

[MS-OXCRPC]: Wire Format Protocol Specification

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's Open Specification Promise (available here: <http://www.microsoft.com/interop/osp>) or the Community Promise (available here: <http://www.microsoft.com/interop/cp/default.mspx>). 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 iplq@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.
- **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
04/04/2008	0.1		Initial Availability.
04/25/2008	0.2		Revised and updated property names and other technical content.
06/27/2008	1.0		Initial Release.
08/06/2008	1.01		Revised and edited technical content.
09/03/2008	1.02		Revised and edited technical content.
10/01/2008	1.03		Revised and edited technical content.
12/03/2008	1.04		Revised and edited technical content.
03/04/2009	1.05		Revised and edited technical content.
04/10/2009	2.0		Updated technical content and applicable product releases.
07/15/2009	3.0	Major	Revised and edited for technical content.
11/04/2009	4.0.0	Major	Updated and revised the technical content.
02/10/2010	5.0.0	Major	Updated and revised the technical content.

Table of Contents

1 Introduction	6
1.1 Glossary	6
1.2 References	7
1.2.1 Normative References	7
1.2.2 Informative References	7
1.3 Protocol Overview	7
1.3.1 Initiating Communication with the Server	8
1.3.2 Issuing Remote Operations for Mailbox Data	8
1.3.3 Terminating Communication with the Server	8
1.3.4 Client/Server Communication Lifetime	8
1.4 Relationship to Other Protocols	9
1.5 Prerequisites/Preconditions	9
1.6 Applicability Statement	10
1.7 Versioning and Capability Negotiation	10
1.8 Vendor-Extensible Fields	10
1.9 Standards Assignments	10
2 Messages	11
2.1 Transport	11
2.2 Common Data Types	11
2.2.1 Simple Data Types	12
2.2.1.1 CXH	12
2.2.1.2 ACXH	12
2.2.1.3 BIG_RANGE_ULONG	12
2.2.1.4 SMALL_RANGE_ULONG	13
2.2.2 Structures	13
2.2.2.1 RPC_HEADER_EXT	13
2.2.2.2 AUX_HEADER	14
2.2.2.3 AUX_PERF_REQUESTID	16
2.2.2.4 AUX_PERF_SESSIONINFO	16
2.2.2.5 AUX_PERF_SESSIONINFO_V2	17
2.2.2.6 AUX_PERF_CLIENTINFO	17
2.2.2.7 AUX_PERF_SERVERINFO	20
2.2.2.8 AUX_PERF_PROCESSINFO	21
2.2.2.9 AUX_PERF_DEFMDB_SUCCESS	22
2.2.2.10 AUX_PERF_DEFGC_SUCCESS	22
2.2.2.11 AUX_PERF_MDB_SUCCESS	23
2.2.2.12 AUX_PERF_MDB_SUCCESS_V2	23
2.2.2.13 AUX_PERF_GC_SUCCESS	24
2.2.2.14 AUX_PERF_GC_SUCCESS_V2	25
2.2.2.15 AUX_PERF_FAILURE	25
2.2.2.16 AUX_PERF_FAILURE_V2	26
2.2.2.17 AUX_CLIENT_CONTROL	27
2.2.2.18 AUX_OSVERSIONINFO<>	28
2.2.2.19 AUX_EXORGINFO	29
2.2.2.20 AUX_PERF_ACCOUNTINFO	29
3 Protocol Details	30
3.1 EMSMDB Server Details	30
3.1.1 Abstract Data Model	30

3.1.2	Timers	31
3.1.3	Initialization	31
3.1.4	Message Processing Events and Sequencing Rules	31
3.1.4.1	Opnum0NotUsedOnWire	32
3.1.4.2	EcDoDisconnect (opnum 1)	32
3.1.4.3	Opnum2NotUsedOnWire	33
3.1.4.4	Opnum3NotUsedOnWire	33
3.1.4.5	EcRRRegisterPushNotification (opnum 4)	33
3.1.4.6	Opnum5NotUsedOnWire	34
3.1.4.7	EcDummyRpc (opnum 6)	34
3.1.4.8	Opnum7NotUsedOnWire	35
3.1.4.9	Opnum8NotUsedOnWire	35
3.1.4.10	Opnum9NotUsedOnWire	35
3.1.4.11	EcDoConnectEx (opnum 10)	35
3.1.4.12	EcDoRpcExt2 (opnum 11)	39
3.1.4.13	Opnum12NotUsedOnWire	42
3.1.4.14	Opnum13NotUsedOnWire	42
3.1.4.15	EcDoAsyncConnectEx (opnum 14)	42
3.1.5	Timer Events	42
3.1.6	Other Local Events	42
3.1.7	Extended Buffer Handling	43
3.1.7.1	Extended Buffer Format	43
3.1.7.1.1	EcDoConnectEx	43
3.1.7.1.1.1	rgbAuxIn	43
3.1.7.1.1.2	rgbAuxOut	43
3.1.7.1.2	EcDoRpcExt2	44
3.1.7.1.2.1	rgbIn	44
3.1.7.1.2.2	rgbOut	44
3.1.7.1.2.3	rgbAuxIn	45
3.1.7.1.2.4	rgbAuxOut	45
3.1.7.2	Compression Algorithm	45
3.1.7.2.1	LZ77 Compression Algorithm	46
3.1.7.2.1.1	Compression Algorithm Terminology	46
3.1.7.2.1.2	Using the Compression Algorithm	46
3.1.7.2.1.3	Compression Process	46
3.1.7.2.1.4	Compression Process Example	47
3.1.7.2.2	DIRECT2 Encoding Algorithm	48
3.1.7.2.2.1	Bitmask	48
3.1.7.2.2.2	Encoding Metadata	48
3.1.7.2.2.3	Metadata Offset	48
3.1.7.2.2.4	Match Length	49
3.1.7.3	Obfuscation Algorithm	51
3.1.7.4	Extended Buffer Packing	51
3.1.8	Auxiliary Buffer	52
3.1.8.1	Client Performance Monitoring	52
3.1.8.2	Server Topology Information	55
3.1.9	Version Checking	56
3.1.9.1	Version Number Comparison	56
3.1.9.2	Server Versions	57
3.1.9.3	Client Versions	58
3.2	EMSMDb Client Details	59
3.2.1	Abstract Data Model	59
3.2.2	Timers	59

3.2.3	Initialization	59
3.2.4	Message Processing Events and Sequencing Rules	59
3.2.4.1	Sending EcDoConnectEx.....	59
3.2.4.2	Sending EcDoRpcExt2	61
3.2.4.3	Handling Server Too Busy	61
3.2.4.4	Handling Connection Failures.....	61
3.2.5	Timer Events.....	61
3.2.6	Other Local Events	61
3.3	AsyncEMSMDB Server Details.....	61
3.3.1	Abstract Data Model.....	61
3.3.2	Timers	62
3.3.3	Initialization	62
3.3.4	Message Processing Events and Sequencing Rules	62
3.3.4.1	EcDoAsyncWaitEx (opnum 0)	63
3.3.5	Timer Events.....	63
3.3.6	Other Local Events	63
3.4	AsyncEMSMDB Client Details	64
3.4.1	Abstract Data Model.....	64
3.4.2	Timers	64
3.4.3	Initialization	64
3.4.4	Message Processing Events and Sequencing Rules	64
3.4.5	Timer Events.....	64
3.4.6	Other Local Events	64
4	Protocol Examples	65
4.1	Client Connecting to Server.....	65
4.2	Client Issuing ROP Commands to Server.....	66
4.3	Client Receiving "Packed" ROP Response from Server	68
4.4	Client Disconnecting from Server.....	69
5	Security.....	70
5.1	Security Considerations for Implementers.....	70
5.2	Index of Security Parameters	70
6	Appendix A: Full IDL/ACF	71
6.1	IDL	71
6.2	ACF.....	73
7	Appendix B: Product Behavior	74
7.1	Protocol Sequences	75
7.1.1	Exchange Server Support.....	75
7.1.2	Office Client Support.....	75
7.2	Authentication Methods	75
7.3	RPC Methods.....	76
7.3.1	Exchange Server Support.....	76
7.3.2	Office Client Support.....	76
7.3.2.1	Accessing Exchange 2003.....	76
7.3.2.2	Accessing Exchange 2007.....	77
7.3.2.3	Accessing Exchange 2010.....	77
7.4	Client Access Licenses	78
8	Change Tracking	79
9	Index.....	82

1 Introduction

The Wire Format protocol is specific to the **EMSMDB** and **AsyncEMSMDB** protocol interfaces between a client and server. These interfaces have traditionally been used by an Outlook client to communicate with an Exchange messaging server. This protocol extends DCE 1.1: **Remote Procedure Call** [\[C706\]](#).

1.1 Glossary

The following terms are defined in [\[MS-OXGLOS\]](#):

ANSI
binding handle
code page
distinguished name (DN)
dynamic endpoint
endpoint
exception
folder
GUID
handle
HTTP
Incremental Change Synchronization (ICS)
Interface Definition Language (IDL)
little-endian
locale
mailbox
message
messaging object
Network Data Representation (NDR)
NTLM
opnum
permissions
public folder
recipient
remote procedure call (RPC)
replica
RPC protocol sequence
remote operation (ROP)
ROP request buffer
ROP response buffer
Server object
stream
store
Unicode
universal unique identifier (UUID)
Well-known endpoint

The following terms are specific to this document:

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 opened by a client are isolated to a Session Context.

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

RPC dynamic endpoint: A network-specific server address that is requested and assigned at run time. For more information, see [\[C706\]](#) part 4

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

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, <http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624>, as an additional source.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, <http://www.opengroup.org/public/pubs/catalog/c706.htm>.

[MS-OXCFXICS] Microsoft Corporation, "[Bulk Data Transfer Protocol Specification](#)", June 2008.

[MS-OXCNOTIF] Microsoft Corporation, "[Core Notifications Protocol Specification](#)", June 2008.

[MS-OXCROPS] Microsoft Corporation, "[Remote Operations \(ROP\) List and Encoding Protocol Specification](#)", June 2008.

[MS-OXCSTOR] Microsoft Corporation, "[Store Object Protocol Specification](#)", June 2008.

[MS-OXGLOS] Microsoft Corporation, "[Exchange Server Protocols Master Glossary](#)", June 2008.

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions", July 2006, <http://go.microsoft.com/fwlink/?LinkId=112246>.

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

1.2.2 Informative References

[MSDN-RPC] Microsoft Corporation, "Remote Procedure Call", <http://go.microsoft.com/fwlink/?LinkId=112245>.

[MSDN-SOCKADDR] Microsoft Corporation, "sockaddr", <http://go.microsoft.com/fwlink/?LinkId=113717>.

1.3 Protocol Overview

This specification describes the RPC interfaces that are used by a client to communicate with a server to access personal messaging data over the Wire Format protocol. This protocol is comprised of the **EMSMDB** and **AsyncEMSMDB** RPC interfaces.

1.3.1 Initiating Communication with the Server

Before a client can retrieve private **mailbox** or **public folder** data from a server on the **EMSMDB** interface, it first makes a call to **EcDoConnectEx** and establishes a **Session Context Handle (CXH)**. The Session Context Handle (CXH) is a RPC context **handle**. The client stores this Session Context Handle (CXH) and uses it on subsequent RPC calls on the **EMSMDB** interface. The server uses the Session Context Handle (CXH) to identify the client and user who is issuing requests and under which context to perform operations against messaging data.

The **EMSMDB** interface function **EcDoConnectEx** is used to create a CXH with the server. The server verifies that the authentication context used to make the RPC function call **EcDoConnectEx** has access rights to perform operations as, or on behalf of, the user whose **distinguished name (DN)** is provided on the RPC call. This is done to validate that the client has permission to perform operations as the user specified in the RPC call. If this access check fails, the server fails the RPC call with an access denied return code.

If the security check passes, the server creates a **Session Context**. A CXH that refers to the Session Context is returned to the client in the response to **EcDoConnectEx**. The returned CXH is used in subsequent calls to the server.

1.3.2 Issuing Remote Operations for Mailbox Data

The client retrieves private mailbox or public folder data through the interface function **EcDoRpcExt2**. There are no separate interface functions to perform different operations against mailbox data. A single interface function is used to submit a group of **remote operation (ROP)** commands to the server. See [\[MS-OXCROPS\]](#) for more details about ROP commands. 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 and response buffers is not understood. See [\[MS-OXCROPS\]](#) for more details about how to interpret the ROP buffers. The **EMSMDB** interface function **EcDoRpcExt2** is just the mechanism in which to pass the ROP request buffer to the server.

In the call to **EcDoRpcExt2**, the client passes the CXH which was created in a successful call to the interface function **EcDoConnectEx**. The server uses the CXH to identify who is issuing the remote operation ROP commands and under which Session Context to perform them.

1.3.3 Terminating Communication with the Server

When a client wants to terminate communication with a server, it calls **EcDoDisconnect**. In the call to **EcDoDisconnect**, the client passes the CXH, which was created in a successful call to the interface function **EcDoConnectEx**. It is suggested that the server clean up any Session Context data associated with this CXH.

1.3.4 Client/Server Communication Lifetime

Figure 1 shows a typical example of the client and server communication lifetime. This is a simplified overview of how the client connects, issues ROP commands, and disconnects from the server.

