[MS-OXCRPC]:

Wire Format Protocol

Intellectual Property Rights Notice for Open Specifications Documentation

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

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

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

Revision Summary

Date	Revision History	Revision Class	Comments
4/4/2008	0.1		Initial Availability.
4/25/2008	0.2		Revised and updated property names and other technical content.
6/27/2008	1.0		Initial Release.
8/6/2008	1.01		Revised and edited technical content.
9/3/2008	1.02		Revised and edited technical content.
10/1/2008	1.03		Revised and edited technical content.
12/3/2008	1.04		Revised and edited technical content.
3/4/2009	1.05		Revised and edited technical content.
4/10/2009	2.0		Updated technical content and applicable product releases.
7/15/2009	3.0	Major	Revised and edited for technical content.
11/4/2009	4.0.0	Major	Updated and revised the technical content.
2/10/2010	5.0.0	Major	Updated and revised the technical content.
5/5/2010	6.0.0	Major	Updated and revised the technical content.
8/4/2010	7.0	Major	Significantly changed the technical content.
11/3/2010	7.1	Minor	Clarified the meaning of the technical content.
3/18/2011	7.1	No change	No changes to the meaning, language, or formatting of the technical content.
8/5/2011	8.0	Major	Significantly changed the technical content.
10/7/2011	9.0	Major	Significantly changed the technical content.
1/20/2012	10.0	Major	Significantly changed the technical content.
4/27/2012	11.0	Major	Significantly changed the technical content.
7/16/2012	11.1	Minor	Clarified the meaning of the technical content.
10/8/2012	12.0	Major	Significantly changed the technical content.
2/11/2013	13.0	Major	Significantly changed the technical content.
7/26/2013	13.1	Minor	Clarified the meaning of the technical content.
11/18/2013	13.1	No Change	No changes to the meaning, language, or formatting of the technical content.
2/10/2014	13.1	No Change	No changes to the meaning, language, or formatting of the technical content.
4/30/2014	14.0	Major	Significantly changed the technical content.
7/31/2014	14.1	Minor	Clarified the meaning of the technical content.

Date	Revision History	Revision Class	Comments
10/30/2014	15.0	Major	Significantly changed the technical content.
3/16/2015	16.0	Major	Significantly changed the technical content.
5/26/2015	17.0	Major	Significantly changed the technical content.
9/14/2015	17.0	No Change	No changes to the meaning, language, or formatting of the technical content.

Table of Contents

1 Introduction 7
1.1 Glossary 7
1.2 References 9
1.2.1 Normative References 10
1.2.2 Informative References 10
1.3 Overview 11
1.4 Relationship to Other Protocols 13
1.5 Prerequisites/Preconditions 13
1.6 Applicability Statement 13
1.7 Versioning and Capability Negotiation 13
1.8 Vendor-Extensible Fields 14
1.9 Standards Assignments 14

2 Messages 15
2.1 Transport 15
2.2 Common Data Types 15
2.2.1 Simple Data Types 16
2.2.1.1 CXH Data Type 16
2.2.1.2 ACXH Data Type 16
2.2.1.3 BIG_RANGE_ULONG Data Type 16
2.2.1.4 SMALL_RANGE_ULONG Data Type 17
2.2.2 Structures 17
2.2.2.1 RPC_HEADER_EXT Structure 17
2.2.2.2 AUX_HEADER Structure 17
2.2.2.2.1 AUX_PERF_REQUESTID Auxiliary Block Structure 20
2.2.2.2.2 AUX_PERF_SESSIONINFO Auxiliary Block Structure 21
2.2.2.2.3 AUX_PERF_SESSIONINFO_V2 Auxiliary Block Structure 21
2.2.2.2.4 AUX_PERF_CLIENTINFO Auxiliary Block Structure 22
2.2.2.2.5 AUX_PERF_SERVERINFO Auxiliary Block Structure 24
2.2.2.2.6 AUX_PERF_PROCESSINFO Auxiliary Block Structure 24
2.2.2.2.7 AUX_PERF_DEFMDB_SUCCESS Auxiliary Block Structure 25
2.2.2.2.8 AUX_PERF_DEFGC_SUCCESS Auxiliary Block Structure 26
2.2.2.2.9 AUX_PERF_MDB_SUCCESS Auxiliary Block Structure 26
2.2.2.2.10 AUX_PERF_MDB_SUCCESS_V2 Auxiliary Block Structure 27
2.2.2.2.11 AUX_PERF_GC_SUCCESS Auxiliary Block Structure 27
2.2.2.2.12 AUX_PERF_GC_SUCCESS_V2 Auxiliary Block Structure 28
2.2.2.2.13 AUX_PERF_FAILURE Auxiliary Block Structure 29
2.2.2.2.14 AUX_PERF_FAILURE_V2 Auxiliary Block Structure 29
2.2.2.2.15 AUX_CLIENT_CONTROL Auxiliary Block Structure 30
2.2.2.2.16 AUX_OSVERSIONINFO Auxiliary Block Structure 31
2.2.2.2.17 AUX_EXORGINFO Auxiliary Block Structure 32
2.2.2.2.18 AUX_PERF_ACCOUNTINFO Auxiliary Block Structure 32
2.2.2.2.19 AUX_ENDPOINT_CAPABILITIES Auxiliary Block Structure 32
2.2.2.2.20 AUX_CLIENT_CONNECTION_INFO Auxiliary Block Structure 33
2.2.2.2.21 AUX_SERVER_SESSION_INFO Auxiliary Block Structure 34
2.2.2.2.22 AUX_PROTOCOL_DEVICE_IDENTIFICATION Auxiliary Block Structure 34

3 Protocol Details 37
3.1 EMSMDB Server Details 37
3.1.1 Abstract Data Model 37
3.1.1.1 Global.Handle 37
3.1.2 Timers 38
3.1.3 Initialization 38
3.1.4 Message Processing Events and Sequencing Rules 38
3.1.4.1 EcDoConnectEx Method (Opnum 10) 40

3.1.4.1.1 Extended Buffer Handling 44
3.1.4.1.1.1 Extended Buffer Format 44
3.1.4.1.1.1.1 rgbAuxIn Input Buffer 44
3.1.4.1.1.1.2 rgbAuxOut Output Buffer 45
3.1.4.1.1.2 Compression Algorithm 45
3.1.4.1.1.2.1 LZ77 Compression Algorithm 45
3.1.4.1.1.2.1.1 Compression Algorithm Terminology 45
3.1.4.1.1.2.1.2 Using the Compression Algorithm 46
3.1.4.1.1.2.1.3 Compression Process 46
3.1.4.1.1.2.1.4 Compression Process Example 46
3.1.4.1.1.2.2 DIRECT2 Encoding Algorithm 47
3.1.4.1.1.2.2.1 Bitmask 47
3.1.4.1.1.2.2.2 Encoding Metadata 48
3.1.4.1.1.2.2.3 Metadata Offset 48
3.1.4.1.1.2.2.4 Match Length 48
3.1.4.1.1.3 Obfuscation Algorithm 50
3.1.4.1.2 Auxiliary Buffer 50
3.1.4.1.2.1 Server Topology Information 51
3.1.4.1.2.2 Processing Auxiliary Buffers Received from the Client 51
3.1.4.1.3 Version Checking 51
3.1.4.1.3.1 Version Number Comparison 52
3.1.4.1.3.2 Server Versions 52
3.1.4.2 EcDoRpcExt2 Method (Opnum 11) 53
3.1.4.2.1 Extended Buffer Handling 55
3.1.4.2.1.1 Extended Buffer Format 55
3.1.4.2.1.1.1 rgbIn Input Buffer 56
3.1.4.2.1.1.2 rgbOut Output Buffer 56
3.1.4.2.1.1.3 rgbAuxIn Input Buffer 57
3.1.4.2.1.1.4 rgbAuxOut Output Buffer 57
3.1.4.2.1.2 Extended Buffer Packing 57
3.1.4.2.2 Auxiliary Buffer 58
3.1.4.2.2.1 Server Topology Information 58
3.1.4.2.2.2 Processing Auxiliary Buffers Received from the Client 58
3.1.4.3 EcDoDisconnect Method (Opnum 1) 58
3.1.4.4 EcDoAsyncConnectEx Method (Opnum 14) 59
3.1.4.5 EcRRegisterPushNotification Method (Opnum 4) 59
3.1.4.6 EcDummyRpc Method (Opnum 6) 61
3.1.4.7 Opnum0NotUsedOnWire Method (Opnum 0) 61
3.1.4.8 Opnum2NotUsedOnWire Method (Opnum 2) 61
3.1.4.9 Opnum3NotUsedOnWire Method (Opnum 3) 61
3.1.4.10 Opnum5NotUsedOnWire Method (Opnum 5) 61
3.1.4.11 Opnum7NotUsedOnWire Method (Opnum 7) 61
3.1.4.12 Opnum8NotUsedOnWire Method (Opnum 8) 62
3.1.4.13 Opnum9NotUsedOnWire Method (Opnum 9) 62
3.1.4.14 Opnum12NotUsedOnWire Method (Opnum 12) 62
3.1.4.15 Opnum13NotUsedOnWire Method (Opnum 13) 62
3.1.5 Timer Events 62
3.1.6 Other Local Events 62
3.2 EMSMDB Client Details 62
3.2.1 Abstract Data Model 62
3.2.2 Timers 62
3.2.3 Initialization 62
3.2.4 Message Processing Events and Sequencing Rules 63
3.2.4.1 Sending the EcDoConnectEx Method 63
3.2.4.1.1 Extended Buffer Handling 64
3.2.4.1.2 Auxiliary Buffer 64
3.2.4.1.2.1 Client Performance Monitoring 65
3.2.4.1.2.2 Processing Auxiliary Buffers Received from the Server 66

3.2.4.1.3 Version Checking 66
3.2.4.1.3.1 Version Number Comparison 66
3.2.4.1.3.2 Client Versions 66
3.2.4.1.3.3 Version Numbers Received from the Server 67
3.2.4.2 Sending the EcDoRpcExt2 Method 67
3.2.4.2.1 Extended Buffer Handling 67
3.2.4.2.2 Auxiliary Buffer 68
3.2.4.2.2.1 Client Performance Monitoring 68
3.2.4.3 Sending the EcDoDisconnect Method 70
3.2.4.4 Handling Server Too Busy 71
3.2.4.5 Handling Connection Failures 71
3.2.4.6 Handling Endpoint Consolidation 71
3.2.5 Timer Events 71
3.2.6 Other Local Events 71
3.3 AsyncEMSMDB Server Details 71
3.3.1 Abstract Data Model 71
3.3.2 Timers 72
3.3.3 Initialization 72
3.3.4 Message Processing Events and Sequencing Rules 72
3.3.4.1 EcDoAsyncWaitEx Method (Opnum 0) 73
3.3.5 Timer Events 74
3.3.6 Other Local Events 74
3.4 AsyncEMSMDB Client Details 74
3.4.1 Abstract Data Model 74
3.4.2 Timers 74
3.4.3 Initialization 74
3.4.4 Message Processing Events and Sequencing Rules 74
3.4.5 Timer Events 75
3.4.6 Other Local Events 75

4 Protocol Examples 76
4.1 Connect to the Server 76
4.2 Issue ROP Commands to the Server 77
4.3 Receive Packed ROP Responses from the Server 79
4.4 Disconnect from the Server 80

5 Security 81
5.1 Security Considerations for Implementers 81
5.2 Index of Security Parameters 81

6 Appendix A: Full IDL 82

7 Appendix B: Product Behavior 84

8 Change Tracking 90

9 Index 91

1 Introduction

The Wire Format Protocol is used by a client to communicate with a server to access personal messaging data by using **remote procedure call (RPC)** interfaces. The Wire Format Protocol uses the **EMSMDB** and **AsyncEMSMDB** protocol interfaces between a client and server. This protocol extends DCE 1.1: Remote Procedure Call, as described in [C706].

Sections 1.8, 2, and 3 of this specification are normative and can contain the terms MAY, SHOULD, MUST, MUST NOT, and SHOULD NOT as defined in [RFC2119]. Sections 1.5 and 1.9 are also normative but do not contain those terms. All other sections and examples in this specification are informative.

1.1 Glossary

The following terms are specific to this document:

asynchronous context handle: A **remote procedure call (RPC)** context handle that is used by a client when issuing RPCs against a server on AsyncEMSMDB interface methods. It represents a handle to a unique session context on the server.

binding handle: A data structure that represents the logical connection between a client and a server.

Client Access License (CAL): A license that gives a user the right to access the services of a server. To legally access the server software, a CAL can be required. A CAL is not a software product.

code page: An ordered set of characters of a specific script in which a numerical index (code-point value) is associated with each character. Code pages are a means of providing support for character sets (1) and keyboard layouts used in different countries. Devices such as the display and keyboard can be configured to use a specific code page and to switch from one code page (such as the United States) to another (such as Portugal) at the user's request.

distinguished name (DN): A name that uniquely identifies an object by using the relative distinguished name (RDN) for the object, and the names of container objects and domains that contain the object. The distinguished name (DN) identifies the object and its location in a tree.

endpoint: A communication port that is exposed by an application server for a specific shared service and to which messages can be addressed.

flags: A set of values used to configure or report options or settings.

globally unique identifier (GUID): A term used interchangeably with **universally unique identifier (UUID)** in Microsoft protocol technical documents (TDs). Interchanging the usage of these terms does not imply or require a specific algorithm or mechanism to generate the value. Specifically, the use of this term does not imply or require that the algorithms described in [RFC4122] or [C706] must be used for generating the **GUID**. See also **universally unique identifier (UUID)**.

handle: Any token that can be used to identify and access an object such as a device, file, or a window.

Hypertext Transfer Protocol (HTTP): An application-level protocol for distributed, collaborative, hypermedia information systems (text, graphic images, sound, video, and other multimedia files) on the World Wide Web.

Incremental Change Synchronization (ICS): A data format and algorithm that is used to synchronize folders and messages between two sources.

Interface Definition Language (IDL): The International Standards Organization (ISO) standard language for specifying the interface for remote procedure calls. For more information, see [C706] section 4.

Kerberos: An authentication (2) system that enables two parties to exchange private information across an otherwise open network by assigning a unique key (called a ticket) to each user that logs on to the network and then embedding these tickets into messages sent by the users. For more information, see [MS-KILE].

little-endian: Multiple-byte values that are byte-ordered with the least significant byte stored in the memory location with the lowest address.

locale: A collection of rules and data that are specific to a language and a geographical area. A locale can include information about sorting rules, date and time formatting, numeric and monetary conventions, and character classification.

mailbox: A **message store** that contains email, calendar items, and other Message objects for a single recipient.

message store: A unit of containment for a single hierarchy of Folder objects, such as a mailbox or public folders.

name service provider interface (NSPI): A method of performing address-book-related operations on Active Directory.

Network Data Representation (NDR): A specification that defines a mapping from **Interface Definition Language (IDL)** data types onto octet streams. **NDR** also refers to the runtime environment that implements the mapping facilities (for example, data provided to **NDR**). For more information, see [MS-RPCE] and [C706] section 14.

NT LAN Manager (NTLM) Authentication Protocol: A protocol using a challenge-response mechanism for authentication (2) in which clients are able to verify their identities without sending a password to the server. It consists of three messages, commonly referred to as Type 1 (negotiation), Type 2 (challenge) and Type 3 (authentication). For more information, see [MS-NLMP].

opnum: An operation number or numeric identifier that is used to identify a specific **remote procedure call (RPC)** method or a method in an interface. For more information, see [C706] section 12.5.2.12 or [MS-RPCE].

permission: A rule that is associated with an object and that regulates which users can gain access to the object and in what manner. See also rights.

public folder: A Folder object that is stored in a location that is publicly available.

recipient: An entity that can receive email messages.

remote operation (ROP): An operation that is invoked against a server. Each ROP represents an action, such as delete, send, or query. A ROP is contained in a ROP buffer for transmission over the wire.

remote procedure call (RPC): A context-dependent term commonly overloaded with three meanings. Note that much of the industry literature concerning RPC technologies uses this term interchangeably for any of the three meanings. Following are the three definitions: (*) The runtime environment providing remote procedure call facilities. The preferred usage for this meaning is "RPC runtime". (*) The pattern of request and response message exchange between two parties (typically, a client and a server). The preferred usage for this meaning is "RPC exchange". (*) A single message from an exchange as defined in the previous definition. The preferred usage for this term is "RPC message". For more information about RPC, see [C706].

replica: A copy of the data that is in a user's **mailbox** at a specific point in time.

ROP request: See **ROP request buffer**.

ROP request buffer: A ROP buffer that a client sends to a server to be processed.

ROP response: See **ROP response buffer**.

ROP response buffer: A ROP buffer that a server sends to a client to be processed.

RPC dynamic endpoint: A network-specific server address that is requested and assigned at run time, as described in [C706].

RPC protocol sequence: A character string that represents a valid combination of a **remote procedure call (RPC)** protocol, a network layer protocol, and a transport layer protocol, as described in [C706] and [MS-RPCE].

Server object: An object on a server that is used as input or created as output for **remote operations (ROPs)**.

Session Context: A server-side partitioning for client isolation. All client actions against a server are scoped to a specific Session Context. All messaging objects and data that is opened by a client are isolated to a Session Context.

session context handle: A **remote procedure call (RPC)** context handle that is used by a client when issuing RPCs against a server on EMSMDB interface methods. It represents a handle to a unique session context on the server.

stream: (1) An element of a compound file, as described in [MS-CFB]. A stream contains a sequence of bytes that can be read from or written to by an application, and they can exist only in storages.

(2) A flow of data from one host to another host, or the data that flows between two hosts.

Unicode: A character encoding standard developed by the Unicode Consortium that represents almost all of the written languages of the world. The **Unicode** standard [UNICODE5.0.0/2007] provides three forms (UTF-8, UTF-16, and UTF-32) and seven schemes (UTF-8, UTF-16, UTF-16 BE, UTF-16 LE, UTF-32, UTF-32 LE, and UTF-32 BE).

universally unique identifier (UUID): A 128-bit value. UUIDs can be used for multiple purposes, from tagging objects with an extremely short lifetime, to reliably identifying very persistent objects in cross-process communication such as client and server interfaces, manager entry-point vectors, and **RPC** objects. UUIDs are highly likely to be unique. UUIDs are also known as **globally unique identifiers (GUIDs)** and these terms are used interchangeably in the Microsoft protocol technical documents (TDs). Interchanging the usage of these terms does not imply or require a specific algorithm or mechanism to generate the UUID. Specifically, the use of this term does not imply or require that the algorithms described in [RFC4122] or [C706] must be used for generating the UUID.

well-known endpoint: A preassigned, network-specific, stable address for a particular client/server instance. For more information, see [C706].

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

1.2 References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents

in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the Errata.

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.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, https://www2.opengroup.org/ogsys/catalog/c706

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

[MS-OXCFXICS] Microsoft Corporation, "Bulk Data Transfer Protocol".

[MS-OXCNOTIF] Microsoft Corporation, "Core Notifications Protocol".

[MS-OXCROPS] Microsoft Corporation, "Remote Operations (ROP) List and Encoding Protocol".

[MS-OXCSTOR] Microsoft Corporation, "Store Object Protocol".

[MS-OXDSCLI] Microsoft Corporation, "Autodiscover Publishing and Lookup Protocol".

[MS-OXOMSG] Microsoft Corporation, "Email Object Protocol".

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

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

[UASDC] Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression", May 1977, http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf

1.2.2 Informative References

[MS-OXABREF] Microsoft Corporation, "Address Book Name Service Provider Interface (NSPI) Referral Protocol".

[MS-OXNSPI] Microsoft Corporation, "Exchange Server Name Service Provider Interface (NSPI) Protocol".

[MS-OXPROTO] Microsoft Corporation, "Exchange Server Protocols System Overview".

[MSDN-RpcBindingSetAuthInfoEx] Microsoft Corporation, "RpcBindingSetAuthInfoEx function", http://msdn.microsoft.com/en-us/library/aa375608(v=VS.85).aspx

[MSDN-SOCKADDR] Microsoft Corporation, "sockaddr", http://msdn.microsoft.com/en-us/library/ms740496.aspx

[MSFT-ConfigStaticUDPPort] Microsoft Corporation, "Configure a Static UDP Port for Push Notifications in an Exchange 2010 Environment (en-US)", http://social.technet.microsoft.com/wiki/contents/articles/2542.configure-a-static-udp-port-for-push-notifications-in-an-exchange-2010-environment.aspx

1.3 Overview

The Wire Format Protocol enables a client to communicate with a server to access personal messaging data. Communications with the server are divided into three major functional areas: (1) initiating and establishing connection with the server, (2) issuing **remote operations (ROPs)** to the server for **mailbox** data, and (3) terminating communications with the server. This functionality is contained in the **EMSMDB** interface, as described in section 3.1 and section 3.2. If events are pending on the server that require client action, the client gets notification of those pending events by using the functionality contained in the **AsyncEMSMDB** interface, as described in section 3.3 and section 3.4.

The following figure shows a simplified overview of client and server communications.

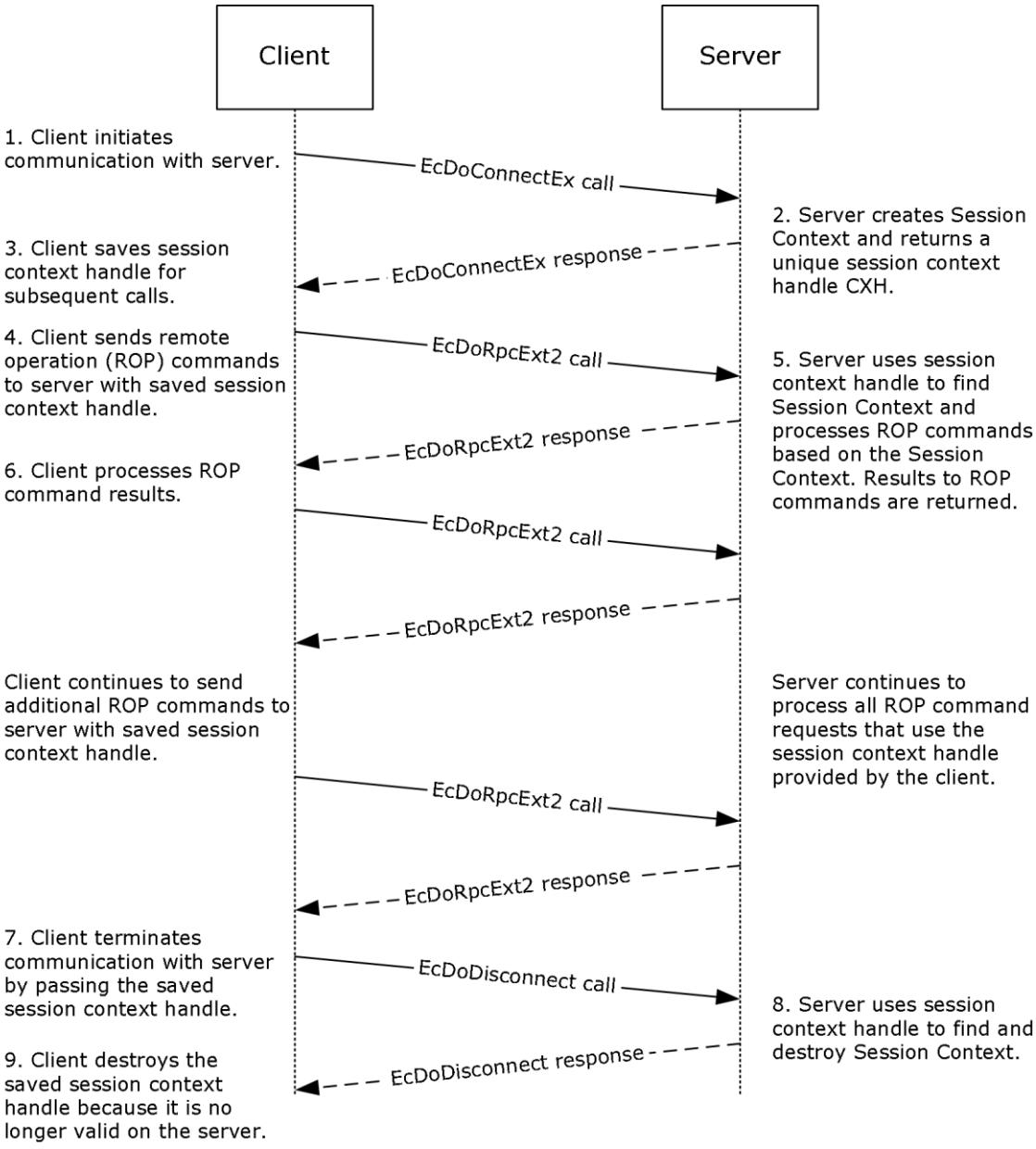


Figure 1: Client/server communications

Before a client can retrieve private mailbox or **public folder** data from a server on the **EMSMDB** interface, it first connects with the server, as described in section 3.2.4.1, and establishes a **session**

context handle. The session context handle is an RPC context **handle** that refers to the **Session Context** created by the server. The client stores this session context handle and uses it on subsequent RPCs on the **EMSMDB** interface.

After the server has returned the session context handle to the client, as described in section 3.1.4.1, the client begins issuing ROPs to the server. The client retrieves private mailbox or public folder data by using the method described in section 3.2.4.2. This single interface function is used to submit a group of ROP commands to the server, and there are no separate interface functions to perform different operations against mailbox data. The **ROP request** operations are tokenized into a request buffer and sent to the server as a byte array. The server parses the **ROP request buffer** and performs actions. The response to these actions is then serialized into a **ROP response buffer** and returned to the client as a byte array. At the **EMSMDB** interface level, the format of these ROP request buffers and ROP response buffers is not understood. For more information about ROP commands and how to interpret the ROP buffers, see [MS-OXCROPS].

To receive notification that events are available on the server related to the Session Context, the client establishes an asynchronous connection to the server to support notification, as described in sections 3.4 and 3.1.4.4. Using the asynchronous context handle returned by the server, the client uses the **AsyncEMSMDB** interface to instruct the server to return notification of an event, as described in section 3.4.4 and section 3.3.4.1.

When the client is finished with the session, the client disconnects from the server as described in section 3.2.4.3.

1.4 Relationship to Other Protocols

This protocol is dependent upon RPC, as described in [C706] and [MS-RPCE], and various network protocol sequences for its transport, as specified in section 2.1.

For conceptual background information and overviews of the relationships and interactions between this and other protocols, see [MS-OXPROTO].

1.5 Prerequisites/Preconditions

The Wire Format Protocol consists of the **EMSMDB** and **AsyncEMSMDB** RPC interfaces and has the same prerequisites as described in [MS-RPCE].

It is assumed that a client has obtained the name of a server that supports this protocol before these interfaces are invoked. How a client accomplishes this task is outside the scope of this specification.

1.6 Applicability Statement

This protocol is applicable to environments that require access to private mailbox and/or public folder messaging end-user data.

1.7 Versioning and Capability Negotiation

This specification covers versioning issues in the following areas:

- **Supported Transports:** This protocol uses multiple **RPC protocol sequences** as described in section 2.1.
- **Protocol Versions:** The **EMSMDB** interface has a single version number of 0.81 and has been extended by adding methods as specified in section 3.1. The **AsyncEMSMDB** interface has a single version number of 0.01.

- **Security and Authentication Methods:** This protocol supports the following authentication methods: **NT LAN Manager (NTLM) Authentication Protocol**, **Kerberos**, and Negotiate. These authentication methods are described in sections 3.1.3 and 3.3.3.
- **Capability Negotiation:** The Ethernet protocol does not support negotiation of the interface version to use. Instead, an implementation is configured with the interface version to use, as described in this specification.

