RFC4122], to generate the UUID.

<21> Section 2.2.1.1.4: Windows–based servers set the **context_handle_attributes** field to zero.

<22> Section 2.2.1.1.7: Without the installation of additional software, Windows supports the following authentication types:

Security Provider

Security Provider Simple and Protected GSS-API Negotiation Mechanism (SPNEGO)
NT LAN Manager (NTLM)
Kerberos
Netlogon

<23> Section 2.2.1.2.2: Windows versions earlier than Windows Server 2003 use the same definition of the structure as what is specified in [C706] Appendix L, Protocol Tower Encoding.

<24> Section 2.2.1.2.4: Windows versions earlier than Windows Server 2003 redefine the same method by:

- Adding the **ptr** attribute to the *object* and *Ifid* parameters.
- Removing the **[idempotent]** method attribute.

The redefined method is as follows:

```
void
ept_lookup (
        [in] handle_t hEpMapper,
        [in] unsigned long inquiry_type,
        [in, ptr] UUID   * object,
        [in, ptr] RPC_IF_ID * Ifid,
        [in] unsigned long vers_option,
        [in, out] ept_lookup_handle_t *entry_handle,
        [in, range(0, 500)] unsigned long max_ents,
        [out] unsigned long *num_ents,
        [out, length_is(*num_ents), size_is(max_ents)]
              ept_entry_t entries[],
        [out] error_status *status
        );
```

Everything else about this method remains as specified in [C706] Appendix O, Endpoint Mapper Interface Definition.

<25> Section 2.2.1.2.5: Windows versions earlier than Windows Server 2003 redefine the method by:

- Adding the **ptr** attribute to the *obj* and *map_tower* parameters.
- Removing the **[idempotent]** method attribute.

The redefined method is as follows:

```
void __RPC_FAR
ept_map (
    [in] handle_t hEpMapper,
    [in, ptr] UUID * obj,
    [in, ptr] twr_p_t
 map_tower,
    [in, out] ept_lookup_handle_t  *entry_handle,
    [in] unsigned long max_towers,
    [out] unsigned long *num_towers,
    [out, ptr, size_is(max_towers),length_is(*num_towers)]
          twr_p_t *ITowers,
    [out] error_status *status
    );
```

Everything else about this method remains as specified in [C706] Appendix O, Endpoint Mapper Interface Definition. Note, this redefinition has no wire impact and therefore, it is interoperable with the [C706] implementation.

<26> Section 2.2.1.2.6: Windows NT 4.0 supports this method. The definition of the method for Windows NT 4.0 Option Pack is as specified in [C706] Appendix O, Endpoint Mapper Interface Definition. Windows 2000, Windows XP, and Windows Server 2003 preserve the Windows NT 4.0 definition of the method. However, the method performs no operation, returning **EPT_S_CANT_PERFORM_OP** in the status field.

Windows Vista and Windows Server 2008 versions of the operating system redefine the method by removing all parameters to the method. The resulting definition is:

```
void    ept_insert(void);
```

This method performs no operation. However, instead of returning **EPT_S_CANT_PERFORM_OP** in the status field, the method raises an **EPT_S_CANT_PERFORM_OP** exception.

<27> Section 2.2.1.2.7: Windows NT 4.0 supports this method. The definition of the method for Windows NT 4.0 is as specified in [C706] Appendix O, Endpoint Mapper Interface Definition. Windows 2000, Windows XP, and Windows Server 2003 preserve the Windows NT 4.0 definition of the method. However, the method performs no operation, returning **EPT_S_CANT_PERFORM_OP** in the status field.

Windows Vista and Windows Server 2008 versions of the operating system redefine the method by removing all parameters to the method. The resulting definition is:

```
void
        ept_delete(
                void
                );
```

This method performs no operation. However, instead of returning **EPT_S_CANT_PERFORM_OP** in the status field, the method raises an **EPT_S_CANT_PERFORM_OP** exception.

<28> Section 2.2.1.2.9: Windows NT 4.0 supports this method. The definition and behavior of the method is as specified in [C706] Appendix O, Endpoint Mapper Interface Definition. Windows 2000, Windows XP, and Windows Server 2003 preserve the Windows NT 4.0 definition of the method. However, the method performs no operation, returning **EPT_S_CANT_PERFORM_OP** in the status field.