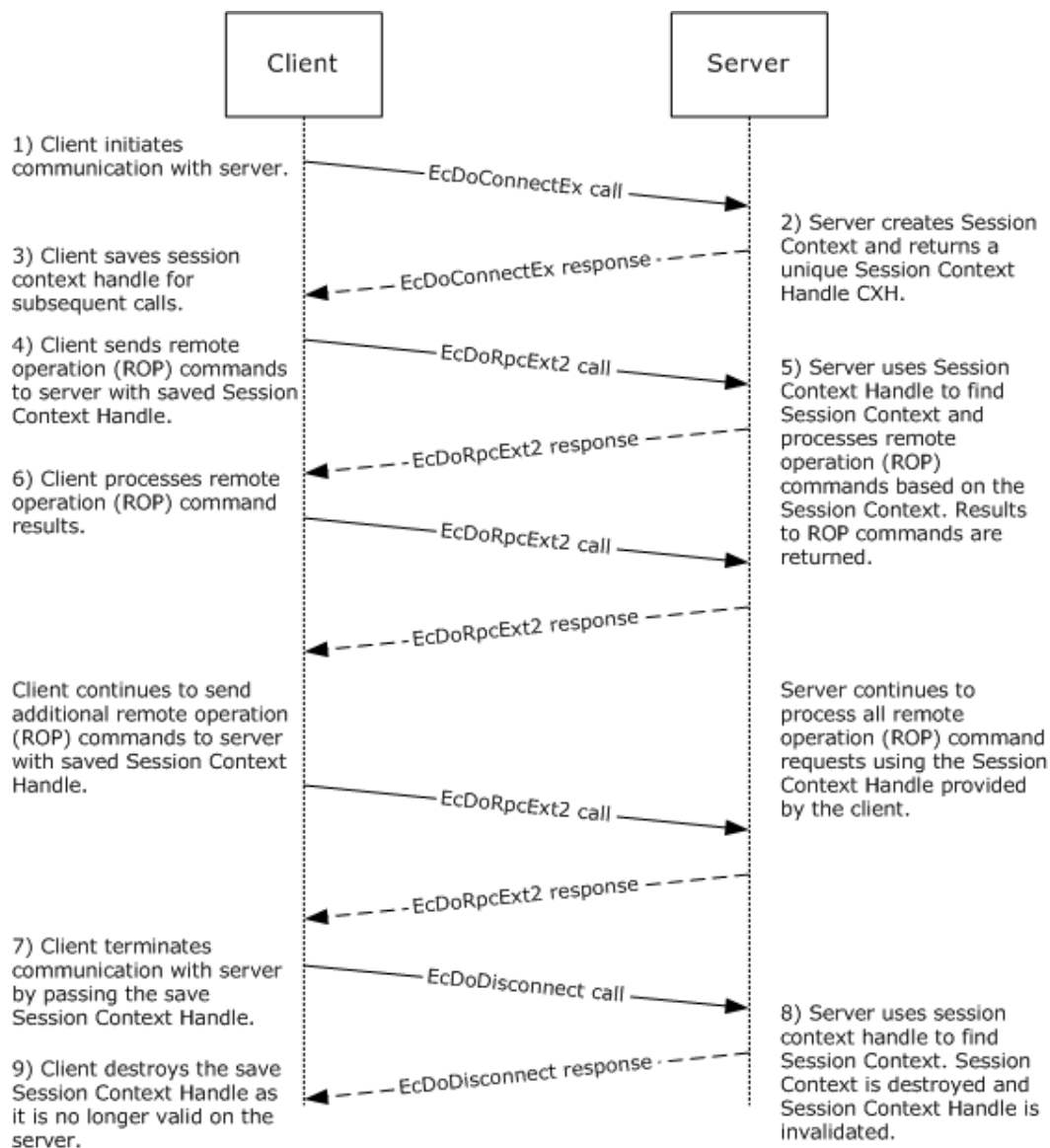


Figure 1: Client/server communications

1.4 Relationship to Other Protocols

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

1.5 Prerequisites/Preconditions

The Wire Format protocol is comprised of the EMSMDB and AsyncEMSMDB RPC interfaces and has the same prerequisites as specified in [\[MS-RPCE\]](#).

It is assumed that a messaging client has obtained the name of a messaging 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

The protocol specified in this document 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 specified in section [2.1](#).
- **Protocol Versions:** The protocol RPC interface **EMSMB** has a single version number of 0.81 and has been extended by adding methods as specified in section [3.1](#). The protocol RPC interface **AsyncEMSMB** has a single version number of 0.01.
- **Security and Authentication Methods:** This protocol supports the following authentication methods: **NTLM**, Kerberos, and Negotiate. These authentication methods are specified 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 must be configured with the interface version to use, as described below in this specification.

1.8 Vendor-Extensible Fields

None.

1.9 Standards Assignments

Parameter	Value	Reference
EMSMB RPC Interface UUID	A4F1DB00-CA47-1067-B31F-00DD010662DA	3.1
AsyncEMSMB RPC Interface UUID	5261574A-4572-206E-B268-6B199213B4E4	3.3
RPC/HTTP protocol sequence endpoint	6001	2.1
LRPC protocol sequence endpoint	MSExchangeIS_LPC <1>	2.1

2 Messages

Unless otherwise specified, buffers and fields in this section are depicted in little-endian byte order.

2.1 Transport

This protocol works over the following protocol sequences:

Protocol Sequence
ncalrpc<2>
ncacn_ip_tcp
ncacn_http

This protocol uses **well-known endpoints** for network protocol sequences "ncalrpc" and "ncacn_http". The following well-known endpoints are used:

Protocol Sequence	Endpoint
ncalrpc	MSExchangeIS_LPC <3>
ncacn_http	6001

For ncacn_ip_tcp, the protocol uses **RPC dynamic endpoints** as specified in [\[MS-OXGLOS\]](#).

This protocol MUST use the UUID specified in section [1.9](#). The RPC version number is 4.0.

This protocol allows any user to establish an authenticated connection to the RPC server 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

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

The following table summarizes the types that are defined in this specification.

Type	Name
Simple Data Type	CXH
Simple Data Type	ACXH
Simple Data Type	BIG_RANGE_ULONG
Simple Data Type	SMALL_RANGE_ULONG
Structure	RPC_HEADER_EXT
Structure	AUX_HEADER

Type	Name
Structure	AUX_PERF_REQUESTID
Structure	AUX_PERF_SESSIONINFO
Structure	AUX_PERF_SESSIONINFO_V2
Structure	AUX_PERF_CLIENTINFO
Structure	AUX_PERF_SERVERINFO
Structure	AUX_PERF_PROCESSINFO
Structure	AUX_PERF_DEFMDB_SUCCESS
Structure	AUX_PERF_DEFGC_SUCCESS
Structure	AUX_PERF_MDB_SUCCESS
Structure	AUX_PERF_MDB_SUCCESS_V2
Structure	AUX_PERF_GC_SUCCESS
Structure	AUX_PERF_GC_SUCCESS_V2
Structure	AUX_PERF_FAILURE
Structure	AUX_PERF_FAILURE_V2
Structure	AUX_CLIENT_CONTROL
Structure	AUX_OSVERSIONINFO<>
Structure	AUX_EXORGINFO

2.2.1 Simple Data Types

The Simple Data Types as identified in the IDL. See Section [6.1](#)

2.2.1.1 CXH

A Session Context Handle to be used with an **EMSMDB** interface.

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

2.2.1.2 ACXH

A Session Context Handle to be used with an **AsyncEMSMDB** interface.

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

2.2.1.3 BIG_RANGE_ULONG

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

An unsigned long that MUST be between 0x0 and 0x1008.

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

2.2.2 Structures

2.2.2.1 RPC_HEADER_EXT

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Version																Flags															
Size																SizeActual															

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

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

Flag	Meaning
Compressed 0x0001	The data that follows the RPC_HEADER_EXT is compressed. The size of the data when uncompressed is in field SizeActual . If this flag is not set, the Size and SizeActual fields MUST be the same.
XorMagic 0x0002	The data following the RPC_HEADER_EXT has been obfuscated. See section 3.1.7.3 for more information about the obfuscation algorithm.
Last 0x0004	Indicates that no other RPC_HEADER_EXT follows the data of the current RPC_HEADER_EXT . This flag is used to indicate 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 **Size**.

2.2.2.2 AUX_HEADER

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Size																Version						Type									

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

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

The following is a list of block types and the corresponding structure that follows the **AUX_HEADER** when the **Version** field is **AUX_VERSION_1**.

Value	Meaning
AUX_TYPE_PERF_REQUESTID 0x01	AUX_PERF_REQUESTID
AUX_TYPE_PERF_CLIENTDINFO 0x02	AUX_PERF_CLIENTINFO
AUX_TYPE_PERF_SERVERINFO 0x03	AUX_PERF_SERVERINFO
AUX_TYPE_PERF_SESSIONINFO 0x04	AUX_PERF_SESSIONINFO
AUX_TYPE_PERF_DEFMDB_SUCCESS 0x05	AUX_PERF_DEFMDB_SUCCESS
AUX_TYPE_PERF_DEFGC_SUCCESS 0x06	AUX_PERF_DEFGC_SUCCESS
AUX_TYPE_PERF_MDB_SUCCESS 0x07	AUX_PERF_MDB_SUCCESS
AUX_TYPE_PERF_GC_SUCCESS 0x08	AUX_PERF_GC_SUCCESS
AUX_TYPE_PERF_FAILURE	AUX_PERF_FAILURE

Value	Meaning
0x09	
AUX_TYPE_CLIENT_CONTROL 0x0A	AUX_CLIENT_CONTROL
AUX_TYPE_PERF_PROCESSINFO 0x0B	AUX_PERF_PROCESSINFO
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_FAILURE
AUX_TYPE_PERF_FG_FAILURE 0x15	AUX_PERF_GC_SUCCESS
AUX_TYPE_OSVERSIONINFO 0x16	AUX_OSVERSIONINFO
AUX_TYPE_EXORGINO 0x17	AUX_EXORGINO

The following is a list of block types and the corresponding structure that follows the **AUX_HEADER** when the **Version** field is **AUX_VERSION_2**.

Type	Meaning
AUX_TYPE_PERF_SESSIONINFO 0x04	AUX_PERF_SESSIONINFO_V2
AUX_TYPE_PERF_MDB_SUCCESS 0x07	AUX_PERF_MDB_SUCCESS_V2
AUX_TYPE_PERF_GC_SUCCESS	AUX_PERF_GC_SUCCESS_V2

Type	Meaning
0x08	
AUX_TYPE_PERF_FAILURE 0x09	AUX_PERF_FAILURE_V2
AUX_TYPE_PERF_PROCESSINFO 0x0B	AUX_PERF_PROCESSINFO AUX_TYPE_PERF_FG_FAILURE
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

Any other block type and version combination that is not understood MUST be ignored.

2.2.2.3 AUX_PERF_REQUESTID

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
SessionID																RequestID															

SessionID (2 bytes): Session identification number.

RequestID (2 bytes): Request identification

2.2.2.4 AUX_PERF_SESSIONINFO

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
SessionID																Reserved															
SessionGuid																															
...																															

...
...

SessionID (2 bytes): Session identification number.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the **stream**. The server MUST ignore the value of this field when reading the stream.

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

2.2.2.5 AUX_PERF_SESSIONINFO_V2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
SessionID																Reserved															
SessionGuid																															
...																															
...																															
...																															
ConnectionID																															

SessionID (2 bytes): Session identification number.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

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

ConnectionID (4 bytes): Connection identification number.

2.2.2.6 AUX_PERF_CLIENTINFO

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
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): Speed of client computer's network adapter (kbits/s).

ClientID (2 bytes): Client-assigned identification number.

MachineNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **MachineName** field.

UserNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **UserName** field.

ClientIPSize (2 bytes): 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.

ClientIPMaskSize (2 bytes): 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.

AdapterNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **AdapterName** field.

MacAddressSize (2 bytes): Size of the network adapter 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.

ClientMode (2 bytes): Determines the mode in which the client is running. The following table specifies valid values.

Mode	Meaning
CLIENTMODE_UNKNOWN 0x00	Client is not designating a mode of operation.
CLIENTMODE_CLASSIC 0x01	Client is running in classic online mode.
CLIENTMODE_CACHED 0x02	Client is running in cached mode.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

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** 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** value.

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

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

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

2.2.2.7 AUX_PERF_SERVERINFO

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ServerID																ServerType															
ServerDNOffset																ServerNameOffset															
ServerDN (variable)																															
...																															
ServerName (variable)																															
...																															

ServerID (2 bytes): Client assigned server identification number.

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

Type	Meaning
SERVERTYPE_UNKNOWN 0x00	Unknown server type.
SERVERTYPE_PRIVATE 0x01	Client server connection servicing private mailbox data.
SERVERTYPE_PUBLIC 0x02	Client server connection servicing public folder data.
SERVERTYPE_DIRECTORY 0x03	Client server connection servicing directory data.
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.

ServerNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ServerName** field.

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

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

2.2.2.8 AUX_PERF_PROCESSINFO

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ProcessID																Reserved_1															
ProcessGuid																															
...																															
...																															
...																															
ProcessNameOffset																Reserved_2															
ProcessName (variable)																															
...																															

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

Reserved_1 (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

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

ProcessNameOffset (2 bytes): The offset from the beginning of the **AUX_HEADER** structure to the **ProcessName** field.

Reserved_2 (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

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

2.2.2.9 AUX_PERF_DEFMDB_SUCCESS

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
TimeSinceRequest																															
TimeToCompleteRequest																															
RequestID																Reserved															

TimeSinceRequest (4 bytes): Number of milliseconds since successful request occurred.

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

RequestID (2 bytes): Request identification number.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

2.2.2.10 AUX_PERF_DEFGC_SUCCESS

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ServerID																SessionID															
TimeSinceRequest																															
TimeToCompleteRequest																															
RequestOperation								Reserved																							

ServerID (2 bytes): Server identification number.

SessionID (2 bytes): Session identification number.

TimeSinceRequest (4 bytes): Number of milliseconds since successful request occurred.

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

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

Reserved (3 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

2.2.2.11 AUX_PERF_MDB_SUCCESS

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ClientID																ServerID															
SessionID																RequestID															
TimeSinceRequest																															
TimeToCompleteRequest																															

ClientID (2 bytes): Client identification number.

ServerID (2 bytes): Server identification number.

SessionID (2 bytes): Session identification number.

RequestID (2 bytes): Request identification number.

TimeSinceRequest (4 bytes): Number of milliseconds since successful request occurred.

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

2.2.2.12 AUX_PERF_MDB_SUCCESS_V2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ProcessID																ClientID															
ServerID																SessionID															
RequestID																Reserved															
TimeSinceRequest																															
TimeToCompleteRequest																															

ProcessID (2 bytes): Process identification number.

ClientID (2 bytes): Client identification number.

ServerID (2 bytes): Server identification number.

SessionID (2 bytes): Session identification number.

RequestID (2 bytes): Request identification number.

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The client MUST ignore the value of this field when reading the stream.

TimeSinceRequest (4 bytes): Number of milliseconds since successful request occurred.