1.8 Vendor-Extensible Fields

None.

1.9 Standards Assignments

This protocol uses the interface entry points and **HTTP** ports listed in the following table.

Parameter	Value	Reference
EMSMDB RPC interface **universally unique identifier (UUID)**	A4F1DB00-CA47-1067-B31F-00DD010662DA	Section 3.1
AsyncEMSMDB RPC interface UUID	5261574A-4572-206E-B268-6B199213B4E4	Section 3.3
RPC over HTTP protocol sequence **endpoint**	6001	Section 2.1

2 Messages

2.1 Transport

This protocol works over the following RPC protocol sequences: <1>

- ncacn_ip_tcp
- ncacn_http

For the network protocol sequence ncacn_http, this protocol MUST use the **well-known endpoint** 6001.

For ncacn_ip_tcp, this protocol MUST use **RPC dynamic endpoints**, as defined in Part 4 of [C706].

This protocol MUST use the UUID specified in section 1.9.

This protocol allows any user to establish an authenticated connection to the RPC server by using an authentication method as specified in [MS-RPCE]. The protocol uses the underlying RPC protocol to retrieve the identity of the caller that made the method call, as specified in [MS-RPCE]. The server uses this identity to perform method-specific access checks.

2.2 Common Data Types

This protocol uses the RPC base types and definitions specified in [C706] and [MS-RPCE], plus additional data types and structures that are defined in section 2.2.1.1 through section 2.2.2.2.22.

The following table lists the types and structures that are defined in this specification. Any structure that is not defined in this specification is reserved and MUST be ignored by the client.

Type	Type name
Simple Data Type	**CXH** (section 2.2.1.1)
Simple Data Type	**ACXH** (section 2.2.1.2)
Simple Data Type	**BIG_RANGE_ULONG** (section 2.2.1.3)
Simple Data Type	**SMALL_RANGE_ULONG** (section 2.2.1.4)
Structure	**RPC_HEADER_EXT** (section 2.2.2.1)
Structure	**AUX_HEADER** (section 2.2.2.2)
Structure	**AUX_PERF_REQUESTID** (section 2.2.2.2.1)
Structure	**AUX_PERF_SESSIONINFO** (section 2.2.2.2.2)
Structure	**AUX_PERF_SESSIONINFO_V2** (section 2.2.2.2.3)
Structure	**AUX_PERF_CLIENTINFO** (section 2.2.2.2.4)
Structure	**AUX_PERF_SERVERINFO** (section 2.2.2.2.5)
Structure	**AUX_PERF_PROCESSINFO** (section 2.2.2.2.6)
Structure	**AUX_PERF_DEFMDB_SUCCESS** (section 2.2.2.2.7)
Structure	**AUX_PERF_DEFGC_SUCCESS** (section 2.2.2.2.8)

Type	Type name
Structure	**AUX_PERF_MDB_SUCCESS** (section 2.2.2.2.9)
Structure	**AUX_PERF_MDB_SUCCESS_V2** (section 2.2.2.2.10)
Structure	**AUX_PERF_GC_SUCCESS** (section 2.2.2.2.11)
Structure	**AUX_PERF_GC_SUCCESS_V2** (section 2.2.2.2.12)
Structure	**AUX_PERF_FAILURE** (section 2.2.2.2.13)
Structure	**AUX_PERF_FAILURE_V2** (section 2.2.2.2.14)
Structure	**AUX_CLIENT_CONTROL** (section 2.2.2.2.15)
Structure	**AUX_OSVERSIONINFO** (section 2.2.2.2.16)
Structure	**AUX_EXORGINFO** (section 2.2.2.2.17)
Structure	**AUX_PERF_ACCOUNTINFO** (section 2.2.2.2.18)
Structure	**AUX_ENDPOINT_CAPABILITIES** (section 2.2.2.2.19)
Structure	**AUX_CLIENT_CONNECTION_INFO** (section 2.2.2.2.20)
Structure	**AUX_SERVER_SESSION_INFO** (section 2.2.2.2.21)
Structure	**AUX_PROTOCOL_DEVICE_IDENTIFICATION** (section 2.2.2.2.22)

2.2.1 Simple Data Types

The **Interface Definition Language (IDL)** for this protocol, as given in section 6, identifies four Simple Data Types, which are defined in section 2.2.1.1 through section 2.2.1.4.

2.2.1.1 CXH Data Type

The **CXH** data type is a session context handle to be used with an **EMSMDB** interface, as specified in section 3.1 and section 3.2.

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

2.2.1.2 ACXH Data Type

The **AXCH** data type is an **asynchronous context handle** to be used with an **AsyncEMSMDB** interface, as specified in section 3.3 and section 3.4.

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

2.2.1.3 BIG_RANGE_ULONG Data Type

The **BIG_RANGE_ULONG** data type is an unsigned long that MUST be between 0x0 and 0x40000.

```
typedef [range(0x0, 0x40000)] unsigned long BIG_RANGE_ULONG;
```

2.2.1.4 SMALL_RANGE_ULONG Data Type

The **SMALL_RANGE_ULONG** data type is an unsigned long that MUST be between 0x0 and 0x1008.

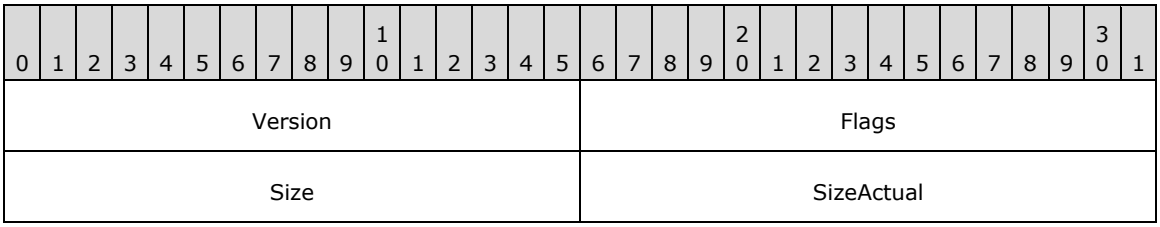
```
typedef [range(0x0, 0x1008)] unsigned long SMALL_RANGE_ULONG;
```

2.2.2 Structures

Unless otherwise specified, buffers and fields in section 2.2.2.1 through section 2.2.2.2.22 are depicted in **little-endian** byte order.

2.2.2.1 RPC_HEADER_EXT Structure

The **RPC_HEADER_EXT** structure provides information about the payload that follows.



Version (2 bytes): The version of the structure. This value MUST be set to 0x0000.

Flags (2 bytes): The **flags** that specify how data that follows this header MUST be interpreted. The flags in the following table are valid.

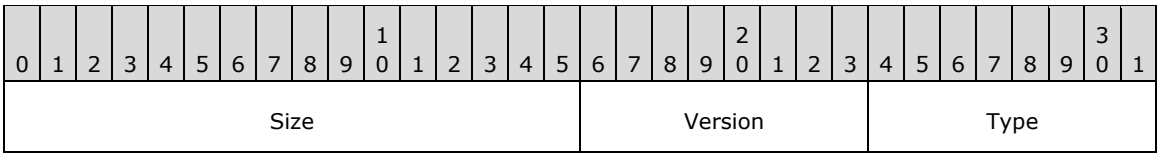
Flag name	Value	Meaning
Compressed	0x0001	The data that follows the **RPC_HEADER_EXT** structure is compressed. The size of the data when uncompressed is in the **SizeActual** field. If this flag is not set, the **Size** and **SizeActual** fields MUST be the same. If this flag is set, the value of the **Size** field MUST be less than the value of the **SizeActual** field.
XorMagic	0x0002	The data following the **RPC_HEADER_EXT** structure has been obfuscated. For more details about the obfuscation algorithm, see section 3.1.4.1.1.3.
Last	0x0004	No other **RPC_HEADER_EXT** structure follows the data of the current **RPC_HEADER_EXT** structure. This flag indicates that there are multiple buffers, each with its own **RPC_HEADER_EXT**, one after the other.

Size (2 bytes): The total length of the payload data that follows the **RPC_HEADER_EXT** structure. This length does not include the length of the **RPC_HEADER_EXT** structure.

SizeActual (2 bytes): The length of the payload data after it has been uncompressed. This field is only useful if the **Compressed** flag is set in the **Flags** field. If the **Compressed** flag is not set, this value MUST be equal to the value of the **Size** field.

2.2.2.2 AUX_HEADER Structure

The **AUX_HEADER** structure provides information about the auxiliary block structures that follow it.



Size (2 bytes): The size of the **AUX_HEADER** structure plus any additional payload data that follows.

Version (1 byte): The version information of the payload data that follows the **AUX_HEADER** structure. This value in conjunction with the **Type** field determines which structure to use to interpret the data that follows the header.

Version	Value
AUX_VERSION_1	0x01
AUX_VERSION_2	0x02

Type (1 byte): The type of auxiliary block data structure that follows the **AUX_HEADER** structure. The value of the **Type** field in conjunction with the **Version** field determines which auxiliary block structure to use to interpret the data that follows the **AUX_HEADER** structure. Several of the types distinguish among the client's foreground request (FG), the client's background request (BG), and the client's global catalog request (GC). A foreground request is a request where the client is waiting for a response from the server before continuing. A background request is a request where the client is operating in cached mode versus online. A global catalog request is a client request sent to the mailbox directory.

The block type names, associated **Type** field values, and the corresponding auxiliary block structure that follows the **AUX_HEADER** structure when the **Version** field is **AUX_VERSION_1** are listed in the following table.

Type name	Value	Auxiliary block structure
AUX_TYPE_PERF_REQUESTID	0x01	**AUX_PERF_REQUESTID** (section 2.2.2.2.1)
AUX_TYPE_PERF_CLIENTINFO	0x02	**AUX_PERF_CLIENTINFO** (section 2.2.2.2.4)
AUX_TYPE_PERF_SERVERINFO	0x03	**AUX_PERF_SERVERINFO** (section 2.2.2.2.5)
AUX_TYPE_PERF_SESSIONINFO	0x04	**AUX_PERF_SESSIONINFO** (section 2.2.2.2.2)
AUX_TYPE_PERF_DEFMDB_SUCCESS	0x05	**AUX_PERF_DEFMDB_SUCCESS** (section 2.2.2.2.7)
AUX_TYPE_PERF_DEFGC_SUCCESS	0x06	**AUX_PERF_DEFGC_SUCCESS** (section 2.2.2.2.8)
AUX_TYPE_PERF_MDB_SUCCESS	0x07	**AUX_PERF_MDB_SUCCESS** (section 2.2.2.2.9)
AUX_TYPE_PERF_GC_SUCCESS	0x08	**AUX_PERF_GC_SUCCESS** (section 2.2.2.2.11)
AUX_TYPE_PERF_FAILURE	0x09	**AUX_PERF_FAILURE**

Type name	Value	Auxiliary block structure
		(section 2.2.2.2.13)
AUX_TYPE_CLIENT_CONTROL	0x0A	**AUX_CLIENT_CONTROL** (section 2.2.2.2.15)
AUX_TYPE_PERF_PROCESSINFO	0x0B	**AUX_PERF_PROCESSINFO** (section 2.2.2.2.6)
AUX_TYPE_PERF_BG_DEFMDB_SUCCESS	0x0C	**AUX_PERF_DEFMDB_SUCCESS**
AUX_TYPE_PERF_BG_DEFGC_SUCCESS	0x0D	**AUX_PERF_DEFGC_SUCCESS**
AUX_TYPE_PERF_BG_MDB_SUCCESS	0x0E	**AUX_PERF_MDB_SUCCESS**
AUX_TYPE_PERF_BG_GC_SUCCESS	0x0F	**AUX_PERF_GC_SUCCESS**
AUX_TYPE_PERF_BG_FAILURE	0x10	**AUX_PERF_FAILURE**
AUX_TYPE_PERF_FG_DEFMDB_SUCCESS	0x11	**AUX_PERF_DEFMDB_SUCCESS**
AUX_TYPE_PERF_FG_DEFGC_SUCCESS	0x12	**AUX_PERF_DEFGC_SUCCESS**
AUX_TYPE_PERF_FG_MDB_SUCCESS	0x13	**AUX_PERF_MDB_SUCCESS**
AUX_TYPE_PERF_FG_GC_SUCCESS	0x14	**AUX_PERF_GC_SUCCESS**
AUX_TYPE_PERF_FG_FAILURE	0x15	**AUX_PERF_FAILURE**
AUX_TYPE_OSVERSIONINFO	0x16	**AUX_OSVERSIONINFO** (section 2.2.2.2.16)
AUX_TYPE_EXORGINFO	0x17	**AUX_EXORGINFO** (section 2.2.2.2.17)
AUX_TYPE_PERF_ACCOUNTINFO	0x18	**AUX_PERF_ACCOUNTINFO** (section 2.2.2.2.18)
AUX_TYPE_ENDPOINT_CAPABILITIES	0x48	**AUX_ENDPOINT_CAPABILITIES**<2> (section 2.2.2.2.19)
AUX_CLIENT_CONNECTION_INFO	0x4A	**AUX_CLIENT_CONNECTION_INFO** (section 2.2.2.2.20)
AUX_SERVER_SESSION_INFO	0x4B	**AUX_SERVER_SESSION_INFO** (section 2.2.2.2.21)
AUX_PROTOCOL_DEVICE_IDENTIFICATION	0x4E	**AUX_PROTOCOL_DEVICE_IDENTIFICATION**

Type name	Value	Auxiliary block structure
		(section 2.2.2.2.22)

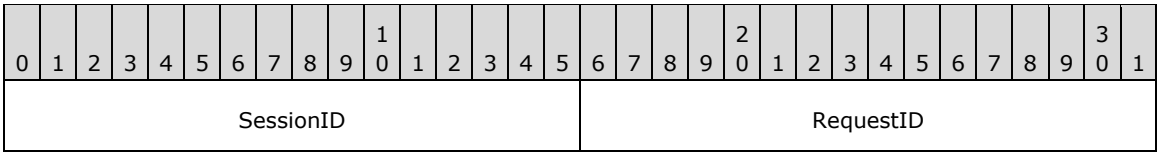
The block type names, associated **Type** field values, and the corresponding auxiliary block structure that follows the **AUX_HEADER** when the **Version** field is **AUX_VERSION_2** are listed in the following table.

Type name	Value	Auxiliary block structure
AUX_TYPE_PERF_SESSIONINFO	0x04	**AUX_PERF_SESSIONINFO_V2** (section 2.2.2.2.3)
AUX_TYPE_PERF_MDB_SUCCESS	0x07	**AUX_PERF_MDB_SUCCESS_V2** (section 2.2.2.2.10)
AUX_TYPE_PERF_GC_SUCCESS	0x08	**AUX_PERF_GC_SUCCESS_V2** (section 2.2.2.2.12)
AUX_TYPE_PERF_FAILURE	0x09	**AUX_PERF_FAILURE_V2** (section 2.2.2.2.14)
AUX_TYPE_PERF_PROCESSINFO	0x0B	**AUX_PERF_PROCESSINFO**
AUX_TYPE_PERF_BG_MDB_SUCCESS	0x0E	**AUX_PERF_MDB_SUCCESS_V2**
AUX_TYPE_PERF_BG_GC_SUCCESS	0x0F	**AUX_PERF_GC_SUCCESS_V2**
AUX_TYPE_PERF_BG_FAILURE	0x10	**AUX_PERF_FAILURE_V2**
AUX_TYPE_PERF_FG_MDB_SUCCESS	0x13	**AUX_PERF_MDB_SUCCESS_V2**
AUX_TYPE_PERF_FG_GC_SUCCESS	0x14	**AUX_PERF_GC_SUCCESS_V2**
AUX_TYPE_PERF_FG_FAILURE	0x15	**AUX_PERF_FAILURE_V2**

The auxiliary block structures are specified in section 2.2.2.2.1 through section 2.2.2.2.22.

2.2.2.2.1 AUX_PERF_REQUESTID Auxiliary Block Structure

The **AUX_PERF_REQUESTID** auxiliary block structure identifies the request associated with the session.

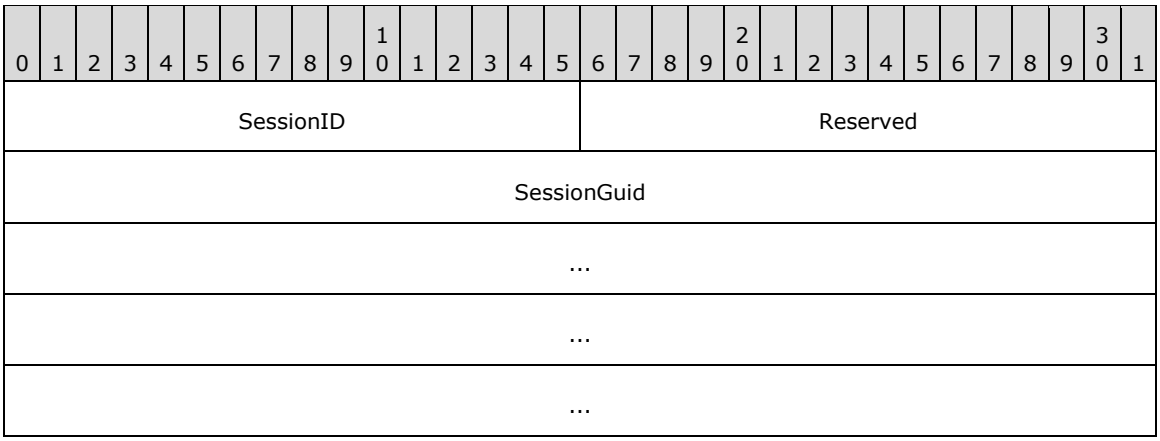


SessionID (2 bytes): The session identification number.

RequestID (2 bytes): The RPC request identification.

2.2.2.2.2 AUX_PERF_SESSIONINFO Auxiliary Block Structure

The **AUX_PERF_SESSIONINFO** auxiliary block structure identifies the client session to associate performance data with.



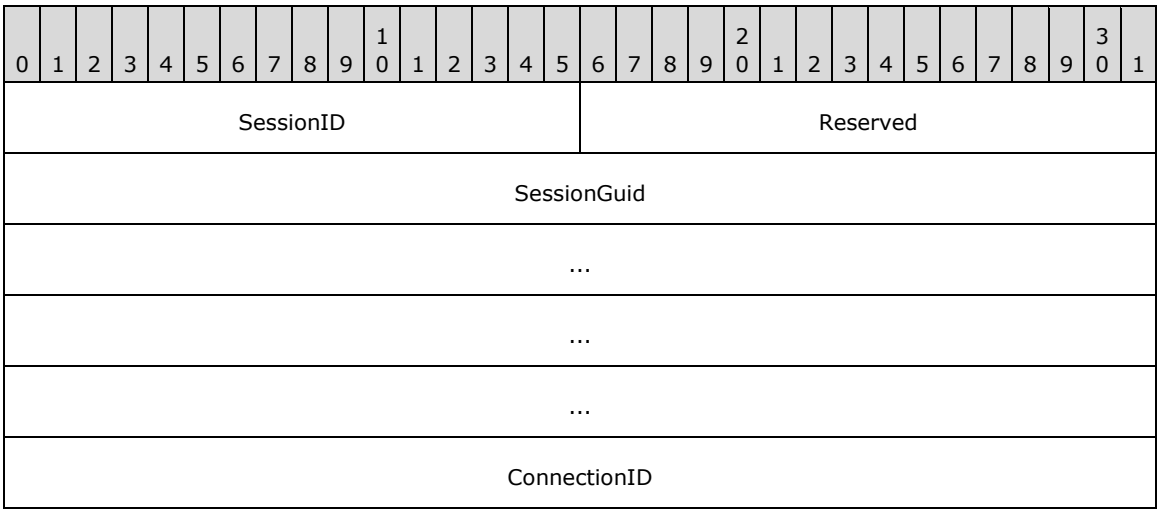
SessionID (2 bytes): The session identification number.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

SessionGuid (16 bytes): The **GUID** representing the client session to associate with the session identification number in the **SessionID** field.

2.2.2.2.3 AUX_PERF_SESSIONINFO_V2 Auxiliary Block Structure

The **AUX_PERF_SESSIONINFO_V2** auxiliary block structure provides diagnostic information about the client session to the server.



SessionID (2 bytes): The session identification number.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

SessionGuid (16 bytes): The GUID representing the client session to associate with the session identification number in the **SessionID** field.

ConnectionID (4 bytes): The connection identification number.

2.2.2.2.4 AUX_PERF_CLIENTINFO Auxiliary Block Structure

The **AUX_PERF_CLIENTINFO** auxiliary block structure identifies which client to associate performance data with.

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
AdapterSpeed																															
ClientID																MachineNameOffset															
UserNameOffset																ClientIPSize															
ClientIPOffset																ClientIPMaskSize															
ClientIPMaskOffset																AdapterNameOffset															
MacAddressSize																MacAddressOffset															
ClientMode																Reserved															
MachineName (variable)																															
...																															
UserName (variable)																															
...																															
ClientIP (variable)																															
...																															
ClientIPMask (variable)																															
...																															
AdapterName (variable)																															
...																															
MacAddress (variable)																															
...																															

AdapterSpeed (4 bytes): The speed of client computer's network adapter, in kilobits per second.

ClientID (2 bytes): The client-assigned client identification number.

MachineNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **MachineName** field. A value of zero indicates that the **MachineName** field is null or empty.

UserNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **UserName** field. A value of zero indicates that the **UserName** field is null or empty.

ClientIPSize (2 bytes): The size of the client IP address referenced by the **ClientIPOffset** field. The client IP address is located in the **ClientIP** field.

ClientIPOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ClientIP** field. A value of zero indicates that the **ClientIP** field is null or empty.

ClientIPMaskSize (2 bytes): The size of the client IP subnet mask referenced by the **ClientIPMaskOffset** field. The client IP mask is located in the **ClientIPMask** field.

ClientIPMaskOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ClientIPMask** field. The size of the IP subnet mask is found in the **ClientIPMaskSize** field. A value of zero indicates that the **ClientIPMask** field is null or empty.

AdapterNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **AdapterName** field. A value of zero indicates that the **AdapterName** field is null or empty.

MacAddressSize (2 bytes): The size of the network adapter Media Access Control (MAC) address referenced by the **MacAddressOffset** field. The network adapter MAC address is located in the **MacAddress** field.

MacAddressOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **MacAddress** field. A value of zero indicates that the **MacAddress** field is null or empty.

ClientMode (2 bytes): A flag that shows the mode in which the client is running. The valid values are specified in the following table.

Client mode flag name	Value	Meaning
CLIENTMODE_UNKNOWN	0x00	Client is not designating a mode of operation.
1. CLIENTMODE_CLASSIC	0x01	Client is running in classic online mode.
2. CLIENTMODE_CACHED	0x02	Client is running in cached mode.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

MachineName (variable): A null-terminated **Unicode** string that contains the client computer name. This variable field is offset from the beginning of the **AUX_HEADER** structure by the **MachineNameOffset** field value.

UserName (variable): A null-terminated Unicode string that contains the user's account name. This variable field is offset from the beginning of the **AUX_HEADER** structure by the **UserNameOffset** field value.

ClientIP (variable): The client's IP address. This field is offset from the beginning of the **AUX_HEADER** structure by the **ClientIPOffset** field value. The size of the client IP address data is found in the **ClientIPSize** field.

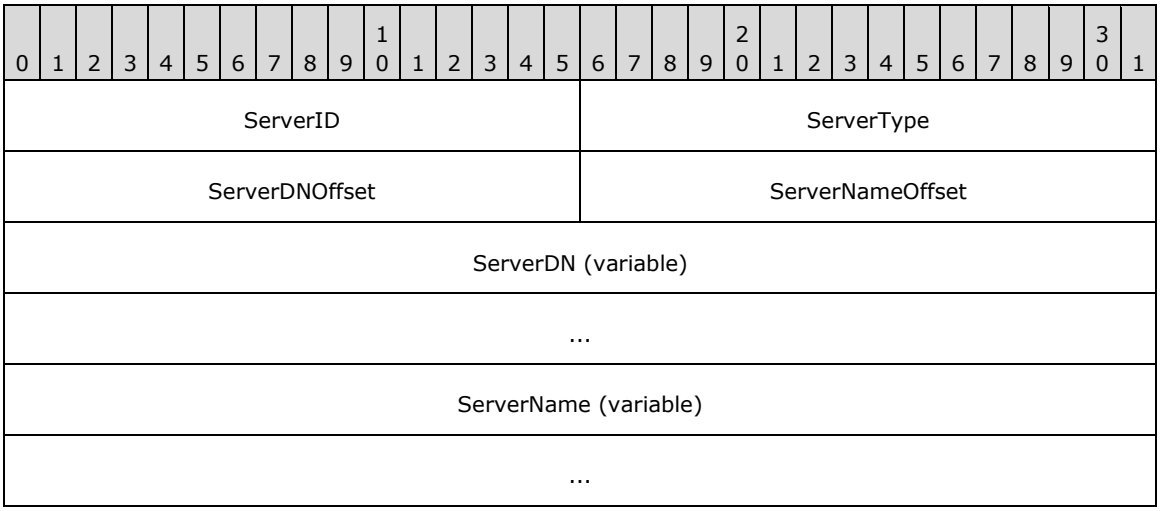
ClientIPMask (variable): The client's IP subnet mask. This field is offset from the beginning of the **AUX_HEADER** structure by the **ClientIPMaskOffset** field value. The size of the client IP mask data is found in the **ClientIPMaskSize** field.

AdapterName (variable): A null-terminated Unicode string that contains the client network adapter name. This field is offset from the beginning of the **AUX_HEADER** structure by the **AdapterNameOffset** field value.

MacAddress (variable): The client's network adapter MAC address. This field is offset from the beginning of the **AUX_HEADER** structure by the **MacAddressOffset** field value. The size of the network adapter MAC address data is found in the **MacAddressSize** field.

2.2.2.2.5 AUX_PERF_SERVERINFO Auxiliary Block Structure

The **AUX_PERF_SERVERINFO** auxiliary block structure identifies which server a client is communicating with to associate the performance data.



ServerID (2 bytes): The client-assigned server identification number.

ServerType (2 bytes): The server type assigned by client. The following table specifies valid values.

Server type name	Value	Meaning
SERVERTYPE_UNKNOWN	0x00	Unknown server type.
3. SERVERTYPE_PRIVATE	0x01	Client/server connection servicing private mailbox data.
4. SERVERTYPE_PUBLIC	0x02	Client/server connection servicing public folder data.
5. SERVERTYPE_DIRECTORY	0x03	Client/server connection servicing directory data.
6. SERVERTYPE_REFERRAL	0x04	Client/server connection servicing referrals.

ServerDNOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ServerDN** field. A value of zero indicates that the **ServerDN** field is null or empty.