Windows Vista and Windows Server 2008 versions of the operating system redefine the method by removing all parameters to the method. The redefined method is as follows:

```
void
        ept_inq_object(
                void
                );
```

This method performs no operation. However, instead of returning **EPT_S_CANT_PERFORM_OP** in the status field, the method raises an **EPT_S_CANT_PERFORM_OP** exception.

<29> Section 2.2.1.2.10: Windows NT 4.0 supports this method. The definition and behavior of the method is as specified in [C706] Appendix O, Endpoint Mapper Interface Definition. Windows 2000, Windows XP, and Windows Server 2003 preserve the Windows NT 4.0 definition of the method.

However, the method performs no operation, returning **EPT_S_CANT_PERFORM_OP** in the status field.

Windows Vista and Windows Server 2008 versions of the operating system redefine the method by removing all parameters to the method. The redefined method is as follows:

```
void
        ept_mgmt_delete(
                void
                );
```

This method performs no operation. However, instead of returning **EPT_S_CANT_PERFORM_OP** in the status field, the method raises an **EPT_S_CANT_PERFORM_OP** exception.

<30> Section 2.2.1.3.2: This type is not defined in Windows versions earlier than Windows Server 2003.

<31> Section 2.2.1.3.3: Windows versions earlier than Windows Server 2003 use the definition of the method specified in [C706] Appendix Q Remote Management Interface.

<32> Section 2.2.1.3.4: Windows versions earlier than Windows Server 2003 use the definition of the method specified in [C706] Appendix Q, Remote Management Interface.

<33> Section 2.2.2.2: This flag is ignored when this flag is present in a PDU. This flag is specified in [C706] section 12.6 Connection-Oriented RPC PDUs.

<34> Section 2.2.2.9: Starting with Windows XP, servers return the RPC extended error information UUID in the Signature field of the **bind_nak** packet if there is RPC extended error information present. Starting with Windows XP, Windows–based clients decode the RPC extended error information, if present, and allow the application to query that information.

<35> Section 2.2.2.11: Windows–based clients starting with Windows XP SP2 and Windows Server 2003 and later versions of Windows send extra zero octets at the end of the authentication token if the security provider indicates a shorter length of the authentication token than the sender of the data estimated initially. Earlier versions of Windows send undefined octets in this case.

<36> Section 2.2.2.13: On Windows 2000 SP4 and subsequent service packs, Windows XP SP2 and subsequent service packs, and Windows Server 2003 and subsequent service packs, Windows does not send the verification trailer for an RPC with the pipe IDL attribute, as specified in [C706] section 4.2. Windows Vista and later versions of Windows sends the verification trailer for an RPC with an pipe IDL attribute only if all the parameters with pipe attribute are [out] only.

<37> Section 2.2.2.13: Support for verification trailers is present in Windows 2000 Server SP4 and subsequent service packs, Windows XP SP2 and subsequent service packs, and Windows Server 2003 and later versions of Windows. What part of the verification trailer is used by Windows, and when it is used is specified in sections 2.2.2.13.3 and 2.2.2.13.4.

<38> Section 2.2.2.13.2: This verification trailer command is sent for the first request on a connection only.

<39> Section 2.2.2.13.3: This verification trailer command is sent for every request when the security provider does not support header signing.

<40> Section 2.2.2.13.4: This verification trailer command is sent on the first request PDU that uses an abstract_syntax and transfer_syntax that were previously sent on a bind or alter_context PDU.

<41> Section 2.2.3: Starting with Windows Vista, Windows no longer supports connectionless RPC messages.

<42> Section 2.2.3.3: PF2_UNRELATED is not used in Windows NT Server 4.0. It is supported in all later versions of Windows that support connectionless RPC messages.

<43> Section 2.2.3.5: Windows–based clients starting with Windows XP SP2 and Windows Server 2003 and later versions of Windows send extra zero octets at the end of the authentication token if the security provider indicates a shorter length of the authentication token than the sender of the data estimated initially. Earlier versions of Windows send undefined octets in this case.

<44> Section 2.2.3.5: These extensions require the model specified in [RFC2743] for all interactions with all security providers. An implementation instructs the [GSS]-compatible security provider to operate in a DCE-compatible manner by setting the DCE Style protocol variable. The following table details what PDU type carries (in its token section) the output of the [GSS] call. Note that the first call to GSS_Init_sec_context generates no token transmitted to the server, and that there is no support for a provider requiring more than two calls to GSS_Init_sec_context or GSS_Accept_sec_context.

<45> Section 2.2.4.3: Arrays of context handles are only supported in Windows Vista and Windows Server 2008.

