[MS-OXCRPC]: Wire Format Protocol Specification

Intellectual Property Rights Notice for Protocol Documentation

- **Copyrights**. This protocol 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 protocols, and may distribute portions of it in your implementations of the protocols or your documentation as necessary to properly document the implementation. This permission also applies to any documents that are referenced in the protocol documentation.

- **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 protocols. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. However, the protocols may be covered by Microsoft's Open Specification Promise (available here: http://www.microsoft.com/interop/osp/default.mspx). If you would prefer a written license, or if the protocols are not covered by the OSP, patent licenses are available by contacting protocol@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.

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.

Preliminary Documentation. This documentation is preliminary documentation for these protocols. Since the documentation may change between this preliminary version and the final version, there are risks in relying on preliminary documentation. To the extent that you incur additional development obligations or any other costs as a result of relying on this preliminary documentation, you do so at your own risk.

Tools. This protocol documentation is intended for use in conjunction with publicly available standard specifications and networking programming art, and assumes that the reader is either familiar with the aforementioned material or has immediate access to it. A protocol specification does not require the use of Microsoft programming tools or programming environments in order for a Licensee to develop an implementation. Licensees who have access to Microsoft programming tools and environments are free to take advantage of them.

Revision Summary			
Author	Date	Version	Comments
Microsoft Corporation	April 4, 2008	0.1	Initial Availability

Table of Contents

1 Introduction

The Wire Format Protocol is specific to the EMSMDB and AsyncEMSMDB protocol interface between a client and server. This interface has been traditionally used by an Outlook client to communicate with an Exchange messaging server.

1.1 Glossary

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

Authentication Level

Authentication Service

Dynamic Endpoint

Endpoint

Globally Unique Identifier (GUID)

Interface Definition Language (IDL)

Microsoft Interface Definition Language (MIDL)

Network Data Representation (NDR)

Opnum

Remote Procedure Call (RPC)

RPC Protocol Sequence

RPC Transfer Syntax

RPC Transport

Security Provider

Universal Unique Identifier (UUID)

Well-Known Endpoint

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

distinguished name (DN)

remote operation (ROP)

The following terms are specific to this document:

Asynchronous Context Handle (ACXH): An RPC context handle used by a client when issuing RPC calls against a server on AsyncEMSMDB interface methods. Represents a handle to a unique Session Context on the server.

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

Session Context Handle (CXH): An RPC context handle used by a client when issuing RPC calls against a server on EMSMDB interface methods. Represents a handle to a unique **Session Context** on the server.

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

[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", April 2008.

[MS-OXCNOTIF] Microsoft Corporation, "Core Notifications Protocol Specification", April 2008.

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

[MS-OXCSTOR] Microsoft Corporation, "Store Object Protocol Specification", April 2008.

[MS-OXGLOS] Microsoft Corporation, "Office Exchange Protocols Master Glossary", April 2008.

[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

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

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

1.3 Protocol Overview (Synopsis)

The [MS-OXCRPC] protocol describes the RPC interfaces used by a messaging client to communicate with a messaging server to access personal messaging data over the Office

Exchange Protocol. The Office Exchange 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 MUST first make a call to EcDoConnectEx and establish a **Session Context Handle** (CXH). The session context handle is a RPC context handle. The client MUST store this Session Context Handle and use it on subsequent RPC calls on the EMSMDB interface. The server uses the Session Context Handle 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 MUST verify that the authentication context used to make the RPC function call EcDoConnectEx has access rights to perform operations as, or on the 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 MUST fail the RPC call with an access denied return code.

If the security check passes, the server MUST create a Session Context. A CXH which refers to the Session Context MUST be returned to the client in the response to EcDoConnectEx. The returned CXH MUST be 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 additional information on ROP commands. The ROP request operations are tokenized into a request buffer and sent to the server as a byte array. The server must then parse the ROP request buffer and perform actions. The response to these actions are 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 additional information on 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.