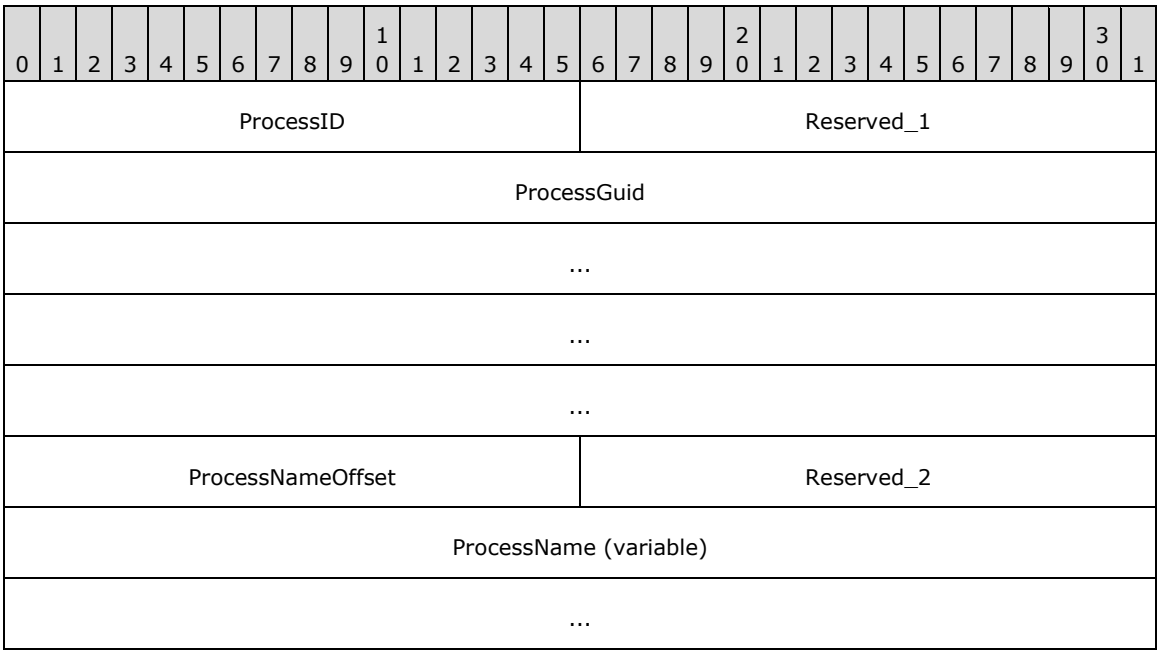
ServerNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ServerName** field. A value of zero indicates that the **ServerName** field is null or empty.

ServerDN (variable): A null-terminated Unicode string that contains the **DN** of the server. This field is offset from the beginning of the **AUX_HEADER** structure by the **ServerDNOffset** field value.

ServerName (variable): A null-terminated Unicode string that contains the server name. This field is offset from the beginning of the **AUX_HEADER** structure by the **ServerNameOffset** field value.

2.2.2.2.6 AUX_PERF_PROCESSINFO Auxiliary Block Structure

The **AUX_PERF_PROCESSINFO** auxiliary block structure identifies the client process to associate performance data with.



ProcessID (2 bytes): The client-assigned process identification number.

Reserved_1 (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

ProcessGuid (16 bytes): The GUID representing the client process to associate with the process identification number in the **ProcessID** field.

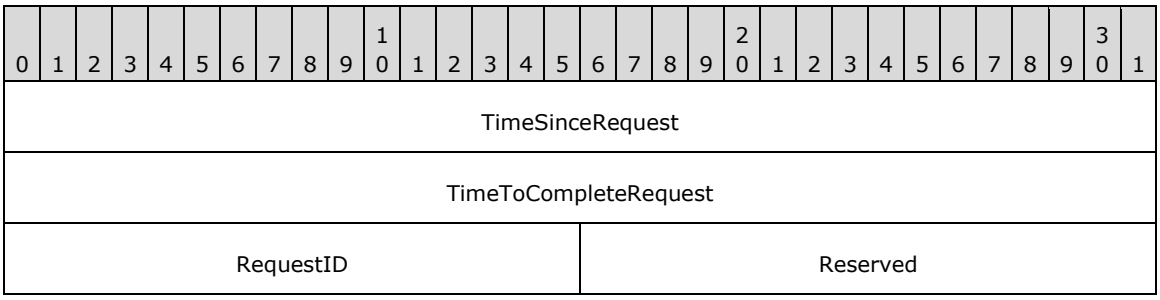
ProcessNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ProcessName** field. A value of zero indicates that the **ProcessName** field is null or empty.

Reserved_2 (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

ProcessName (variable): A null-terminated Unicode string that contains the client process name. This field is offset from the beginning of the **AUX_HEADER** structure by the **ProcessNameOffset** field value.

2.2.2.2.7 AUX_PERF_DEFMDB_SUCCESS Auxiliary Block Structure

The **AUX_PERF_DEFMDB_SUCCESS** auxiliary block structure reports a previously successful RPC to the messaging server.



TimeSinceRequest (4 bytes): The number of milliseconds since a successful request occurred.

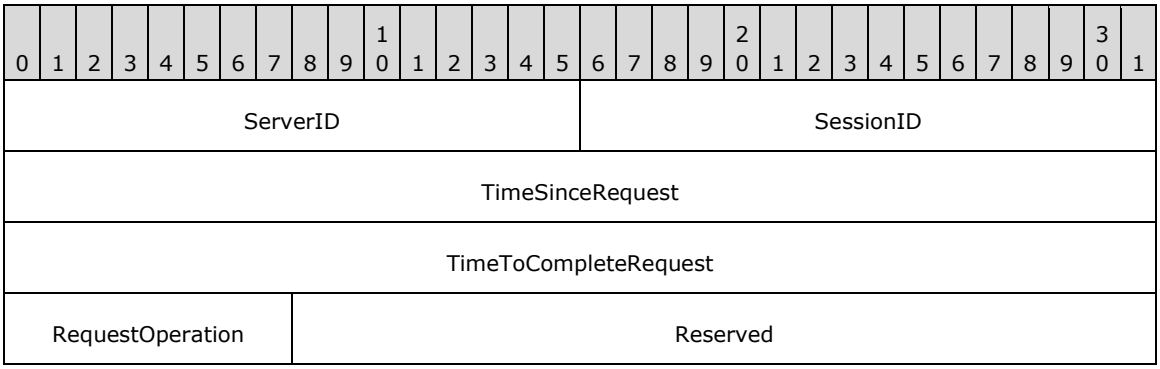
TimeToCompleteRequest (4 bytes): The number of milliseconds the successful request took to complete.

RequestID (2 bytes): The request identification number.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

2.2.2.2.8 AUX_PERF_DEFGC_SUCCESS Auxiliary Block Structure

The **AUX_PERF_DEFGC_SUCCESS** auxiliary block structure reports a previously successful call to the directory service.



ServerID (2 bytes): The server identification number.

SessionID (2 bytes): The session identification number.

TimeSinceRequest (4 bytes): The number of milliseconds since a successful request occurred.

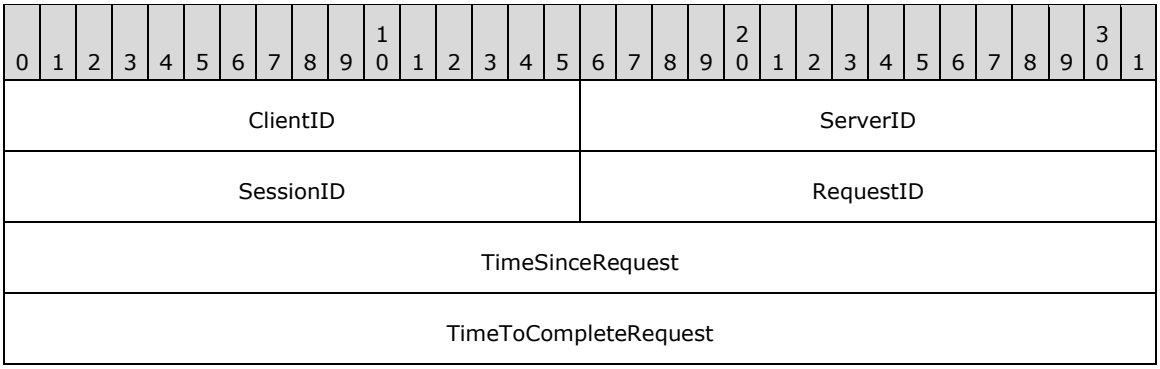
TimeToCompleteRequest (4 bytes): The number of milliseconds the successful request took to complete.

RequestOperation (1 byte): The client-defined operation that was successful.

Reserved (3 bytes): Padding to enforce alignment of the data on a 4-byte field.

2.2.2.2.9 AUX_PERF_MDB_SUCCESS Auxiliary Block Structure

The **AUX_PERF_MDB_SUCCESS** auxiliary block structure reports a previously successful RPC to the messaging server.



ClientID (2 bytes): The client identification number.

ServerID (2 bytes): The server identification number.

SessionID (2 bytes): The session identification number.

RequestID (2 bytes): The request identification number.

TimeSinceRequest (4 bytes): The number of milliseconds since a successful request occurred.

TimeToCompleteRequest (4 bytes): The number of milliseconds the successful request took to complete.

2.2.2.2.10 AUX_PERF_MDB_SUCCESS_V2 Auxiliary Block Structure

The **AUX_PERF_MDB_SUCCESS_V2** auxiliary header structure reports a previously successful RPC to the server.

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
ProcessID																ClientID															
ServerID																SessionID															
RequestID																Reserved															
TimeSinceRequest																															
TimeToCompleteRequest																															

ProcessID (2 bytes): The process identification number.

ClientID (2 bytes): The client identification number.

ServerID (2 bytes): The server identification number.

SessionID (2 bytes): The session identification number.

RequestID (2 bytes): The request identification number.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

TimeSinceRequest (4 bytes): The number of milliseconds since a successful request occurred.

TimeToCompleteRequest (4 bytes): The number of milliseconds the successful request took to complete.

2.2.2.2.11 AUX_PERF_GC_SUCCESS Auxiliary Block Structure

The **AUX_PERF_GC_SUCCESS** auxiliary block structure reports a previously successful call to the directory service.

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
ClientID																ServerID															
SessionID																Reserved_1															

TimeSinceRequest	
TimeToCompleteRequest	
RequestOperation	Reserved_2

ClientID (2 bytes): The client identification number.

ServerID (2 bytes): The server identification number.

SessionID (2 bytes): The session identification number.

Reserved_1 (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

TimeSinceRequest (4 bytes): The number of milliseconds since a successful request occurred.

TimeToCompleteRequest (4 bytes): The number of milliseconds the successful request took to complete.

RequestOperation (1 byte): The client-defined operation that was successful.

Reserved_2 (3 bytes): Padding to enforce alignment of the data on a 4-byte field.

2.2.2.2.12 AUX_PERF_GC_SUCCESS_V2 Auxiliary Block Structure

The **AUX_PERF_GC_SUCCESS_V2** auxiliary block structure reports a previously successful call to the directory service.

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
ProcessID																ClientID															
ServerID																SessionID															
TimeSinceRequest																															
TimeToCompleteRequest																															
RequestOperation																Reserved															

ProcessID (2 bytes): The process identification number.

ClientID (2 bytes): The client identification number.

ServerID (2 bytes): The server identification number.

SessionID (2 bytes): The session identification number.

TimeSinceRequest (4 bytes): The number of milliseconds since a successful request occurred.

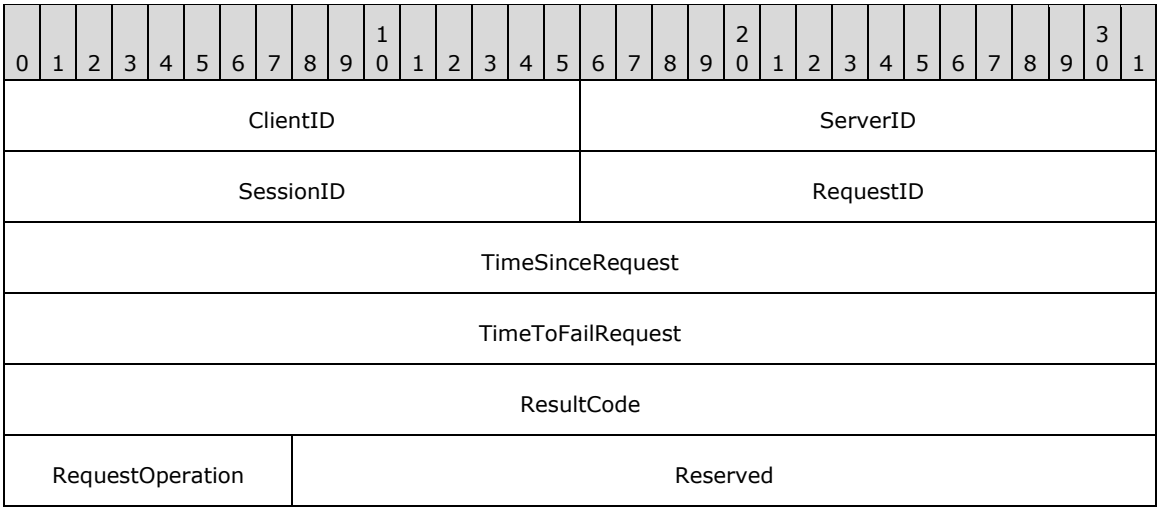
TimeToCompleteRequest (4 bytes): The number of milliseconds the successful request took to complete.

RequestOperation (1 byte): The client-defined operation that was successful.

Reserved (3 bytes): Padding to enforce alignment of the data on a 4-byte field.

2.2.2.2.13 AUX_PERF_FAILURE Auxiliary Block Structure

The **AUX_PERF_FAILURE** auxiliary block structure reports a previously failed call to the messaging server or the directory service.



ClientID (2 bytes): The client identification number.

ServerID (2 bytes): The server identification number.

SessionID (2 bytes): The session identification number.

RequestID (2 bytes): The request identification number.

TimeSinceRequest (4 bytes): The number of milliseconds since a request failure occurred.

TimeToFailRequest (4 bytes): The number of milliseconds the failed request took to complete.

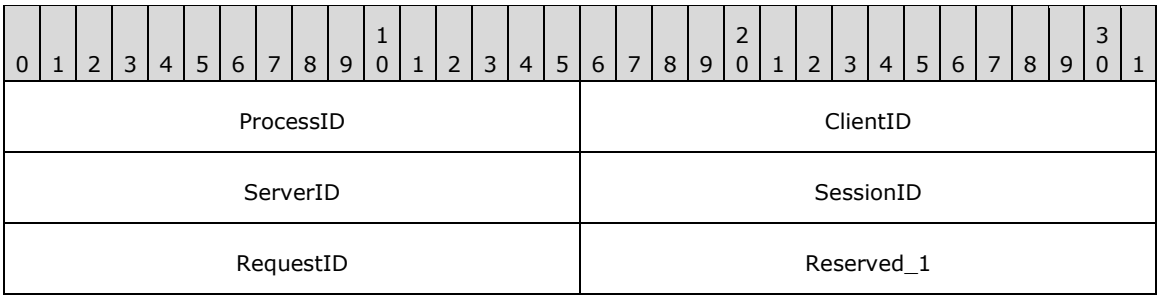
ResultCode (4 bytes): The error code returned for the failed request. Returned error codes are implementation-specific.

RequestOperation (1 byte): The client-defined operation that failed.

Reserved (3 bytes): Padding to enforce alignment of the data on a 4-byte field.

2.2.2.2.14 AUX_PERF_FAILURE_V2 Auxiliary Block Structure

The **AUX_PERF_FAILURE_V2** auxiliary block structure reports a previously failed call to the messaging server or the directory service.



TimeSinceRequest	
TimeToFailRequest	
ResultCode	
RequestOperation	Reserved_2

ProcessID (2 bytes): The process identification number.

ClientID (2 bytes): The client identification number.

ServerID (2 bytes): The server identification number.

SessionID (2 bytes): The session identification number.

RequestID (2 bytes): The request identification number.

Reserved_1 (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

TimeSinceRequest (4 bytes): The number of milliseconds since a request failure occurred.

TimeToFailRequest (4 bytes): The number of milliseconds the request failure took to complete.

ResultCode (4 bytes): The error code returned for the failed request. Returned error codes are implementation-specific.

RequestOperation (1 byte): The client-defined operation that failed.

Reserved_2 (3 bytes): Padding to enforce alignment of the data on a 4-byte field.

2.2.2.2.15 AUX_CLIENT_CONTROL Auxiliary Block Structure

The **AUX_CLIENT_CONTROL** auxiliary block structure reports a change in client behavior.

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
EnableFlags																															
ExpiryTime																															

EnableFlags (4 bytes): The flags that instruct the client to either enable or disable behavior. The flag values and their meanings are described in the following table. To disable a client behavior, the server does not set the flag to the specified value.

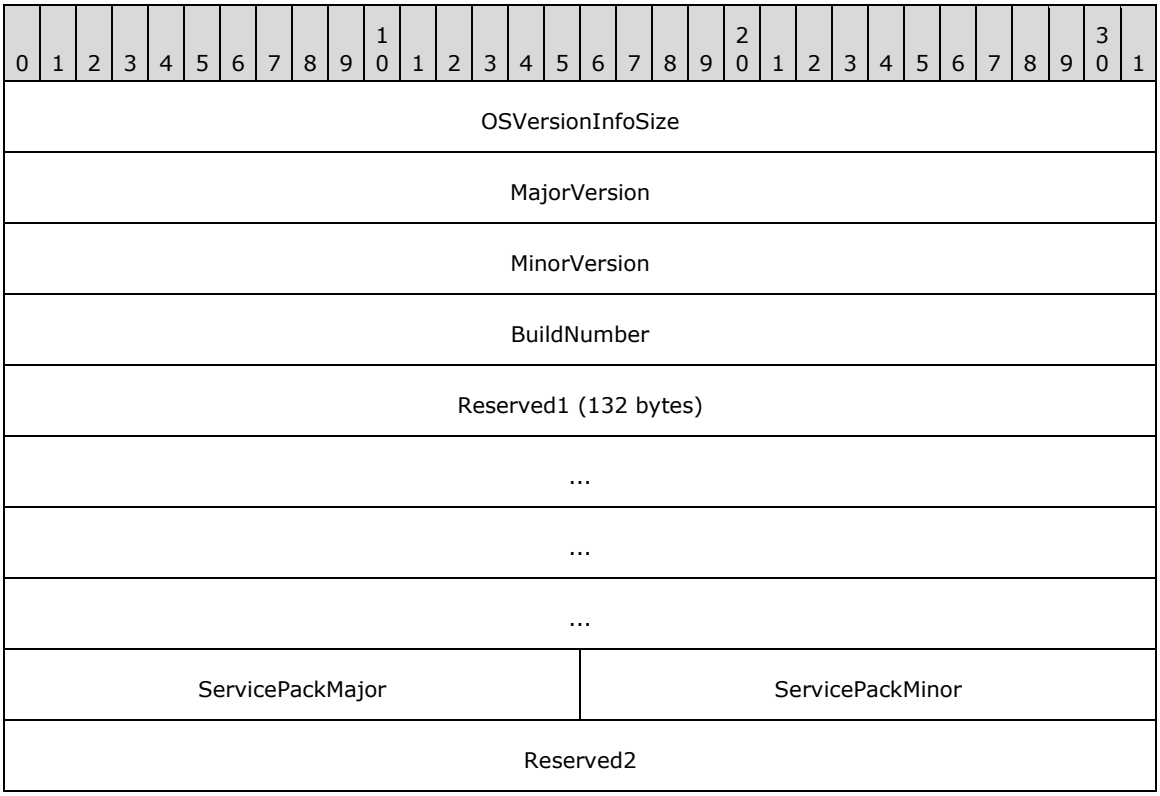
Flag name	Value	Meaning
ENABLE_PERF_SENDTOSERVER	0x00000001	Client MUST start sending performance information to server.
ENABLE_COMPRESSION	0x00000004	Client MUST compress information up to the server. Compression MUST ordinarily be the default behavior, but this allows the server to "disable" compression.
ENABLE_HTTP_TUNNELING	0x00000008	Client MUST use RPC over HTTP if configured.

Flag name	Value	Meaning
ENABLE_PERF_SENDGCDATA	0x00000010	Client MUST include performance data of the client that is communicating with the directory service.

ExpiryTime (4 bytes): The number of milliseconds the client keeps unsent performance data before the data is expired. Expired data is not transmitted to the server. This prevents the server from receiving stale performance information that is stored on the client.

2.2.2.2.16 AUX_OSVERSIONINFO Auxiliary Block Structure

The **AUX_OSVERSIONINFO** auxiliary block structure sends the server's operating system version information to the client.



OSVersionInfoSize (4 bytes): The size of this **AUX_OSVERSIONINFO** structure.

MajorVersion (4 bytes): The major version number of the operating system of the server.

MinorVersion (4 bytes): The minor version number of the operating system of the server.

BuildNumber (4 bytes): The build number of the operating system of the server.

Reserved1 (132 bytes): Reserved and MUST be ignored when received.

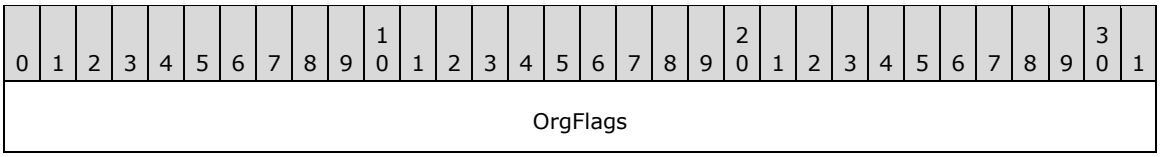
ServicePackMajor (2 bytes): The major version number of the latest operating system service pack that is installed on the server.

ServicePackMinor (2 bytes): The minor version number of the latest operating system service pack that is installed on the server.

Reserved2 (4 bytes): Reserved and MUST be ignored when received.

2.2.2.2.17 AUX_EXORGINFO Auxiliary Block Structure

The **AUX_EXORGINFO** auxiliary block structure informs the client of the presence of public folders within the organization.

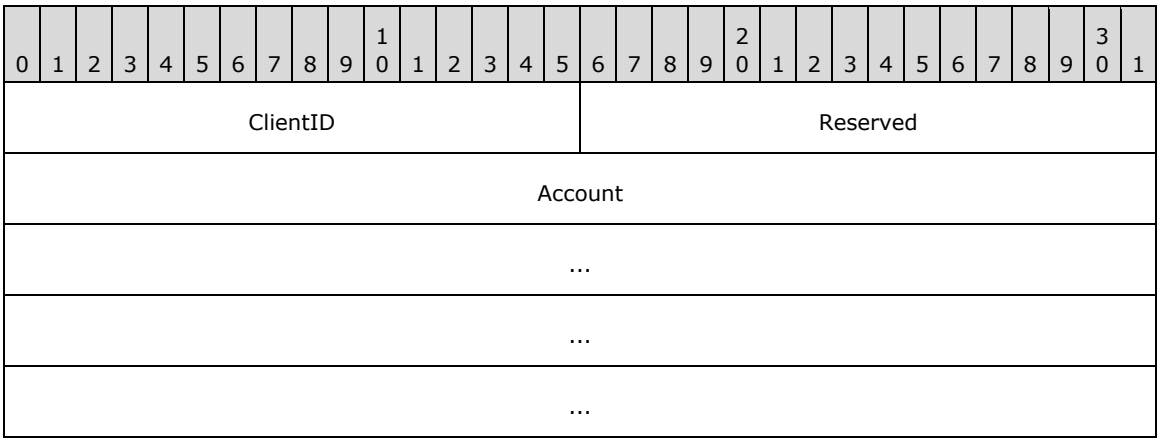


OrgFlags (4 bytes): The flags indicating the server organizational information. The following table specifies the valid values.

Flag name	Value	Meaning
PUBLIC_FOLDERS_ENABLED	0x00000001	Organization has public folders.
USE_AUTODISCOVER_FOR_PUBLIC_FOLDER_CONFIGURATION	0x0000002	The client SHOULD<3> configure public folders using the Autodiscover Publishing and Lookup Protocol, as specified in [MS-OXDSCLI].

2.2.2.2.18 AUX_PERF_ACCOUNTINFO Auxiliary Block Structure

The **AUX_PERF_ACCOUNTINFO** auxiliary block structure reports the client account information to the server.



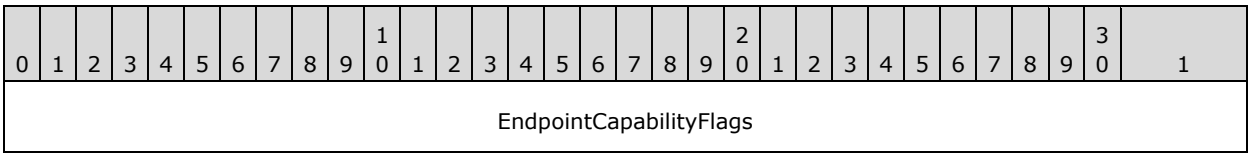
ClientID (2 bytes): The client-assigned identification number. Maps to the **ClientID** of the **AUX_PERF_CLIENTINFO** structure, as specified in section 2.2.2.2.4.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

Account (16 bytes): A GUID representing the client account information that relates to the client identification number in the **ClientID** field.

2.2.2.2.19 AUX_ENDPOINT_CAPABILITIES Auxiliary Block Structure

The **AUX_ENDPOINT_CAPABILITIES** auxiliary block structure informs the client that the server supports multiple interfaces on a single HTTP endpoint.

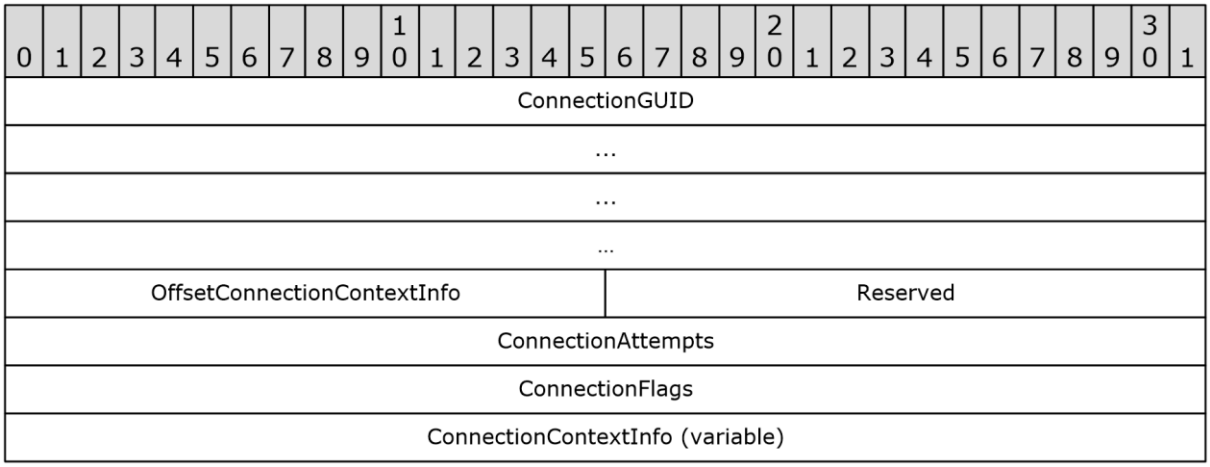


EndpointCapabilityFlag (4 bytes): A flag that indicates that the server combines capabilities on a single endpoint. The valid flag values are specified in the following table.

Flag name	Value	Meaning
ENDPOINT_CAPABILITIES_SINGLE_ENDPOINT	0x00000001	The server supports combined Directory Service Referral interface (RFRI), **name service provider interface (NSPI)**, and EMSMDB interface on a single HTTP endpoint. For more information about RFRI, see [MS-OXABREF]. For more information about NSPI, see [MS-OXNSPI]. The server MAY<4> process requests for different interfaces independently even when requests are transmitted on the same connection. A call to one interface is not to be blocked by a previous call to a different interface on the same connection.

2.2.2.2.20 AUX_CLIENT_CONNECTION_INFO Auxiliary Block Structure

The **AUX_CLIENT_CONNECTION_INFO** auxiliary block structure provides information about the client connection to be logged by the server.



ConnectionGUID (16 bytes): The GUID of the connection to the server.

OffsetConnectionContextInfo (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ConnectionContextInfo** field. A value of zero indicates that the **ConnectionContextInfo** field is null or empty.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field.