<46> Section 2.2.4.5: In the Windows version of the Microsoft Interface Definition Language (MIDL), this is accomplished by compiling with the ms_union MIDL compiler option on MIDL compilers, starting with version 3.01.75.

<47> Section 2.2.4.7: Windows supports a subset of the expressions allowed in C language in both NDR64 transfer syntax and when target level 6.0 strict NDR/NDR64 data consistency check is requested. The subset is the same in both cases.

<48> Section 2.2.4.11: Windows implementation indicates the octet stream as invalid if the provided byte count is not big enough to contain all the memory needed to unmarshal the pointer indicated by the other pointer parameter. byte_count is not supported in NDR64 transfer syntax.

<49> Section 2.2.5: NDR64 is available on the Windows XP–based client running on x64, on 64-bit versions of the Windows Server 2003 family, and on later 64-bit versions of Windows. NDR64 is not available for connectionless RPC.

<50> Section 2.2.5.3.2.1: A conformant array can contain, at most, 2^{31}-1 elements in Windows.

<51> Section 2.2.5.3.2.2: A varying array can contain, at most, 2^{31}-1 elements in Windows.

<52> Section 2.2.5.3.2.3: A conformant varying array can contain, at most, 2^{31}-1-o elements where o is the offset.

<53> Section 2.2.5.3.4.5: A chunk can contain, at most, 2^{31}–1 elements of the pipe.

<54> Section 2.2.6.1: If the endianness is not 0x10 indicating little-endian, Windows assumes big-endian, as specified in section 2.2.6.1.

<55> Section 2.2.7.1: During unmarshaling, Windows ignores the value of the **InterfaceID** field.

<56> Section 3.1.1.5.1.1.2: The Windows system always selects the leftmost [in] handle as the binding handle.

<57> Section 3.1.1.5.3.2: This level of strict NDR/NDR64 data consistency check is enabled by using target robust compiler option, using an Microsoft Interface Definition Language (MIDL) compiler. Target level 5.0 strict NDR/NDR64 data consistency check is available in Windows 2000 and later.

<58> Section 3.1.1.5.3.2.2.1: If the maximum memory size exceeds 2^{31}-1 bytes for a conformant structure, conformant varying structure, conformant array, conformant varying array, or conformant and varying string, the octet stream is indicated as invalid.

<59> Section 3.1.1.5.3.2.2.5: Interfaces using auto handle are rejected in this level of consistency check.

<60> Section 3.1.1.5.3.3: This level of strict NDR/NDR64 data consistency check is enabled by using the target NT60 compiler option, using a Microsoft Interface Definition Language (MIDL) compiler. Target level 6.0 strict NDR/NDR64 data consistency check is available in Windows Vista and Windows Server 2008.

<61> Section 3.1.1.5.3.3.1.2: This behavior is available in Windows Vista when the Interface Definition Language (IDL) file is compiled for target level 6.0 strict NDR/NDR64 data consistency check. This behavior is turned off if the IDL file is compiled with Microsoft Interface Definition Language (MIDL) command option backward_compat maybenull_sizeis.

<62> Section 3.1.1.5.4: By default, Windows XP SP2 and later and Windows Vista does not allow remote anonymous calls.

By default, Windows Server 2003 and Windows Server 2008 allows remote anonymous calls.

For details and how to change this behavior see [RPCIFRESTRICTION]

<63> Section 3.1.2.5.1.6: These additional client conformant validation checks are available in Windows Vista and Windows Server 2008. Users can disable these validations through registry and/or application compatibility settings. There is no validation support for multiple dimension conformant/varying arrays. A subset of the rules specified in this section are available in Windows Server 2003 SP1, as listed. These validations can be disabled by Windows registry settings.

- Validations are available for parameter-level correlation only. There is no support for embedded pointers, arrays, or structures.
- Validations are available for NDR transfer syntax only. There is no support for NDR64 transfer syntax.
- Conformant array, conformant varying array, or conformant varying string parameter must be declared earlier in the parameter list before the parameter describing the conformance.
- Conformance can only be specified by dereference of another parameter, the value of another parameter plus one, the value of another parameter minus one, the value of another parameter multiplied by two, or the value of another parameter divided by two.

There is no validation support for a conformant varying string whose maximum count is not specified by another parameter.

<64> Section 3.1.3.3.1: On Windows, the endpoint mapper does not listen on a protocol sequence until at least one server using dynamic endpoints on the system starts to listen on that protocol sequence.