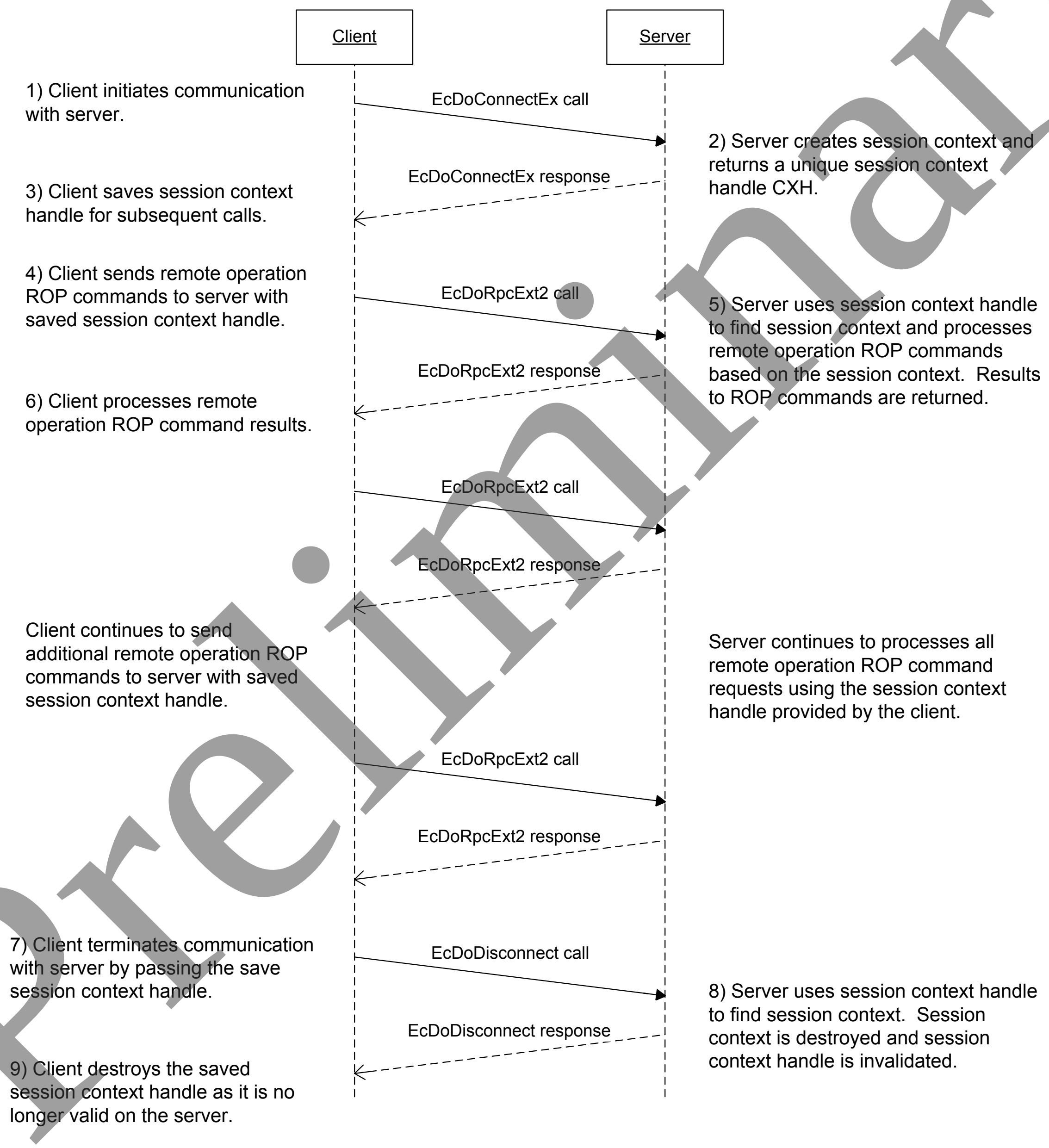
The client MUST pass in the call to EcDoRpcExt2 the CXH which was created in a successful call to the interface function EcDoConnectEx. The server uses the CXH to identify who is issuing the remote operation ROP commands and under which Session Context to perform them.

1.3.3 Terminating Communication with the Server

When a client wishes to terminate communication with a server, it MUST call EcDoDisconnect. The client MUST pass in the call to EcDoDisconnect the CXH which was created in a successful call to the interface function EcDoConnectEx. The server SHOULD cleanup any Session Context data associated with this CXH.

1.3.4 Client/Server Communication Lifetime

The following sequence diagram shows a typical example of the client and server communication lifetime. This is a simplified overview of how the client connects, issues remote operation ROP commands, and disconnects from the server.