ConnectionAttempts (4 bytes): The number of connection attempts.

ConnectionFlags (4 bytes): A flag designating the mode of operation. A value of 0x0001 for this field means that the client is running in cached mode. A value of 0x0000 means that the client is not designating a mode of operation.

ConnectionContextInfo (variable): A null-terminated Unicode string that contains opaque connection context information to be logged by the server. This field is offset from the beginning of the **AUX_HEADER** structure by the **OffsetConnectionContextInfo** field value.

2.2.2.2.21 AUX_SERVER_SESSION_INFO Auxiliary Block Structure

The **AUX_SERVER_SESSION_INFO** auxiliary block structure provides server information to be logged by the client.

OffsetServerSessionContextInfo (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ServerSessionContextInfo** field. A value of zero indicates that the **ServerSessionContextInfo** field is null or empty.

ServerSessionContextInfo (variable): A null-terminated Unicode string that contains opaque server session context information to be logged by the client. This field is offset from the beginning of the **AUX_HEADER** structure by the **OffsetServerSessionContextInfo** field value.

2.2.2.2.22 AUX_PROTOCOL_DEVICE_IDENTIFICATION Auxiliary Block Structure

The **AUX_PROTOCOL_DEVICE_IDENTIFICATION** auxiliary block structure identifies man-in-middle equipment used in messaging applications.

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
DeviceManufacturerOffset																DeviceModelOffset															
DeviceSerialNumberOffset																DeviceVersionOffset															
DeviceFirmwareVersionOffset																DeviceManufacturer (variable)															
...																															
DeviceModel (variable)																															
...																															
DeviceSerialNumber (variable)																															
...																															
DeviceVersion (variable)																															
...																															
DeviceFirmwareVersion (variable)																															
...																															

DeviceManufacturerOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure, as specified in section 2.2.2.2, to the **DeviceManufacturer** field. A value of zero indicates that the **DeviceManufacturer** field is null or empty.

DeviceModelOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **DeviceModel** field. A value of zero indicates that the **DeviceModel** field is null or empty.

DeviceSerialNumberOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **DeviceSerialNumber** field. A value of zero indicates that the **DeviceSerialNumber** field is null or empty.

DeviceVersionOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **DeviceVersion** field. A value of zero indicates that the **DeviceVersion** field is null or empty.

DeviceFirmwareVersionOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **DeviceFirmwareVersion** field. A value of zero indicates that the **DeviceFirmwareVersion** field is null or empty.

DeviceManufacturer (variable): A null-terminated Unicode string that contains the name of the manufacturer of the device. This field is offset from the beginning of the **AUX_HEADER** structure by the value of the **DeviceManufacturerOffset** field.

DeviceModel (variable): A null-terminated Unicode string that contains the model name of the device. This field is offset from the beginning of the **AUX_HEADER** structure by the value of the **DeviceModelOffset** field.

DeviceSerialNumber (variable): A null-terminated Unicode string that contains the serial number of the device. This field is offset from the beginning of the **AUX_HEADER** structure by the value of the **DeviceSerialNumberOffset** field.

DeviceVersion (variable): A null-terminated Unicode string that contains the version number of the device. This field is offset from the beginning of the **AUX_HEADER** structure by the value of the **DeviceVersionOffset** field.

DeviceFirmwareVersion (variable): A null-terminated Unicode string that contains the firmware version of the device. This field is offset from the beginning of the **AUX_HEADER** structure by the value of the **DeviceFirmwareVersionOffset** field.

3 Protocol Details

The Wire Format Protocol contains two RPC interfaces: **EMSMDB**, as specified in section 3.1 and section 3.2, and **AsyncEMSMDB** as specified in section 3.3 and section 3.4.

For some functionality through the **EMSMDB** interface, the client is required to first establish a session context handle by a successful call to the **EcDoConnectEx** method, as specified in section 3.1.4.1. The session context handle is an RPC context handle. All method calls that require a valid session context handle are listed in the following table.

Session context handle–based methods	Interface
EcDoDisconnect	**EMSMDB**
EcRRegisterPushNotification	**EMSMDB**
EcDoRpcExt2	**EMSMDB**
EcDoAsyncConnectEx	**EMSMDB**

For some functionality through the **AsyncEMSMDB** interface, the client is required to call specific interface methods first to establish an asynchronous context handle. The asynchronous context handle is an RPC context handle. To establish an asynchronous context handle, a call to the **EcDoAsyncConnectEx** method on the **EMSMDB** interface MUST be successful. All method calls that require a valid asynchronous context handle are listed in the following table.

Asynchronous context handle–based methods	Interface
EcDoAsyncWaitEx	**AsyncEMSMDB**

3.1 EMSMDB Server Details

The server responds to messages it receives from the client.

3.1.1 Abstract Data Model

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

This protocol includes the following abstract data model (ADM) element:

Global.Handle, as specified in section 3.1.1.1.

3.1.1.1 Global.Handle

The following ADM element is maintained by the server for each session context.

Global.Handle: Some methods on this interface require session context handle information to be stored on the server and used across multiple interface calls for a long duration of time. For these method calls, this protocol is stateful. The server stores this session context information and provides a session context handle (the **Global.Handle** ADM element) to the client to make subsequent interface calls by using this same session context information.

The server keeps a list of all active sessions and their associated session context information. Each session context is identified by a **Global.Handle** ADM element. After a session context has been established, a client can access messaging resources through this session context. The server keeps track of all open resources or any state information specific to the session on the session context. This can include but is not limited to resources, such as folders, messages, tables, attachments, streams, associated asynchronous context handles, and notification callbacks.

The server isolates all resources associated with one session context from all other session contexts on the server. Access to resources on one session context is not allowed using a session context handle of another session context.

When the session context handle is destroyed or the client connection is lost, the session context and all session context information is destroyed, all open resources are closed, and all **Server objects** that are associated with the session context are released.

3.1.2 Timers

None.

3.1.3 Initialization

The server initializes the RPC session by doing the following:

1. The server MUST register the different protocol sequences that will allow the server to communicate with the client. The supported protocol sequences are specified in section 2.1, including named endpoints.
2. The server MUST register the authentication methods that are allowed on the **EMSMDB** interface. The server SHOULD<5> register the following authentication methods. A client authenticates using one of the following authentication methods:
   - RPC_C_AUTHN_WINNT
   - RPC_C_AUTHN_GSS_KERBEROS<6>
   - RPC_C_AUTHN_GSS_NEGOTIATE
   - RPC_C_AUTHN_NONE
3. Start listening for RPCs.
4. Register the **EMSMDB** interface.
5. Register the **EMSMDB** interface to all the registered **binding handles** created previously.

3.1.4 Message Processing Events and Sequencing Rules

This protocol MUST indicate to the RPC runtime that it is to perform a strict **Network Data Representation (NDR)** data consistency check at target level 5.0, as specified in [MS-RPCE].

The methods that this interface includes are listed in the following table.<7> The phrase "Reserved" means that the client MUST NOT send the **opnum**, and the server behavior is undefined.

Methods in RPC Opnum Order

Method	Description
Opnum0NotUsedOnWire	Reserved.

Method	Description
	opnum: 0
EcDoDisconnect	Closes a Session Context with the server. The Session Context is destroyed and all associated server state, objects, and resources that are associated with the Session Context are released. The method requires an active session context handle to be returned from the **EcDoConnectEx** method, as specified in section 3.1.4.1. opnum: 1
Opnum2NotUsedOnWire	Reserved. opnum: 2
Opnum3NotUsedOnWire	Reserved. opnum: 3
EcRRegisterPushNotification	Registers a callback address with the server for a Session Context. The callback address is used to notify the client of a pending event on the server. The method requires an active session context handle to be returned from the **EcDoConnectEx** method. opnum: 4
Opnum5NotUsedOnWire	Reserved. opnum: 5
EcDummyRpc	This call returns a SUCCESS. A client can use it to determine whether it can communicate with the server. opnum: 6
Opnum7NotUsedOnWire	Reserved. opnum: 7
Opnum8NotUsedOnWire	Reserved. opnum: 8
Opnum9NotUsedOnWire	Reserved. opnum: 9
EcDoConnectEx	Creates a session context handle on the server to be used in subsequent calls to the **EcDoDisconnect** (section 3.1.4.3), **EcDoRpcExt2** (section 3.1.4.2), and **EcDoAsyncConnectEx** (section 3.1.4.4) methods. opnum: 10
EcDoRpcExt2	Passes generic ROP commands to the server for processing within a Session Context. The method requires an active session context handle to be returned from the **EcDoConnectEx** method. opnum: 11
Opnum12NotUsedOnWire	Reserved. opnum: 12
Opnum13NotUsedOnWire	Reserved. opnum: 13
EcDoAsyncConnectEx	Binds a session context handle that is returned in the **EcDoConnectEx** method to a new asynchronous context handle that can be used in calls to the **EcDoAsyncWaitEx** method (section 3.3.4.1) in the **AsyncEMSMDB** interface. The method requires an active session context handle to be returned from the **EcDoConnectEx** method. opnum: 14

3.1.4.1 EcDoConnectEx Method (Opnum 10)

The **EcDoConnectEx** method establishes a new Session Context with the server. The Session Context is persisted on the server until the client disconnects by using the **EcDoDisconnect** method, as specified in section 3.1.4.3. The **EcDoConnectEx** method returns a session context handle to be used by a client in subsequent calls.

```
long   stdcall EcDoConnectEx(
    [in] handle_t hBinding,
     [out, ref] CXH * pcxh,
     [in, string] unsigned char * szUserDN,
     [in] unsigned long ulFlags,
     [in] unsigned long ulConMod,
     [in] unsigned long cbLimit,
     [in] unsigned long ulCpid,
     [in] unsigned long ulLcidString,
     [in] unsigned long ulLcidSort,
     [in] unsigned long ulIcxrLink,
     [in] unsigned short usFCanConvertCodePages,
     [out] unsigned long * pcmsPollsMax,
     [out] unsigned long * pcRetry,
     [out] unsigned long * pcmsRetryDelay,
     [out] unsigned short * picxr,
     [out, string] unsigned char **szDNPrefix,
     [out, string] unsigned char **szDisplayName,
     [in] unsigned short rgwClientVersion[3],
     [out] unsigned short rgwServerVersion[3],
     [out] unsigned short rgwBestVersion[3],
     [in, out] unsigned long * pulTimeStamp,
     [in, size_is(cbAuxIn)] unsigned char rgbAuxIn[],
     [in] unsigned long cbAuxIn,
     [out, length_is(*pcbAuxOut), size_is(*pcbAuxOut)] unsigned char rgbAuxOut[],
     [in, out] SMALL_RANGE_ULONG *pcbAuxOut
);
```

hBinding: A valid RPC binding handle.

pcxh: A session context handle for the client. On success, the server MUST return a unique value to be used as a session context handle.

On failure, the server MUST return a zero value as the session context handle.

szUserDN: The DN of the user who is calling the **EcDoConnectEx** method in a directory service. The value of the *szUserDN* parameter is similar to the following: "/o=First Organization/ou=First Administrative Group/cn=recipients/cn=janedow".

ulFlags: A flag value that designates the type of connection being established. On input, this parameter contains connection bits that MUST be set; all flag values not in the following table are reserved connection flags.

Value	Meaning
0x00000000	Requests connection without administrator privilege.
0x00000001	Requests administrator behavior, which causes the server to check that the user has administrator privilege.
0x00008000	If this flag is not passed and the client version (as specified by the *rgwClientVersion* parameter) is less than 12.00.0000.000 and no public folders are configured within the messaging system, the server MUST fail the connection attempt with error code ecClientVerDisallowed. The

Value	Meaning
	AUX_EXORGINFO auxiliary block structure, specified in section 2.2.2.2.17, informs the client of the presence of public folders within the organization. The use of the **AUX_EXORGINFO** auxiliary block structure is further defined in section 3.1.4.1.2.1. If this flag is passed and the client version (as specified by the *rgwClientVersion* parameter) is less than 12.00.0000.000, the server MUST NOT fail the connection attempt due to public folders not being configured within the messaging system. If the client version (as specified by the *rgwClientVersion* parameter) is greater than or equal to 12.00.0000.000, the server MUST NOT fail the connection attempt due to public folders not being configured within the messaging system (regardless of whether or not this flag is passed).

ulConMod: A client-derived 32-bit hash value of the DN passed in the *szUserDN* parameter. The server determines which public folder **replica** to use when accessing public folder information when more than one replica of a folder exists. The hash can be used to distribute client access across replicas in a deterministic way for load balancing.

cbLimit: MUST be set to zero when sent and MUST be ignored when received.

ulCpid: The **code page** in which text data is sent. If the Unicode format is not requested by the client on subsequent calls that use this Session Context, the *ulCpid* parameter sets the code page to be used in subsequent calls.

ulLcidString: The local ID for everything other than sorting.

ulLcidSort: The local ID for sorting.

ulIcxrLink: A value used to link the Session Context created by this call with a currently existing Session Context on the server. To request Session Context linking, the client MUST pass the value of 0xFFFFFFFF. To link to an existing Session Context, this value is the session index value returned in the *piCxr* parameter from a previous call to the **EcDoConnectEx** method. In addition to passing the session index in the *ulIcxrLink* parameter, the client sets the *pulTimeStamp* parameter to the value that was returned in the *pulTimeStamp* parameter from the previous call to the **EcDoConnectEx** method. These two values MUST be used by the server to identify an active session with the same session index and session creation time stamp. If a session is found, the server MUST link the Session Context created by this call with the one found.<8>

A server allows Session Context linking for the following reasons:

1. To consume a single **Client Access License (CAL)** for all the connections made from a single client computer. This gives a client the ability to open multiple independent connections using more than one Session Context on the server but be seen to the server as only consuming a single CAL.<9>

2. To get pending notification information for other sessions on the same client computer. For details, see [MS-OXCNOTIF].

Note that the *ulIcxrLink* parameter is defined as a 32-bit value. Other than passing 0xFFFFFFFF for no Session Context linking, the server only uses the low-order 16 bits as the session index. This value is the value returned in the *piCxr* parameter from a previous call to the **EcDoConnectEx** method, which is the session index and defined as a 16-bit value.

usFCanConvertCodePages: This parameter is reserved. The client MUST pass a value of 0x0001.

pcmsPollsMax: An implementation-dependent value that specifies the number of milliseconds that a client waits between polling the server for event information. If the client or server does not support making asynchronous RPCs for notifications as specified in section 3.3.4.1, or the client is unable to receive notifications via UDP datagrams, as specified in [MS-OXCNOTIF] section 3.2.5.4 and [MS-OXCNOTIF] section 3.2.5.5.2, the client can poll the server to determine whether any events are pending for the client.

pcRetry: An implementation-dependent value that specifies the number of times a client retries future RPCs using the session context handle returned in this call. This is for client RPCs that fail with RPC status code RPC_S_SERVER_TOO_BUSY (0x000006BB). This is a suggested retry count for the client and is not to be enforced by the server. For more details about circumstances under which the RPC_S_SERVER_TOO_BUSY status code is returned, see [MS-OXCROPS] section 3.2.4.2. For more details about how the client handles the RPC_S_SERVER_TOO_BUSY status code, see section 3.2.4.4.

pcmsRetryDelay: An implementation-dependent value that specifies the number of milliseconds a client waits before retrying a failed RPC. If any future RPC to the server using the session context handle returned in this call fails with RPC status code RPC_S_SERVER_TOO_BUSY (0x000006BB), the client waits the number of milliseconds specified in this output parameter before retrying the call. The number of times a client retries is returned in the *pcRetry* parameter. This is a suggested delay for the client and is not to be enforced by the server.

piCxr: A session index value that is associated with the session context handle returned from this call. This value in conjunction with the session creation time stamp value returned in the *pulTimeStamp* parameter will be passed to a subsequent call to the **EcDoConnectEx** method if the client requests to link two Session Contexts.<10> The server MUST NOT assign the same session index value to two active Session Contexts. The server is free to return any 16-bit value for the session index.

The server MUST also use the session index when returning a **RopPending** **ROP response** ([MS-OXCROPS] section 2.2.14.3) on calls to the **EcDoRpcExt2** method, as specified in section 3.1.4.2, to tell the client which Session Context has pending notifications. If Session Contexts are linked, a **RopPending** ROP response can be returned for any linked Session Context.

szDNPrefix: An implementation-dependent value that specifies a DN prefix that is used to build message **recipients**. An empty value indicates that there is nothing to prepend to recipient entries on messages.

szDisplayName: The display name of the user associated with the *szUserDN* parameter.

rgwClientVersion: The client protocol version that the server uses to determine what protocol functionality the client supports. For more details about how version numbers are interpreted from the wire data, see section 3.2.4.1.3.

rgwServerVersion: The server protocol version that the client uses to determine what protocol functionality the server supports. For details about how version numbers are interpreted from the wire data, see section 3.1.4.1.3.

rgwBestVersion: The minimum client protocol version that the server supports. This information is useful if the call to the **EcDoConnectEx** method fails with return code ecVersionMismatch. On success, the server returns the value passed in the *rgwClientVersion* parameter by the client. The server cannot perform any client protocol version negotiation. The server can either return the minimum client protocol version required to access the server and fail the call with ecVersionMismatch (0x80040110) or allow the client and return the value passed by the client in the *rgwClientVersion* parameter. The server implementation sets the minimum client protocol version that is supported by the server. For details about how version numbers are interpreted from the wire data, see section 3.1.4.1.3.1.

pulTimeStamp: The creation time of the newly created Session Context. On input, a value used with the *ulIcxrLink* parameter to link the Session Context created by this call with an existing Session Context. If the *ulIcxrLink* parameter is not set to 0xFFFFFFFF, the client MUST pass in the value of the *pulTimeStamp* parameter returned from the server on a previous call to the **EcDoConnectEx** method. For more details, see the *ulIcxrLink* and *piCxr* parameter descriptions earlier in this section. If the server supports Session Context linking, the server verifies that there is a Session Context state with the unique identifier in the *ulIcxrLink* parameter, and the Session Context state has a creation time stamp equal to the value passed in this parameter. If so, the server MUST link the Session Context created by this call with the one found. If no such Session Context state is found, the server does not fail the **EcDoConnectEx** method call but simply does not link the Session Contexts.<11>

On output, the server has to return a time stamp in which the new Session Context was created. The server saves the Session Context creation time stamp within the Session Context state for later use if a client attempts to link Session Contexts.

rgbAuxIn: An auxiliary payload buffer prefixed by an **RPC_HEADER_EXT** structure, as specified in section 2.2.2.1. Information stored in this structure determines how to interpret the data that follows the structure. The length of the auxiliary payload buffer that includes the **RPC_HEADER_EXT** structure is contained in the *cbAuxIn* parameter.

For details about how to access the embedded auxiliary payload buffer, see section 3.1.4.1.1. For details about how to interpret the auxiliary payload data, see section 3.1.4.1.2.

cbAuxIn: The length of the *rgbAuxIn* parameter. If this value on input is larger than 0x00001008 bytes in size, the server SHOULD<12> fail with the RPC status code RPC_X_BAD_STUB_DATA (0x000006F7). If this value is greater than 0x00000000 and less than 0x00000008, the server SHOULD<13><14> fail with ecRpcFailed (0x80040115). For more information on returning RPC status codes, see [C706].

rgbAuxOut: An auxiliary payload buffer prefixed by an **RPC_HEADER_EXT** structure (section 2.2.2.1). On output, the server can return auxiliary payload data to the client in this parameter. The server MUST include an **RPC_HEADER_EXT** structure before the auxiliary payload data.

For details about how to access the embedded auxiliary payload buffer, see section 3.1.4.1.1. For details about how to interpret the auxiliary payload data, see section 3.1.4.1.2.

pcbAuxOut: The length of the *rgbAuxOut* parameter. If this value on input is larger than 0x00001008, the server MUST fail with the RPC status code RPC_X_BAD_STUB_DATA (0x000006F7).

On output, this parameter contains the size of the data to be returned in the *rgbAuxOut* parameter.

Return Values: If the method succeeds, the return value is 0. If the method fails, the return value is an implementation-specific error code or one of the protocol-defined error codes listed in the following table.

Error code name	Value	Meaning
ecAccessDenied<15>	0x80070005	The authentication context associated with the binding handle does not have enough privilege or the *szUserDN* parameter is empty.
ecNotEncrypted	0x00000970	The server is configured to require encryption and the authentication for the binding handle contained in the *hBinding* parameter is not set with RPC_C_AUTHN_LEVEL_PKT_PRIVACY. For more information about setting the authentication and authorization, see [MSDN-RpcBindingSetAuthInfoEx]. The client attempts the call again with new binding handle that is encrypted.
ecClientVerDisallowed	0x000004DF	1. The server requires encryption, but the client is not encrypted and the client does not support receiving error code ecNotEncrypted being returned by the server. For details about which client versions do not support receiving error code ecNotEncrypted, see section 3.1.4.1.3 and section 3.2.4.1.3. 2. The client version has been blocked by the administrator.
ecLoginFailure	0x80040111	Server is unable to log in user to the mailbox or public folder database.
ecUnknownUser	0x000003EB	The server does not recognize the *szUserDN* parameter as a valid enabled mailbox. For more details, see [MS-OXCSTOR] section 3.1.4.1.
ecLoginPerm	0x000003F2	The connection is requested for administrative access, but the authentication context associated with the binding handle does not have enough privilege.

Error code name	Value	Meaning
ecVersionMismatch	0x80040110	The client and server versions are not compatible. The client protocol version is earlier than that required by the server.
ecCachedModeRequired	0x000004E1	The server requires the client to be running in cache mode. For details about which client versions understand this error code, see section 3.2.4.1.3.
ecRpcHttpDisallowed	0x000004E0	The server requires the client to not be connected via RPC over HTTP. For details about which client versions understand this error code, see section 3.1.4.1.3.
ecProtocolDisabled	0x000007D8	The server disallows the user to access the server via this protocol interface. This could be done if the user is only capable of accessing their mailbox information through a different means (for example, Webmail, POP, or IMAP). For details about which client versions understand this error code, see section 3.1.4.1.3.

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

3.1.4.1.1 Extended Buffer Handling

The **EcDoConnectEx** method contains request and response buffers that use an extended buffer mechanism where the payload is preceded by a header. The header contains the flags specified in section 2.2.2.1 that determine whether the payload has been compressed, determine whether the payload has been obfuscated, and determine whether another extended buffer and payload exist after the current payload. A single payload MUST NOT exceed 32 KB in size.

An extended buffer is used in the *rgbAuxIn* and *rgbAuxOut* parameters on the **EcDoConnectEx** method as specified in section 3.1.4.1.1.1.1 and section 3.1.4.1.1.1.2.

The client or server can choose not to compress the payload if the payload is small enough that compression would not yield much benefit. The client or server can choose not to obfuscate the payload if the payload has already been compressed. The client or server can choose not to obfuscate the payload if the client is connected using RPC layer encryption.

The extended buffer format, compression algorithm, obfuscation algorithm, and extended buffer packing for the **EcDoConnectEx** method are specified in section 3.1.4.1.1.1 through section 3.1.4.1.3 and their subsections.

3.1.4.1.1.1 Extended Buffer Format

The extended buffer format is used in the **EcDoConnectEx** method for the *rgbAuxIn* and *rgbAuxOut* parameters. The way the extended buffer is used for the different fields in the **EcDoConnectEx** method is specified in section 3.1.4.1.1.1.1 and section 3.1.4.1.1.1.2.

3.1.4.1.1.1.1 rgbAuxIn Input Buffer

The *rgbAuxIn* parameter input buffer contains an **RPC_HEADER_EXT** structure, as specified in section 2.2.2.1, followed by payload data.

The **RPC_HEADER_EXT** structure provides information about the payload that follows it.

The **RPC_HEADER_EXT** structure MUST contain the **Last** flag in the **Flags** field.

If the **Compressed** flag is present in the **Flags** field, the payload data MUST be compressed by the client and MUST be uncompressed by the server before it can be interpreted. For details about the compression algorithm, see section 3.1.4.1.1.2.

If the **XorMagic** flag is present in the **Flags** field, the payload data MUST be obfuscated by the client and MUST be reverted by the server before it can be interpreted. For details about the obfuscation algorithm, see section 3.1.4.1.1.3.

If both the **Compressed** and **XorMagic** flags are present in the **Flags** field, the payload MUST first be compressed and then obfuscated by the client, and then MUST first be reverted and then uncompressed by the server before it can be interpreted.

The payload contains auxiliary information, specified in section 3.2.4.1.2, that can be passed from the client to the server. The payload data contains an **AUX_HEADER** structure, as specified in section 2.2.2.2, followed by an auxiliary block structure as specified in the auxiliary block structure table.

3.1.4.1.1.2 rgbAuxOut Output Buffer

The *rgbAuxOut* parameter output buffer contains an **RPC_HEADER_EXT** structure, as specified in section 2.2.2.1, followed by payload data.

The **RPC_HEADER_EXT** structure provides information about the payload that follows it.

The **RPC_HEADER_EXT** structure MUST contain the **Last** flag in the **Flags** field.

If the **Compressed** flag is present in the **Flags** field, the payload data MUST be compressed by the server and MUST be uncompressed by the client before it can be interpreted. For details about the compression algorithm, see section 3.1.4.1.1.2.

If the **XorMagic** flag is present in the **Flags** field, the payload data MUST be obfuscated by the server and MUST be reverted by the client before it can be interpreted. For details about the obfuscation algorithm, see section 3.1.4.1.1.3.

If both the **Compressed** and **XorMagic** flags are present in the **Flags** field, the payload data MUST first be compressed and then obfuscated by the server and then MUST first be reverted and then uncompressed by the client before it can be interpreted.

Payload data contains auxiliary information passed from the server to the client, as specified in section 3.1.4.1.2. The payload contains an **AUX_HEADER** structure, as specified in section 2.2.2.2, followed by an auxiliary block structure as specified in the auxiliary block structure table.

3.1.4.1.1.2 Compression Algorithm

Based on the **Compressed** flag in the **Flags** field that is passed in the **RPC_HEADER_EXT** structure, as specified in section 2.2.2.1, of the extended buffer, the payload data is compressed or decompressed by the server and client by using the Lempel-Ziv 1977 (LZ77) compression algorithm, as specified in [UASDC], and the DIRECT2 encoding algorithm.

The LZ77 compression algorithm is specified in section 3.1.4.1.1.2.1 and its subsections. The basic encoding algorithm DIRECT2 is specified in section 3.1.4.1.1.2.2 and its subsections.

3.1.4.1.1.2.1 LZ77 Compression Algorithm

The LZ77 compression algorithm is used to analyze input data and determine how to reduce the size of that input data by replacing redundant information with metadata. Sections of the data that are identical to sections of the data that have been encoded are replaced by small metadata that indicates how to expand those sections again. The encoding algorithm is used to take that combination of data and metadata and serialize it into a stream of bytes that can later be decoded and decompressed.

3.1.4.1.1.2.1.1 Compression Algorithm Terminology

The following terms are associated with the compression algorithm.

byte: The basic data element in the input stream.

window: A buffer that indicates the number of bytes from the **coding position** backward. A **window** of size W contains the last W processed bytes.

3.1.4.1.1.2.1.2 Using the Compression Algorithm

To use the LZ77 compression algorithm:

1. Set the coding position to the beginning of the input stream.
2. Find the longest match in the window for the lookahead buffer.
3. Output the P,C pair, where P is the pointer to the match in the window, and C is the first byte in the lookahead buffer that does not match.
4. If the lookahead buffer is not empty, move the coding position (and the window) L+1 bytes forward.
5. Return to step 2.