<65> Section 3.1.3.5.1: Certain implementations of Windows 2000, Windows XP, and Windows Server 2003 may choose to limit the size of server stub memory allocation via configuration settings. This behavior is available in Windows Server 2003 SP2 and subsequent service packs.

<66> Section 3.2: Starting with Windows Vista, Windows no longer supports connectionless RPC protocol variants. Attempts to listen on the server on connectionless protocol sequences and attempts to make RPCs on the client result in an **RPC_S_CANNOT_SUPPORT** (0x6e4) error.

<67> Section 3.2.1.1.1: As specified in [C706] sections 9.5.5, 10.1, and 10.2 allow conforming implementations broad latitude in implementing the sliding window algorithm for REQUEST and RESPONSE fragments. The Windows–based client and server call objects share a common packet-windowing implementation that maintains separate windows for the data to be sent and received. The Windows behavior is compatible with clients and servers that use other windowing implementations conforming to [C706]. The following section specifies the implementation of the sliding window algorithm. For a particular call, the send window contains the following properties:

- **Fragment_base**: An unsigned 32-bit integer representing the first unacknowledged fragment of the buffer to be sent. It is zero initially and advanced when the receiver acknowledges one or more additional fragments.
- **Maximum_window_size**: An unsigned 32-bit integer representing the maximum number of unacknowledged fragments. It is set to one for the first call of an activity; maximum supported value is 32; updated by the **window_size** field of a FACK or NOCALL. At the end of the call, the window size is stored in the activity, and the next call begins with the stored value.
- **Burst_length**: Number of fragments to transmit at one time. Initially one; limited by the maximum window size. It is incremented when a FACK is received, and cut in half when a receive times out. The minimum value is 0. For details, see Packet Transmission Behavior at the end of this list.
- **Send_serial_number**: Serial number of the next packet to be sent. Initially zero; incremented after every sent fragment and (for client) PING packet.
- **Fack_serial_number**: The latest serial number acknowledged by the recipient. Initially zero; updated when a received FACK or NOCALL carries a higher value in its **serial_num** field.
- **Maximum_pdu_length**: The size of the largest packet that can be sent and received by the transport. Set to 1,024 bytes for the first call of an activity. At the end of the call, the current value is stored in the activity, and the next call begins with the stored value. When a FACK or (from client) NOCALL is received, the value is updated to the lower of the local transport limit and the value in the packet's **max_tsdu** field.
- **Maximum_fragment_length**: The largest amount of stub data that fits in a single protocol data unit (PDU). It is equal to the maximum PDU length minus 0x80 bytes for the remote procedure call (RPC) header and the number of bytes required for the security trailer. It is updated whenever maximum_pdu_length is updated.

Packet Transmission Behavior. A client call sends fragments in the following three cases:

1. When the call is first instantiated. The call sets the burst length to the current server window size in the call's client address space (CAS), and then sends a burst of fragments.
2. When a FACK or NOCALL-with-body is received from the server. The client call increments its current burst length, and then sends a burst of fragments.
3. When the packet retransmission timer is triggered (for more information, see section 3.2.2.2.1). The client reduces the current burst length by half, and then sends a burst of fragments.

When the client or server must send a burst of fragments, it attempts to send burst_length fragments. The sender first attempts to extend the window by sending never-before-sent fragments. All fragments except the last are sent with the **PF_NOFACK** flag set. The last fragment sent clears the **PF_NOFACK** flag unless (a) it is the final fragment of the call data, or (b) it is overlapping a previous async call of the activity (that is, the **PF2_UNRELATED** flag is set). Otherwise it, too, is sent with the **PF_NOFACK** flag. If fewer than burst_length are sent because the call data is too short or the send window limit is reached, the burst length is reduced by half. If no fragments at all are sent, the lowest unacknowledged fragment is resent with the **PF_NOFACK** flag cleared. The receive window contains the following properties:

- For each received fragment, its fragment number and serial number.
- Receive serial number: the latest serial number received in this call.