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

2.2.2.13 AUX_PERF_GC_SUCCESS

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ClientID																ServerID															
SessionID																Reserved_1															
TimeSinceRequest																															
TimeToCompleteRequest																															
RequestOperation								Reserved_2																							

ClientID (2 bytes): Client identification number.

ServerID (2 bytes): Server identification number.

SessionID (2 bytes): Session identification number.

Reserved_1 (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

TimeSinceRequest (4 bytes): Number of milliseconds since successful request occurred.

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

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

Reserved_2 (3 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

2.2.2.14 AUX_PERF_GC_SUCCESS_V2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ProcessID																ClientID															
ServerID																SessionID															
TimeSinceRequest																															
TimeToCompleteRequest																															
RequestOperation																Reserved															

ProcessID (2 bytes): Process identification number.

ClientID (2 bytes): Client identification number.

ServerID (2 bytes): Server identification number.

SessionID (2 bytes): Session identification number.

TimeSinceRequest (4 bytes): Number of milliseconds since successful request occurred.

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

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

Reserved (3 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

2.2.2.15 AUX_PERF_FAILURE

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ClientID																ServerID															
SessionID																RequestID															
TimeSinceRequest																															
TimeToFailRequest																															

ResultCode	
RequestOperation	Reserved

ClientID (2 bytes): Client identification number.

ServerID (2 bytes): Server identification number.

SessionID (2 bytes): Session identification number.

RequestID (2 bytes): Request identification number.

TimeSinceRequest (4 bytes): Number of milliseconds since failure request occurred.

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

ResultCode (4 bytes): Error code return of failed request. Returned error codes are implementation specific.

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

Reserved (3 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

2.2.2.16 AUX_PERF_FAILURE_V2

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ProcessID																ClientID															
ServerID																SessionID															
RequestID																Reserved_1															
TimeSinceRequest																															
TimeToFailRequest																															
ResultCode																															
RequestOperation																Reserved_2															

ProcessID (2 bytes): Process identification number.

ClientID (2 bytes): Client identification number.

ServerID (2 bytes): Server identification number.

SessionID (2 bytes): Session identification number.

RequestID (2 bytes): Request identification number.

Reserved_1 (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

TimeSinceRequest (4 bytes): Number of milliseconds since failure request occurred.

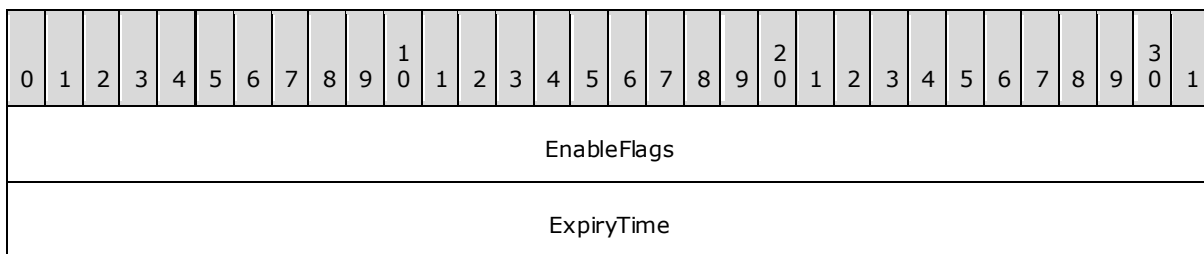
TimeToFailRequest (4 bytes): Number of milliseconds the failure request took to complete.

ResultCode (4 bytes): Error code return of failed request. Returned error codes are implementation specific.

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

Reserved_2 (3 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

2.2.2.17 AUX_CLIENT_CONTROL



EnableFlags (4 bytes): The following table describes the flags that instruct the client to either enable or disable behavior. To disable behavior, do not set the flag to the specified value.

Flag	Meaning
ENABLE_PERF_SENDSERVER 0x00000001	Client MUST start sending performance information to server.
ENABLE_PERF_SENDSMAILBOX 0x00000002	Client MUST start sending performance information as logs to a special location in the user's mailbox.
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 utilize RPC/ HTTP if configured.
ENABLE_PERF_SENDCDATA 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 SHOULD keep unspent 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.

2.2.2.18 AUX_OSVERSIONINFO<>

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
OSVersionInfoSize																															
MajorVersion																															
MinorVersion																															
BuildNumber																															
Reserved1																															
...																															
...																															
...																															
...																															
...																															
...																															
...																															
(Reserved1 cont'd for 25 rows)																															
ServicePackMajor																ServicePackMinor															
Reserved2																															

OSVersionInfoSize (4 bytes): Size of the **AUX_OSVERSIONINFO** structure.

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

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

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

Reserved1 (132 bytes): Reserved. Content MUST be ignored by client.

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

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

Reserved2 (4 bytes): Reserved. Content MUST be ignored by client.

2.2.2.19 AUX_EXORGINFO

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
OrgFlags																															

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

Flag	Meaning
PUBLIC_FOLDERS_ENABLED 0x00000001	Organization has public folders.

2.2.2.20 AUX_PERF_ACCOUNTINFO

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ClientID																Reserved															
Account																															
...																															
...																															
...																															

ClientID (2 bytes): Client assigned identification number. Maps to the **ClientID** of the **AUX_PERF_CLIENTINFO** structure (section [2.2.2.6](#)).

Reserved (2 bytes): Padding to enforce alignment of the data on a 4-byte field. The client can fill this field with any value when writing the stream. The server MUST ignore the value of this field when reading the stream.

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

3 Protocol Details

The Wire Format protocol is comprised of two RPC interfaces: **EMSMDB** and **AsyncEMSMDB**. This section describes the details of each interface.

For some functionality through the **EMSMDB** interface, the client is required to call interface method **EcDoConnectEx** first to establish a Session Context Handle (CXH). The CXH is an RPC context handle. To establish a CXH, a call to **EcDoConnectEx** MUST be successful. The following table lists all method calls that require a valid CXH.

CXH 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 (ACXH)**. The ACXH is an RPC context handle. To establish an ACXH, a call to **EcDoAsyncConnectEx** on the EMSMDB interface MUST be successful. The following table lists all method calls that require a valid ACXH context handle.

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

Some methods on this interface require CXH 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 MUST store this Session Context information and provide a CXH to the client to make subsequent interface calls by using this same Session Context information.

The server MUST keep a list of all active sessions and their associated Session Context information. Each Session Context MUST be identified by a CXH. After a Session Context has been established, a client can access messaging resources through this Session Context. The server MUST keep 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 (ACXHs), and notification callbacks.

The server MUST isolate all resources associated with one Session Context from all other Session Contexts on the server. Access to resources on one Session Context MUST NOT be allowed using a CXH of another Session Context.

When the CXH is destroyed or the client connection is lost, the Session Context and all Session Context information MUST be destroyed, all open resources MUST be closed, and all **Server objects** that are associated with the Session Context MUST be released.

3.1.2 Timers

None.

3.1.3 Initialization

The server MUST first register the different protocol sequences that will allow the server to communicate with the client. This is done by calling the RPC function **RpcServerUseProtseqEp**. For protocol sequences and details about this function, see [\[MSDN-RPC\]](#). The supported protocol sequences are specified in section [2.1](#). Note some protocol sequences use named endpoints, which are also specified in section [2.1](#).

The server MUST register the different authentication methods that are allowed on the **EMSMDB** interface. This is done by calling the RPC function **RpcServerRegisterAuthInfo**. For details about this function and the authentication methods, see [\[MSDN-RPC\]](#).

The server MUST start listening for RPC calls by calling RPC function **RpcServerListen**. For details about this function, see [\[MSDN-RPC\]](#).

The server MUST register the **EMSMDB** interface. This is done by calling the RPC function **RpcServerRegisterIfEx**. For details about this function, see [\[MSDN-RPC\]](#).

The last step is to register the **EMSMDB** interface to all the registered **binding handles** created previously in calls to **RpcServerUseProtseq** or **RpcServerUseProtseqEp**. This is done by first acquiring all the binding handle information through RPC function **RpcServerInqBindings** and then calling RPC function **RpcEpRegister** with the binding information. For details about these functions, see [\[MSDN-RPC\]](#).

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 [\[MS-RPCE\]](#) section 3.

The following table lists the methods that this interface includes. The phrase "Reserved for local use" means that the client MUST NOT send the opnum, and the server behavior is undefined [<4>](#) since it does not affect interoperability. All methods MUST NOT throw exceptions.

Method	Description
Opnum0NotUsedOnWire	Reserved for local use. 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 (CXH) to be returned from EcDoConnectEx .

Method	Description
Opnum2NotUsedOnWire	Reserved for local use. opnum: 2
Opnum3NotUsedOnWire	Reserved for local use. 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 CXH to be returned from EcDoConnectEx .
Opnum5NotUsedOnWire	Reserved for local use. opnum: 5
EcDummyRpc	This call returns a SUCCESS. A client can use it to determine whether it can communicate with the server.
Opnum7NotUsedOnWire	Reserved for local use. opnum: 7
Opnum8NotUsedOnWire	Reserved for local use. opnum: 8
Opnum9NotUsedOnWire	Reserved for local use. opnum: 9
EcDoConnectEx	Creates a CXH on the server to be used in subsequent calls to EcDoDisconnect , EcDoRpcExt2 , and EcDoAsyncConnectEx .
EcDoRpcExt2	Passes generic remote operation (ROP) commands to the server for processing within a Session Context. The method requires an active CXH to be returned from EcDoConnectEx .
Opnum12NotUsedOnWire	Reserved for local use. opnum: 12
Opnum13NotUsedOnWire	Reserved for local use. opnum: 13
EcDoAsyncConnectEx	Binds a CXH that is returned in EcDoConnectEx to a new Asynchronous Context Handle (ACXH) which can be used in calls to EcDoAsyncWaitEx in interface AsyncEMSMDB . The method requires an active Session Context handle to be returned from EcDoConnectEx .

3.1.4.1 Opnum0NotUsedOnWire

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

3.1.4.2 EcDoDisconnect (opnum 1)

The method **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. This call requires an active Session Context Handle (CXH) to be returned from method **EcDoConnectEx**.

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

pcxh: On input, contains the CXH of the Session Context that the client wants to disconnect. On output, the server MUST clear the CXH to a zero value. Setting the value to zero instructs the RPC layer of the server to destroy the RPC context handle.

Error Codes: 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 [\[MS-RPCE\]](#).

3.1.4.3 Opnum2NotUsedOnWire

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

3.1.4.4 Opnum3NotUsedOnWire

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

3.1.4.5 EcRRegisterPushNotification (opnum 4)

The method **EcRRegisterPushNotification** registers a callback address with the server for a Session Context. The callback address is used to notify the client of pending events on the server. This call requires an active Session Context Handle (CXH) to be returned from method **EcDoConnectEx**.[<5>](#)

The server MUST store the callback address and the opaque context data in the Session Context. Whenever the server wants to notify the client of pending events, it 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: On input, the client MUST pass a valid CXH that was created by calling **EcDoConnectEx**. The server uses the CXH to identify the Session Context to use for this call. On output, the server MUST return the same CXH on success.

The server can destroy the CXH by returning a zero CXH. The server might want to destroy the CXH for the following reasons:

The CXH that was passed in is invalid.

An attempt was made to access a mailbox that is in the process of being moved.

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

rgbContext: This parameter contains opaque client-generated context data that is sent back to the client at the callback address, passed in parameter *rgbCallbackAddress*, when the server wants to

notify 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: This parameter contains the size of the opaque client context data that is passed in parameter *rgbContext*. The server MUST fail this call with error code *ecTooBig* if this parameter is larger than 0x00000010.

grbitAdviseBits: This parameter MUST be 0xFFFFFFFF.

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

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

hNotification: If the call completes successfully, this output parameter will contain a handle to the notification callback on the server.

Error Codes: If the method succeeds, the return value is 0. If the method fails, the error codes listed in the following table are returned. Additional implementation-specific error codes might be returned.

Name	Value	Meaning
<i>ecInvalidParam</i>	0x80070057	A parameter passed was not valid for the call.
<i>ecNotSupported</i>	0x80040102	The callback address is not support on the server.
<i>ecTooBig</i>	0x80040305	Opaque context data is too large.

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

3.1.4.6 Opnum5NotUsedOnWire

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

3.1.4.7 EcDummyRpc (opnum 6)

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

```
long __stdcall EcDummyRpc (
```

```
    [in] handle_t hBinding
);
```

hBinding: A valid RPC binding handle.

Error Codes: The function MUST always succeed and return 0.

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

3.1.4.8 Opnum7NotUsedOnWire

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

3.1.4.9 Opnum8NotUsedOnWire

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

3.1.4.10 Opnum9NotUsedOnWire

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

3.1.4.11 EcDoConnectEx (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 **EcDoDisconnect**. This method returns a Session Context Handle (CXH) 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 **szDNPrefixed,
    [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: On success, the server MUST return a unique value to be used as a CXH. This unique value serves as the CXH for the client.

On failure, the server MUST return a zero value as the CXH.

szUserDN: User's distinguished name (DN). String containing the DN of the user who is making the **EcDoConnectEx** call in a directory service. Value: "/o=First Organization/ou=First Administrative Group/cn=recipients/cn=janedow".

ulFlags: For ordinary client calls this value MUST be 0x00000000. For Administrative privilege calls this value MUST be 0x00000001.

Value	Meaning
0x00000000	Ordinary client connection.
0x00000001	Administrator privilege requested for connection.

ulConMod: The connection modulus is a client-derived 32-bit hash value of the DN passed in field **szUserDN** and can be used by the server to decide 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: This field is reserved. A client MUST pass a value of 0x00000000.

ulCpid: The **code page** in which text data is sent if Unicode format is not requested by the client on subsequent calls using this Session Context.

ulLcidString: The local ID for everything other than sorting.

ulLcidSort: The local ID for sorting.

ulIcxrLink: This value is used to link the Session Context created by this call with an existing Session Context on the server. If no session linking is requested, and the client wants session context linking, it MUST pass the value of 0xFFFFFFFF. To link to an existing Session Context, this value is the session index value returned in field **piCxr** from a previous **EcDoConnectEx** call. In addition to passing the session index, the value in **pulTimeStamp** will be returned in the **pulTimeStamp** field from the previous **EcDoConnectEx** call. 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. <6>

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.
2. To get pending notification information for other sessions on the same client computer. For details, see [\[MS-OXCNOTIF\]](#) section 3.1.4.1.1.