1) Client initiates communication with server.

2) Server creates session context and returns a unique session context handle CXH.

3) Client saves session context handle for subsequent calls.

4) Client sends remote operation ROP commands to server with saved session context handle.

5) Server uses session context handle to find session context and processes remote operation ROP commands based on the session context. Results to ROP commands are returned.

6) Client processes remote operation ROP command results.

Client continues to send additional remote operation ROP commands to server with saved session context handle.

Server continues to processes all remote operation ROP command requests using the session context handle provided by the client.

7) Client terminates communication with server by passing the save session context handle.

8) Server uses session context handle to find session context. Session context is destroyed and session context handle is invalidated.

9) Client destroys the saved session context handle as it is no longer valid on the server.

Figure 1: Client/server communications

1.4 Relationship to Other Protocols

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

1.5 Prerequisites/Preconditions

The Office Exchange Protocol is a set of RPC interfaces and has the same prerequisites as specified in [MS-RPCE].

It is assumed that an Office Exchange Protocol client has obtained the name of a remote machine that supports the Office Exchange Protocol before these protocols are invoked. How a client does this is outside of the scope of this specification.

1.6 Applicability Statement

The protocol describe herein is applicable to environments which require access to private mailbox and/or public folder messaging end-user data.

1.7 Versioning and Capability Negotiation

This document 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 EMSMDB has a single version number of 0.81. The protocol RPC interface AsyncEMSMDB has a single version number of 0.01.
- **Protocol Versions:** The protocol RPC interface EMSMDB has a single interface version, but that interface has been extended by adding additional methods at the end. The use of these methods is specified in section 3.1.
- **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:** None.

1.8 Vendor-Extensible Fields

None.

1.9 Standards Assignments

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

2 Messages

2.1 Transport

This protocol works over the following protocol sequences listed in the table below.

Protocol Sequence
ncalrpc
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
ncacn_http	6001

For all other network protocol sequences the protocol uses RPC Dynamic Endpoints as specified in Part 4 of [C706].

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 underlying RPC protocol to retrieve the identity of the caller that made the method call as specified in [MS-RPCE]. The server SHOULD use this identity to perform method specific access checks.

2.2 Common Data Types

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

2.2.1 Simply Data Types

2.2.1.1 CXH

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

2.2.1.2 ACXH

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

2.2.2 Structures

2.2.2.1 RPC_HEADER_EXT

```
typedef struct _RPC_HEADER_EXT {
      unsigned short     Version;
      unsigned short     Flags;
      unsigned short     Size;
      unsigned short     SizeActual;
} RPC_HEADER_EXT;
```

Version (2 bytes): Defines the version of the header. There is only one version of the header at this time so this value MUST be set to 0x0000.

Flags (2 bytes): Flags which specify how data following this header MUSTSHOULD be interpreted. The following flags are valid.

Flag	Value	Description
Compressed	0x0001	The data following 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 on obfuscation algorithm..
Last	0x0004	Indicates there is not another RPC_HEADER_EXT following the data of the current RPC_HEADER_EXT. This flag is used to indicate 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 following 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

```
typedef struct _AUX_HEADER {
      unsigned short Size;
      unsigned char Version;
      unsigned char Type;
} AUX_HEADER;
```

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

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

Version	Value
AUX_VERSION_1	0x01
AUX_VERSION_2	0x02

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

This is a list of block types and the corresponding structure following the AUX_HEADER when the Version field is AUX_VERSION_1.

Type	Value	Payload
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	0x09	AUX_PERF_FAILURE
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

Type	Value	Payload
AUX_TYPE_PERF_BG_GC_SUCCESS	0x0F	AUX_PERF_GC_SUCCESS
AUX_TYPE_PERF_BG_FAILURE	0x10	AUX_PERF_FAILURE
AUX_TYPE_PERF_FG_DEFMDB_SUCCESS	0x11	AUX_PERF_DEFMDB_SUCCESS
AUX_TYPE_PERF_FG_DEFGC_SUCCESS	0x12	AUX_PERF_DEFGC_SUCCESS
AUX_TYPE_PERF_FG_MDB_SUCCESS	0x13	AUX_PERF_MDB_SUCCESS
AUX_TYPE_PERF_FG_GC_SUCCESS	0x14	AUX_PERF_GC_SUCCESS
AUX_TYPE_PERF_FG_FAILURE	0x15	AUX_PERF_FAILURE
AUX_TYPE_OSINFO	0x16	AUX_OSINFO
AUX_TYPE_EXORGINO	0x17	AUX_EXORGINFO

This is a list of block types and the corresponding structure following the AUX_HEADER when the Version field is AUX_VERSION_2.