When a packet with **PF_NOFACK** cleared is received, the recipient sends a FACK with a version-zero body. The **max_tdsu** field is set to the maximum PDU length for the transport (for more information, see section 2.1.2). The **max_frag_size** field is set to the maximum unfragmented packet length for the transport (for more information, see section 2.1.2). The **window_size** field is calculated by dividing a version-specific constant by the number of calls currently using the port. The version-specific constant is 0x10000 for RPC servers that run on Windows 2000 Professional or Windows XP, and is 0x40000 for RPC servers that run on Windows 2000 Server or Windows Server 2003. RPC clients use 0x2000. For client ports, the number of calls is typically one, but may be higher if multiple asynchronous calls are in progress. If the resulting window size is less than one, it is set to one. If the resulting window size is greater than 32, it is set to 32. The **serial_num** field is set to the current value of **receive_serial_number**. The **selack_num**, **selack**, and **header** fragnum fields are set based on the fragments received, as specified in [C706] section 12. When an RPC receives a fragment with a length signifying a **maximum_pdu_length** larger than the current value in the send window, the implied length is calculated by rounding the total packet length down to the nearest multiple of 8. The activity's **maximum_pdu_length** is then set to the lower of this rounded value and the local transport limit. Therefore, the new value takes effect with the next call of the activity.

<68> Section 3.2.1.5.1: The Windows NT 4.0 client never calls **convc_indy**. The Windows NT 4.0 client does not implement **conv_who_are_you_auth_more**, and the Windows NT 4.0 server never calls **conv_who_are_you_auth_more**.

<69> Section 3.2.1.5.1: Windows NT 4.0 does not have support for Kerberos.

<70> Section 3.2.1.5.2: Windows NT 4.0 does not support multiple simultaneous active calls in a single activity. Versions of Windows later than Windows NT 4.0 support multiple simultaneous active calls in a single activity.

<71> Section 3.2.2.2.1: The timer interval is initially one second. When a call in **STATE_DISPATCHED** receives a WORKING packet or a NOCALL packet with a body that specifies a window size of zero, the timer interval is doubled. The interval is limited to a maximum of 32 seconds. In addition, when a call's **F_CANCELED** flag is set, the timer interval is limited to the max of two seconds or the cancel time out. If the timer expires, Windows considers the previously transmitted packet lost, and the client sends a new packet.

<72> Section 3.2.2.2.2: The timer interval is initially infinite and can be changed by the client application.

<73> Section 3.2.2.2.3: The timer interval is two seconds.

<74> Section 3.2.2.2.4: Windows NT 4.0 does not implement this timer. Windows 2000 and later versions of Windows use a single shared timer for all activities.

<75> Section 3.2.2.3.2: In Windows, Kerberos and the SPNEGO Protocol allow a higher-level client protocol to pass in a server principal name (SPN), which is semantically equivalent to the *target_name* parameter of the **gss_init_sec_context** call in the [GSS] model (as specified in [RFC2743]) and, optionally, a set of credentials to be used when contacting the server. If no credentials are supplied, the credentials associated with the log-on session for the currently executing thread are used. Details on log-on sessions and credentials are as specified in [MS-SECO].

<76> Section 3.2.2.5.2: Windows silently discards PING packets.

<77> Section 3.2.2.5.6: Windows follows the guidance specified in section 3.2.2.5.6. If the client has accepted six consecutive NOCALL packets containing a packet body with a send window greater than 0, the call state is changed to STATE_FAULT.

<78> Section 3.2.3.1.2: Windows maintains two rows representing the current context and the previous context.

<79> Section 3.2.3.1.3: In Windows NT 4.0, at most, one call can be in progress per activity. When a packet of a higher sequence number is accepted, the call with the lower sequence is canceled, and the higher number becomes the new lowest-allowed-sequence.

<80> Section 3.2.3.2.1: In Windows NT 4.0, the timer interval is always three seconds. In Windows 2000 and higher versions of Windows, the interval is effectively infinite: The server sends a burst of packets only in response to a client packet.

<81> Section 3.2.3.4.1: In Windows, the server implementation of the application protocol layer indicates to the RPC runtime that the error should be handled at the RPC protocol layer by raising an exception.

<82> Section 3.2.3.4.3: An administrator on the server machine can deploy authorization policies that restrict access to individual RPC servers or all RPC servers on that machine. In such cases, the server sends back to the client a fault PDU (as specified in [C706] section 12.5.3.5, The Fault PDU) where the status field of the fault PDU is set to status code 5.

<83> Section 3.2.3.5.3: Windows servers follow this clause, except that the **dc_rpc_cl_pkt_hdr_t.auth_proto check** is skipped when the PDU type is PING or the **maybe** flag is set in the **dc_rpc_cl_pkt_hdr_t.flags1** field.