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 **piCxr** from a previous **EcDoConnectEx** call, which is the session index and defined as a 16-bit value.

usFCanConvertCodePages: This field is reserved. The client MUST pass a value of 0x01.

pcmsPollsMax: The server returns 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 RPC calls for notifications (see **EcDoAsyncWaitEx**), or the client is unable to receive notifications via UDP datagrams (see **EcRRegisterPushNotifications**), the client can poll the server to determine whether any events are pending for the client. For details about notifications, see [\[MS-OXCNOTIF\]](#) section 3.2.4.1.3 and section [3.2.3.2.1](#).

pcRetry: The server returns the number of times a client retries future RPC calls using the CXH returned in this call. This is for client RPC calls that fail with RPC status code `RPC_S_SERVER_TOO_BUSY`. This is a suggested retry count for the client and is not to be enforced by the server.

pcmsRetryDelay: The server returns the number of milliseconds a client waits before retrying a failed RPC call. If any future RPC call to the server using the CXH returned in this call fails with RPC status code `RPC_S_SERVER_TOO_BUSY`, it waits the number of milliseconds specified in this output parameter before retrying the call. The number of times a client retries is returned in parameter *pcRetry*. This is a suggested delay for the client and is not to be enforced by the server.

piCxr: The server returns a session index value that is associated with the CXH returned from this call. This value in conjunction with the session creation time stamp value returned in **pulTimeStamp** will be passed to a subsequent **EcDoConnectEx** call, if the client wants to link two Session Contexts. The server MUST NOT assign two active Session Contexts the same session index value. 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](#) response command on calls to **EcDoRpcExt2** to tell the client which Session Context has pending notifications. If Session Contexts are linked, a [RopPending](#) can be returned for any linked Session Context. For details about [RopPending](#), see [\[MS-OXCROPS\]](#) section 3.1.5.1.3 and [\[MS-OXCNOTIF\]](#) section 3.1.4.1.1.

szDNPrefix: The server returns the distinguished name (DN) of the server.

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

rgwClientVersion: The client passes the client protocol version the server uses to determine what protocol functionality the client supports. For more information about how version numbers are interpreted from the wire data, see section [3.1.9](#).

rgwServerVersion: The server returns the server protocol version 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 [3.1.9](#).

rgwBestVersion: The server returns the minimum client protocol version the server supports. This information is useful if the **EcDoConnectEx** call fails with return code `ecVersionMismatch`. On success, the server returns the value passed in **rgwClientVersion** 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**, or the server can allow the client and return the value passed by the client in **rgwClientVersion**. It is up to the server implementation to set 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 [3.1.9](#).

pulTimeStamp: On input, this parameter and parameter *ulIcxrLink* are used for linking the Session Context created by this call with an existing Session Context. If the *ulIcxrLink* parameter is not 0xFFFFFFFF, the client MUST pass in the **pulTimeStamp** value returned from the server on a previous call to **EcDoConnectEx** (see the *ulIcxrLink* and *piCxr* parameters for more details). If the server supports Session Context linking, the server verifies that there is a Session Context state with the unique **identifier ulIcxrLink** and it 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** call, but simply not do linking. <7>

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 do Session Context linking.

rgbAuxIn: This parameter contains an 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 that includes the **RPC_HEADER_EXT** header is contained in parameter *cbAuxIn*.

See section [3.1.7](#) for details about how to access the embedded auxiliary payload buffer. See section [3.1.8](#) for details about how to interpret the auxiliary payload data.

cbAuxIn: On input, this parameter contains the length of the auxiliary payload buffer passed in the *rgbAuxIn* parameter. The server MUST fail with the RPC status code **RPC_X_BAD_STUB_DATA** (0x000006F7), if this value on input is larger than 0x00001008 bytes in size. For more information on returning RPC Status Codes, see [\[C706\]](#). <8>

rgbAuxOut: On output, the server can return auxiliary payload data to the client. The server MUST include an **RPC_HEADER_EXT** header before the auxiliary payload data.

See section [3.1.7](#) for details about how to access the embedded auxiliary payload buffer. See section [3.1.8](#) for details about how to interpret the auxiliary payload data.

pcbAuxOut: On input, this parameter contains the maximum length of the *rgbAuxOut* buffer. The server MUST fail with the RPC status code **RPC_X_BAD_STUB_DATA** (0x000006F7) if this value on input is larger than 0x00001008. For more information on returning RPC Status Codes, see [\[C706\]](#).

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

Error Codes: 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.

Name	Value	Meaning
ecRpcAuthentication	0x000004B6	The <i>szUserDN</i> parameter does not reference a user or references a guest user or a built-in user.
ecNotEncrypted	0x00000970	The server is configured to require encryption and the binding handle, <i>hBinding</i> , authentication is not set with RPC_C_AUTHN_LEVEL_PKT_PRIVACY . For more information about setting the authentication and authorization, see 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

Name	Value	Meaning
		ecNotEncrypted being returned by the server. See section 3.1.9 for details about which client versions do not support receiving error code ecNotEncrypted. 2. The client version has been blocked by the administrator.
ecLoginFailure	0x80040111	1. The user does not have any access to a private mailbox or public folder messaging data. 2. There are no private mailboxes or public folders on the server. 3. The server is exiting or is about to exit.
ecUnknownUser	0x000003EB	The server does not recognize the <i>szUserDN</i> 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.
ecVersionMismatch	0x80040110	The client and server versions are not compatible. The client protocol version is older than that required by the server.
ecCachedModeRequired	0x000004E1	The server requires the client to be running in cache mode. See section 3.1.9 for details about which client versions understand this error code.
ecRpcHttpDisallowed	0x000004E0	The server requires the client to not be connected via RPC/HTTP. See section 3.1.9 for details about which client versions understand this error code.
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, IMAP, and so on). See section 3.1.9 for details about which client versions understand this error code.

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

3.1.4.12 EcDoRpcExt2 (opnum 11)

The method **EcDoRpcExt2** passes generic remote operation (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 (CXH) returned from method **EcDoConnectEx**.

```

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_ULONGLONG *pcbAuxOut,
    [out] unsigned long *pulTransTime
);

```

pcxh: On input, the client MUST pass a valid Session Context Handle (CXH) that was created by calling **EcDoConnectEx**. The server uses the CXH to identify the Session Context to use for this call. On output, the server MUST return the same CXH on success.

The server can destroy the CXH by returning a zero CXH. The server might destroy the Session CXH for the following reasons:

1. It determines that the **ROP request** payload in the *rgbIn* buffer is malformed or length parameters are invalid.
2. The CXH passed in is invalid.
3. It is trying to access a mailbox that is in the process of being moved.
4. If an administrator blocks a client that has an active connection.

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

Name	Value	Meaning
NoCompression	0x00000001	The server MUST NOT compress ROP response payload (<i>rgbOut</i>) or auxiliary payload (<i>rgbAuxOut</i>). If flag is absent, server MUST compress.
NoXorMagic	0x00000002	The server MUST NOT obfuscate the ROP response payload (<i>rgbOut</i>) or auxiliary payload (<i>rgbAuxOut</i>). If flag is absent, server SHOULD obfuscate.
Chain	0x00000004	The server allows chaining of ROP response payloads.

See section [3.1.7.1.2](#) for details about how to use these flags.

On output, the server MUST return 0x00000000. The meaning of the output **flags** are reserved for future use.

rgbIn: This buffer contains the ROP request payload. The ROP request payload is prefixed with an **RPC_HEADER_EXT header**. Information stored in this header determines how to interpret the data following the header. See [3.1.7](#) for details about how to access the embedded ROP request payload. The length of the ROP request payload including the **RPC_HEADER_EXT** header is contained in parameter *cbIn*.

For more information about **ROP buffers**, see [\[MS-OXCROPS\]](#).

cbIn: On input, this parameter contains the length of the ROP request payload passed in the *rgbIn* parameter. The server MUST fail with error code *ecRpcFormat* if the request buffer is larger than 0x00008007 bytes in size or is smaller than the size of the **RPC_HEADER_EXT** (0x00000008 bytes). <9>The ROP request payload includes the size of the ROPs plus the size of **RPC_HEADER_EXT**. For more details, see [\[MS-OXCROPS\]](#).

rgbOut: On success, this buffer contains the ROP response payload. 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 [3.1.7](#). The size of the ROP response payload plus the `RPC_HEADER_EXT` header is returned in `pcbOut`.

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

pcbOut: On input, this parameter contains the maximum size of the `rgbOut` buffer. The server MUST fail with error code `ecRpcFormat` if the value in `pcbOut` on input is less than `0x00008007`.
<10>The server MUST fail with the RPC status code of `RPC_X_BAD_STUB_DATA` (`0x000006F7`) if the value in `pcbOut` on input is larger than `0x00040000`. For more information on returning RPC Status Codes, see [\[C706\]](#).

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 as there is no ROP response payload. The client ignores any data returned on failure.

rgbAuxIn: This parameter contains an 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 parameter `cbAuxIn`.

See section [3.1.7](#) for details about how to access the embedded auxiliary payload buffer. See section [3.1.8](#) for details about how to interpret the auxiliary payload data.

cbAuxIn: On input, this parameter contains the length of the auxiliary payload buffer passed in the `rgbAuxIn` parameter. The server MUST fail with the RPC status code `RPC_X_BAD_STUB_DATA` (`0x000006F7`) if the request buffer is larger than `0x00001008` bytes in size. For more information on returning RPC status codes, see [\[C706\]](#).
<11>

rgbAuxOut: On output, the server can return auxiliary payload data to the client. The server MUST include a `RPC_HEADER_EXT` header before the auxiliary payload data.

See section [3.1.7](#) for details about how to access the embedded auxiliary payload buffer. See section [3.1.8](#) for details about how to interpret the auxiliary payload data.

pcbAuxOut: On input, this parameter contains the maximum length of the `rgbAuxOut` buffer. The server MUST fail with the RPC status code `RPC_X_BAD_STUB_DATA` (`0x000006F7`) if this value on input is larger than `0x00001008`. For more information on returning RPC status codes, see [\[C706\]](#).

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

pulTransTime: 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 function call and, using this output parameter, can determine whether time was spent transmitting the request/response on the network or processing time on the server.

Error Codes: If the method succeeds, the return value is 0. If the method fails, the error codes listed in the following table are returned. Additional implementation-specific error codes could be returned.

Name	Value	Meaning
<code>ecRpcFormat</code>	<code>0x000004B6</code>	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 [\[MS-RPCE\]](#).

3.1.4.13 Opnum12NotUsedOnWire

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

3.1.4.14 Opnum13NotUsedOnWire

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

3.1.4.15 EcDoAsyncConnectEx (opnum 14)

The method **EcDoAsyncConnectEx** binds a Session Context Handle (CXH) returned from method **EcDoConnectEx** to a new Asynchronous Context Handle (ACXH) that can be used in calls to **EcDoAsyncWaitEx** in interface **AsyncEMSMDB**. This call requires an active CXH to be returned from method **EcDoConnectEx**.

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 : Client MUST pass a valid CXH that was created by calling **EcDoConnectEx**. The server uses the CXH to identify the **Session Context** to use for this call.

pacxh: On success, the server returns an ACXH that is associated with the Session Context passed in parameter CXH. On a failure the returned value is a Null. This ACXH can be used to make a call to **EcDoAsyncWaitEx** on interface **AsyncEMSMDB**.

Error Codes: If the method succeeds, the return value is 0. If the method fails, the error codes listed in the following table are returned. Additional implementation-specific error codes could be returned.

Name	Value	Meaning
ecRejected	0x000007EE	Server has asynchronous RPC notifications disabled. Client either polls for notifications or calls EcRRegisterPushNotifications .

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

3.1.5 Timer Events

None.

3.1.6 Other Local Events

None.

3.1.7 Extended Buffer Handling

Interface methods **EcDoConnectEx** and **EcDoRpcExt2** contain request and response buffers that use an extended buffer mechanism where the payload is preceded by a header. The header contains flags that determine whether or not the payload has been compressed, obfuscated, or another extended buffer and payload exists after the current payload. A single payload MUST NOT exceed 32 KB in size.

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

The following sections detail the extended buffer format, compression algorithm, obfuscation algorithm, and extended buffer packing.

3.1.7.1 Extended Buffer Format

See section [2.2.2.1](#) for details about the structure and individual fields.

The client or server can choose not to compress the payload if the payload is small. The client or server can choose to not obfuscate the payload if the payload has already been compressed. The client or server can choose to not obfuscate the payload if the client is connected using RPC layer encryption.

The extended buffer is used in both the **EcDoConnectEx** and **EcDoRpcExt2** for a variety of different fields. The information in the following sections describes how the extended buffer is used for the different fields on each method.

3.1.7.1.1 EcDoConnectEx

3.1.7.1.1.1 rgbAuxIn

The input buffer *rgbAuxIn* has the following format:

RPC_HEADER_EXT	Payload
----------------	----------------

The header MUST contain the Last flag in the flags field.

If the Compressed flag is present in the flags field, the content of the payload MUST be compressed by the client and MUST be uncompressed by the server before it can be interpreted. See section [3.1.7.2](#) for details about how to compress and uncompress payload data.

If the XorMagic flag is present in the flags field, the content of the payload MUST be obfuscated by the client and MUST be reverted by the server before it can be interpreted. See section [3.1.7.3](#) for details about how to obfuscate and revert obfuscated payload data.

The payload is auxiliary information that can be passed from the client to the server. See section [3.1.8](#) for details about how to interpret this data.

3.1.7.1.1.2 rgbAuxOut

The output buffer *rgbAuxOut* has the following format:

RPC_HEADER_EXT	Payload
----------------	----------------

The header MUST contain the Last flag in the flags field.

If the Compressed flag is present in the flags field, the content of the payload MUST be compressed by the server and MUST be uncompressed by the client before it can be interpreted. See section [3.1.7.2](#) details about how to compress and uncompress payload data.

If the XorMagic flag is present in the flags field, the content of the payload MUST be obfuscated by the server and MUST be reverted by the client before it can be interpreted. See section [3.1.7.3](#) for details about how to obfuscate and revert obfuscated payload data.