3.1.4.1.1.2.1.3 Compression Process

The compression algorithm searches the window for the longest match with the beginning of the lookahead buffer and then outputs a pointer to that match. Because even a 1-byte match might not be found, the output cannot contain only pointers. The compression algorithm solves this problem by outputting after the pointer the first byte in the lookahead buffer after the match. If no match is found, the algorithm outputs a NULL pointer and the byte at the coding position.

3.1.4.1.1.2.1.4 Compression Process Example

The following table shows the input stream that is used for this compression example. The bytes in the input, "AABCBBABC," occupy the first nine positions of the stream.

Input stream

Stream position	1	2	3	4	5	6	7	8	9
Byte value	A	A	B	C	B	B	A	B	C

The output from the compression process is shown in the following table, which includes the following columns.

Step: Indicates the number of the encoding step. A step in the table finishes every time that the encoding algorithm makes an output. With the compression algorithm, this process happens in each pass through step 3.

Pos: Indicates the coding position. The first byte in the input stream has the coding position 1.

Match: Shows the longest match found in the window.

Byte: Shows the first byte in the lookahead buffer after the match.

Output: Presents the output in the format (B,L)C, where (B,L) is the pointer (P) to the match. This gives the following instructions to the decoder: Go back B bytes in the window and copy L bytes to the output. C is the explicit byte.

Note One or more pointers might be included before the explicit byte that is shown in the Byte column.

Compression process output

Step	Pos	Match	Byte	Output
1.	1	--	A	(0,0)A
2.	2	A	B	(1,1)B
3.	4	--	C	(0,0)C
4.	5	B	B	(2,1)B
5.	7	A B	C	(5,2)C

The result of compression, conceptually, is the output column—that is, a series of bytes and optional metadata that indicates whether that byte is preceded by some sequence of bytes that is already in the output.

Because representing the metadata itself requires bytes in the output stream, it is inefficient to represent a single byte that has previously been encoded by 2 bytes of metadata (offset and length). The overhead of the metadata bytes equals or exceeds the cost of outputting the bytes directly. Therefore, the server protocol only considers sequences of bytes to be a match if the sequences have 3 or more bytes in common.

3.1.4.1.1.2.2 DIRECT2 Encoding Algorithm

The basic notion of the DIRECT2 encoding algorithm is that data appears unchanged in the compressed representation (it is not recommended to try to further compress the data by, for example, applying Huffman compression to that payload), and metadata is encoded in the same output stream, and in line with, the data.

The key to decoding the compressed data is recognizing what bytes are metadata and what bytes are data. The decoder MUST be able to identify the presence of metadata in the compressed and encoded data stream. To provide this information to the decoder, bitmasks are inserted periodically in the byte stream.

The bitmasks that enable the decoder to distinguish data from metadata and the process of encoding the metadata are specified in section 3.1.4.1.1.2.2.1 through section 3.1.4.1.1.2.2.4.

3.1.4.1.1.2.2.1 Bitmask

To distinguish data from metadata in the compressed byte stream, the data stream begins with a 4-byte bitmask that indicates to the decoder whether the next byte to be processed is data ("0" value in the bit), or if the next byte (or series of bytes) is metadata ("1" value in the bit). If a "0" bit is encountered, the next byte in the input stream is the next byte in the output stream. If a "1" bit is encountered, the next byte or series of bytes is metadata that MUST be interpreted further.

For example, a bitmask of 0x01000000 indicates that the first seven bytes are actual data, followed by encoded metadata that starts at the eighth byte. The metadata is followed by 24 additional bytes of data.

The bitmask also contains a "1" in the bit following the last encoded element to indicate the end of the compressed data. For example, given a hypothetical 8-bit bitmask, the string "ABCABCDEF" would be compressed as (0,0)A(0,0)B(0,0)C(3,3)D(0,0)E(0,0)F. Its bitmask would be b'00010001' (0x11). This would indicate 3 bytes of data, followed by metadata, followed by an additional 3 bytes, finally terminated with a "1" to indicate the end of the stream.

The final end bit is always necessary, even if an additional bitmask has to be allocated. If the string in the above example was "ABCABCDEFG", for example, it would require an additional bitmask. It would

begin with the bitmask b'00010000', followed by the compressed data, and followed by another bitmask with a "1" as the next bit to indicate the end of the stream.

When the bitmask has been consumed, the next four bytes in the input stream are another bitmask.

3.1.4.1.1.2.2.2 Encoding Metadata

In the output stream, actual data bytes are stored unchanged. To indicate whether the next byte or bytes are data or metadata, bitmasks are stored periodically. If the next bit in the bitmask is "1", the next set of bytes in the input data stream is metadata. This metadata contains an offset back to the start of the data to be copied to the output stream, and the length of the data to be copied.

To represent the metadata as efficiently as possible, the encoding of that metadata is not fixed in length. The encoding algorithm supports the largest possible floating compression window to increase the probability of finding a large match; the larger the window, the greater the number of bytes that are required for the offset. The encoding algorithm also supports the longest possible match; the longer the match length, the greater the number of bytes that are required to encode the length.

3.1.4.1.1.2.2.3 Metadata Offset

This protocol assumes the metadata is 2 bytes in length, where the high-order 13 bits are a first complement of the offset, and the low-order 3 bits are the length. The offset is only encoded with those 13 bits; this value cannot be extended and defines the maximum size of the compression floating window. For example, the metadata 0x0018 is converted into the offset b'000000000011', and the length b'000'. In integers, the offset is '-4', computed by inverting the offset bits, treating the result as a 2s complement, and converting to integer.

3.1.4.1.1.2.2.4 Match Length

Unlike the metadata offset, the match length is extensible. If the length is less than 10 bytes, it is encoded in the 3 low-order bits of the 2-byte metadata. Although 3 bits seems to allow for a maximum length of 6 (the value b'111' is reserved), because the minimum match is 3 bytes, these 3 bits actually allow for the expression of lengths from 3 to 9. The match length goes from L = b'000' + 3 bytes, to L = b'110' + 3 bytes. Because smaller lengths are much more common than the larger lengths, the algorithm tries to optimize for smaller lengths. To encode a length between 3 and 9, we use the 3 bits that are "in-line" in the 2-byte metadata.

If the length of the match is greater than 9 bytes, an initial bit pattern of b'111' is put in the 3 bits. This does not signify a length of 10 bytes, but instead a length that is greater than or equal to 10, which is included in the low-order nibble of the following byte.

Every other time that the length is greater than 9, an additional byte follows the initial 2-byte metadata. The first time that the additional byte is included, the low-order nibble is used as the additive length. The next nibble is "reserved" for the next metadata instance when the length is greater than 9. Therefore, the first time that the decoder encounters a length that is greater than 9, it reads the next byte from the data stream and the low-order nibble is extracted and used to compute length for this metadata instance. The high-order nibble is remembered and used the next time that the decoder encounters a metadata length that is greater than 9. The third time that a length that is greater than 9 is encountered, another extra byte is added after the 2-byte metadata, with the low-order nibble used for this length and the high-order nibble reserved for the fourth length that is greater than 9, and so on.

If the nibble from this "shared" byte is all 1s (for example, b'1111'), another byte is added after the shared byte to hold more length. In this manner, a length of 24 is encoded as follows:

b'111' (in the 3 bits in the original 2 bytes of metadata), plus

b'1110' (in the nibble of the 'shared' byte of extended length)

b'111' means 10 bytes plus b'1110', which is 14, which results in a total of 24.

If the length is more than 24, the next byte is also used in the length calculation. In this manner, a length of 25 is encoded as follows:

b'111' (in the 3 bits in the original 2 bytes of metadata), plus

b'1111' (in the nibble of the 'shared' byte of extended length), plus

b'00000000' (in the next byte)

This scheme is good for lengths of up to 279 (a length of 10 in the 3 bits in the original 2 bytes of metadata, plus a length of 15 in the nibble of the "shared" byte of extended length, plus a length of up to 254 in the extra byte).

A "full" (all b'1') bit pattern (b'111', b'1111', and b'11111111') means that there is more length in the following 2 bytes.

The final 2 bytes of length differ from the length information that comes earlier in the metadata. For lengths that are equal to 280 or greater, the length is calculated only from these last 2 bytes and is not added to the previous length bits. The value in the last 2 bytes, a 16-bit integer, is 3 bytes less than the metadata length. These last 2 bytes allow for a match length of up to 32,768 bytes + 3 bytes (the minimum match length).

The following table summarizes the length representation in metadata.

Note Length is computed from the bits that are included in the metadata plus the minimum match length of 3.

Length representation in metadata

Match length	Length bits in the metadata
24	b'111' (3 bits in the original 2 bytes of metadata) + b'1110' (in the high- or low-order nibble, as appropriate, of the shared byte)
25	b'111' (3 bits in the original 2 bytes of metadata) + b'1111' (in the high- or low-order nibble, as appropriate, of the shared byte) + b'00000000' (in the next byte)
26	b'111' (3 bits in the original 2 bytes of metadata) + b'1111' (in the high- or low-order nibble, as appropriate, of the shared byte) + b'00000001' (in the next byte)
279	b'111' (3 bits in the original 2 bytes of metadata) + b'1111' (in the high- or low-order nibble, as appropriate, of the shared byte) + b'11111110' (in the next byte)
280	b'111' (3 bits in the original 2 bytes of metadata) b'1111' (in the high- or low-order nibble, as appropriate, of the shared byte)

Match length	Length bits in the metadata
	b'11111111' (in the next byte) 0x0115 (in the next 2 bytes). These 2 bytes represent a length of 277 + 3 (minimum match length). **Note** All the length is included in the final 2 bytes and is not additive, as were the previous length calculations for lengths that are smaller than 280 bytes.
281	b'111' (3 bits in the original 2 bytes of metadata) b'1111' (in the high- or low-order nibble, as appropriate, of the shared byte) b'11111111' (in the next byte) 0x0116 (in the next 2 bytes). This is 278 + 3 (minimum match length). **Note** All the length is included in the final 2 bytes and is not additive, as were the previous length calculations for lengths that are smaller than 280 bytes.

A "full" bit pattern in that last half word does not mean that more metadata is coming after the last bytes.

The LZ77 compression algorithm produces a well-compressed encoding for small valued lengths, but as the length increases, the encoding becomes less well compressed. A match length of greater than 278 bytes requires a relatively large number of bits: 3 + 4 + 8 + 16. This includes 3 bits in the original 2 bytes of metadata, 4 bits in the nibble in the "shared" byte, 8 bits in the next byte, and 16 bits in the final 2 bytes of metadata.

3.1.4.1.1.3 Obfuscation Algorithm

Based on the **XorMagic** flag in the **Flags** field that is passed in the **RPC_HEADER_EXT** structure, as specified in section 2.2.2.1, of the extended buffer, the payload data specified in section 3.1.4.1.1.1.1 and section 3.1.4.1.1.1.2 is obfuscated to obscure any easily readable messaging data being transmitted between the client and server across the network. This is not intended as a security feature. If a client requests to have secure communications with the server, it MUST use RPC-level packet encryption.

The algorithm used to obscure data is straightforward and simple. Every byte of the data to be obfuscated has the **XOR** operator applied with the value 0xA5.

3.1.4.1.2 Auxiliary Buffer

The **EcDoConnectEx** method allows for additional data to travel between the server and the client. This additional data is transferred in the auxiliary buffers of the method. The *rgbAuxIn* parameter payload, as specified in in section 3.1.4.1.1.1.1, is for auxiliary data being sent from the client to the server. The *rgbAuxOut* parameter payload, as specified in section 3.1.4.1.1.1.2, is for auxiliary data being sent from the server to the client.

Unlike the ROP request and ROP response payloads in the *rgbIn* and *rgbOut* parameters, there is no request and response nature to the auxiliary buffers. The data sent from the server to the client is informational data that the client might use to alter its behavior against the server.

The data being transferred from the server to the client enables the server to tell the client about topology characteristics of the messaging system.

All information in the auxiliary buffer MUST be added with an **AUX_HEADER** structure preceding the actual auxiliary information. For details about the **AUX_HEADER** and how it is formatted, see section 2.2.2.2. Within the **AUX_HEADER** structure, the **Version** and **Type** fields combined determine which auxiliary block structure follows the **AUX_HEADER** structure. Details about how to format the **AUX_HEADER** structure to indicate which auxiliary block follows are provided in section 2.2.2.2.

If the server receives an **AUX_HEADER** auxiliary block with a version and type it does not recognize (that is, does not support), it MUST skip over the entire block without throwing an error. The **AUX_HEADER** structure contains the length of the **AUX_HEADER** plus the following auxiliary block structure in the **Size** field, so the information can be skipped.

3.1.4.1.2.1 Server Topology Information

The auxiliary blocks sent from the server to the client in the *rgbAuxOut* parameter auxiliary buffer on the **EcDoConnectEx** method to provide server topology information are described in the following table. Each of these auxiliary blocks MUST be preceded by a properly formatted **AUX_HEADER** structure, as specified in see section 2.2.2.2.

Sent by server to client in the EcDoConnectEx method

Auxiliary block structure	Description
AUX_CLIENT_CONTROL (section 2.2.2.2.15)	Sent to the client to request a change in client behavior. This is a means for the server to dynamically change client behavior. For details about what client behavior the server can adjust, see section 2.2.2.2.15. The client alters its behavior based on this request.
AUX_OSVERSIONINFO (section 2.2.2.2.16)	Sent to the client as informational data to help the client decide whether it needs to alter its behavior against the server. The data provided to the client is the server's operating system version and operating system service pack information.<16>
AUX_EXORGINFO (section 2.2.2.2.17)	Sent to the client as informational data to help the client decide whether it needs to alter its behavior against the server. The data provided informs the client of the presence of public folders within the organization. A client MUST NOT try to open a public **message store** if the server informs the client that it is not present or disabled. If this block is not returned to the client, the client assumes that public folders are available within the organization.
AUX_SERVER_SESSION_INFO (section 2.2.2.2.21)	Sent by the server to the client as server information data to be logged by the client.<17>
AUX_PROTOCOL_DEVICE_IDENTIFICATION (section 2.2.2.2.22)	This information is returned to the client as diagnostic information by any device or system operating between the client and the server.

3.1.4.1.2.2 Processing Auxiliary Buffers Received from the Client

Auxiliary buffers received from the client can contain reserved fields that are inserted in the buffer as padding to enforce alignment of the data on a 4-byte field. The server MUST ignore the value of these fields when reading the stream.

The data sent to the server from the client in the auxiliary input buffer is purely informational, and the server is not required to respond in the auxiliary output buffer.

3.1.4.1.3 Version Checking

When the server receives the client version in the **EcDoConnectEx** method, the server returns its version to the client. The server version information indicates what functionality is supported on the server.

3.1.4.1.3.1 Version Number Comparison

On the wire, version numbers are passed as three **WORD** values, as specified in [MS-DTYP]. In the **EcDoConnectEx** method, as specified in section 3.1.4.1, the *rgwClientVersion*, *rgwServerVersion*, and *rgwBestVersion* parameters are each passed as three **WORD** values.

Version numbers are now expressed in the format "XX.XX.XXXX.XXX". For example, "08.01.0215.000" represents a specific server build. The first number is the product major version. The second number is the product minor version. The third number is the build major number. The fourth number is the build minor number.

In order to make version comparisons, a three-**WORD** value version number (as transmitted over the wire) is converted into a four-**WORD** value version number. A scheme, referred to a version number normalization, was devised that converts from the three-**WORD** on-the-wire format of the version into a four-number version.

All received version parameters are converted into four-number versions before any version checks are performed. A function that converts the three-**WORD** value wire version format into a four-number (four-**WORD**) format that can then be used for version comparisons is described in the following pseudocode example.

```
// This routine converts a three-WORD version value into a normalized
// four-WORD version value.
//
// Version[] is an array of 3 WORD values on the wire.
// NormalizedVersion[] is an array of 4 WORD values for comparison.
//

IF high-bit of Version[1]is set THEN
    SET NormalizedVersion[0] to high-byte of Version[0]
    SET NormalizedVersion[1] to low-byte of Version[0]
    SET NormalizedVersion[2] to Version[1] with high-bit cleared
    SET NormalizedVersion[3] to Version[2]

ELSE
    SET NormalizedVersion[0] to Version[0]
    SET NormalizedVersion[1] to 0
    SET NormalizedVersion[2] to Version[1]
    SET NormalizedVersion[3] to Version[2]
ENDIF
```

The first **WORD** of the three-**WORD** version number is divided into two **BYTE** values, as specified in [MS-DTYP], one being the product major version and the other being the product minor version. On the wire, the client and server MUST determine whether the version that is being passed is in the old scheme or the new scheme. If the highest bit of the second **WORD** value on the wire is set, the version on the wire is in the new scheme. Otherwise, it is interpreted as the old scheme where the product minor version is not sent.

3.1.4.1.3.2 Server Versions

A server implementation determines which level of support it will offer clients. Based on this level of support, it MUST return a server version that corresponds to that support. A server cannot mix and match functionality. To support functionality at a given server version level, the server MUST support all functionality from previous server version levels.

The server version values that are returned to the client on the **EcDoConnectEx** method call are shown in the following table.

Server version	Description
6.0.6755.0	The server supports passing the sentinel value 0xBABE in the **BufferSize** field of a **RopFastTransferSourceGetBuffer** ROP request ([MS-OXCROPS] section 2.2.12.3). This is the minimum server version returned to the client.
8.0.295.0	The server supports passing the sentinel value 0xBABE in the **ByteCount** field of a **RopReadStream** ROP request ([MS-OXCROPS] section 2.2.9.2).
8.0.324.0	The server supports the USE_PER_MDB_REPLID_**MAPPING** (0x01000000) flag in the **OpenFlags** field of a **RopLogon** ROP request ([MS-OXCROPS] section 2.2.3.1).
8.0.358.0	The server supports the **EcDoAsyncConnectEx** and **EcDoAsyncWaitEx** RPC methods.
14.0.324.0	The server supports passing the **ConversationMembers** flag (0x80) in the **TableFlags** field of a **RopGetContentsTable** ROP request ([MS-OXCROPS] section 2.2.4.14).
14.0.616.0	The server supports passing the **HardDelete** flag (0x02) in the **ImportDeleteFlags** field of a **RopSynchronizationImportDeletes** ROP request ([MS-OXCROPS] section 2.2.13.5).
14.1.67.0	The server supports passing the **FailOnConflict** flag (0x40) in the **ImportFlag** field of a **RopSynchronizationImportMessageChange** ROP request ([MS-OXCROPS] section 2.2.13.2).

3.1.4.2 EcDoRpcExt2 Method (Opnum 11)

The **EcDoRpcExt2** method passes generic ROP commands to the server for processing within a Session Context. Each call can contain multiple ROP commands. The server returns the results of each ROP command to the client. This call requires an active session context handle returned from the **EcDoConnectEx** method.

```
long   stdcall EcDoRpcExt2(
     [in, out, ref] CXH * pcxh,
     [in, out] unsigned long *pulFlags,
     [in, size_is(cbIn)] unsigned char rgbIn[],
     [in] unsigned long cbIn,
     [out, length_is(*pcbOut), size_is(*pcbOut)] unsigned char rgbOut[],
     [in, out] BIG_RANGE_ULONG *pcbOut,
     [in, size is(cbAuxIn)] unsigned char rgbAuxIn[],
     [in] unsigned long cbAuxIn,
     [out, length_is(*pcbAuxOut), size_is(*pcbAuxOut)] unsigned char rgbAuxOut[],
     [in, out] SMALL_RANGE_ULONG *pcbAuxOut,
     [out] unsigned long *pulTransTime
);
```

pcxh: A session context handle. On input, the client MUST pass a valid session context handle that was created by calling the **EcDoConnectEx** method. The server uses the session context handle to identify the Session Context to use for this call. On output, the server MUST return the same session context handle on success.

The server can destroy the session context handle by returning a zero session context handle. Reasons for destroying the session context handle are implementation-dependent; following are examples of why the server might destroy the session context handle:

- The server determines that the ROP request payload in the *rgbIn* buffer is malformed or length parameters are not valid.
- The session context handle that was passed in is not valid.
- An attempt was made to access a mailbox that is in the process of being moved.

- An administrator has blocked a client that has an active connection.

pulFlags: The flags that describe the server output characteristics. On input, this parameter contains flags that tell the server how to build the *rgbOut* parameter.

Flag name	Value	Meaning
NoCompression	0x00000001	The server MUST NOT compress ROP response payload (*rgbOut*) or auxiliary payload (*rgbAuxOut*). For details about server behavior when this flag is absent, see section 3.1.4.2.1.1.
NoXorMagic	0x00000002	The server MUST NOT obfuscate the ROP response payload (*rgbOut*) or auxiliary payload (*rgbAuxOut*). For details about server behavior when this flag is absent, see section 3.1.4.2.1.1.
Chain	0x00000004	The client allows chaining of ROP response payloads.

For details about how to use these flags, see section 3.1.4.2.1.1.

On output, the server SHOULD<18> set this parameter to 0x00000000. The output flags not in the table are reserved for future use.

rgbIn: The ROP request payload. The ROP request payload is prefixed with an **RPC_HEADER_EXT** header, as specified in section 2.2.2.1. Information stored in this header determines how to interpret the data following the header. For details about how to access the embedded ROP request payload, see section 3.1.4.2.1. The length of the ROP request payload including the **RPC_HEADER_EXT** header is contained in the *cbIn* parameter.

For more information about ROP buffers, see [MS-OXCROPS].

cbIn: The length of the ROP request payload. On input, this parameter contains the length of the ROP request payload passed in the *rgbIn* parameter. The ROP request payload includes the size of the ROPs plus the size of the **RPC_HEADER_EXT** structure. The server SHOULD<19> fail with the RPC status code of RPC_X_BAD_STUB_DATA (0x000006F7) if the request buffer is larger than 0x00040000 bytes in size. For more information on returning RPC status codes, see [C706]. If the request buffer is smaller than the size of the **RPC_HEADER_EXT** structure (0x00000008 bytes), the server SHOULD<20> fail with error code ecRpcFailed (0x80040115).

rgbOut: The ROP response payload. The size of the payload is specified in the *pcbOut* parameter. Like the ROP request payload, the ROP response payload is also prefixed by a **RPC_HEADER_EXT** header. For details about how to format the ROP response payload, see section 3.1.4.2.1. The size of the ROP response payload plus the **RPC_HEADER_EXT** header is returned in the *pcbOut* parameter.

pcbOut: The maximum size of the *rgbOut* parameter. On input, this parameter contains the maximum size of the *rgbOut* parameter. If the value in the *pcbOut* parameter on input is less than 0x00000008, the server SHOULD<21> fail with error code ecRpcFailed (0x80040115). If the value in the *pcbOut* parameter on input is larger than 0x00040000, the server MUST fail with the RPC status code of RPC_X_BAD_STUB_DATA (0x000006F7).

On output, this parameter contains the size of the ROP response payload, including the size of the **RPC_HEADER_EXT** header in the *rgbOut* parameter. The server returns 0x00000000 on failure because there is no ROP response payload. The client ignores any data returned on failure.

rgbAuxIn: The auxiliary payload buffer. The auxiliary payload buffer is prefixed by an **RPC_HEADER_EXT** structure. Information stored in this header determines how to interpret the data following the header. The length of the auxiliary payload buffer including the **RPC_HEADER_EXT** header is contained in the *cbAuxIn* parameter.

For details about how to access the embedded auxiliary payload buffer, see section 3.1.4.2.1. For details about how to interpret the auxiliary payload data, see section 3.1.4.2.2.

cbAuxIn: The length of the auxiliary payload buffer. On input, this parameter contains the length of the auxiliary payload buffer passed in the *rgbAuxIn* parameter. If the request buffer value is larger than 0x00001008 bytes in size, the server SHOULD<22> fail with the RPC status code RPC_X_BAD_STUB_DATA (0x000006F7).<23>

rgbAuxOut: The auxiliary payload buffer. On output, the server MAY<24> return auxiliary payload data to the client. The server MUST include a **RPC_HEADER_EXT** header before the auxiliary payload data.

pcbAuxOut: The maximum length of the auxiliary payload buffer. On input, this parameter contains the maximum length of the *rgbAuxOut* parameter. If this value on input is larger than 0x00001008, the server MUST fail with the RPC status code RPC_X_BAD_STUB_DATA (0x000006F7).

On output, this parameter contains the size of the data to be returned in the *rgbAuxOut* parameter.

pulTransTime: The time it took to execute this method. On output, the server stores the number of milliseconds the call took to execute. This is the total elapsed time from when the call is dispatched on the server to the point in which the server returns the call. This is diagnostic information the client can use to determine the cause of a slow response time from the server. The client can monitor the total elapsed time across the RPC method call and, using this output parameter, can determine whether time was spent transmitting the request/response on the network or processing it on the server.

Return Values: If the method succeeds, the return value is 0. If the method fails, the return value is an implementation-specific error code or the protocol-defined error code listed in the following table.

Error code name	Value	Meaning
ecRpcFormat	0x000004B6	The format of the request was found to be invalid. This is a generic error that means the length was found to be invalid or the content was found to be invalid.

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

3.1.4.2.1 Extended Buffer Handling

The **EcDoRpcExt2** method, as specified in section 3.1.4.2, contains request and response buffers that use an extended buffer mechanism where the payload is preceded by a header. The header contains the flags specified in section 2.2.2.1 that determine whether the payload has been compressed, determine whether the payload has been obfuscated, and determine whether another extended buffer and payload exist after the current payload. A single payload MUST NOT exceed 32 KB in size.

An extended buffer is used in the *rgbIn*, *rgbOut*, *rgbAuxIn*, and *rgbAuxOut* parameters on the **EcDoRpcExt2** method.

For specification of the compression algorithm used in compressing an extended buffer, see section 3.1.4.1.1.2. For specification of the obfuscation algorithm used to obscure readable messaging content in an extended buffer, see section 3.1.4.1.1.3.

The extended buffer format and extended buffer packing are specified in section 3.1.4.2.1.1 and section 3.1.4.2.1.2.

3.1.4.2.1.1 Extended Buffer Format

The client or server can choose not to compress the payload if the payload is small enough that compression would not yield much benefit. The client or server can also choose to not obfuscate the payload if the payload has already been compressed or if the client is connected using RPC layer

encryption. These options are specified in the **Flags** field of the **RPC_HEADER_EXT** structure in section 2.2.2.1.

The extended buffer is used in the **EcDoRpcExt2** method, as specified in section 3.1.4.2, for a variety of different fields. The information in section 3.1.4.2.1.1.1 through section 3.1.4.2.1.1.4 describes how the extended buffer is used for the different parameters in this method.

3.1.4.2.1.1.1 rgbIn Input Buffer

The *rgbIn* parameter input buffer contains an **RPC_HEADER_EXT** structure, as specified in section 2.2.2.1, followed by payload data.

The **RPC_HEADER_EXT** structure MUST contain the **Last** flag in the **Flags** field.

If the **Compressed** flag is present in the **Flags** field of the **RPC_HEADER_EXT** structure, the payload data MUST be compressed by the client and MUST be uncompressed by the server before it can be interpreted. For details about the compression algorithm, see section 3.1.4.1.1.2.

If the **XorMagic** flag is present in the **Flags** field of the **RPC_HEADER_EXT** structure, the payload data MUST be obfuscated by the client and MUST be reverted by the server before it can be interpreted. For details about the obfuscation algorithm, see section 3.1.4.1.1.3.

If both the **Compressed** and **XorMagic** flags are present in the **Flags** field of the **RPC_HEADER_EXT** structure the payload data MUST first be compressed and then obfuscated by the client, and then MUST first be reverted and then uncompressed by the server before it can be interpreted.

The payload data is ROP request information that can be passed from the client to the server. For details about how to interpret this data, see [MS-OXCROPS].