<84> Section 3.2.3.5.4: Windows NT 4.0 has the following behavior when receiving this packet. Find or create an activity object for the activity ID in the header. If the activity's lowest-allowed-sequence number is higher than the packet sequence number, discard the packet. If no active call exists with the packet sequence, create a call with that sequence in STATE_INIT, and add it to the activity. Set the activity's lowest-allowed-sequence to the packet sequence. Process the packet according to the call state.

<85> Section 3.2.3.5.5: Windows servers answer the PING only if its serial number is higher than the serial number of any client packet previously seen in this call.

<86> Section 3.3.1.5.1: Servers return a PDU indicating an error depending on the received PDU with the invalid version number, as specified in section 3.3.3.5.7.

<87> Section 3.3.1.5.2.1: The table below lists the security providers that Windows assumes uses three legs, as specified in section 3.3.1.5.2.1:

Security Provider

- NT LAN Manager (NTLM)

- NetLogon

<88> Section 3.3.1.5.3: Windows–based clients starting with Windows Server 2003 SP1 and Windows Vista use security context multiplexing on this connection.

<89> Section 3.3.1.5.3: –based clients starting with Windows Vista support keeping the connection open after sending the orphaned PDU. Also, Windows–based servers starting with Windows Vista support keeping the connection open after receiving the orphaned PDU.

<90> Section 3.3.1.5.3: Windows Server 2003 SP1 and subsequent service packs and Windows Vista and Windows Server 2008 versions of Windows: These versions of Windows have support for bind time feature negotiation. The server uses the message processing rules in this section, and clients always indicate support for bind time feature negotiation and for security context multiplexing. For versions of Windows earlier than Windows Server 2003 SP1 and Windows Vista, the server uses the behavior specified in [C706], and the client does not indicate support for bind time feature negotiation and security context multiplexing. Windows allows a client to disable proposing use of the bind time feature negotiation through configuration.

<91> Section 3.3.1.5.4: Windows–based clients starting with Windows Server 2003 SP1 and subsequent service packs and Windows Vista and Windows Server 2008 versions of Windows do not send authentication information in this case. Windows–based servers for those versions of Windows still send authentication information in this case.

<92> Section 3.3.1.5.4: A Windows–based client that is capable of security context multiplexing does not build more than 1,000 security contexts per connection.

<93> Section 3.3.1.5.7: In some cases, Windows–based clients negotiate a transfer syntax in parallel with marshaling data using transfer syntax NDR. In such cases, the client always proposes NDR as one of the transfer syntaxes, and, if the server accepts a transfer syntax different from NDR, the client attempts to renegotiate transfer syntax NDR, which is used to send the requests already marshaled; but the server-accepted transfer syntax in the first negotiation is used for requests that have not started transfer syntax negotiation by the time the first negotiation completed.

<94> Section 3.3.1.5.9: Windows versions starting with Windows 2000 set PFC_CONC_MPX and support concurrent multiplexing on a connection.

<95> Section 3.3.1.5.11: In Windows releases earlier than Windows Vista, the RPC client closes the connection after sending the orphaned PDU.

<96> Section 3.3.2.2.1: Only NCACN_IP_TCP makes use of this timer. The RPC runtime on the client instructs the TCP/IP stack on the client to use a potentially smaller value than the default for the TCP keep-alives to monitor the state of the connection. The value used for the timer is determined by a higher-level protocol. A higher-level protocol passes a value between 0 and 10, and, starting from Windows 2000, the RPC runtime on the client uses these values as an indication of how long it should wait for a response from the server before it turns on keep-alives. The value passed in by a higher-level protocol is interpreted according to the following table.

Time-out parameter	Actual delay before turning on keep-alives (in seconds)
0 (RPC_C_BINDING_MIN_TIMEOUT)	120
1	240
2	360
3	480

Time-out parameter	Actual delay before turning on keep-alives (in seconds)
4	600
5(RPC_C_BINDING_DEFAULT_TIMEOUT)	720
6	840
7	960
8	1,080
9 (RPC_C_BINDING_MAX_TIMEOUT)	1,200
10 (RPC_C_BINDING_INFINITE_TIMEOUT)	Never

The default is time-out parameter 5. Once the keep-alives are turned on, the implementation of these extensions instruct the TCP/IP stack to send one keep-alive packet every second.

<97> Section 3.3.2.2.3: A Windows–based client considers a connection as idle if it has not been used for 10 seconds. If the number of connections in the association is more than 500, or the sum of the number of security contexts on all connections in the association is more than 500, the connection is considered idle if it has not been used for 5 seconds. The implementation is structured in a way that makes these time intervals optional and a lower bound only.