The payload is auxiliary information that can be passed from the server to the client. See section [3.1.8](#) for details about how to interpret this data.

3.1.7.1.2 EcDoRpcExt2

The flags passed to the server in field **pulFlags** by the client request that the server compress or obfuscate the response data returned in field **rgbOut** and **rgbAuxOut**. If the client requests no compression or no obfuscation through the flags NoCompression or NoXorMagic, the server MUST honor the client request. If the client requests compression or obfuscation through the absence of either flags NoCompression or NoXorMagic, the server MAY honor the client request. The client MUST NOT assume a response will be compressed or obfuscated if requested and has the ability to handle data which is not compressed or not obfuscated.

3.1.7.1.2.1 rgbIn

The input buffer *rgbIn* has the following format:

RPC_HEADER_EXT	Payload
----------------	----------------

The header MUST contain the Last flag in the flags field.

If the Compressed flag is present in the flags field, the content of the payload MUST be compressed by the client and MUST be uncompressed by the server before it can be interpreted. See section [3.1.7.2](#) for details about how to compress and uncompress payload data.

If the XorMagic flag is present in the flags field, the content of the payload MUST be obfuscated by the client and MUST be reverted by the server before it can be interpreted. See section [3.1.7.3](#) for details about how to obfuscate and revert obfuscated payload data.

The payload is remote operation (ROP) request information that can be passed from the client to the server. See [\[MS-OXCROPS\]](#) for details about how to interpret this data.

3.1.7.1.2.2 rgbOut

The output buffer *rgbOut* has the following format:

RPC_HEADER_EXT	Payload	RPC_HEADER_EXT	Payload	...	RPC_HEADER_EXT	Payload
----------------	----------------	----------------	----------------	-----	----------------	----------------

There might be multiple extended buffers contained in the single output buffer. They will each have an RPC_HEADER_EXT header followed by a payload.

All headers except for the last MUST NOT contain the Last flag in the flags field. The last header MUST contain the Last flag in the flags field.

If the Compressed flag is present in the flags field, the content of the payload following the header MUST be compressed by the server and MUST be uncompressed by the client before it can be interpreted. See section [3.1.7.2](#) for details about how to compress and uncompress payload data.

If the XorMagic flag is present in the flags field, the content of the payload following the header MUST be obfuscated by the server and MUST be reverted by the client before it can be interpreted. See section [3.1.7.3](#) for details about how to obfuscate and revert obfuscated payload data.

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

Each payload contains remote operation (ROP) response information that is returned from the server to the client. See [\[MS-OXCROPS\]](#) for details about how to interpret this data.

3.1.7.1.2.3 rgbAuxIn

The input buffer *rgbAuxIn* has the following format:

RPC_HEADER_EXT	Payload
----------------	----------------

The header MUST contain the Last flag in the flags field.

If the Compressed flag is present in the flags field, the content of the payload MUST be compressed by the client and MUST be uncompressed by the server before it can be interpreted. See section [3.1.7.2](#) for details about how to compress and uncompress payload data.

If the XorMagic flag is present in the flags field, the content of the payload MUST be obfuscated by the client and MUST be reverted by the server before it can be interpreted. See section [3.1.7.3](#) for details about how to obfuscate and revert obfuscated payload data.

The payload is auxiliary information that can be passed from the client to the server. See section [3.1.8](#) for details about how to interpret this data.

3.1.7.1.2.4 rgbAuxOut

The output buffer *rgbAuxOut* has the following format:

RPC_HEADER_EXT	Payload
----------------	----------------

The header MUST contain the Last flag in the flags field.

If the Compressed flag is present in the flags field, the content of the payload MUST be compressed by the server and MUST be uncompressed by the client before it can be interpreted. See section [3.1.7.2](#) for more details about how to compress and uncompress payload data.

If the XorMagic flag is present in the flags field, the content of the payload MUST be obfuscated by the server and MUST be reverted by the client before it can be interpreted. See section [3.1.7.3](#) for more details about how to obfuscate and revert obfuscated payload data.

The payload is auxiliary information that can be passed from the server to the client. See section [3.1.8](#) for more details about how to interpret this data.

3.1.7.2 Compression Algorithm

Based on flags that are passed in **RPC_HEADER_EXT** header of the extended buffer, the payload is compressed or decompressed by the server and client by using the LZ77 compression algorithm and the DIRECT2 encoding algorithm.

This section describes the compression algorithm LZ77 and the basic encoding algorithm DIRECT2 that are used by the Wire Format protocol.

3.1.7.2.1 LZ77 Compression Algorithm

The 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.

3.1.7.2.1.1 Compression Algorithm Terminology

The following terms are associated with the compression algorithm.

input stream : The sequence of bytes to be compressed.

byte: The basic data element in the input stream.

coding position: The position of the byte in the input stream that is currently being coded (the beginning of the **lookahead buffer**).

lookahead buffer: The byte sequence from the coding position to the end of 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.

pointer: Information about the beginning of the **match** in the window (referred to as "B" in the example later in this section) and also specifies its length (referred to as "L" in the example later in this section).

match: The string that is used to find a match of the byte sequence between the **lookahead buffer** and the **window**.

3.1.7.2.1.2 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.

3.1.7.2.1.3 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**.

3.1.7.2.1.4 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

Pos	1	2	3	4	5	6	7	8	9
Byte	A	A	B	C	B	B	A	B	C

The following table shows the output from the compression process. The table 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 two 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 three or more bytes in common.

3.1.7.2.2 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. Bitmasks are inserted periodically in the byte stream to provide this information to the decoder.

This section describes the bitmasks that enable the decoder to distinguish data from metadata. It also describes the process of encoding the metadata.

3.1.7.2.2.1 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.

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

3.1.7.2.2.2 Encoding Metadata

In the output stream, actual data **bytes** are stored unchanged. Bitmasks are stored periodically to indicate whether the next byte or bytes are data or metadata. 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 needed 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 needed to encode the length.

3.1.7.2.2.3 Metadata Offset

This protocol assumes the metadata is two **bytes** in length, where the high-order 13 bits are a first complement of the offset, and the low-order three 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.

3.1.7.2.2.4 Match Length

Unlike the metadata offset, the **match** length is extensible. If the length is less than 10 **bytes**, it is encoded in the three low-order bits of the 2-byte metadata. Although three bits seems to allow for a maximum length of six (the value b'111' is reserved), because the minimum match is three bytes, these three bits actually allow for the expression of lengths from three to nine. 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 three and nine, we use the three bits that are "in-line" in the 2-byte metadata.

If the length of the match is greater than nine bytes, an initial bit pattern of b'111' is put in the three 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 nine, 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 nine. Therefore, the first time that the decoder encounters a length that is greater than nine, 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 nine. The third time that a length that is greater than nine 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 nine, 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 three bits in the original two 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 three bits in the original two 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 278 (a length of 10 in the three bits in the original two 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 two bytes.

The final two 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 two bytes, and is not added to the previous length bits. The value in the last two bytes, a 16-bit integer, is three less than the metadata length. These last two 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 three.

Length representation in metadata

Match Length	Length Bits in the Metadata
24	b'111' (three bits in the original two bytes of metadata) + b'1110' (in the high or lower-order nibble, as appropriate, of the shared byte)
25	b'111' (three bits in the original two bytes of metadata) + b'1111' (in the high or lower-order nibble, as appropriate, of the shared byte) + b'0000000' (in the next byte)
26	b'111' (three bits in the original two bytes of metadata) + b'1111' (in the high or lower-order nibble, as appropriate, of the shared byte) + b'00000001' (in the next byte)
279	b'111' (three bits in the original two bytes of metadata) + b'1111' (in the high or lower-order nibble, as appropriate, of the shared byte) + b'11111110' (in the next byte)
280	b'111' (three bits in the original two bytes of metadata) b'1111' (in the high or lower-order nibble, as appropriate, of the shared byte) b'11111111' (in the next byte) 0x0115 (in the next two bytes). These two bytes represent a length of 277 + 3 (minimum match length). Note All the length is included in the final two bytes and is not additive, as were the previous length calculations for lengths that are smaller than 280 bytes.
281	b'111' (three bits in the original two bytes of metadata) b'1111' (in the high or lower-order nibble, as appropriate, of the shared byte) b'11111111' (in the next byte) 0x0116 (in the next two bytes). This is 278 + 3 (minimum match length). Note All the length is included in the final two 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 three bits in the

original two bytes of metadata, four bits in the nibble in the 'shared' byte, eight bits in the next byte, and 16 bits in the final two bytes of metadata.

3.1.7.3 Obfuscation Algorithm

Obfuscation is used 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 wants 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 XOR applied with the value 0xA5.

3.1.7.4 Extended Buffer Packing

As mentioned in section [3.1.7.1.2.2](#), the *rgbOut* field of method **EcDoRpcExt2** can contain more than one extended buffer, each with an **RPC_HEADER_EXT** header. This concept is called "packing". The server has the ability to "pack" additional response data into the *rgbOut* field based on whether the client has requested this functionality through passing flag Chain in the *pulFlags* field and whether the remote operation (ROP) in the *rgbIn* request buffer on the **EcDoRpcExt2** method support "packing". The ROP commands that support "packing" are [RopQueryRows](#), [RopReadStream](#), and [RopFastTransferSourceGetBuffer](#). See [\[MS-OXCROPS\]](#) section 2.2.11.3 for details about these ROP commands.

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](#), [RopReadStream](#), and [RopFastTransferSourceGetBuffer](#) from the *rgbIn* request buffer and it was the last ROP command in the request buffer and the client has requested "packing" through the Chain flag and there is residual room in the *rgbOut* response buffer, the server can add additional data to the *rgbOut* response buffer 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](#), [RopReadStream](#), or [RopFastTransferSourceGetBuffer](#). The additional response data is also limited to 32 KB in size. The additional **ROP response** is placed into the *rgbOut* buffer following the previous header and payload with its own **RPC_HEADER_EXT** header. The server can then compress and/or obfuscate this payload if the client requests and set the appropriate flags in the header indicating how the payload has been altered. If there is still more residual room in the *rgbOut* buffer, the server can continue to produce more response data until there is not enough room in the *rgbOut* buffer to hold another response.

The server MUST stop adding additional "packed" buffers to the *rgbOut* response buffer if the residual size of the *rgbOut* response buffer is less than 8 KB for [RopReadStream](#) and [RopFastTransferSourceGetBuffer](#) and 32 KB for [RopQueryRows](#). The server MUST NOT place more than 96 individual payloads into a single *rgbOut* response buffer.

When it adds additional response data, the server MUST alter the request to reflect what has already been done. For example, if the client requests to read 1,000 rows in [RopQueryRows](#) and the first payload contains 100 rows, the additional response data MUST be processed "as if" the client only request 900 rows. The server MUST NOT return more data to the client than the client originally requested.

For [RopQueryRows](#), the server MUST adjust the row count when adding additional response data. For [RopReadStream](#), the server MUST adjust the number of bytes to read when adding additional response data. There is no specific limit for [RopFastTransferSourceGetBuffer](#), but the server MUST stop if no more data is indicated for the fast transfer stream. For [RopFastTransferSourceGetBuffer](#), the client requests that the server return all the server data. See [\[MS-OXCROPS\]](#) section 2.2.11.3 for details about how to properly format [RopFastTransferSourceGetBuffer](#) in this way.

3.1.8 Auxiliary Buffer

Methods **EcDoConnectEx** and **EcDoRpcExt2** allow for additional data to travel between the client and server. This additional data is transferred in the auxiliary buffers of both methods. The *rgbAuxIn* is for auxiliary data being sent from the client to the server and *rgbAuxOut* is for auxiliary data being sent from the server to the client.

Unlike the ROP request and response payloads *rgbIn* and *rgbOut*, 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 sent from the server to the client is also informational data that the client might use to alter its behavior against the server.

The data being transferred in the auxiliary buffers is divided into two different categories. The first 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. The server is free to analyze this information and provide feedback to help diagnose any potential networking or communications issues with the client/server messaging network infrastructure.

The second category of auxiliary information is server-to-client oriented and enables the server to tell the client about topology characteristics of the messaging system. The client can use this information to change how it interacts with the server.

All information in the auxiliary buffer MUST be added with an **AUX_HEADER** preceding the actual auxiliary information. See section [2.2.2.2](#) for details about the **AUX_HEADER** and how it is formatted. Within the **AUX_HEADER** header the fields **Version** and **Type** combined determine which auxiliary block follows the header. Section [2.2.2.2](#) provides details about how to format the **AUX_HEADER** header to indicate which auxiliary block follows.

If the client or server receives an auxiliary **AUX_HEADER** block with a version and type it does not identify, it MUST skip over the entire block. The **AUX_HEADER** contains the length of the **AUX_HEADER** plus the following auxiliary block in the field **Size**, and so skipping the information can be done. The client or server does not throw an error if there is an auxiliary block that it does not identify. This will allow for future expansion to the auxiliary blocks without affecting legacy clients or servers.

3.1.8.1 Client Performance Monitoring

The following are sent from the client to the server in the *rgbAuxIn* auxiliary buffer on method **EcDoConnectEx**. Each of these auxiliary blocks MUST be preceded by a properly formatted **AUX_HEADER** header.

Sent by client to server in EcDoConnectEx

Block	Description
AUX_PERF_CLIENTINFO (see section 2.2.2.6)	Sent to the server as diagnostic information about the client for more robust reporting of networking issues. <12> The client MUST assign a unique ClientID parameter for each AUX_PERF_CLIENTINFO block sent to the server. The ClientID is also used in other performance blocks to identify which client to associate the performance data with.
AUX_PERF_PROCESSINFO (see section 2.2.2.8)	Sent to the server as diagnostic information about the client process for more robust reporting of networking issues. The client MUST assign

Block	Description
	a unique ProcessID for each AUX_PERF_PROCESSINFO block sent to the server. The ProcessID is also used in other performance blocks to identify which client process to associate the performance data with.
AUX_PERF_SESSIONINFO (see section 2.2.2.4)	Sent to the server as diagnostic information about the client session for more robust reporting of networking issues. The client MUST assign a unique SessionID for each AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 block sent to the server. The SessionID is also used in other performance blocks to identify which client session to associate the performance data with. If writing a client, it is recommended that AUX_PERF_SESSIONINFO_V2 be used instead. A server still supports this older session information auxiliary block. This block can also be passed in the EcDoRpcExt2 auxiliary input buffer.
AUX_PERF_SESSIONINFO_V2 (see section 2.2.2.5)	Sent to the server as diagnostic information about the client session for more robust reporting of networking issues. The client MUST assign a unique SessionID for each AUX_PERF_SESSIONINFO_V2/ AUX_PERF_SESSIONINFO block sent to the server. The SessionID 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 auxiliary input buffer.