3.1.4.2.1.1.2 rgbOut Output Buffer

The *rgbOut* parameter output buffer can contain multiple extended buffers in a single output buffer. Each of the extended buffers contains an **RPC_HEADER_EXT** structure, as specified in section 2.2.2.1, followed by **Payload** data.

All **RPC_HEADER_EXT** structures in the output buffer except for the last MUST NOT contain the **Last** flag in the **Flags** field of the **RPC_HEADER_EXT** structure. The last **RPC_HEADER_EXT** structure in the output buffer MUST contain the **Last** flag in its **Flags** field.

If the **Compressed** flag is present in the **Flags** field of an **RPC_HEADER_EXT** structure, the payload data associated with that **RPC_HEADER_EXT** structure MUST be compressed by the server and MUST be uncompressed by the client before it can be interpreted. For details about the compression algorithm, see section 3.1.4.1.1.2.

If the **XorMagic** flag is present in the **Flags** field of an **RPC_HEADER_EXT** structure, the payload data associated with that **RPC_HEADER_EXT** structure MUST be obfuscated by the server and MUST be reverted by the client before it can be interpreted. For details about the obfuscation algorithm, see section 3.1.4.1.1.3.

If both the **Compressed** and **XorMagic** flags are present in the **Flags** field of an **RPC_HEADER_EXT** structure, the payload data associated with that **RPC_HEADER_EXT** structure MUST first be compressed and then obfuscated by the server, and then MUST first be reverted and then uncompressed by the client before it can be interpreted.

Compression or obfuscation can be done differently for each **RPC_HEADER_EXT** structure and its related payload data. The server MUST be able to treat each **RPC_HEADER_EXT** structure and payload data combination independently and interpret the contents solely on the flags specified in the associated **RPC_HEADER_EXT** structure.

Each payload contains ROP response information that is returned from the server to the client. For details about how to interpret this data, see [MS-OXCROPS].

3.1.4.2.1.1.3 rgbAuxIn Input Buffer

The format of the *rgbAuxIn* parameter input buffer for the **EcDoRpcExt2** method, as specified in section 3.1.4.2, is the same as that of the *rgbAuxIn* parameter input buffer for the **EcDoConnectEx** method, as specified in section 3.1.4.1.1.1.1.

3.1.4.2.1.1.4 rgbAuxOut Output Buffer

The format of the *rgbAuxOut* parameter input buffer for the **EcDoRpcExt2** method, as specified in section 3.1.4.2, is the same as that of the *rgbAuxOut* parameter input buffer for the **EcDoConnectEx** method, as specified in section 3.1.4.1.1.1.2.

3.1.4.2.1.2 Extended Buffer Packing

As mentioned in section 3.1.4.2.1.1.2, the *rgbOut* parameter of the **EcDoRpcExt2** method, as specified in section 3.1.4.2, can contain more than one extended buffer, each with an **RPC_HEADER_EXT** structure, as specified in section 2.2.2.1. This concept is called packing. The server has the ability to "pack" additional response data into the *rgbOut* parameter based on whether the client has requested this functionality through passing the **Chain** flag in the *pulFlags* parameter and whether the ROP in the *rgbIn* request on the **EcDoRpcExt2** method supports packing. The ROP commands that support packing are the **RopQueryRows** ROP ([MS-OXCROPS] section 2.2.5.4), the **RopReadStream** ROP ([MS-OXCROPS] section 2.2.9.2), and the **RopFastTransferSourceGetBuffer** ROP ([MS-OXCROPS] section 2.2.12.3).

When processing ROP requests, the server MUST NOT produce more than 32 KB worth of response data for all ROP requests. However, when the server finishes processing a **RopQueryRows** ROP ([MS-OXCROPS] section 2.2.5.4), **RopReadStream** ROP ([MS-OXCROPS] section 2.2.9.2), or **RopFastTransferSourceGetBuffer** ROP ([MS-OXCROPS] section 2.2.12.3) from the *rgbIn* parameter request and it was the last ROP command in the request and the client has requested packing through the **Chain** flag and there is residual room in the *rgbOut* parameter response, the server can add additional data to the *rgbOut* parameter response, each with its own **RPC_HEADER_EXT** header.

For the server to produce additional response data, it MUST build a response as if the client sent another request with only a **RopQueryRows** ROP, **RopReadStream** ROP, or **RopFastTransferSourceGetBuffer** ROP. The additional response data is also limited to 32 KB in size. The additional ROP response is placed into the *rgbOut* parameter buffer following the previous header and associated payload with its own **RPC_HEADER_EXT** structure. The server can then compress and/or obfuscate this payload if the client requests and set the **Flags** field of the **RPC_HEADER_EXT** structure to indicate how the payload has been altered. If more residual room remains in the *rgbOut* parameter, the server can continue to produce more response data until the *rgbOut* parameter does not have enough room to hold another response.

The server MUST stop adding additional packed responses to the *rgbOut* parameter response if the residual size of the *rgbOut* parameter response is less than 8 KB for the **RopReadStream** ROP and **RopFastTransferSourceGetBuffer** ROP and 32 KB for the **RopQueryRows** ROP. The server MUST NOT place more than 96 individual payloads into a single *rgbOut* parameter response.

When it adds additional response data, the server MUST alter its processing of the request to reflect what has already been done. For example, if the client requests to read 1,000 rows in the **RopQueryRows** ROP and the first payload contains 100 rows, the additional response data is processed as if the client requested only 900 rows. The server MUST NOT return more data to the client than the client originally requested.

For the **RopQueryRows** ROP, the server MUST adjust the row count when adding additional response data. For the **RopReadStream** ROP, the server MUST adjust the number of bytes to read when adding additional response data. There is no specific limit for the **RopFastTransferSourceGetBuffer**

ROP, but the server MUST stop packing additional extended buffers that contain the **RopFastTransferSourceGetBuffer** ROP when there is no more data for the fast transfer stream. For the **RopFastTransferSourceGetBuffer** ROP, the client requests that the server return all the server data. For details about how to properly format the **RopFastTransferSourceGetBuffer** ROP in this way, see [MS-OXCROPS] section 2.2.12.3.

3.1.4.2.2 Auxiliary Buffer

The **EcDoRpcExt2** method, as specified in section 3.1.4.2, allows for additional data to travel between the server and client by using auxiliary buffers as specified in section 3.1.4.1.2.

3.1.4.2.2.1 Server Topology Information

The following block MAY<25> be sent from the server to the client in the *rgbAuxOut* parameter auxiliary buffer, as specified in section 3.1.4.1.1.2, on the **EcDoRpcExt2** method, as specified in section 3.1.4.2. Each of these auxiliary blocks MUST be preceded by a properly formatted **AUX_HEADER** structure, as specified in section 2.2.2.2.

Sent by server to client in the EcDoRpcExt2 method

Block structure name	Description
AUX_CLIENT_CONTROL (section 2.2.2.2.15)	Sent to the client to request a change in client behavior. This is a means for the server to dynamically change client behavior. For details about what client behavior the server can adjust, see section 2.2.2.2.15. The client alters its behavior based on this request.

3.1.4.2.2.2 Processing Auxiliary Buffers Received from the Client

Auxiliary buffers received from the client can contain reserved fields that are inserted in the buffer as padding to enforce alignment of the data on a 4-byte field width. The server MUST ignore the value of these fields when reading the stream.

The data sent to the server from the client in the auxiliary input buffer is purely informational, and the server is not required to respond in the auxiliary output buffer.

3.1.4.3 EcDoDisconnect Method (Opnum 1)

The **EcDoDisconnect** method closes a Session Context with the server. The server destroys the Session Context and releases all associated server state, objects, and resources that are associated with the Session Context. This call requires an active session context handle from the **EcDoConnectEx** method, as specified in section 3.1.4.1.

```
long __stdcall EcDoDisconnect(
    [in, out, ref] CXH * pcxh
);
```

pcxh: A session context handle. On input, this parameter is the session context handle of the Session Context that the client is disconnecting. On output, the server MUST set the *pcxh* parameter to a zero value. Setting the value to zero instructs the RPC layer of the server to destroy the RPC context handle.

Return Values: If the method succeeds, the return value is 0. If the method fails, the return value is an implementation-specific error code.

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

3.1.4.4 EcDoAsyncConnectEx Method (Opnum 14)

The **EcDoAsyncConnectEx** method binds a session context handle returned from the **EcDoConnectEx** method, as specified in section 3.1.4.1, to a new asynchronous context handle that can be used in calls to the **EcDoAsyncWaitEx** method in the **AsyncEMSMDB** interface, as specified in section 3.3.4.1. This call requires that an active session context handle be returned from the **EcDoConnectEx** method.

This method is part of notification handling. For more information about notifications, see [MS-OXCNOTIF].

```
long __stdcall EcDoAsyncConnectEx(
     [in] CXH cxh,
     [out, ref] ACXH * pacxh
);
```

cxh: A session context handle. The client MUST pass a valid session context handle that was created by calling the **EcDoConnectEx** method. The server uses the session context handle to identify the Session Context to use for this call.

pacxh: An asynchronous context handle. On success, the server returns an asynchronous context handle that is associated with the Session Context passed in the *cxh* parameter. On failure, the returned value is NULL. The asynchronous context handle can be used to make a call to the **EcDoAsyncWaitEx** method on the **AsyncEMSMDB** interface.

Return Values: If the method succeeds, the return value is 0. If the method fails, the return value is an implementation-specific error code or the protocol-defined error code listed in the following table.

Error code name	Value	Meaning
ecRejected<26>	0x000007EE	Server has asynchronous RPC notifications disabled. Client either polls for notifications or calls the **EcRRegisterPushNotifications** method (section 3.1.4.5).

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

3.1.4.5 EcRRegisterPushNotification Method (Opnum 4)

The **EcRRegisterPushNotification** method registers a callback address with the server for a Session Context. The server MAY<27> support the **EcRRegisterPushNotification** method.

The callback address is used to notify the client of pending events on the server. This call requires an active session context handle from the **EcDoConnectEx** method, as specified in section 3.1.4.1. The server MUST store the callback address and the opaque context data in the Session Context. To notify the client of pending events, the server sends a packet containing just the opaque context data to the callback address. The callback address specifies which network transport is to be used to send the data packet.

For more information about notification handling, see [MS-OXCNOTIF].

```
long __stdcall EcRRegisterPushNotification(
     [in, out, ref] CXH * pcxh,
```

```
        [in] unsigned long iRpc,
        [in, size_is(cbContext)] unsigned char rgbContext[],
        [in] unsigned short cbContext,
        [in] unsigned long grbitAdviseBits,
        [in, size_is(cbCallbackAddress)] unsigned char rgbCallbackAddress[],
        [in] unsigned short cbCallbackAddress,
        [out] unsigned long *hNotification
    );
```

pcxh: A session context handle. On input, the client MUST pass a valid session context handle that was created by calling the **EcDoConnectEx** method. The server uses the session context handle to identify the Session Context to use for this call. On output, the server MUST return the same session context handle on success.

The server can destroy the session context handle by returning a zero for the *pcxh* parameter. Reasons for destroying the session context handle are implementation-dependent; following are examples of why the server might destroy the session context handle:

- The session context handle that was passed in is not valid.
- An attempt was made to access a mailbox that is in the process of being moved.

iRpc: The server MUST ignore this value. The client MUST pass a value of 0x00000000.

rgbContext: Opaque client-generated context data that is sent back to the client at the callback address, passed in the *rgbCallbackAddress* parameter, when the server notifies the client of pending event information. The server MUST save this data within the Session Context and use it when sending a notification to the client.

cbContext: The size of the opaque client context data that is passed in the *rgbContext* parameter. If the value of this parameter is larger than 0x00000010, the server MUST fail this call with error code ecTooBig.

grbitAdviseBits: This parameter MUST be set to 0xFFFFFFFF.

rgbCallbackAddress: The callback address for the server to use to notify the client of a pending event. The size of this data is in the *cbCallbackAddress* parameter.

The data contained in this parameter follows the format of a **sockaddr** structure. For information about the **sockaddr** structure, see [MSDN-SOCKADDR].

The server supports the address families AF_INET and AF_INET6 for a callback address that corresponds to the protocol sequence types that are specified in section 2.1.

If an address family is requested that is not supported, the server MUST return error code ecInvalidParam. If the address family is supported but the communications stack of the server does not support the address type, the server MUST return error code ecNotSupported.

cbCallbackAddress: The length of the callback address in the *rgbCallbackAddress* parameter. The size of this parameter depends on the address family being used. If this size does not correspond to the size of the **sockaddr** structure based on address family, the server MUST return error code ecInvalidParam.

hNotification: A handle to the notification callback. If the call completes successfully, the *hNotification* parameter contains a handle to the notification callback on the server.

Return Values: If the method succeeds, the return value is 0. If the method fails, the return value is an implementation-specific error code or one of the protocol-defined error codes listed in the following table.

Error code name	Value	Meaning
ecInvalidParam	0x80070057	A parameter passed was not valid for the call.
ecNotSupported	0x80040102	The callback address is not supported on the server.
ecTooBig	0x80040305	Opaque context data is too large.

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

3.1.4.6 EcDummyRpc Method (Opnum 6)

The **EcDummyRpc** method returns a SUCCESS. A client can use it to determine whether it can communicate with the server.

```
long __stdcall EcDummyRpc(
    [in] handle_t hBinding
);
```

hBinding: A valid RPC binding handle.

Return Values: The function MUST always succeed and return 0.

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

3.1.4.7 Opnum0NotUsedOnWire Method (Opnum 0)

The **Opnum0NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.4.8 Opnum2NotUsedOnWire Method (Opnum 2)

The **Opnum2NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.4.9 Opnum3NotUsedOnWire Method (Opnum 3)

The **Opnum3NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.4.10 Opnum5NotUsedOnWire Method (Opnum 5)

The **Opnum5NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.4.11 Opnum7NotUsedOnWire Method (Opnum 7)

The **Opnum7NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.4.12 Opnum8NotUsedOnWire Method (Opnum 8)

The **Opnum8NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.4.13 Opnum9NotUsedOnWire Method (Opnum 9)

The **Opnum9NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.4.14 Opnum12NotUsedOnWire Method (Opnum 12)

The **Opnum12NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.4.15 Opnum13NotUsedOnWire Method (Opnum 13)

The **Opnum13NotUsedOnWire** method is reserved for local use. The client MUST NOT send this method.

3.1.5 Timer Events

None.

3.1.6 Other Local Events

None.

3.2 EMSMDB Client Details

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 specification does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this specification.

For some functionality on the **EMSMDB** interface, it is required that the client store a session context handle, as specified in section 3.1.1.1, and use it on subsequent interface calls that require a session context handle.

3.2.2 Timers

None.

3.2.3 Initialization

The client creates an RPC connection to the remote server according to the details specified in section 2.1.

Establishing a connection with the server requires authentication. The RPC binding handle MUST have an authentication method defined.

3.2.4 Message Processing Events and Sequencing Rules

This protocol MUST indicate to the RPC runtime that it is to perform a strict NDR data consistency check at target level 5.0, as specified in [MS-RPCE].

Upon the completion of the RPC method, the client returns the result unmodified to the higher layer. Some method calls require an RPC context handle, which is created in another method call. For details about method dependencies, see section 3.

A client SHOULD<28> use different RPC methods based on the product version being run on the server that it is accessing.

3.2.4.1 Sending the EcDoConnectEx Method

When issuing the **EcDoConnectEx** method on the **EMSMDB** interface, some parameters require additional client-side consideration beyond what is stated in section 3.1.4.1. The parameters for which the client has specific handling are as follows:

hBinding: A valid RPC binding handle that MUST have a server name, protocol sequence, and authentication method defined. Some protocol sequences have named endpoints that MUST be used. For details about how to create a binding handle, see section 2.1.

pcxh: On success, this parameter will contain the session context handle. On failure, this value is NULL. The session context handle MUST be stored on the client and used in subsequent calls on the **EMSMDB** interface that require a valid session context handle.

ulConMod: The connection modulus hash is determined by the client for a connection. How the client determines the hash value is an implementation detail. The client ensures that for a particular DN passed in the *szUserDN* parameter, the hash value is always the same. It is acceptable to have the same hash value for different DNs. The client is free to send any 32-bit value.

cbLimit: A client MUST pass a value of 0x00000000.

ulIcxrLink: This value is used to link the Session Context that is created by this call with an existing Session Context on the server that was created by a previous call to the **EcDoConnectEx** method.<29>

A client can link two Session Contexts for the following reasons:

1. To consume a single CAL for all the connections made from a single client computer. This gives a client the ability to open multiple independent connections by using more than one Session Context on the server but be seen to the server as only consuming a single CAL.<30>

2. To get pending notification information for other sessions on the same client computer. For details, see [MS-OXCNOTIF].

If a client is not requesting to link two Session Contexts or if this is the first call to the **EcDoConnectEx** method, the client MUST pass a value of 0xFFFFFFFF.

Note that the *ulIcxrLink* parameter is defined as a 32-bit value. Other than passing 0xFFFFFFFF if there is no Session Context link, the client passes a value with the high-order 16-bits set to zero, and the low-order 16-bits MUST be the value returned in the *piCxr* parameter from a previous **EcDoConnectEx** method call.

usFCanConvertCodePages: The client MUST pass a value of 0x0001.

pcmsPollsMax: On success, this value is the number of milliseconds the client waits before polling the server for notification information. On failure, the value of this field is undefined and SHOULD be ignored. Other more dynamic options are available to the client for receiving notifications from the server. The client saves this value and associates it with the session context handle.

pcRetry: On success, this value is the number of times the client retries a subsequent **EMSMDB** interface method call that uses the session context handle that is returned in the *pcxh* parameter. On failure, the value of this field is undefined and SHOULD be ignored. For details about retrying RPCs, see section 3.2.4.4. The client saves this value and associates it with the session context handle.

pcmsRetryDelay: On success, this value is the number of milliseconds a client waits before retrying a subsequent **EMSMDB** interface method call that uses the session context handle that is returned in the *pcxh* parameter. On failure, the value of this field is undefined and SHOULD be ignored. The client saves this value and associates it with the session context handle.

piCxr: On success, this value is a 16-bit session index that can be used in conjunction with the value returned in the *pulTimeStamp* parameter to link two Session Contexts on the server. On failure, the value of this field is undefined and SHOULD be ignored. For details about how to link Session Contexts and the reason why a client might request to do so, see the *ulIcxrLink* parameter.<31>

The client saves this value and associates it with the session context handle. It is the session index returned in a **RopPending** ROP response ([MS-OXCROPS] section 2.2.14.3) on calls to the **EcDoRpcExt2** method, as specified in section 3.1.4.2. The **RopPending** ROP response tells the client that a Session Context on the server has pending notifications. If a client links Session Contexts, a **RopPending** ROP can be returned for any linked Session Context.

rgwClientVersion: The client MUST pass the version number of the highest client protocol version it supports. This value will provide information to the server about the protocol functionality that the client supports. For details about how version numbers are interpreted from the wire data and the expected client behavior, see section 3.2.4.1.3.

rgwServerVersion: On success, this value is the server protocol version that the client uses to determine what protocol functionality the server supports. On failure, the value of this field is undefined and SHOULD be ignored. For details about how version numbers are interpreted from the wire data and the expected server behavior, see section 3.1.4.1.3. The client saves this value and associates it with the session context handle.

pulTimeStamp: If a client requests to link the Session Context that is created by this call to a previously created Session Context, the client MUST pass on input the session creation time stamp returned in the *pulTimeStamp* parameter on a previous **EcDoConnectEx** method call. If the client is not requesting to link Session Contexts, the client passes value 0x00000000.<32>

On success, this value is the Session Context creation time stamp. On failure, the value of this field is undefined and SHOULD be ignored. The server saves the Session Context creation time stamp and associates it with the session context handle.

3.2.4.1.1 Extended Buffer Handling

The **EcDoConnectEx** method, as specified in section 3.1.4.1, contains request and response buffers that use an extended buffer mechanism where the payload is preceded by a header. The handling of the extended buffer is specified in section 3.1.4.1.1.

Compression, as specified in section 3.1.4.1.1.2, or obfuscation, as specified in section 3.1.4.1.1.3, can be done differently for each header and associated payload. The client MUST be able to treat each header and associated payload independently and interpret the contents solely on the flags specified in the structure.

3.2.4.1.2 Auxiliary Buffer

The **EcDoConnectEx** method, as specified in section 3.1.4.1, allows for additional data to travel between the client and server. This additional data is transferred in the auxiliary buffers of the method. The *rgbAuxIn* parameter is for auxiliary data being sent from the client to the server.

Unlike the ROP request and ROP response payloads in the *rgbIn* and *rgbOut* parameters, there is no request and response nature to the auxiliary buffers. The data sent to the server from the client in the auxiliary input buffer is purely informational, and the server is not required to respond in the auxiliary output buffer.

The data being transferred in the auxiliary buffers from the client to the server is client-side performance information, which is statistical information that the client can keep regarding its communication with the messaging server or the directory service. Part of this information is for when the client fails to communicate with the messaging server or the directory service. The client can then report this information to the server the next time it communicates.

All information in the auxiliary buffer MUST be added with an **AUX_HEADER** structure preceding the actual auxiliary information. Within the **AUX_HEADER** structure, the **Version** and **Type** fields combined determine which auxiliary block follows the header. For details about how to format the **AUX_HEADER** structure to indicate which auxiliary block follows, see section 2.2.2.2.

If the client receives an auxiliary **AUX_HEADER** structure block with a version and type it does not recognize (that is, does not support), it MUST skip over the entire block (header and auxiliary payload) without throwing an error. The **AUX_HEADER** structure block contains the length of the **AUX_HEADER** structure itself plus the following auxiliary block structure in its **Size** field, so the information can be skipped.

3.2.4.1.2.1 Client Performance Monitoring

The following blocks are sent from the client to the server in the *rgbAuxIn* parameter auxiliary buffer on the **EcDoConnectEx** method to support client performance monitoring. Each of these auxiliary blocks MUST be preceded by a properly formatted **AUX_HEADER** structure. The client can fill reserved fields in these blocks with any value when writing the **stream (2)**.

Auxiliary block structure	Description
AUX_PERF_CLIENTINFO (section 2.2.2.2.4)	Sent to the server as diagnostic information about the client for more robust reporting of networking issues.<33> The client MUST assign a unique client identifier for each **AUX_PERF_CLIENTINFO** block sent to the server. The client identifier is also used in other performance blocks to identify which client to associate the performance data with.
AUX_PERF_PROCESSINFO (section 2.2.2.2.6)	Sent to the server as diagnostic information about the client process for more robust reporting of networking issues. The client MUST assign a unique process identifier for each **AUX_PERF_PROCESSINFO** auxiliary block structure sent to the server. The process identifier is also used in other performance blocks to identify which client process to associate the performance data with.
AUX_PERF_SESSIONINFO (section 2.2.2.2.2)	Sent to the server as diagnostic information about the client session for more robust reporting of networking issues. The client MUST assign a unique session identifier for each **AUX_PERF_SESSIONINFO** or **AUX_PERF_SESSIONINFO_V2** auxiliary block structure sent to the server. The session identifier is also used in other performance blocks to identify which client session to associate the performance data with. It is recommended that the **AUX_PERF_SESSIONINFO_V2** auxiliary block structure be used instead of this block structure. A server still supports this older session information auxiliary block. This block can also be passed in the **EcDoRpcExt2** method auxiliary input buffer (section 3.1.4.2.1.1.3).

Auxiliary block structure	Description
AUX_PERF_SESSIONINFO_V2 (section 2.2.2.2.3)	Sent to the server as diagnostic information about the client session for more robust reporting of networking issues. The client MUST assign a unique session identifier for each **AUX_PERF_SESSIONINFO_V2** or **AUX_PERF_SESSIONINFO** auxiliary block structure sent to the server. The session identifier is also used in other performance blocks to identify which client session to associate the performance data with. This block can also be passed in the **EcDoRpcExt2** method auxiliary input buffer.
AUX_CLIENT_CONNECTION_INFO (section 2.2.2.2.20)	Sent to the server as information about the client connection to be logged by the server.<34>
AUX_PROTOCOL_DEVICE_IDENTIFICATION (section 2.2.2.2.22)	Sent to the server as diagnostic information by any device or system operating between the client and the server.

3.2.4.1.2.2 Processing Auxiliary Buffers Received from the Server

Auxiliary buffers received from the server can contain reserved fields that are inserted in the buffer as padding to enforce alignment of the data on a 4-byte field. The client MUST ignore the value of these fields when reading the stream.

The data received from the server is informational data that the client can use to alter its behavior against the server.

3.2.4.1.3 Version Checking

In the **EcDoConnectEx** method, as specified in section 3.2.4.1, the client passes the client version to the server. The client version information indicates to the server what functionality the client supports.

3.2.4.1.3.1 Version Number Comparison

Version number comparison is specified in section 3.1.4.1.3.1.

3.2.4.1.3.2 Client Versions

A client implementation determines which level of support it will offer servers. Based on this level of support, it MUST pass a client version that corresponds to that support. A client cannot mix and match functionality. To support functionality at one client version level, it MUST support all functionality from previous client version levels.

The following table shows client versions that are passed to the server on the **EcDoConnectEx** method, as specified in section 3.1.4.1, where the client can expect the server behavior to change if the version that is transferred on the wire is equal to or greater than the client version numbers as listed in the table.

Client version	Description
11.0.0.0	The client supports receiving Unicode strings for all string properties on recipient row data that is returned from the server on the **RopReadRecipients** ROP ([MS-OXCROPS] section 2.2.6.6), the **RopOpenMessage** ROP ([MS-OXCROPS] section 2.2.6.1), and the **RopOpenEmbeddedMessage** ROP ([MS-OXCROPS] section 2.2.6.16). This is the minimum version that a client supports to implement the protocol.

Client version	Description
11.00.0000.4920	The client supports receiving ecServerBusy (0x00000480) in the **ReturnValue** field of the **RopFastTransferSourceGetBuffer** ROP response ([MS-OXCROPS] section 2.2.12.3). The **BackoffTime** field is present when the **ReturnValue** field contains ecServerBusy. If the value of the **ReturnValue** field is not ecServerBusy, the **BackoffTime** field is not present. For details about the **RopFastTransferSourceGetBuffer** ROP, see [MS-OXCFXICS] sections 2.2.3.1.1.5 and 3.2.5.8.1.5.
12.00.0000.000	The client supports receiving the errors ecCachedModeRequired, ecRpcHttpDisallowed, and ecProtocolDisabled on the **EcDoConnectEx** method call; otherwise, the client will get back ecClientVerDisallowed instead. The client supports topologies that do not have public folders available. For client versions earlier than 12.00.0000.000, the server MUST fail the **EcDoConnectEx** method call with ecClientVerDisallowed when no public folders are configured within the messaging system unless the **EcDoConnectEx** method parameter flag 0x00008000 is passed in the *ulFlags* parameter.
12.00.3118.000	The client supports receiving an **AUX_EXORGINFO** block in the *rgbAuxOut* parameter (section 3.1.4.1.1.2), on the **EcDoConnectEx** method. The server SHOULD return the **AUX_EXORGINFO** auxiliary block structure in the *rgbAuxOut* parameter on the **EcDoConnectEx** method call.
12.00.3619.000	The client supports receiving the error ecNotEncrypted on the **EcDoConnectEx** method call; otherwise, the client will get back ecClientVerDisallowed. This error is returned when the server is configured to only allow encrypted connections and the client is trying to connect on a nonencrypted connection.
12.00.3730.000	The client supports send optimization for **Incremental Change Synchronization (ICS)** using the **PidTagTargetEntryId** property ([MS-OXOMSG] section 2.2.1.69). For more details, see [MS-OXCFXICS] section 3.3.4.3.3.2.1.2.
12.00.4207.000	The client supports packing of the **RopReadStream** ROP ([MS-OXCROPS] section 2.2.9.2) in the ROP response buffer of the **EcDoRpcExt2** method (section 3.1.4.2). The **RopReadStream** ROP MUST be the last ROP in the request buffer on the **EcDoRpcExt2** method call. For details about extended buffer packing, see section 3.1.4.2.1.2.
12.00.4228.0000	The client supports receiving the **RopBackoff** ROP ([MS-OXCROPS] section 2.2.15.2) in the ROP response buffer of the **EcDoRpcExt2** method call. For details, see [MS-OXCROPS] section 3.1.5.1.1.