<98> Section 3.3.2.2.4: Windows NT Server 4.0 and Windows 2000 do not set this timer. Later versions of Windows 3.0 set this timer.

<99> Section 3.3.2.3.2: In Windows, most security providers allow a higher-level client protocol to pass in something called a server principal name (SPN) and, optionally, a set of credentials to be used when contacting the server. If no credentials are supplied, the credentials associated with the log-on session for the currently executing thread are used.

<100> Section 3.3.2.4.1.3: The RPC runtime on the client can obtain the credentials from a higher-level protocol that can supply a user name/domain/password, or it can use the implicit credentials of the log-on session that is attached to the thread on which the call is made.

<101> Section 3.3.2.5.1: Windows–based clients return error code 0x6c0 (**RPC_S_PROTOCOL_ERROR**) to the client application in this case.

0x6c0

<102> Section 3.3.2.6.4: The following table lists the Windows behavior for the various security providers:

Security provider	Security information applied for endpoint mapper requests
Kerberos	NT LAN Manager (NTLM)
NT LAN Manager (NTLM)	NT LAN Manager (NTLM)
Simple and Protected GSS-API Negotiation Mechanism	NT LAN Manager (NTLM)

Security provider	Security information applied for endpoint mapper requests
Netlogon	None

In Windows, the application of this protection is triggered through configuration or APIs available to higher layers.

<103> Section 3.3.3.2.1: Only NCACN_IP_TCP makes use of this timer. The RPC runtime on the server instructs the TCP/IP stack on the server to use a potentially smaller value than the default for the TCP keep-alives to monitor the state of the connection. The value used for the timer is determined by a higher-level protocol. A higher-level protocol passes a value between 0 and 10, and, starting from Windows Server 2003, the RPC runtime on the server uses these values as an indication of how long it should wait for a packet from the client before it turns on keep-alives. The value passed in by a higher-level protocol is interpreted according to the same table that is specified in section 3.3.2.2.1. The default is parameter value 5. Once the keep-alives are turned on, the implementation of these extensions instruct the TCP/IP stack to send one keep-alive packet every second.

<104> Section 3.3.3.3.1.3: The name of the security provider module is retrieved from the registry by using the authentication_type constant supplied by the higher-level protocol.

<105> Section 3.3.3.4.1: In Windows, the server implementation of the application protocol layer indicates to the RPC runtime that the error should be handled at the RPC protocol layer by raising an exception.

<106> Section 3.3.3.4.2: Windows–based servers never send shutdown packets.

<107> Section 3.3.3.4.3: On Windows, only NCACN_NP is capable of retrieving the identity of the client. The RPC runtime returns error 0x6e4 (RPC_S_CANNOT_SUPPORT) for all other RPC transports.

<108> Section 3.3.3.5.4: This behavior can be turned off by higher-level protocols or machine configuration. Note that the limit on Windows 2000 is 1 MB; Windows NT 4.0 does not implement such a limit.

<109> Section 3.3.3.5.6: This message handling is not present on Windows NT 4.0, Windows 2000, and Windows XP versions earlier than Service Pack 2.

8 Appendix C: RPC Extensions Conformance to [C706] Requirements

The following table documents the conformance of RPC extensions to the [C706] specification against the list of conformance requirements specified in [C706] section 1.3. In cases where these extensions do not conform to [C706], specific section references are provided only where the conformance failure is specified explicitly by an individual section. Conformance failures specified across numerous sections are not accompanied by lists of those sections.