The following are sent from the client to the server in the *rgbAuxIn* auxiliary buffer on method **EcDoRpcExt2**. Each of these auxiliary blocks MUST be preceded by a properly formatted **AUX_HEADER** header (see section [2.2.2.2](#)).

Sent by client to server in EcDoRpcExt2

Block	Description
AUX_PERF_SESSIONINFO (see section 2.2.2.4)	Sent to the server as diagnostic information about the client session for more robust reporting of networking issues. The client MUST assign a unique SessionID for each AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 block sent to the server. The SessionID is also used in other performance blocks to identify which client session to associate the performance data with. If writing a client, it is recommended that AUX_PERF_SESSIONINFO_V2 be used instead. A server still supports this older session information auxiliary block. This block can also be passed in the EcDoConnectEx auxiliary input buffer.
AUX_PERF_SESSIONINFO_V2 (see section 2.2.2.5)	Sent to the server as diagnostic information about the client session for more robust reporting of networking issues. The client MUST assign a unique SessionID for each AUX_PERF_SESSIONINFO_V2/ AUX_PERF_SESSIONINFO block sent to the server. The SessionID 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 auxiliary input buffer.

Block	Description
AUX_PERF_SERVERINFO (see section 2.2.2.7)	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 ServerID for each AUX_PERF_SERVERINFO block sent to the server. The ServerID is also used in other performance blocks to identify which server a client is communicating with to associate the performance data.
AUX_PERF_REQUESTID (see section 2.2.2.3)	Sent to the server as diagnostic information about a particular request for more robust reporting of networking issues. The client MUST assign a unique RequestID for each AUX_PERF_REQUESTID block sent to the server. The RequestID is also used in other performance blocks to identify which request to associate the performance data with. This block requires an AUX_PERF_SESSIONINFO or AUX_PERF_SESSIONINFO_V2 to have been previously sent to the server for the SessionID field within this block.
AUX_PERF_DEFMDB_SUCCESS (see section 2.2.2.9)	Sent to the server as diagnostic information to report a previously successful RPC call to the messaging server. This block requires an AUX_PERF_REQUESTID to have been previously sent to the server for the RequestID field within this block.
AUX_PERF_DEFGC_SUCCESS (see section 2.2.2.10)	Sent to the server as diagnostic information to report a previously successful call to the Active Directory directory service. This block requires an AUX_PERF_SERVERINFO and AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 to have been previously sent to the server for the ServerID and SessionID fields within this block.
AUX_PERF_MDB_SUCCESS (see section 2.2.2.11)	Sent to the server as diagnostic information to report a previously successful RPC call to the messaging server. This block requires an AUX_PERF_REQUESTID , AUX_PERF_CLIENTINFO , AUX_PERF_SERVERINFO , and AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 to have been previously sent to the server for the RequestID , ClientID , ServerID , and SessionID fields within this block. If writing a client, it is recommended that AUX_PERF_MDB_SUCCESS_V2 be used instead. A server still supports this older session information auxiliary block.
AUX_PERF_MDB_SUCCESS_V2 (see section 2.2.2.12)	Sent to the server as diagnostic information to report a previously successful RPC call to the messaging server. This block requires an AUX_PERF_REQUESTID , AUX_PERF_PROCESSINFO , AUX_PERF_CLIENTINFO , AUX_PERF_SERVERINFO , and AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 to have been previously sent to the server for the RequestID , ProcessID , ClientID , ServerID , and SessionID fields within this block.
AUX_PERF_GC_SUCCESS (see section 2.2.2.13)	Sent to the server as diagnostic information to report a previously successful call to the directory service. This block requires an AUX_PERF_CLIENTINFO , AUX_PERF_SERVERINFO , and AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 to have been previously sent to the

Block	Description
	server for the ClientID , ServerID , and SessionID fields within this block. If writing a client, it is recommended that AUX_PERF_GC_SUCCESS_V2 be used instead. A server still supports this older session information auxiliary block.
AUX_PERF_GC_SUCCESS_V2 (see section 2.2.2.14)	Sent to the server as diagnostic information to report a previously successful call to the directory service. This block requires an AUX_PERF_PROCESSINFO , AUX_PERF_CLIENTINFO , AUX_PERF_SERVERINFO , and AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 to have been previously sent to the server for the ProcessID , ClientID , ServerID , and SessionID fields within this block.
AUX_PERF_FAILURE (see section 2.2.2.15)	Sent to the server as diagnostic information to report a previously FAILED call to the messaging server or the directory service. This block requires an AUX_PERF_REQUESTID , AUX_PERF_CLIENTINFO , AUX_PERF_SERVERINFO , and AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 to have been previously sent to the server for the RequestID , ClientID , ServerID , and SessionID fields within this block. If writing a client, it is recommended that AUX_PERF_FAILURE_V2 be used instead. A server still supports this older session information auxiliary block.
AUX_PERF_FAILURE_V2 (see section 2.2.2.16)	Sent to the server as diagnostic information to report a previously FAILED call to the messaging server or the directory service. This block requires an AUX_PERF_REQUESTID , AUX_PERF_PROCESSINFO , AUX_PERF_CLIENTINFO , AUX_PERF_SERVERINFO , and AUX_PERF_SESSIONINFO/ AUX_PERF_SESSIONINFO_V2 to have been previously sent to the server for the RequestID , ProcessID , ClientID , ServerID , and SessionID fields within this block.

3.1.8.2 Server Topology Information

The following are sent from the server to the client in the *rgbAuxOut* auxiliary buffer on method **EcDoConnectEx**. Each of these auxiliary blocks MUST be preceded by a properly formatted **AUX_HEADER** header (see section [2.2.2.2](#)).

Sent by server to client in EcDoConnectEx

Block	Description
AUX_CLIENT_CONTROL (see section 2.2.2.17)	Sent to the client to request a change in client behavior. This is a means for the server to dynamically change client behavior. See section 2.2.2.17 for details about what client behavior the server can adjust. The client alters its behavior based on this request.
AUX_OSVERSIONINFO (see section 2.2.2.18)	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 servers operating system version and operating system service pack information. <13>

Block	Description
AUX_EXORGINFO (see section 2.2.2.19)	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 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.

The following are sent from the server to the client in the `rgbAuxOut` auxiliary buffer on method **EcDoRpcExt2**. Each of these auxiliary blocks **MUST** be preceded by a properly formatted **AUX_HEADER** header (see section [2.2.2.2](#)).

Sent by server to client in EcDoRpcExt2

Block	Description
AUX_CLIENT_CONTROL (see section 2.2.2.17)	Sent to the client to request a change in client behavior. This is a means for the server to dynamically change client behavior. See section 2.2.2.17 for details about what client behavior the server can adjust. The client alters its behavior based on this request.

3.1.9 Version Checking

In the method **EcDoConnectEx**, the client passes the client version to the server. In response, the server returns its version to the client. The server version information indicates to the client what functionality is supported on the server. The client version information indicates to the server what functionality the client supports.

Sometimes the functionality represents a change in the protocol wire format. This section describes the following:

- How version numbers are compared.
- Specific server versions and their associated functionality.
- Specific client versions and their associated functionality.

3.1.9.1 Version Number Comparison

On the wire, client and server version numbers are passed as three WORD values. See section [3.1.4.11](#) for details about the **EcDoConnectEx** method. In this method, the fields **rgwClientVersion**, **rgwServerVersion**, and **rgwBestVersion** are all passed as three WORD values. However, manipulation **MUST** be performed before the numbers can be compared.

Because versions that are passed on the wire were historically represented as only three numbers, the version number was expressed as "XX.XXXX.XXX." The first number represented the product major version. The second number was the build major number. The third number was the build minor number. However, this representation prevented the inclusion of a required fourth number, the product minor number, which is used when shipping service packs.

Microsoft changed the versioning to be represented as "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.

However, the version size on the wire did not change: it is still represented as three WORD values. A scheme was devised that converts from the three WORD on-the-wire-format of the version into a four-number version. This is referred to as version number normalization.