3.2.4.1.3.3 Version Numbers Received from the Server

The client can assume that the described functionality exists if the version number that is passed in the RPC buffer is equal to or greater than the server version number in which the functionality was added, as specified in section 3.1.4.1.3.2.

3.2.4.2 Sending the EcDoRpcExt2 Method

When issuing the **EcDoRpcExt2** method, as specified in section 3.1.4.2, some parameters require additional client-side consideration beyond what is stated in section 3.1.4.2. The client has specific handling for the following parameter:

pcxh: The client MUST pass a valid session context handle that was created by calling the **EcDoConnectEx** method. If the value of the *pcxh* parameter on output is zero, the Session Context on the server has been destroyed.

3.2.4.2.1 Extended Buffer Handling

The **EcDoRpcExt2** method, as specified in section 3.1.4.2, contains request and response buffers that use an extended buffer mechanism in which the payload is preceded by a header. Extended buffer handling is specified in section 3.1.4.1.1.

Compression, as specified in section 3.1.4.1.1.2, or obfuscation, as specified in section 3.1.4.1.1.3, can be done differently for each header and associated payload section. The client MUST be able to treat each header and its associated payload independently and to interpret the payload contents solely on the flags specified in the header.

3.2.4.2.2 Auxiliary Buffer

The **EcDoRpcExt2** method, as specified in section 3.1.4.2, allows for additional data to travel between the client and server. This additional data is transferred in the auxiliary buffers of the method. The *rgbAuxIn* parameter payload is for auxiliary data being sent from the client to the server.

Unlike the ROP request and response *rgbIn* and *rgbOut* parameter payloads, there is no request and response nature to the auxiliary buffers. The data sent to the server from the client in the auxiliary input buffer is purely informational, and the server is not required to respond in the auxiliary output buffer.

The data being transferred in the auxiliary buffers from the client to the server is client-side performance information, which is statistical information the client can keep regarding its communication with the messaging server or the directory service. Part of this information is for when the client fails to communicate with the messaging server or the directory service. The client can then report this information to the server the next time it communicates.

All information in the auxiliary buffer MUST be added with an **AUX_HEADER** structure preceding the actual auxiliary block information. For details about the **AUX_HEADER** structure and how it is formatted, see section 2.2.2.2. Within the **AUX_HEADER** header, the **Version** field and **Type** field are combined to determine which auxiliary block follows the header. For details about how to format the **AUX_HEADER** header to indicate which auxiliary block follows, see section 2.2.2.2.

If the client receives an auxiliary **AUX_HEADER** structure block with a version and type it does not recognize (that is, does not support), it MUST skip over the entire block, including the following auxiliary block, without throwing an error. The **AUX_HEADER** structure block contains the length of the **AUX_HEADER** structure plus the following auxiliary block in the **Size** field, and so the information can be skipped.

3.2.4.2.2.1 Client Performance Monitoring

The following blocks are sent from the client to the server in the *rgbAuxIn* parameter, as specified in section 3.1.4.1.1.1, on the **EcDoRpcExt2** method, as specified in section 3.1.4.2, to support client performance monitoring. Each of these auxiliary blocks MUST be preceded by a properly formatted **AUX_HEADER** structure, as specified in section 2.2.2.2. The client can fill reserved fields in these blocks with any value when writing the stream.

Auxiliary block structure	Description
AUX_PERF_SESSIONINFO (section 2.2.2.2.2)	Sent to the server as diagnostic information about the client session for more robust reporting of networking issues. The client MUST assign a unique session identifier for each **AUX_PERF_SESSIONINFO** or **AUX_PERF_SESSIONINFO_V2** auxiliary block structure sent to the server. The session identifier is also used in other performance blocks to identify which client session to associate the performance data with. It is recommended that the **AUX_PERF_SESSIONINFO_V2** auxiliary block structure be used instead of this auxiliary block structure. A server still supports this older session information auxiliary block structure.

Auxiliary block structure	Description
	This block can also be passed in the **EcDoConnectEx** method auxiliary input buffer.
AUX_PERF_SESSIONINFO_V2 (section 2.2.2.2.3)	Sent to the server as diagnostic information about the client session for more robust reporting of networking issues. The client MUST assign a unique session identifier for each **AUX_PERF_SESSIONINFO_V2** or **AUX_PERF_SESSIONINFO** auxiliary block structure sent to the server. The session identifier is also used in other performance blocks to identify which client session to associate the performance data with. This block can also be passed in the **EcDoConnectEx** method auxiliary input buffer.
AUX_PERF_SERVERINFO (section 2.2.2.2.5)	Sent to the server as diagnostic information about the server that the client is communicating with for more robust reporting of networking issues. The client MUST assign a unique server identifier for each **AUX_PERF_SERVERINFO** auxiliary block structure sent to the server. The server identifier is also used in other performance blocks to identify which server a client is communicating with to associate the performance data.
AUX_PERF_REQUESTID (section 2.2.2.2.1)	Sent to the server as diagnostic information about a particular request for more robust reporting of networking issues. The client MUST assign a unique request identifier for each **AUX_PERF_REQUESTID** auxiliary block structure sent to the server. The request identifier is also used in other performance blocks to identify which request to associate the performance data with. The client SHOULD acquire the **SessionID** field value used within this block by previously sending either an **AUX_PERF_SESSIONINFO** auxiliary block structure or an **AUX_PERF_SESSIONINFO_V2** auxiliary block structure to the server.
AUX_PERF_DEFMDB_SUCCESS (section 2.2.2.2.7)	Sent to the server as diagnostic information to report a previously successful RPC to the messaging server. The client can fill the **Reserved** field in this auxiliary buffer with any value when writing the stream. The client acquires the **RequestID** field value used within this block by previously sending an **AUX_PERF_REQUESTID** auxiliary block structure to the server.
AUX_PERF_DEFGC_SUCCESS (section 2.2.2.2.8)	Sent to the server as diagnostic information to report a previously successful call to the directory service. The client acquires the values of the **ServerID** and **SessionID** fields used within this block by previously sending an **AUX_PERF_SERVERINFO** auxiliary block structure and either an **AUX_PERF_SESSIONINFO** auxiliary block structure or an **AUX_PERF_SESSIONINFO_V2** auxiliary block structure to the server.
AUX_PERF_MDB_SUCCESS (section 2.2.2.2.9)	Sent to the server as diagnostic information to report a previously successful RPC to the messaging server. The client acquires the values of the **RequestID**, **ClientID**, **ServerID**, and **SessionID** fields used within this block by previously sending the **AUX_PERF_REQUESTID** auxiliary block structure, the **AUX_PERF_CLIENTINFO** auxiliary block structure, the **AUX_PERF_SERVERINFO** auxiliary block structure, and either the **AUX_PERF_SESSIONINFO** auxiliary block structure or the **AUX_PERF_SESSIONINFO_V2** auxiliary block structure to the server. It is recommended that the **AUX_PERF_MDB_SUCCESS_V2** auxiliary block structure be used instead of this older auxiliary block structure. A server still supports this older session information auxiliary block.
AUX_PERF_MDB_SUCCESS_V2 (section 2.2.2.2.10)	Sent to the server as diagnostic information to report a previously successful RPC to the messaging server. The client acquires the values of the **RequestID**, **ProcessID**, **ClientID**, **ServerID**, and **SessionID** fields used within this block by previously

Auxiliary block structure	Description
	sending the **AUX_PERF_REQUESTID** auxiliary block structure, the **AUX_PERF_PROCESSINFO** auxiliary block structure, the **AUX_PERF_CLIENTINFO** auxiliary block structure, the **AUX_PERF_SERVERINFO** auxiliary block structure, and either the **AUX_PERF_SESSIONINFO** auxiliary block structure or the **AUX_PERF_SESSIONINFO_V2** auxiliary block structure to the server.
AUX_PERF_GC_SUCCESS (section 2.2.2.2.11)	Sent to the server as diagnostic information to report a previously successful call to the directory service. The client acquires the values of the **ClientID**, **ServerID**, and **SessionID** fields used within this block by previously sending the **AUX_PERF_CLIENTINFO** auxiliary block structure, the **AUX_PERF_SERVERINFO** auxiliary block structure, and either the **AUX_PERF_SESSIONINFO** auxiliary block structure or the **AUX_PERF_SESSIONINFO_V2** auxiliary block structure to the server. It is recommended that the **AUX_PERF_GC_SUCCESS_V2** auxiliary block structure be used instead of this auxiliary block structure. A server still supports this older session information auxiliary block.
AUX_PERF_GC_SUCCESS_V2 (section 2.2.2.2.12)	Sent to the server as diagnostic information to report a previously successful call to the directory service. The client acquires the values of the **ProcessID**, **ClientID**, **ServerID**, and **SessionID** fields used within this block by previously sending the **AUX_PERF_PROCESSINFO** auxiliary block structure, the **AUX_PERF_CLIENTINFO** auxiliary block structure, the **AUX_PERF_SERVERINFO** auxiliary block structure, and either the **AUX_PERF_SESSIONINFO** auxiliary block structure or the **AUX_PERF_SESSIONINFO_V2** auxiliary block structure to the server.
AUX_PERF_FAILURE (section 2.2.2.2.13)	Sent to the server as diagnostic information to report a previously failed call to the messaging server or the directory service. The client acquires the values of the **RequestID**, **ClientID**, **ServerID**, and **SessionID** fields used within this block by previously sending the **AUX_PERF_REQUESTID** auxiliary block structure, the**AUX_PERF_CLIENTINFO** auxiliary block structure, the **AUX_PERF_SERVERINFO**, and either the **AUX_PERF_SESSIONINFO** auxiliary block structure or the **AUX_PERF_SESSIONINFO_V2** auxiliary block structure to the server. It is recommended that **AUX_PERF_FAILURE_V2** auxiliary block structure be used instead of this auxiliary block structure. A server still supports this older session information auxiliary block.
AUX_PERF_FAILURE_V2 (section 2.2.2.2.14)	Sent to the server as diagnostic information to report a previously failed call to the messaging server or the directory service. The client acquires the values of the **RequestID**, **ProcessID**, **ClientID**, **ServerID**, and **SessionID** fields used within this block by previously sending the **AUX_PERF_REQUESTID** auxiliary block structure, the **AUX_PERF_PROCESSINFO** auxiliary block structure, the **AUX_PERF_CLIENTINFO** auxiliary block structure, the **AUX_PERF_SERVERINFO**, and either the **AUX_PERF_SESSIONINFO** auxiliary block structure or the **AUX_PERF_SESSIONINFO_V2** auxiliary block structure to the server.

3.2.4.3 Sending the EcDoDisconnect Method

A client terminates communication with a server by calling the **EcDoDisconnect** method, as described in section 3.1.4.3. In the call to the **EcDoDisconnect** method, the client passes the session context

handle that was created in a successful call to the interface **EcDoConnectEx** method, as described in section 3.1.4.1. It is suggested that the server clean up any Session Context data associated with this session context handle.

3.2.4.4 Handling Server Too Busy

All methods that require a valid session context handle are to be retried if the call fails with RPC status RPC_S_SERVER_TOO_BUSY (0x000006BB). The number of times the client retries and the amount of time the client waits before retrying is based on the *pcRetry* and *pcmsRetryDelay* parameters returned on the **EcDoConnectEx** method, as specified in section 3.1.4.1. The **EcDoConnectEx** method is the only method that creates a session context handle, so successful processing of this method is a prerequisite for any method that requires a session context handle. For more details about circumstances under which the RPC_S_SERVER_TOO_BUSY status code is returned, see [MS-OXCROPS] section 3.2.4.2.

3.2.4.5 Handling Connection Failures

If the client's connection to the server fails or if the server prematurely disconnects a client by clearing the session context handle in the response to an **EMSMDB** interface RPC, the client cleans up any saved session state information and closes the session context handle if it is not already set to zero. The binding handle of the session is to be closed.

A client can choose to reconnect to the server automatically by creating a new binding handle and calling the **EcDoConnectEx** method, as specified in section 3.1.4.1. This creates a new Session Context on the server. Note that all Server objects previously opened on the server will no longer exist, and the client MUST issue ROP commands to re-create or reopen the Server objects.

3.2.4.6 Handling Endpoint Consolidation

During the first connection to the server, the client does not know whether the server supports port consolidation. If the client receives the **AUX_ENDPOINT_CAPABILITIES** auxiliary block structure, as specified in section 2.2.2.2.19, in the server's response to the **EcDoConnectEx** method, as specified in section 3.1.4.1, initiated by the client, then the client SHOULD<35> save the information so that on subsequent connections to that server the client can consolidate the RFRI, NSPI, and **EMSMDB** interfaces to a single port, such as port 6001. There is no requirement that the client consolidate the interfaces because this behavior is completely optional.
There is always a one reconnection lag until the client connects in the most optimal way.

3.2.5 Timer Events

None.

3.2.6 Other Local Events

None.

3.3 AsyncEMSMDB Server Details

The server responds to messages it receives from the client.

3.3.1 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 specification does not mandate that implementations

adhere to this model as long as their external behavior is consistent with that described in this specification.

The ADM for this interface is the same as that for the **EMSMDB** interface, as specified in section 3.1.1.

The **AsyncEMSMDB** uses an asynchronous **Global.Handle** ADM element, as defined in section 3.1.1.1. The **Global.Handle** ADM element maps to the session context that is associated with a session context handle. There is only one asynchronous **Global.Handle** ADM element for a session context.

All methods on the **AsyncEMSMDB** interface that use an asynchronous context handle are performed against the Session Context that is associated with the asynchronous **Global.Handle** ADM element (or asynchronous context handle).

The server keeps a mapping between the asynchronous **Global.Handle** ADM element and an active Session Context on the server. A Session Context can be created and destroyed through the **EMSMDB** interface.

When the Session Context is destroyed or the client connection is lost, the asynchronous context handle becomes invalid and will be rejected if used.

3.3.2 Timers

None.

3.3.3 Initialization

To initialize the **AsyncEMSMDB** interface, the server MUST do the following:

1. Register the different protocol sequences that will allow clients to communicate with the server. The supported protocol sequences are specified in section 2.1. Note that some protocol sequences use named endpoints, which are also specified in section 2.1.
2. Register the authentication methods that are allowed on the **AsyncEMSMDB** interface:
   - RPC_C_AUTHN_WINNT
   - RPC_C_AUTHN_GSS_KERBEROS
   - RPC_C_AUTHN_GSS_NEGOTIATE

   A client uses one of these authentication methods to authenticate.
3. Start listening for RPCs.
4. Register the **AsyncEMSMDB** interface.
5. Register the **AsyncEMSMDB** interface to all the registered binding handles created previously.

3.3.4 Message Processing Events and Sequencing Rules

This protocol MUST indicate to the RPC runtime that it is to perform a strict NDR data consistency check at target level 5.0, as specified in [MS-RPCE].

This interface includes the following method.<36>

Method	Description
EcDoAsyncWaitEx	An asynchronous call that the server will not complete until there are pending events on

Method	Description
	the Session Context. The method requires an active asynchronous context handle returned from the **EcDoAsyncConnectEx** method on the **EMSMDB** interface. Opnum: 0

3.3.4.1 EcDoAsyncWaitEx Method (Opnum 0)

The **EcDoAsyncWaitEx** method is an asynchronous call that the server does not complete until events are pending on the Session Context, up to a 5-minute duration of no client activity. If no events are available within 5 minutes of the time that the client last accessed the server<37> through a call to the **EcDoRpcExt2** method, as specified in section 3.1.4.2, the server returns the call and does not set the **NotificationPending** flag in the *pulFlagsOut* parameter. If an event is pending, the server completes the call immediately and returns the **NotificationPending** flag in the *pulFlagsOut* parameter. This call requires an active asynchronous context handle to be returned from the **EcDoAsyncConnectEx** method on the **EMSMDB** interface, as specified in section 3.1.4.1. The asynchronous context handle is associated with the Session Context.

This method is part of notification handling. For more information about notifications, see [MS-OXCNOTIF].

```
long __stdcall EcDoAsyncWaitEx(
     [in] ACXH acxh,
     [in] unsigned long ulFlagsIn,
     [out] unsigned long *pulFlagsOut
);
```

acxh: An asynchronous context handle. On input, the client MUST pass a valid asynchronous context handle that was created by calling the **EcDoAsyncConnectEx** method on the **EMSMDB** interface. The server uses the asynchronous context handle to identify the Session Context to use for this call. If the asynchronous context handle is not valid, the server successfully completes the call, setting the **NotificationPending** flag in the *pulFlagsOut* parameter.

ulFlagsIn: Unused. Reserved for future use. Client MUST pass a value of 0x00000000.

pulFlagsOut: The output flags for the client. Flag values are specified in the following table.

Flag name	Value	Description
NotificationPending	0x00000001	Signals that events are pending for the client on the Session Context on the server. The client MUST call the **EcDoRpcExt2** method (with additional data in the ROP request buffer if there is additional data to send to the server, or with an empty ROP request buffer if there is no additional data to send to the server). The server will return the event details in the ROP response buffer.

Return Values: If the method succeeds, the return value is 0. If the method fails, the return value is an implementation-specific error code or one of the protocol-defined error codes listed in the following table.

Error code name	Value	Meaning
Rejected	0x000007EE	An **EcDoAsyncWaitEx** method call is already outstanding on this asynchronous context handle.<38>

Error code name	Value	Meaning
Exiting	0x000003ED	The server is shutting down.

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

3.3.5 Timer Events

None.

3.3.6 Other Local Events

None.

3.4 AsyncEMSMDB Client Details

3.4.1 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 specification does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this specification.

For some functionality on the **AsyncEMSMDB** interface, it is required that the client store an asynchronous context handle, as described in section 3.3.1, and use it on subsequent interface calls that require an asynchronous context handle.

3.4.2 Timers

None.

3.4.3 Initialization

This interface can only be used after first obtaining an asynchronous context handle from the **EcDoAsyncConnectEx** method from the **EMSMDB** interface, as specified in section 3.1.4.1.

3.4.4 Message Processing Events and Sequencing Rules

This protocol MUST indicate to the RPC runtime that it is to perform a strict NDR data consistency check at target level 5.0, as specified in [MS-RPCE].

A client SHOULD<39> use different RPC methods based on the product version being run on the server that it is accessing.

Upon the completion of the RPC method, the client returns the result unmodified to the higher layer. Some method calls require an RPC context handle, which is created in another method call. For details about method dependencies, see section 3.

3.4.5 Timer Events

None.

3.4.6 Other Local Events

None.

4 Protocol Examples

The following examples show how a client and server use this protocol connection, submit ROP commands, and disconnect.

4.1 Connect to the Server

To begin, the client creates an RPC binding handle to the server with the "ncacn_ip_tcp" protocol sequence and the RPC_C_AUTHN_WINNT authentication method.

The client then establishes a Session Context with the server by using the **EcDoConnectEx** method, as described in section 3.1.4.1, and sets the parameters as follows:

hBinding: Binding handle returned from the **EcDoConnectEx** method call.

pcxh: Pointer to session context handle to hold output value. In this example, the client initializes the session context handle to zero.

szUserDN: "/o=First Organization/ou=First Administrative Group/CN=recipients/CN=janedow" (This is the user's DN. A string that contains the DN of the user who is making the **EcDoConnectEx** method call in a directory service.)

ulFlags: 0x00000000 (Regular user access.)

ulConMod: 0x00340567 (Client-computed hash on the *szUserDN* parameter value.)

cbLimit: 0x00000000

ulCpid: 0x000004E4 (Code page 1252.)

ulLcidString: 0x00000409 (**Locale** 1033 "en-us".)

ulLcidSort: 0x00000409 (Locale 1033 "en-us".)

ulIcxrLink: 0xFFFFFFFF (No session link.)

usFCanConvertCodePages: 0x0001

rgwClientVersion: Pointer to unsigned short array containing values 0x000C, 0x183E, and 0x03E8. Client supports protocol client version 12.6206.1000.

pulTimeStamp: Pointer to unsigned long value 0x00000000.

rgbAuxIn: NULL pointer value.

cbAuxIn: 0x00000000

rgbAuxOut: Pointer to buffer of size 0x1008.

pcbAuxOut: Pointer to unsigned long value 0x00001008.

The server processes the **EcDoConnectEx** method request. The server verifies that authentication context associated with the *hBinding* parameter binding handle has ownership privileges to a directory service object that contains a DN that is in the *szUserDN* parameter. The server creates a Session Context and assigns a session context handle (using 0x00001234 for this example). The server returns the following output values:

pcxh: Value at session context handle pointer is 0x00001234. Note that the actual RPC context handle returned to the client in this parameter might not be what the server returned. The RPC layer on the server and client might map the context handle. The context handle returned to the

client is guaranteed to be unique and will map back to the server-assigned context handle if used on subsequent calls to the server.

pcmsPollsMax: Value at unsigned long pointer is 0x0000EA60. (The client is instructed to poll for events every 60 seconds.)

pcRetry: Value at unsigned long pointer is 0x00000006. (The client is instructed to retry six times before failing.)

pcmsRetryDelay: Value at unsigned long pointer is 0x00001770. (The client is instructed to wait 10 seconds between each retry.)

picxr: Value at unsigned short pointer is a server-assigned session index with value 0x0304.

szDNPrefix: Value at unsigned CHAR pointer is a pointer to a null-terminated ANSI string with value "/o=First Group/ou=First Administrative Group/CN=Configuration/CN=Servers/CN=MBX-SRV-02".

szDisplayName: Value at unsigned CHAR pointer is a pointer to a null-terminated ANSI string with value "MBX-SRV-02".

rgwServerVersion: Value at unsigned short array contains values 0x0008, 0x82B4, and 0x0003. (Server supports protocol server version 8.0.692.3.)

rgwBestVersion: Value at unsigned short array contains values 0x000C, 0x183E, and 0x03E8.

pulTimeStamp: Value at unsigned long pointer is a 32-bit value that represents the internal server time when the Session Context was created.

rgbAuxOut: The server returns the following extended buffer and payload containing auxiliary information.

RPC_HEADER_EXT				**Payload**			
				AUX_HEADER			**AUX_EXORGINFO**
Version	**Flags**	**Size**	**SizeActual**	**Size**	**Version**	**Type**	**OrgFlags**
0x0000	0x0004	0x0008	0x0008	0x0008	0x01	0x17	0x00000001

(Payload is not compressed and not obfuscated.)

pcbAuxOut: Value at unsigned long pointer is 0x00000010. (The *rgbAuxOut* parameter is 16 bytes in length.)

Return Value: 0x00000000

4.2 Issue ROP Commands to the Server

The client has already established a Session Context with the server and has a valid session context handle. For more information, see section 4.1.

The client sends ROP commands to server by using the **EcDoRpcExt2** method, as described in section 3.1.4.2, and by using the session context handle returned from the **EcDoConnectEx** method RPC, as described in section 3.1.4.1.

pcxh: Pointer to session context handle value, which is 0x00001234.

pulFlags: Pointer to unsigned long containing value 0x00000003. (Client requests server to not compress or perform the **XOR** operation on the payload of the *rgbOut* and *rgbAuxOut* parameters.)

rgbIn: Client passes extended buffer and payload containing ROP commands to be processed by server. For details about ROP commands, see [MS-OXCROPS].

RPC_HEADER_EXT				**Payload**		
				ROP request commands		
Version	**Flags**	**Size**	**SizeActual**	**RopSize**	**ROPs**	**ServerObjectHandleTable (SOHT)**
0x0000	0x0004	0x0152	0x0152	0x0142	320 bytes	16 bytes

(Payload is not compressed and not obfuscated.)

cbIn: 0x0000015A

rgbAuxIn: Null pointer value.

cbAuxIn: 0x00000000

rgbOut: Pointer to buffer of size 0x00018008.

pcbOut: Pointer to unsigned long value 0x00018008.

rgbAuxOut: Pointer to buffer of size 0x1008.

pcbAuxOut: Pointer to unsigned long value 0x00001008.

The server processes the **EcDoRpcExt2** method request. The server verifies that the session context handle is for a valid Session Context for this user. The server processes the ROP request commands and returns the ROP response results to the client with the following output values:

pcxh: Value at session context handle pointer is 0x00001234.

pulFlags: Value at unsigned long is 0x00000000.

rgbOut: Server returns the following extended buffer and payload containing ROP response commands.

RPC_HEADER_EXT				**Payload**		
				ROP response commands		
Version	**Flags**	**Size**	**SizeActual**	**RopSize**	**ROPs**	**SOHT**
0x0000	0x0004	0x0052	0x0052	0x0042	64 bytes	16 bytes

(Payload is not compressed and not obfuscated.)

pcbOut: 0x0000005A

rgbAuxOut: Server returns nothing in the auxiliary output buffer.

pcbAuxOut: 0x00000000

pulTransTime: Value at unsigned long pointer is 0x00000010. (The number of milliseconds it took the server to process the **EcDoRpcExt2** method RPC.)

Return Value: 0x00000000

4.3 Receive Packed ROP Responses from the Server

The client has already established a Session Context with the server and has a valid session context handle. For more information, see section 4.1.

The client sends ROP commands to server by calling the **EcDoRpcExt2** method, as described in section 3.1.4.2, by using the session context handle that is returned from the **EcDoConnectEx** method call, as described in section 3.1.4.1. The last ROP request contains the **RopReadStream** ROP ([MS-OXCROPS] section 2.2.9.2), so the client requests response chaining (for example, packing).

pcxh: Pointer to session context handle value, which is 0x00001234.

pulFlags: Pointer to unsigned long containing value 0x00000007. (Client requests that the server not compress or perform an **XOR** operation on the payload of the *rgbOut* and *rgbAuxOut* parameters. Client requests response chaining.)

rgbIn: Client passes extended buffer and payload containing ROP commands to be processed by server. For details about ROP commands, see [MS-OXCROPS].

RPC_HEADER_EXT				**Payload**		
				ROP request commands		
Version	**Flags**	**Size**	**SizeActual**	**RopSize**	**ROPs**	**SOHT**
0x0000	0x0004	0x0152	0x0152	0x0142	320 bytes (last ROP command is **RopReadStream**)	16 bytes

(Payload is not compressed and not obfuscated.)

cbIn: 0x0000015A

rgbAuxIn: Null pointer value.

cbAuxIn: 0x000000

rgbOut: Pointer to buffer of size 0x00018008.

pcbOut: Pointer to unsigned long value 0x00018008.

rgbAuxOut: Pointer to buffer of size 0x1008.

pcbAuxOut: Pointer to unsigned long value 0x00001008.

The server processes the **EcDoRpcExt2** method request. The server verifies that the session context handle is for a valid Session Context for this user. The server processes the ROP request commands and returns ROP response results to client. The last ROP was the **RopReadStream** ROP, and the client has requested chaining; there is more data to return in the stream being read, there is more room in the *rgbOut* parameter output buffer, and the server adds another extended buffer and payload. The server returns the following output values.

pcxh: Value at session context handle pointer is 0x00001234.

pulFlags: Value at unsigned long is 0x00000000.

rgbOut: Server returns two extended buffer header and payload pairs containing ROP response commands. The last payload contains only the **RopReadStream** ROP command.