Conformance requirements from [C706] section 1.3	Status of this specification
Implementations must support the endpoint selection rules in Endpoint Selection on page 23.	Conforms except for name service parts. Name service specification is Remote Procedure Call Location Services Protocol.
Implementations must support the manager selection rules in Interface and Manager Selection on page 24.	Conforms except for name service parts. Name service specification is Remote Procedure Call Location Services Protocol.
Implementations must support the search algorithm in section 2.4.5.	Name service specification is Remote Procedure Call Location Services Protocol'.
Implementations must support the API naming, syntax, and semantics, as defined in Chapter 3. Implementations may extend the set of status codes documented in Chapter 3.	The APIs in [C706] are implementation details that are not required under the Decision, and the Commission has indicated that Microsoft should not supply implementation details.
Implementations must support the naming, syntax, and semantics for IDL, as specified in Chapter 4.	Conforms to aspects that affect the protocol and interoperability except where noted in this specification in sections 2.2.4 through 2.2.7 and 3.1.1.5.
Implementations must support the naming, syntax, and semantics for stubs, as specified in Chapter 5.	Conforms except where noted in this specification in sections 2.2.4 through 2.2.7, 3.1.1.5, and 3.1.2.4.1.
Implementations must support the semantics specified in Chapter 6.	Conforms except where noted in this specification.
Implementations must support the NSI syntax and naming, as specified in section 6.2 on page 358.	Name service specification is Remote Procedure Call Location Services Protocol.
Implementations must support the service semantics specified in Chapter 7.	The service semantics in [C706] are implementation details that are not required under the Decision, and the Commission has indicated that Microsoft should not supply implementation details.
Implementations must follow the conformance rules specified in Chapter 9.	This specification redefines the state machines of [C706] in sections 3.2.1.1.1, 3.2.2.1, 3.2.3.1, 3.3.2 and 3.3.3.
Implementations must support the syntax of the PDU encodings in Chapter 12.	Conforms except where noted in this specification in sections 2.2.2 and 2.2.3.
Implementations must support the Authentication Verifier encodings, as specified in Chapter 13.	Conforms except where noted in this specification.

Conformance requirements from [C706] section 1.3	Status of this specification
Implementations must support the rules and encodings for NDR, as specified in Chapter 14.	Conforms except where noted in this specification in sections 2.2.4 through 2.2.7, 3.1.1.5, 3.1.2.4.1, 3.1.2.5.1 and 3.1.3.5.
Implementations must support the syntax, semantics, and encoding for UUIDs, as specified in Appendix A.	Conforms except where noted in this specification in section 2.2.1.1.3.
Implementations must support the naming and semantics for protocol sequence strings, as specified in Appendix B.	Conforms except where noted in this specification in section 2.1.
Implementations must support the naming and semantics for the name_syntax arguments, as specified in Appendix C.	Name service specification, as specified in Remote Procedure Call Location Services Protocol'.
Implementations must support the naming and semantics for security parameters, as specified in Appendix D.	Not conformant as specified in this specification.
Implementations must support the naming and encodings for comm_status and fault_status, as specified in Appendix E.	Conformant with respect to status codes sent between network nodes.
Implementations must support the mapping from IDL types to NDR types, and from NDR types to defined ISO C types, as specified in Appendix F.	The mappings in question are specified in [C706], and are performed locally and are implementation details that are not required under the Decision; the Commission has indicated that Microsoft should not supply implementation details.
Implementations must support the portable character set, as specified in Appendix G.	Conforms.
Implementations must use the endpoint mapper ports, as specified in Appendix H, for the corresponding protocols.	Conforms except where noted in this specification in section 2.1.
Implementations must adhere to the rules for protocol identifier assignment, as specified in Appendix I.	Not conformant, as specified in this specification in section 2.1.
Implementations must adhere to the mappings for directory service attributes, as specified in the DCE: Directory Services specification.	This specification does not extend or update the Directory Services specification.
Implementations must provide defaults for the protocol machine values specified in Appendix K.	This specification redefines the protocol state machines that are specified in [C706], in sections 3.2.1.1.1, 3.2.2.1, 3.2.3.1, 3.3.2 and 3.3.3. Some specific values from Appendix K are used in this specification where noted.
Implementations must obey the special protocol tower encoding rules specified in Appendix L.	Conforms.
Implementations must support the syntax and semantics of the dce_error_inq_text routine	The APIs specified in [C706] are implementation details that are not required under the Decision; the

Conformance requirements from [C706] section 1.3	Status of this specification
specified in Appendix M.	Commission has indicated that Microsoft should not supply implementation details.
Implementations must adhere to the mappings for transfer syntax UUIDs, as specified in Appendix N.	Conforms except where noted in this specification.
Implementations must support the endpoint mapper semantics, as specified in Appendix O.	Conforms except where noted in this specification in section 2.2.1.2.
Implementations must support the conversation manager semantics, as specified in Appendix P.	Conforms except where noted in this specification in section 3.2.
Implementations must support the remote management semantics, as specified in Appendix Q.	Conforms except where noted in this specification in section 2.2.1.3.
Implementations must register all protocol sequences with OSF, as specified in [C706] Appendix B.	The protocol sequence strings listed in section 2.1 that are not already specified in [C706] Appendix B have not been registered with OSF.

9 Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W