All versions are converted into four-number versions before any version checks are performed. The following pseudo-code example describes a function that converts the three WORD value wire version format into a four-number format that can then be used for version comparisons.

```
// 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 is divided into two BYTE values, one being the product major version and the other being the product minor version. On the wire, the client and server need to know 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.

3.1.9.2 Server Versions

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

Server version	Description
6.0.6755.0	The server supports passing the sentinel value 0xBABE in the BufferSize field of a RopFastTransferSourceGetBuffer request. For details, see [MS-OXCROPS] section 2.2.11.3.
8.0.295.0	The server supports passing the sentinel value 0xBABE in the ByteCount field of a RopReadStream request. For details, see [MS-OXCROPS] section 2.2.8.2.
8.0.324.0	The server supports the flag CLI_WITH_PER_MDB_FIX in the OpenFlags field of a RopLogon request. For details, see [MS-OXCROPS] section 2.2.2.1 and [MS-OXCSTOR] section 3.2.5.1.
8.0.358.0	The server supports the EcDoAsyncConnectEx and EcDoAsyncWaitEx RPC function calls.

A server implementation needs to determine 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 one server version level, the server MUST support all functionality from previous server version levels.

3.1.9.3 Client Versions

The following table shows client versions that are passed to the server on the **EcDoConnectEx** method call, 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 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 RopReadRecipients , RopOpenMessage , and RopOpenEmbeddedMessage . For details, see [MS-OXCROPS] section 2.2.5.6, section 2.2.5.1 , and section 2.2.5.16 .
11.00.0000.4920	The client supports receiving ecServerBusy in the ReturnValue field of the RopFastTransferSourceGetBuffer response. The BackoffTime field will be present when the ReturnValue is ecServerBusy. If ReturnValue is not ecServerBusy, the BackoffTime field is not present. For details, see [MS-OXCROPS] section 2.2.11.3 and [MS-OXCFXICS] section 2.2.3.1.1.5.
12.00.0000.000	The client supports receiving the errors ecCachedModeRequired, ecRpcHttpDisallowed, and ecProtocolDisabled on the EcDoConnectEx call; otherwise, the client will get back ecClientVerDisallowed instead.
12.00.3118.000	The client supports receiving an AUX_EXORGINFO block in the <i>rgbAuxOut</i> buffer on the EcDoConnectEx call.
12.00.3619.000	The client supports receiving the errors ecNotEncrypted on the EcDoConnectEx 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 PidTagTargetEntryId . See [MS-OXCFXICS] for more details.
12.00.4207.000	The client supports "packing" of RopReadStream in the ROP response buffer of the EcDoRpcExt2 RPC call. The RopReadStream MUST be the last ROP in the request buffer on the EcDoRpcExt2 call. See section 3.1.7.4 for details about extended buffer "packing".
12.00.4228.0000	The client supports receiving RopBackoff in the ROP response buffer of the EcDoRpcExt2 call. For details, see [MS-OXCROPS] section 3.1.5.1.1.

A client implementation needs to determine 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.

3.2 EMSMDB Client Details

3.2.1 Abstract Data Model

For some functionality on the **EMSMDB** interface, it is required that the client store a Session Context Handle (CXH) and use it on subsequent interface calls that require a CXH context handle.

3.2.2 Timers

No protocol timers are required beyond the internal timers that are used in RPC to implement resiliency to network outages. For details, see [\[MS-RPCE\]](#).

3.2.3 Initialization

The client creates an RPC connection to the remote server using the details described 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 section [3](#) of [\[MS-RPCE\]](#).

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](#).

3.2.4.1 Sending EcDoConnectEx

When issuing the interface call **EcDoConnectEx**, some parameters need additional client-side consideration beyond what is stated in section [3.1.4.11](#). The following is a list of parameters for which the client has specific handling:

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. See section [2.1](#) for details about how to create a binding handle.

pcxh: On success, this field will contain the Session Context Handle (CXH). On failure, this value is Null. The CXH MUST be stored on the client and used in subsequent calls on the **EMSMDB** interface that require a valid CXH.

ulConMod: The connection modulus hash is determined by the client for a connection. How the client determines the hash value is not important. The client ensures that for a particular distinguished name passed in field **szUserDN**, the hash value is always the same. It is acceptable to have the same hash value for different distinguished names. 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 **EcDoConnectEx**.

A client can link two Session Contexts 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.
2. To get pending notification information for other sessions on the same client computer. See [RopPending](#) in [\[MS-OXCNOTIF\]](#) section 3.1.4.1.1 for details.

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

Note that the *ulIcxrLink* parameter is defined as a 32-bit value. Other than passing 0xFFFFFFFF for 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 field *piCxr* from a previous **EcDoConnectEx** call.

usFCanConvertCodePages: The client MUST pass a value of 0x01.

pcmsPollsMax: On success, this value is the number of milliseconds the client waits before polling the server for notification information. Other more dynamic options are available to the client for receiving notifications from the server. See [\[MS-OXCNOTIF\]](#) for details about working with Notifications. The client saves this value and associates it with the CXH.

pcRetry: On success, this value is the number of times the client retries a subsequent **EMSMDB** method call that uses the CXH that is returned in field *pcxh*. On failure, this value is Null. See section [3.2.4.3](#) for details about retrying RPC calls. The client saves this value and associates it with the CXH.

pcmsRetryDelay: On success, this value is the number of milliseconds a client waits before retrying a subsequent **EMSMDB** method call that uses the CXH that is returned in field *pcxh*. On failure, this value is Null. See section [3.2.4.3](#) for details about retrying RPC calls. The client saves this value and associates it with the CXH.

piCxr: On success, this value is a 16-bit session index that can be used in conjunction with the value returned in *pulTimeStamp* to link two Session Contexts on the server. On failure, this value is Null. See field *ulIcxrLink* for details about how to link Session Contexts and the reason why a client might want to do so.

The client saves this value and associates it with the CXH. It is the session index returned in a [RopPending](#) response command on calls to **EcDoRpcExt2**. The [RopPending](#) response command tells the client that a Session Context on the server has pending notifications. If a client links Session Contexts, a [RopPending](#) can be returned for any linked Session Context. See [\[MS-OXCROPS\]](#) section 3.1.5.1.3 and [\[MS-OXCNOTIF\]](#) section 3.1.4.1.1 for details about [RopPending](#).

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 [3.1.9](#).

rgwServerVersion: On success, this value is the server protocol version that the client uses to determine what protocol functionality the server supports. On failure, this value is Null. For details about how version numbers are interpreted from the wire data and the expected server behavior, see section [3.1.9](#). The client saves this value and associates it with the CXH.

pulTimeStamp: If a client wants 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 *pulTimeStamp* on a previous **EcDoConnectEx** call. If the client does not want to link Session Contexts, the client passes value 0x00000000.

On success, this value is the Session Context creation time stamp. On failure, this value is Null. The server saves the Session Context creation time stamp and associate it with the CXH.

3.2.4.2 Sending EcDoRpcExt2

When issuing the interface call **EcDoRpcExt2** some parameters need additional client-side consideration beyond what is stated in section [3.1.4.12](#). The following is a parameter for which the client has specific handling:

pcxh: The client MUST pass a valid Session Context Handle (CXH) that was created by calling **EcDoConnectEx**. On output, the server might have prematurely closed the client's session by clearing the CXH to zero. If the value on output is zero, the Session Context on the server has been destroyed.

3.2.4.3 Handling Server Too Busy

All method calls that require a valid Session Context Handle (CXH) are to be retried if the call fails with RPC status `RPC_S_SERVER_TOO_BUSY`. The number of times the client retries and the amount of time the client waits before retrying is based on fields *pcRetry* and *pcmsRetryDelay* returned on **EcDoConnectEx**. **EcDoConnectEx** is the only method that creates a CXH, so it is a prerequisite for any method that requires a CXH.

3.2.4.4 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 (CXH) in the response to an **EMSMDB** method call, the client cleans up any saved session state information and close the CXH if it is not already set to zero. The binding handle of the session is to be closed.

A client might choose to reconnect to the server automatically by creating a new binding handle and calling **EcDoConnectEx**. This will create 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 if the client wants to recreate or reopen the Server objects.

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

The abstract data model for this interface is the same as that for the **EMSMDB** interface. See section [3.1.1](#) for details about Session Context and Session Context Handles (CXHs).

Some methods on this interface require Session Context 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 **MUST** store this Session Context information and provide a CXH to the client to make subsequent interface calls using this same Session Context information.

The **AsyncEMSMDB** uses Asynchronous Context Handles (ACXH), which are RPC context handles. Every ACXHMUST map to the Session Context that is associated with a CXH. There is only one ACXH for a Session Context.

All methods on the **AsyncEMSMDB** interface that use an ACXH **MUST** be performed against the Session Context that is associated with the ACXH.

The server **MUST** keep a mapping between the ACXHand an active Session Context on the server. Session Context can be created and destroyed through the **EMSMDB** interface.

When the Session Context is destroyed or the client connection is lost, the ACXH **MUST** also be destroyed.

3.3.2 Timers

None.

3.3.3 Initialization

The server first **MUST** register the different protocol sequences that will allow clients to communicate with the server. This is done by calling RPC function **RpcServerUseProtseqEp**. See [\[MSDN-RPC\]](#) for details about this function and protocol sequences. 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](#).

The server **MUST** register the different authentication methods that are allowed on the **AsyncEMSMDB** interface. This is done by calling RPC function **RpcServerRegisterAuthInfo**. See [\[MSDN-RPC\]](#) for details about this function and authentication methods.

The server **MUST** start listening for RPC calls by calling RPC function **RpcServerListen**. See [\[MSDN-RPC\]](#) for details about this function.

The server **MUST** register the **AsyncEMSMDB** interface. This is done by calling RPC function **RpcServerRegisterIfEx**. See [\[MSDN-RPC\]](#) for details about this function.

The last step is to register the **AsyncEMSMDB** interface to all the registered binding handles created previously in calls to **RpcServerUseProtseq** or **RpcServerUseProtseqEp**. This is done by first acquiring all the binding handle information through RPC function **RpcServerInqBindings**, and then calling RPC function **RpcEpRegister** with the binding information. See [\[MSDN-RPC\]](#) for details about these functions.

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 [\[MS-RPCE\]](#) Section 3.

This interface includes the following method:

Method	opnum	Description
EcDoAsyncWaitEx	0	Asynchronous call that the server will not complete until there are pending events on the Session Context. The method requires an active Asynchronous Context Handle (ACXH) returned from EcDoAsyncConnectEx on interface EMSMDB .

3.3.4.1 EcDoAsyncWaitEx (opnum 0)

The method **EcDoAsyncWaitEx** is an asynchronous call that the server will not complete until there are pending events on the Session Context up to a five minute duration. If no events are available within five minutes, the server will return the call and will not set the NotificationPending flag in the *pulFlagsOut* field. If an event is pending, the server will complete the call immediately and return the NotificationPending flag in the *pulFlagsOut* field. This call requires an active Asynchronous Context Handle (ACXH) returned from **EcDoAsyncConnectEx** on interface **EMSMDB**. The ACXH 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: On input, the client MUST pass a valid ACXH that was created by calling **EcDoAsyncConnectEx** on interface **EMSMDB**. The server uses the ACXH to identify the Session Context to use for this call.

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

pulFlagsOut: Output flags for the client.

Flag	Value	Description
NotificationPending	0x00000001	Signals that events are pending for the client on the Session Context on the server. The client calls EcDoRpcExt2 with an empty remote operation (ROP) request buffer. The server will return the event details in the ROP response buffer.

3.3.5 Timer Events

None.

3.3.6 Other Local Events

None.

3.4 AsyncEMSMDB Client Details

3.4.1 Abstract Data Model

For some functionality on the **AsyncEMSMDB** interface, it is required that the client store an Asynchronous Context Handle (ACXH) and use it on subsequent interface calls that require an ACXH.

3.4.2 Timers

No protocol timers are required beyond those internal timers used in RPC to implement resiliency to network outages. For details, see [\[MS-RPCE\]](#).

3.4.3 Initialization

This interface can only be used after first obtaining an Asynchronous Context Handle (ACXH) from the method **EcDoAsyncConnectEx** from interface **EMSMDB**.

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 [\[MS-RPCE\]](#) section 3.

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 are examples of how a client and server use this protocol connection, submit ROP commands, and disconnect.

4.1 Client Connecting to Server

1. Client creates an RPC binding handle to the server with the "ncacn_ip_tcp" protocol sequence and the RPC_C_AUTHN_WINNT authentication method.
2. Client makes **EMSMDB** interface method call **EcDoConnectEx** with the following parameters to establish a Session Context with the server:

hBinding: binding handle created in step 1.

pcxh: Pointer to CXH to hold output value. In this example the client initializes CXH to zero.

szUserDN: User's distinguished name. String that contains the distinguished name of the user who is making the **EcDoConnectEx** call in a directory service. Value: "/o=First Organization/ou=First Administrative Group/**CN=recipients**/CN=janedow".

ulFlags: Value 0x00000000. Regular user access.

ulConMod: Value 0x00340567. Client computed hash on szUserDN value.

cbLimit: Value 0x00000000.

ulCpid: Value 0x000004E4. code page 1252.

ulLcidString: Value 0x00000409. **locale** 1033 "en-us".

ulLcidSort: Value 0x00000409. locale 1033 "en-us".

ulIcxrLink: Value 0xFFFFFFFF. No session link.

usFCanConvertCodePages: Value 0x01.

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: Value 0x00000000.

rgbAuxOut: Pointer to buffer of size 0x1008.

pcbAuxOut: Pointer to unsigned long value 0x00001008.

3. Server processes **EcDoConnectEx** request. Verifies that authentication context associated with *hBinding* handle has ownership privileges to a directory service object that contains a distinguished name in field *szUserDN*. Server creates Session Context and assigns a CXH (using 0x00001234 for this example). Server returns the following output values:

pcxh: Value at CXH pointer is 0x00001234. Note that the actual RPC context handle returned to the client in this field 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. In this example the client is instructed to poll for events every 60 seconds.

pcRetry: Value at unsigned long pointer is 0x00000006. In this example the client is instructed to retry six times before failing.

pcmsRetryDelay: Value at unsigned long pointer is 0x00001770. In this example 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: 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. Field *rgbAuxOut* is 16 bytes in length.

Return Value: Value is 0x00000000.

4.2 Client Issuing ROP Commands to Server

1. Client has already established a Session Context with the server and has a valid Session Context Handle (CXH). For more information, see steps 1 through 3 of section [4.1](#)
2. Client sends ROP commands to server by calling **EcDoRpcExt2** using the CXH returned from the **EcDoConnectEx** call.

pcxh: Pointer to CXH value which is 0x00001234.

pulFlags: Pointer to unsigned long containing value 0x00000003. Client requests server to not compress or XOR payload of *rgbOut* and *rgbAuxOut*.

rgbIn: Client passes extended buffer and payload containing ROP commands to be processed by server. See [\[MS-OXCROPS\]](#) for details about ROP commands.

RPC_HEADER_EXT				Payload		
				ROP Request Commands		
Version	flags	Size	SizeActual	RopSize	ROPs	ServerObjectHandleTable
0x0000	0x0004	0x0152	0x0152	0x0142	320 bytes	16 bytes

Payload is not compressed and not obfuscated.

cbIn: Value of 0x0000015A.

rgbAuxIn: Null pointer value.

cbAuxIn: Value of 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.

3. Server processes **EcDoRpcExt2** request. Server verifies that CXH is for a valid Session Context for this user. Server processes ROP request commands and returns ROP response results to client. Server returns the following output values:

pcxh: Value at CXH 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	ServerObjectHandleTable
0x0000	0x0004	0x0052	0x0052	0x0042	64 bytes	16 bytes

Payload is not compressed and not obfuscated.

pcbOut: Value is 0x0000005A.

rgbAuxOut: Server returns nothing in the auxiliary output buffer.

pcbAuxOut: Value is 0x00000000.

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

Return Value: Value is 0x00000000.

4.3 Client Receiving "Packed" ROP Response from Server

1. Client has already established a Session Context with the server and has a valid Session Context Handle (CXH). For more information, see steps 1 through 3 of section [4.1](#).
2. Client sends ROP commands to server by calling **EcDoRpcExt2** using the CXH that is returned from the **EcDoConnectEx** call. The last ROP request contains [RopReadStream](#), and so client requests response chaining (for example, "packing").

pcxh: Pointer to CXH value, which is 0x00001234.

pulFlags: Pointer to unsigned long containing value 0x00000007. Client requests server to not compress or XOR payload of *rgbOut* and *rgbAuxOut*. Client requests response chaining.

rgbIn: Client passes extended buffer and payload containing ROP commands to be processed by server. See [\[MS-OXCROPS\]](#) for details about ROP commands.

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: Value of 0x0000015A.

rgbAuxIn: Null pointer value.

cbAuxIn: Value of 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.

3. Server processes **EcDoRpcExt2** request. Server verifies that CXH is for a valid Session Context for this user. Server processes ROP request commands and returns ROP response results to client. The last ROP was [RopReadStream](#), and the client has requested chaining; there is more data to return in the stream being read, there is more room in the *rgbOut* output buffer and the server adds another extended buffer and payload. The server returns the following output values:

pcxh: Value at CXH 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](#) command.

RPC_HEADER_EXT flags: 0x0000 Size: 0x7FFE	Payload			RPC_HEADER_EXT flags: 0x0004 Size: 0x2008	Payload		
	ROP response Commands				ROP response Command		
	RopSize	ROPs	SOHT		RopSize	ROP	SOHT
	0x7FEE		16 bytes			16 bytes	

Payloads are not compressed and not obfuscated.

pcbOut: Value is 0x0000A016.

rgbAuxOut: Server returns nothing in the auxiliary output buffer.

pcbAuxOut: Value is 0x00000000.

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

Return Value: Value is 0x00000000.

4.4 Client Disconnecting from Server

1. Client has already established a Session Context with the server and has a valid Session Context Handle (CXH). For more information, see steps 1 through 3 of section [4.1](#).
2. Client is exiting and wants to destroy the Session Context on the server. Client issues **EcDoDisconnect** using the CXH that was returned from the **EcDoConnectEx** call.

pcxh: Pointer to CXH value, which is 0x00001234.

3. Server processes **EcDoDisconnect** request. Server verifies that CXH is for a valid Session Context for this user. Server destroys Session Context and invalidates CXH. Server returns the following output values:

pcxh: Value at CXH pointer is 0x00000000.

Return Value: Value is 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 (CXH), which can only be created from a successful call to **EcDoConnectEx**. 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 calls 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 distinguished name passed in the *szUserDN* field. 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 distinguished name. If the authentication context does not have adequate permissions, then the server fails the call and does not create a Session Context.

Although the protocol allows for data compression and data obfuscation on method call **EcDoRpcExt2**, 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 fields of all method calls on the **EMSMDB** and **AsyncEMSMDB** interfaces.

5.2 Index of Security Parameters

None.

6 Appendix A: Full IDL/ACF

For ease of implementation, the full **IDL** and ACF is provided in the following sections, where "ms-rpce.IDL" refers to the IDL found in [\[MS-RPCE\]](#) Appendix A. The syntax uses the IDL syntax extensions as specified in [\[MS-RPCE\]](#) section 2.2.4 and [\[MS-RPCE\]](#) section 3.1.5.1. For example, as specified in [\[MS-RPCE\]](#) section 2.2.4.8, a pointer_default declaration is not required and pointer_default(unique) is assumed.

6.1 IDL

```
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  
  );  
}
```

6.2 ACF

The ACF specifies attributes that affect only local performance rather than the network contract.