RPC_HEADER_EXT	Payload			RPC_HEADER_EXT	Payload		
Flags: 0x0000 **Size:** 0x7FFE	ROP response commands			**Flags:** 0x0004 **Size:** 0x2008	ROP response command		
	RopSize 0x7FEE	**ROPs**	**SOHT** 16 bytes		**RopSize** 0x1FF8	**ROP**	**SOHT** 16 bytes

(Payloads are not compressed and not obfuscated.)

pcbOut: 0x0000A016

rgbAuxOut: Server returns nothing in the auxiliary output buffer.

pcbAuxOut: 0x00000000

pulTransTime: Value at unsigned long pointer is 0x00000010. (The number of milliseconds it took the server to process the **EcDoRpcExt2** method call.)

Return Value: 0x00000000

4.4 Disconnect from the Server

The client has already established a Session Context with the server and has a valid session context handle. For more information, see section 4.1.

The client is exiting and requests to destroy the Session Context on the server. The client calls the **EcDoDisconnect** method, as described in section 3.1.4.3, using the session context handle that was returned from the **EcDoConnectEx** method call, as described in section 3.1.4.1.

pcxh: Pointer to session context handle value, which is 0x00001234.

The server processes the **EcDoDisconnect** method request. The server verifies that the session context handle is for a valid Session Context for this user. The server destroys the Session Context and invalidates the session context handle. The server returns the following output values.

pcxh: Value at session context handle pointer is 0x00000000.

Return Value: 0x00000000

5 Security

5.1 Security Considerations for Implementers

To reduce exploits of server code, it is recommended that anonymous access to the server not be granted. To make method calls on the **EMSMDB** and **AsyncEMSMDB** interfaces, only properly authenticated RPC binding handles are allowed.

Most of the **EMSMDB** and **AsyncEMSMDB** interface methods require a session context handle, which can only be created from a successful call to the **EcDoConnectEx** method, as described in section 3.1.4.1. The server verifies that the authentication context on the RPC binding handle has sufficient **permissions** to access the server and create a Session Context. These method RPCs are used by the client to create a Session Context with the server. They are also used to declare to the server who is attempting to access messaging data on the server through the DN passed in the *szUserDN* parameter. It is recommended that the server verify that the authentication context on the RPC binding handle has ownership permissions to the directory service object that is associated with the DN. If the authentication context does not have adequate permissions, the server fails the call and does not create a Session Context.

Although the protocol allows for data compression and data obfuscation on the **EcDoRpcExt2** method specified in section 3.1.4.2, it is recommended that data compression and data obfuscation not be used in place of proper encryption. It is recommended that RPC-level encryption be used by the client when establishing a connection with the server. This will properly encrypt all parameters of all method RPCs on the **EMSMDB** and **AsyncEMSMDB** interfaces.

5.2 Index of Security Parameters

None.

6 Appendix A: Full IDL

For ease of implementation, the following full IDL is provided, where "ms-dtyp.idl" refers to the IDL found in [MS-DTYP] Appendix A. The syntax uses the IDL syntax extensions defined in [MS-RPCE]. For example, as noted in [MS-RPCE], a pointer_default declaration is not required and pointer_default(unique) is assumed.

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

typedef [context_handle] void * CXH;
typedef [context_handle] void * ACXH;
// Special restricted types to prevent allocation of big buffers.
typedef [range(0x0, 0x40000)] unsigned long BIG_RANGE_ULONG;
typedef [range(0x0, 0x1008)] unsigned long SMALL_RANGE_ULONG;

 [ uuid (A4F1DB00-CA47-1067-B31F-00DD010662DA),
  version(0.81),
  pointer_default(unique)]
interface emsmdb
{
long   __stdcall Opnum0Reserved(
);

long __stdcall EcDoDisconnect(
[in, out, ref] CXH * pcxh
);

long __stdcall Opnum2Reserved(
);

long __stdcall Opnum3Reserved(
);

long __stdcall EcRRegisterPushNotification(
[in, out, ref] CXH * pcxh,
[in] unsigned long iRpc,
[in, size_is(cbContext)]unsigned char rgbContext[],
[in] unsigned short cbContext,
[in] unsigned long grbitAdviseBits,
[in, size_is(cbCallbackAddress)] unsigned char rgbCallbackAddress[],
[in] unsigned short cbCallbackAddress,
[out] unsigned long *hNotification
);

long __stdcall Opnum5Reserved(
);

long __stdcall EcDummyRpc(
[in] handle_t hBinding
);

long   __stdcall Opnum7Reserved(
);

long __stdcall Opnum8Reserved(
);

long   __stdcall Opnum9Reserved(
);

long __stdcall EcDoConnectEx(
[in] handle_t hBinding,
[out, ref] CXH * pcxh,
[in, string] unsigned char * szUserDN,
[in] unsigned long ulFlags,
[in] unsigned long ulConMod,
[in] unsigned long cbLimit,
```

```
[in] unsigned long ulCpid,
[in] unsigned long ulLcidString,
[in] unsigned long ulLcidSort,
[in] unsigned long ulIcxrLink,
[in] unsigned short usFCanConvertCodePages,
[out] unsigned long * pcmsPollsMax,
[out] unsigned long * pcRetry,
[out] unsigned long * pcmsRetryDelay,
[out] unsigned short * picxr,
[out, string] unsigned char **szDNPrefix,
[out, string] unsigned char **szDisplayName,
[in] unsigned short rgwClientVersion[3],
[out] unsigned short rgwServerVersion[3],
[out] unsigned short rgwBestVersion[3],
[in, out] unsigned long * pulTimeStamp,
[in, size_is(cbAuxIn)] unsigned char rgbAuxIn[],
[in] unsigned long cbAuxIn,
[out, length_is(*pcbAuxOut), size_is(*pcbAuxOut)] unsigned char rgbAuxOut[],
[in, out] SMALL_RANGE_ULONG *pcbAuxOut
);

long __stdcall EcDoRpcExt2(
[in, out, ref] CXH * pcxh,
[in, out] unsigned long *pulFlags,
[in, size_is(cbIn)] unsigned char rgbIn[],
[in] unsigned long cbIn,
[out, length_is(*pcbOut), size_is(*pcbOut)] unsigned char rgbOut[],
[in, out] BIG_RANGE_ULONG *pcbOut,
[in, size is(cbAuxIn)] unsigned char rgbAuxIn[],
[in] unsigned long cbAuxIn,
[out, length is(*pcbAuxOut), size is(*pcbAuxOut)] unsigned char rgbAuxOut[],
[in, out] SMALL_RANGE_ULONG *pcbAuxOut,
[out] unsigned long *pulTransTime
);

long   stdcall Opnum12Reserved(
);

long __stdcall Opnum13Reserved(
);

long   stdcall EcDoAsyncConnectEx(
[in] CXH cxh,
[out, ref] ACXH * pacxh
);

}

[ uuid (5261574A-4572-206E-B268-6B199213B4E4),
  version(0.01),
  pointer_default(unique)]
interface asyncemsmdb
{
long   stdcall EcDoAsyncWaitEx(
[in] ACXH acxh,
[in] unsigned long ulFlagsIn,
[out] unsigned long *pulFlagsOut
);

}
```

7 Appendix B: Product Behavior

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

- Microsoft Exchange Server 2003
- Microsoft Exchange Server 2007
- Microsoft Exchange Server 2010
- Microsoft Exchange Server 2013
- Microsoft Exchange Server 2016
- Microsoft Office Outlook 2003
- Microsoft Office Outlook 2007
- Microsoft Outlook 2010
- Microsoft Outlook 2013
- Microsoft Outlook 2016

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

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

<1> Section 2.1: The following tables indicate which product versions support a given protocol sequence.

Protocol sequence	Exchange 2003	Exchange 2007	Exchange 2010	Exchange 2013	Exchange 2016
ncacn_ip_tcp	X	X	X		
ncacn_http	X	X	X	X	X

Protocol sequence	Office Outlook 2003	Office Outlook 2007	Outlook 2010	Outlook 2013	Outlook 2016
ncacn_ip_tcp	X	X	X	X	X
ncacn_http	X	X	X	X	X

<2> Section 2.2.2.2: Exchange 2003, Exchange 2007, and Exchange 2010 do not return the **AUX_ENDPOINT_CAPABILITIES** auxiliary block structure. Office Outlook 2003, the initial release version of Office Outlook 2007, Microsoft Office Outlook 2007 Service Pack 1, Microsoft Office Outlook 2007 Service Pack 2 (SP2), and the initial release version of Microsoft Outlook 2010 ignore the

AUX_ENDPOINT_CAPABILITIES auxiliary block structure. Microsoft Outlook 2010 Service Pack 1 (SP1), Outlook 2013, and Outlook 2016 support the **AUX_ENDPOINT_CAPABILITIES** auxiliary block structure.

<3> Section 2.2.2.2.17: Exchange 2003, Exchange 2007, and Exchange 2010 do not set this flag. Office Outlook 2003, Office Outlook 2007, and Outlook 2010 ignore this flag.

<4> Section 2.2.2.2.19: Exchange 2003, Exchange 2007, and Exchange 2010 do not support combined RFRI, NSPI, and EMSMDB interfaces on the same connection.

<5> Section 3.1.3: The following tables indicate which product versions support each authentication method.

Authentication method	Exchange 2003 and Office Outlook 2003	Exchange 2007 and Office Outlook 2007	Exchange 2010 and Outlook 2010	Exchange 2013 and Outlook 2013	Exchange 2016 and Outlook 2016
RPC_C_AUTHN_WINNT	X	X	X	X	X
RPC_C_AUTHN_GSS_KERBEROS	X	X	X		
RPC_C_AUTHN_GSS_NEGOTIATE	X	X	X	X	X
RPC_C_AUTHN_NONE				X	X

<6> Section 3.1.3: Exchange 2003, Exchange 2007, Exchange 2010, Office Outlook 2003, Office Outlook 2007, and Outlook 2010 use "exchangeMDB/<Mailbox server FQDN>" as the service principal name (SPN) for the Kerberos authentication method.

<7> Section 3.1.4: The following table indicates which **EMSMDB** methods are supported in which product versions.

Method	Exchange 2003	Exchange 2007	Exchange 2010	Exchange 2013	Exchange 2016
EcDoDisconnect	X	X	X	X	X
EcRRegisterPushNotification	X	X	See section 3.1.4.5.		
EcDummyRpc	X	X	X	X	X
EcDoConnectEx	X	X	X	X	X
EcDoRpcExt2	X	X	X	X	X
EcDoAsyncConnectEx		X	X	X	X

<8> Section 3.1.4.1: Exchange 2010, Exchange 2013, and Exchange 2016 do not support Session Context linking. If the value of the *ulIcxrLink* parameter is not 0xFFFFFFFF, the server will not attempt to search for a session with the same Session Context and link to them. It will then return the same value in the *pulTimeStamp* parameter that was passed in.

<9> Section 3.1.4.1: In Exchange 2003 and the initial release version of Exchange 2007, the server counts individual connections for CAL accounting, so Session Context linking is useful in a call to the **EcDoConnectEx** method on the **EMSMDB** interface.

<10> Section 3.1.4.1: Exchange 2010, Exchange 2013, and Exchange 2016 do not support Session Context linking.

<11> Section 3.1.4.1: Exchange 2010, Exchange 2013, and Exchange 2016 do not support Session Context linking. If the *ulIcxrLink* parameter is not 0xFFFFFFFF, the server will not attempt to search for a session with the same Session Context and link to it. Rather, it will then return the same value in the *pulTimeStamp* parameter that was passed in.

<12> Section 3.1.4.1: Exchange 2007 fails with return code 0x80040110 when the value of the *cbAuxIn* parameter on input is larger than 0x00001008.

<13> Section 3.1.4.1: The initial release version of Exchange 2010 will fail with ecInvalidParam (0x80070057) if the *cbAuxIn* parameter is greater than 0x00000000 and less than 0x00000008.

<14> Section 3.1.4.1: Exchange 2007 does not fail if the *cbAuxIn* parameter is greater than 0x00000000 and less than 0x00000008.

<15> Section 3.1.4.1: Exchange 2003 and Exchange 2007 return ecRpcAuthentication (0x000004B6) if the authentication context associated with the binding handle does not have enough privilege and if the *szUserDn* parameter is not empty. If the *szUserDN* parameter is empty, Exchange 2003 and Exchange 2007 return ecNone (0x00000000).

<16> Section 3.1.4.1.2.1: Exchange 2010, Exchange 2013, and Exchange 2016 do not support sending the **AUX_OSVERSIONINFO** auxiliary block.

<17> Section 3.1.4.1.2.1: Exchange 2003 and Office Outlook 2003 do not support the **AUX_SERVER_SESSION_INFO** auxiliary block.

<18> Section 3.1.4.2: Exchange 2010 returns the same value on output as was input, not 0x00000000.

<19> Section 3.1.4.2: Exchange 2003, Exchange 2007 and Exchange 2010 will fail with error code ecRpcFormat (0x000004B6) if the request buffer is larger than 0x00008007 bytes in size. Microsoft Exchange Server 2010 Service Pack 2 (SP2), Microsoft Exchange Server 2013 Service Pack 1 (SP1), and Exchange 2016 will fail with error code ecRpcFailed (0x80040115) if the request buffer is larger than 0x00008007 bytes in size.

<20> Section 3.1.4.2: Exchange 2003, Exchange 2007, and Microsoft Exchange Server 2010 Service Pack 1 (SP1) fail with error code ecRpcFormat (0x000004B6) if the value in the *cbIn* parameter is less than 0x00000008. The initial release version of Exchange 2010 will not allow a *cbIn* parameter value smaller than 0x00000008.

<21> Section 3.1.4.2: Exchange 2003 and Exchange 2007 will fail with ecRpcFormat (0x000004B6) if the output buffer is less than 0x00008007. Exchange 2013 and Exchange 2016 will succeed if output buffer is less than 0x00000008, but no request ROPs will have been processed.

<22> Section 3.1.4.2: Exchange 2007 fails with return code 0x80040110 if the request buffer value of the *cbAuxIn* parameter is larger than 0x00001008 bytes in size.

<23> Section 3.1.4.2: Exchange 2010 will fail with ecRpcFailed (0x80040115) if the value of the *cbAuxIn* parameter is greater than 0x00000000 and less than 0x00000008.

<24> Section 3.1.4.2: Exchange 2003, Exchange 2007, Exchange 2013, and Exchange 2016 support returning data in the *rgbAuxOut* parameter.

<25> Section 3.1.4.2.2.1: Exchange 2003, Exchange 2007, Exchange 2013, and Exchange 2016 support returning data in the *rgbAuxOut* parameter.

<26> Section 3.1.4.4: Exchange 2010, Exchange 2013, and Exchange 2016 do not return the ecRejected error code.

<27> Section 3.1.4.5: Exchange 2003 and Exchange 2007 do support the **EcRRegisterPushNotification** method. The initial release version of Exchange 2010 and Exchange 2010 SP1 do not support the **EcRRegisterPushNotification** method and always return ecNotSupported. Exchange 2010 SP2 supports the **EcRRegisterPushNotification** method when a registry key is created to support push notifications, as described in [MSFT-ConfigStaticUDPPort]. Exchange 2013 and Exchange 2016 do not support the **EcRRegisterPushNotification** method and always returns ecNotSupported.

<28> Section 3.2.4: The following table indicates which **EMSMDB** interface methods are used by a client when accessing a server that is running Exchange 2003.

Method	Office Outlook 2003	Office Outlook 2007	Outlook 2010	Outlook 2013	Outlook 2016
EcDoDisconnect	X	X	X	X	X
EcRRegisterPushNotification	X	X	X		
EcDummyRpc					
EcDoConnectEx	X	X	X	X	X
EcDoRpcExt2	X	X	X	X	X
EcDoAsyncConnectEx					

The following table indicates which **EMSMDB** interface methods are used by a client when it is accessing a server that is running Exchange 2007.

Method	Office Outlook 2003	Office Outlook 2007	Outlook 2010	Outlook 2013	Outlook 2016
EcDoDisconnect	X	X	X	X	X
EcRRegisterPushNotification	X	X	X		
EcDummyRpc					
EcDoConnectEx	X	X	X	X	X
EcDoRpcExt2	X	X	X	X	X
EcDoAsyncConnectEx		X	X	X	X

The following table indicates which **EMSMDB** interface methods are used by a client when it is accessing a server that is running Exchange 2010.

Method	Office Outlook 2003	Office Outlook 2007	Outlook 2010	Outlook 2013	Outlook 2016
EcDoDisconnect	X	X	X	X	X
EcRRegisterPushNotification	See section 3.1.4.5.				
EcDummyRpc					
EcDoConnectEx	X	X	X	X	X

Method	Office Outlook 2003	Office Outlook 2007	Outlook 2010	Outlook 2013	Outlook 2016
EcDoRpcExt2	X	X	X	X	X
EcDoAsyncConnectEx		X	X	X	X

The following table indicates which **EMSMDB** interface methods are used by a client when it is accessing a server that is running Exchange 2013 or Exchange 2016.

Method	Office Outlook 2003	Office Outlook 2007	Outlook 2010	Outlook 2013	Outlook 2016
EcDoDisconnect	X	X	X	X	X
EcRRegisterPushNotification	See section 3.1.4.5.				
EcDummyRpc					
EcDoConnectEx	X	X	X	X	X
EcDoRpcExt2	X	X	X	X	X
EcDoAsyncConnectEx		X	X	X	X

<29> Section 3.2.4.1: Exchange 2010, Exchange 2013, and Exchange 2016 not support Session Context linking.

<30> Section 3.2.4.1: In Exchange 2003 and the initial release version of Exchange 2007, the server counts individual connections for CAL accounting, so Session Context linking is useful in the **EcDoConnectEx** method on the **EMSMDB** interface.

<31> Section 3.2.4.1: Exchange 2010, Exchange 2013, and Exchange 2016 do not support Session Context linking.

<32> Section 3.2.4.1: Exchange 2010, Exchange 2013, and Exchange 2016 do not support Session Context linking.

<33> Section 3.2.4.1.2.1: Outlook 2010 by default does not populate the **MachineName**, **UserName**, **ClientIP**, and **MacAddress** fields within the **AUX_PERF_CLIENTINFO** auxiliary block structure.

<34> Section 3.2.4.1.2.1: Exchange 2003 and Office Outlook 2003 do not support the **AUX_CLIENT_CONNECTION_INFO** auxiliary block structure.

<35> Section 3.2.4.6: Office Outlook 2003, the initial release version of Office Outlook 2007, Office Outlook 2007 SP1, Office Outlook 2007 SP2, and Outlook 2010 do not support port consolidation. Microsoft Office Outlook 2007 Service Pack 3 (SP3) supports port consolidation. Clients that do not support port consolidation ignore the **AUX_ENDPOINT_CAPABILITIES** auxiliary block structure, as described in section 2.2.2.2.19. Office Outlook 2007 SP3, Outlook 2013, and Outlook 2016 support port consolidation.

<36> Section 3.3.4: The following table indicates which **AsyncEMSMDB** interface methods are supported in which product versions.

Method	Exchange 2003	Exchange 2007	Exchange 2010	Exchange 2013	Exchange 2016
EcDoAsyncWaitEx		X	X	X	X

<37> Section 3.3.4.1: Exchange 2003 and Exchange 2007 complete the call every 5 minutes regardless of the client's last activity time.

<38> Section 3.3.4.1: Exchange 2007 and Exchange 2010 also reject the request if the asynchronous context handle is not valid.

<39> Section 3.4.4: The **AsyncEMSMDB** interface methods are not used by a client when accessing a server that is running Exchange 2003. The following table indicates which **AsyncEMSMDB** interface methods are used by a client when accessing a server that is running Exchange 2007, Exchange 2010, Exchange 2013, or Exchange 2016.

Method	Office Outlook 2003	Office Outlook 2007	Outlook 2010	Outlook 2013	Outlook 2016
EcDoAsyncWaitEx		X	X	X	X

8 Change Tracking

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

9 Index

A

Abstract data model
  client (section 3.2.1 62, section 3.4.1 74)
  server (section 3.1.1 37, section 3.3.1 71)
Applicability 13
asyncemsmdb interface 71
AUX_HEADER structure 17

C

Capability negotiation 13
Change tracking 90
Client
  abstract data model (section 3.2.1 62, section 3.4.1 74)
  Handling Connection Failures method 71
  Handling Endpoint Consolidation method 71
  Handling Server Too Busy method 71
  initialization (section 3.2.3 62, section 3.4.3 74)
  local events (section 3.2.6 71, section 3.4.6 75)
  message processing (section 3.2.4 63, section 3.4.4 74)
  Sending the EcDoConnectEx Method method 63
  Sending the EcDoDisconnect Method method 70
  Sending the EcDoRpcExt2 Method method 67
  sequencing rules (section 3.2.4 63, section 3.4.4 74)
  timer events (section 3.2.5 71, section 3.4.5 75)
  timers (section 3.2.2 62, section 3.4.2 74)
Common data types 15
  Simple Data Types 16
Connect to the server example 76

D

Data model - abstract
  client (section 3.2.1 62, section 3.4.1 74)
  server (section 3.1.1 37, section 3.3.1 71)
Data types
  common - overview 15
Disconnect from the server example 80

E

EcDoAsyncConnectEx Method (Opnum 14) method 59
EcDoAsyncWaitEx Method (Opnum 0) method 73
EcDoConnectEx Method (Opnum 10) method 40
EcDoDisconnect Method (Opnum 1) method 58
EcDoRpcExt2 Method (Opnum 11) method 53
EcDummyRpc Method (Opnum 6) method 61
EcRRegisterPushNotification Method (Opnum 4) method 59
emsmdb interface 37
Events
  local - client (section 3.2.6 71, section 3.4.6 75)
  local - server (section 3.1.6 62, section 3.3.6 74)
  timer - client (section 3.2.5 71, section 3.4.5 75)
  timer - server (section 3.1.5 62, section 3.3.5 74)
Examples
  connect to the server 76
  disconnect from the server 80
  issue rop commands to the server 77
  overview 76
  receive packed rop responses from the server 79

F

Fields - vendor-extensible 14
Full IDL 82

G

Glossary 7

H

Handling Connection Failures method 71
Handling Endpoint Consolidation method 71
Handling Server Too Busy method 71

I

IDL 82
Implementer - security considerations 81
Index of security parameters 81
Informative references 10
Initialization
  client (section 3.2.3 62, section 3.4.3 74)
  server (section 3.1.3 38, section 3.3.3 72)
Interfaces - server
  asyncemsmdb 71
  emsmdb 37
Introduction 7
Issue rop commands to the server example 77

L

Local events
  client (section 3.2.6 71, section 3.4.6 75)
  server (section 3.1.6 62, section 3.3.6 74)

M

Message processing
  client (section 3.2.4 63, section 3.4.4 74)
  server (section 3.1.4 38, section 3.3.4 72)
Messages
  common data types 15
  transport 15
Methods
  EcDoAsyncConnectEx Method (Opnum 14) 59
  EcDoAsyncWaitEx Method (Opnum 0) 73
  EcDoConnectEx Method (Opnum 10) 40
  EcDoDisconnect Method (Opnum 1) 58
  EcDoRpcExt2 Method (Opnum 11) 53
  EcDummyRpc Method (Opnum 6) 61
  EcRRegisterPushNotification Method (Opnum 4) 59
  Handling Connection Failures 71
  Handling Endpoint Consolidation 71
  Handling Server Too Busy 71
  Opnum0NotUsedOnWire Method (Opnum 0) 61
  Opnum12NotUsedOnWire Method (Opnum 12) 62

Opnum13NotUsedOnWire Method (Opnum 13) 62
Opnum2NotUsedOnWire Method (Opnum 2) 61
Opnum3NotUsedOnWire Method (Opnum 3) 61
Opnum5NotUsedOnWire Method (Opnum 5) 61
Opnum7NotUsedOnWire Method (Opnum 7) 61
Opnum8NotUsedOnWire Method (Opnum 8) 62
Opnum9NotUsedOnWire Method (Opnum 9) 62
Sending the EcDoConnectEx Method 63
Sending the EcDoDisconnect Method 70
Sending the EcDoRpcExt2 Method 67

N

Normative references 10

O

Opnum0NotUsedOnWire Method (Opnum 0) method 61
Opnum12NotUsedOnWire Method (Opnum 12) method 62
Opnum13NotUsedOnWire Method (Opnum 13) method 62
Opnum2NotUsedOnWire Method (Opnum 2) method 61
Opnum3NotUsedOnWire Method (Opnum 3) method 61
Opnum5NotUsedOnWire Method (Opnum 5) method 61
Opnum7NotUsedOnWire Method (Opnum 7) method 61
Opnum8NotUsedOnWire Method (Opnum 8) method 62
Opnum9NotUsedOnWire Method (Opnum 9) method 62
Overview (synopsis) 11

P

Parameters - security index 81
Preconditions 13
Prerequisites 13
Product behavior 84
Protocol Details
  overview 37
Protocol details overview 37

R

Receive packed rop responses from the server example 79
References 9
  informative 10
  normative 10
Relationship to other protocols 13
RPC_HEADER_EXT structure 17

S

Security
  implementer considerations 81
  parameter index 81
Sending the EcDoConnectEx Method method 63
Sending the EcDoDisconnect Method method 70
Sending the EcDoRpcExt2 Method method 67
Sequencing rules
  client (section 3.2.4 63, section 3.4.4 74)
  server (section 3.1.4 38, section 3.3.4 72)
Server
  abstract data model (section 3.1.1 37, section 3.3.1 71)
  asyncemsmdb interface 71
  EcDoAsyncConnectEx Method (Opnum 14) method 59
  EcDoAsyncWaitEx Method (Opnum 0) method 73
  EcDoConnectEx Method (Opnum 10) method 40
  EcDoDisconnect Method (Opnum 1) method 58
  EcDoRpcExt2 Method (Opnum 11) method 53
  EcDummyRpc Method (Opnum 6) method 61
  EcRRegisterPushNotification Method (Opnum 4) method 59
  emsmdb interface 37
  initialization (section 3.1.3 38, section 3.3.3 72)
  local events (section 3.1.6 62, section 3.3.6 74)
  message processing (section 3.1.4 38, section 3.3.4 72)
  Opnum0NotUsedOnWire Method (Opnum 0) method 61
  Opnum12NotUsedOnWire Method (Opnum 12) method 62
  Opnum13NotUsedOnWire Method (Opnum 13) method 62
  Opnum2NotUsedOnWire Method (Opnum 2) method 61
  Opnum3NotUsedOnWire Method (Opnum 3) method 61
  Opnum5NotUsedOnWire Method (Opnum 5) method 61
  Opnum7NotUsedOnWire Method (Opnum 7) method 61
  Opnum8NotUsedOnWire Method (Opnum 8) method 62
  Opnum9NotUsedOnWire Method (Opnum 9) method 62
  overview (section 3.1 37, section 3.3 71)
  sequencing rules (section 3.1.4 38, section 3.3.4 72)
  timer events (section 3.1.5 62, section 3.3.5 74)
  timers (section 3.1.2 38, section 3.3.2 72)
Simple Data Type common data types 16
Standards assignments 14
Structures
  AUX_HEADER 17
  RPC_HEADER_EXT 17

T

Timer events
  client (section 3.2.5 71, section 3.4.5 75)
  server (section 3.1.5 62, section 3.3.5 74)
Timers
  client (section 3.2.2 62, section 3.4.2 74)
  server (section 3.1.2 38, section 3.3.2 72)
Tracking changes 90
Transport 15

V

Vendor-extensible fields 14

Versioning 13