```
typedef [context_handle_noserialize] ACXH;  
  
interface asyncemsmdb  
{  
  [async] EcDoAsyncWaitEx();  
}
```

7 Appendix B: Product Behavior

The information in this specification is applicable to the following product versions. References to product versions include released service packs.

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

Exceptions, if any, are noted below. If a service pack number appears with the product version, behavior changed in that service pack. The new behavior also applies to subsequent service packs of the product unless otherwise specified.

Unless otherwise specified, any statement of optional behavior in this specification 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 product does not follow the prescription.

[<1> Section 1.9:](#) Exchange 2010 does not support MExchangeIS_LPC.

[<2> Section 2.1:](#) Exchange 2010 does not support ncalrpc.

[<3> Section 2.1:](#) Exchange 2010 does not support MExchangeIS_LPC.

[<4> Section 3.1.4:](#) Opnums reserved for local use apply to Office/Exchange as follows.

opnum	Description
0	Not used by Office/Exchange.
2	Not used by Office/Exchange.
3	Not used by Office/Exchange.
5	Not used by Office/Exchange.
7	Not used by Office/Exchange.
8	Not used by Office/Exchange.
9	Not used by Office/Exchange.
12	Not used by Office/Exchange.
13	Not used by Office/Exchange.

[<5> Section 3.1.4.5:](#) Exchange Server 2010 does not support **EcRRegisterPushNotification** when client connection services are deployed on an Exchange server that does not also have a mailbox store installed. The returned value will always be ecNotSupported.

<6> [Section 3.1.4.11](#): Exchange Server 2010 does not support Session Context linking. If ulIcxrLink is not 0xFFFFFFFF, then 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 that was passed in.

<7> [Section 3.1.4.11](#): Exchange Server 2010 does not support Session Context linking when client connection services are deployed on an Exchange server that does not also have a mailbox store installed. If ulIcxrLink is not 0xFFFFFFFF, then 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 that was passed in.

<8> [Section 3.1.4.11](#): The call will fail with ecRpcFailed (0x80040115) if cbAuxIn is greater than 0x00000000 and less than 0x00000008.

<9> [Section 3.1.4.12](#): Exchange Server 2010: The call will fail with ecRpcFailed (0x80040115) when cbIn is greater than 0x00000000 and less than 0x00000008. It will not fail when cbIn is 0x00000000.

<10> [Section 3.1.4.12](#): Exchange Server 2010 does not require that the server fail if the output buffer is less than 0x00008007 bytes. It will fail with ecRpcFailed (0x80040115) if the output buffer is not more than 0x00000008 bytes in size.

<11> [Section 3.1.4.12](#): Exchange Server 2010: It will fail with ecRpcFailed (0x80040115) if the cbAuxIn parameter is greater than 0x00000000 and less than 0x00000008

<12> [Section 3.1.8.1](#): Outlook 2010 by default does not populate MachineName, UserName, ClientIP, and MacAddress within AUX_PERF_CLIENTINFO.

<13> [Section 3.1.8.2](#): Exchange Server 2010 does not support sending the AUX_OSVERSIONINFO block when client connection services are deployed on an Exchange server that does not also have a mailbox store installed.

7.1 Protocol Sequences

7.1.1 Exchange Server Support

Exchange 2003 SP2 allows all RPC protocol sequences listed in section 2.1.

Exchange 2007 SP1 allows only the following RPC protocol sequences: ncalrpc, ncacl_ip_tcp and ncacl_http.

Exchange 2010 allows only the following RPC protocol sequences: ncacl_ip_tcp and ncacl_http.

7.1.2 Office Client Support

Office 2003 SP3 uses only the following RPC protocol sequences: ncacl_ip_tcp and ncacl_http.

Office 2007 SP1 uses only the following RPC protocol sequences: ncacl_ip_tcp and ncacl_http.

7.2 Authentication Methods

The following [table](#) lists the authentication methods supported by Exchange 2003 SP2 and Exchange 2007 SP1. A client authenticates using one of these authentication methods.

Authentication Method

Authentication Method
RPC_C_AUTHN_WINNT
RPC_C_AUTHN_GSS_KERBEROS
RPC_C_AUTHN_GSS_NEGOTIATE

7.3 RPC Methods

7.3.1 Exchange Server Support

The following table indicates which RPC methods are supported in which versions of Exchange.

EMSMDB Interface:

Method	Exchange 2003 SP2	Exchange 2007 SP1	Exchange 2010
EcDoDisconnect	X	X	X
EcRRegisterPushNotification	X	X	X
EcDummyRpc	X	X	X
EcDoConnectEx	X	X	X
EcDoRpcExt2	X	X	X
EcDoAsyncConnectEx		X	X

AsyncEMSMDB Interface:

Method	Exchange 2003 SP2	Exchange 2007 SP1	Exchange 2010
EcDoAsyncWaitEx		X	X

7.3.2 Office Client Support

An Office client will use different RPC methods based on the version of Exchange that it is accessing.

7.3.2.1 Accessing Exchange 2003

The following table indicates which RPC methods are used by an Office client when accessing a computer that is running Exchange 2003.

EMSMDB Interface:

Method	Office 2003 SP2	Office 2007 SP1
EcDoDisconnect	X	X

Method	Office 2003 SP2	Office 2007 SP1
EcRRegisterPushNotification	X	X
EcDummyRpc		
EcDoConnectEx	X	X
EcDoRpcExt2	X	X
EcDoAsyncConnectEx		

AsyncEMSMDB Interface:

Method	Office 2003 SP2	Office 2007 SP1
EcDoAsyncWaitEx		

7.3.2.2 Accessing Exchange 2007

The following table indicates which RPC methods are used by an Office client when it is accessing a computer that is running Exchange 2007.

EMSMDB Interface:

Method	Office 2003 SP2	Office 2007 SP1
EcDoDisconnect	X	X
EcRRegisterPushNotification	X	X
EcDummyRpc		
EcDoConnectEx	X	X
EcDoRpcExt2	X	X
EcDoAsyncConnectEx		X

AsyncEMSMDB Interface:

Method	Office 2003 SP2	Office 2007 SP1
EcDoAsyncWaitEx		X

7.3.2.3 Accessing Exchange 2010

The following table indicates which RPC methods are used by an Office client when it is accessing a computer that is running Exchange 2010.

EMSMDB Interface:

Method	Office 2003 SP2	Office 2007 SP1
EcDoDisconnect	X	X
EcRRegisterPushNotification	X	X
EcDummyRpc		
EcDoConnectEx	X	X
EcDoRpcExt2	X	X
EcDoAsyncConnectEx		X

AsyncEMSMDB Interface:

Method	Office 2003 SP2	Office 2007 SP1
EcDoAsyncWaitEx		X

7.4 Client Access Licenses

As of Exchange 2007 SP1, the server no longer counts individual connections for **Client Access License** accounting, so **Session Context** linking is not required in method call **EcDoConnectEx** on the **EMSMDB** interface.

8 Change Tracking

This section identifies changes made to [MS-OXCRPC] protocol documentation between November 2009 and February 2010 releases. Changes are classed as major, minor, or editorial.

Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements or functionality.
- An extensive rewrite, addition, or deletion of major portions of content.
- A protocol is deprecated.
- The removal of a document from the documentation set.
- Changes made for template compliance.

Minor changes do not affect protocol interoperability or implementation. Examples are updates to fix technical accuracy or ambiguity at the sentence, paragraph, or table level.

Editorial changes apply to grammatical, formatting, and style issues.

No changes means that the document is identical to its last release.

Major and minor changes can be described further using the following revision types:

- New content added.
- Content update.
- Content removed.
- New product behavior note added.
- Product behavior note updated.
- Product behavior note removed.
- New protocol syntax added.
- Protocol syntax updated.
- Protocol syntax removed.
- New content added due to protocol revision.
- Content updated due to protocol revision.
- Content removed due to protocol revision.
- New protocol syntax added due to protocol revision.
- Protocol syntax updated due to protocol revision.
- Protocol syntax removed due to protocol revision.
- New content added for template compliance.
- Content updated for template compliance.

- Content removed for template compliance.
- Obsolete document removed.

Editorial changes always have the revision type "Editorially updated."

Some important terms used in revision type descriptions are defined as follows:

Protocol syntax refers to data elements (such as packets, structures, enumerations, and methods) as well as interfaces.

Protocol revision refers to changes made to a protocol that affect the bits that are sent over the wire.

Changes are listed in the following table. If you need further information, please contact protocol@microsoft.com.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Revision Type
1 Introduction	48461 Pluralized "interface".	N	Content update.
1.1 Glossary	48161 Moved the terms ACXH and CXH to the list of [MS-OXGLOS] terms.	N	Content update.
1.3.2 Issuing Remote Operations for Mailbox Data	48492 Updated normative language.	N	Content update.
2.2.2.2 AUX HEADER	48461 Updated "between" to "among".	N	Content update.
2.2.2.20 AUX PERF ACCOUNTINFO	53721 New topic.	Y	New content added.
3.1.4 Message Processing Events and Sequencing Rules	48486 Modified Opnum descriptions.	Y	Content update.
3.1.4 Message Processing Events and Sequencing Rules	51635 Modified endnotes to contain unused opnums.	Y	Content update.
3.1.4.1 Opnum0NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.1 Opnum0NotUsedOnWire	51635 Modified Opnum name.	Y	Content update.
3.1.4.3 Opnum2NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.3 Opnum2NotUsedOnWire	51635 Modified Opnum name.	Y	Content update.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Revision Type
3.1.4.4 Opnum3NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.4 Opnum3NotUsedOnWire	51635 Modified Opnum name.	Y	Content update.
3.1.4.6 Opnum5NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.6 Opnum5NotUsedOnWire	51635 Modified Opnum name.	Y	Content update.
3.1.4.8 Opnum7NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.8 Opnum7NotUsedOnWire	51635 Modified Opnum name.	Y	Content update.
3.1.4.9 Opnum8NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.9 Opnum8NotUsedOnWire	51635 Modified Opnum name.	Y	Content update.
3.1.4.10 Opnum9NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.10 Opnum9NotUsedOnWire	51635 Modified Opnum name.	Y	Product behavior note updated.
3.1.4.13 Opnum12NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.13 Opnum12NotUsedOnWire	51635 Modified Opnum name.	Y	Content update.
3.1.4.14 Opnum13NotUsedOnWire	48486 Modified Opnum name.	Y	Content update.
3.1.4.14 Opnum13NotUsedOnWire	51635 Modified Opnum name.	Y	Content update.
3.1.7.4 Extended Buffer Packing	51643 Added a link to [MS-OXCROPS].	N	Content update.
3.1.8.1 Client Performance Monitoring	54275 Added information about Outlook 2010 behavior.	N	New product behavior note added.
3.1.9.2 Server Versions	51643 Added section links.	N	Content update.

9 Index

A

- Abstract data model
 - [client](#) 59
 - [server](#) 30
- [Applicability](#) 10
- [AUX_CLIENT_CONTROL packet](#) 27
- [AUX_EXORGINFO packet](#) 29
- [AUX_HEADER packet](#) 14
- [AUX_OSVERSIONINFO packet](#) 28
- [AUX_PERF_ACCOUNTINFO packet](#) 29
- [AUX_PERF_CLIENTINFO packet](#) 17
- [AUX_PERF_DEFGC_SUCCESS packet](#) 22
- [AUX_PERF_DEFMDB_SUCCESS packet](#) 22
- [AUX_PERF_FAILURE packet](#) 25
- [AUX_PERF_FAILURE_V2 packet](#) 26
- [AUX_PERF_GC_SUCCESS packet](#) 24
- [AUX_PERF_GC_SUCCESS_V2 packet](#) 25
- [AUX_PERF_MDB_SUCCESS packet](#) 23
- [AUX_PERF_MDB_SUCCESS_V2 packet](#) 23
- [AUX_PERF_PROCESSINFO packet](#) 21
- [AUX_PERF_REQUESTID packet](#) 16
- [AUX_PERF_SERVERINFO packet](#) 20
- [AUX_PERF_SESSIONINFO packet](#) 16
- [AUX_PERF_SESSIONINFO_V2 packet](#) 17

C

- [Capability negotiation](#) 10
- [Change tracking](#) 79
- Client
 - [abstract data model](#) 59
 - [EMSMDB interface](#) 59
 - [initialization](#) 59
 - [local events](#) 61
 - [message processing](#) 59
 - [overview](#) 59
 - [sequencing rules](#) 59
 - [timer events](#) 61
 - [timers](#) 59
- [Common data types](#) 11

D

- Data model – abstract
 - [client](#) 59
 - [server](#) 30
- Data types
 - [common - overview](#) 11

E

- EMSMDB interface ([section 3.1](#) 30, [section 3.2](#) 59)
- Events
 - [local - client](#) 61
 - [local - server](#) 42
 - [timer - client](#) 61
 - [timer - server](#) 42

- [Examples - overview](#) 65

F

- [Fields - vendor-extensible](#) 10
- [Full IDL](#) 71

G

- [Glossary](#) 6

I

- [IDL](#) 71
- [Implementer - security considerations](#) 70
- [Index of security parameters](#) 70
- [Informative references](#) 7
- Initialization
 - [client](#) 59
 - [server](#) 31
- Interfaces – client
 - [EMSMDB](#) 59
- Interfaces – server
 - [EMSMDB](#) 30
- [Introduction](#) 6

L

- Local events
 - [client](#) 61
 - [server](#) 42

M

- Message processing
 - [client](#) 59
 - [server](#) 31
- Messages
 - [common data types](#) 11
 - [overview](#) 11
 - [transport](#) 11

N

- [Normative references](#) 7

O

- [Overview](#) 7

P

- [packet](#) 13
- [Parameters - security index](#) 70
- [Preconditions](#) 9
- [Prerequisites](#) 9
- [Product behavior](#) 74

R

References

- [informative](#) 7
- [normative](#) 7
- [Relationship to other protocols](#) 9

S

Security

- [implementer considerations](#) 70
- [overview](#) 70
- [parameter index](#) 70

Sequencing rules

- [client](#) 59
- [server](#) 31

Server

- [abstract data model](#) 30
- [EMSMDb interface](#) 30
- [initialization](#) 31
- [local events](#) 42
- [message processing](#) 31
- [overview](#) 30
- [sequencing rules](#) 31
- [timer events](#) 42
- [timers](#) 31

- [Standards assignments](#) 10

T

Timer events

- [client](#) 61
- [server](#) 42

Timers

- [client](#) 59
- [server](#) 31
- [Tracking changes](#) 79
- [Transport](#) 11

V

- [Vendor-extensible fields](#) 10
- [Versioning](#) 10