Type	Value	Payload
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	0x08	AUX_PERF_GC_SUCCESS_V2
AUX_TYPE_PERF_FAILURE	0x09	AUX_PERF_FAILURE_V2

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

2.2.2.3 AUX_PERF_REQUESTID

```
typedef struct _AUX_PERF_REQUESTID {
       unsigned short SessionID;
       unsigned short RequestID;
} AUX_PERF_REQUESTID;
```

SessionID (2 bytes): Session identification number.

RequestID (2 bytes): Request identification number.

2.2.2.4 AUX_PERF_SESSIONINFO

```
typedef struct _AUX_PERF_SESSIONINFO {
      unsigned short SessionID;
      GUID SessionGuid;
} AUX_PERF_SESSIONINFO;
```

SessionID (2 bytes): Session identification number.

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

```
typedef struct _AUX_PERF_SESSIONINFO_V2 {
      unsigned short SessionID;
      GUID SessionGuid;
      unsigned long ConnectionID;
} AUX_PERF_SESSIONINFO_V2;
```

SessionID (2 bytes): Session identification number.

SessionGuid (2 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

```
typedef struct _AUX_PERF_CLIENTINFO {
      unsigned long AdapterSpeed;
      unsigned short ClientID;
      unsigned short MachineNameOffset;
      unsigned short UserNameOffset;
      unsigned short ClientIPSize;
      unsigned short ClientIPOffset;
      unsigned short ClientIPMaskSize;
      unsigned short ClientIPMaskOffset;
      unsigned short AdapterNameOffset;
      unsigned short MacAddressSize;
      unsigned short MacAddressOffset;
      unsigned short ClientMode;
} AUX_PERF_CLIENTINFO;
```

AdapterSpeed (4 bytes): Speed of client machines network adaptor (kbits/s)

ClientID (2 bytes): Client assigned identification number.

MachineNameOffset (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to a null-terminated Unicode string which contains the client machine name.

UserNameOffset (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to a null-terminated Unicode string which contains the user's account name.

ClientIPSize (2 bytes): Size of the client IP address referenced by field ClientIPOffset.

ClientIPOffset (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to the clients IP address. Size of the IP address data is found in field ClientIPSize.

ClientIPMaskSize (2 bytes): Size of the client IP subnet mask referenced by field ClientIPMaskOffset.

ClientIPMaskOffset (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to the clients IP subnet mask. Size of the IP subnet mask is found in field ClientIPMaskSize.

AdapterNameOffset (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to a null-terminated Unicode string which contains the client network adapter name.

MacAddressSize (2 bytes): Size of the network adapter MAC address referenced by field MacAddressOffset.

MacAddressOffset (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to the client network adapter MAC address. Size of the network adapter MAC address is found in field MacAddressSize.

ClientMode (2 bytes): Determines the mode in which the client is running.

Mode	Value	Description
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.

2.2.2.7 AUX_PERF_SERVERINFO

```
typedef struct _AUX_PERF_SERVERINFO {
      unsigned short ServerID;
      unsigned short ServerType;
      unsigned short ServerDNOffset;
      unsigned short ServerNameOffset;
} AUX_PERF_SERVERINFO;
```

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

ServerType (2 bytes): Server type assigned by client.

Type	Value	Description
SERVERTYPE_UNKNOWN	0x00	Unknown server type.
SERVERTYPE_PRIVATE	0x01	Client server connection servicing private mailbox data.
SERVERTYPE_PUBLIC	0x02	Client server connecting servicing public

Type	Value	Description
		folder data.
SERVERTYPE_DIRECTORY	0x03	Client server connection servicing directory data.
SERVERTYPE_REFERRAL	0x04	Client server connection servicing referrals.

ServerDNOffset (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to a null-terminated Unicode string which contains the server's distinguished name.

ServerNameOffset (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to a null-terminated Unicode string which contains the server name.

2.2.2.8 AUX_PERF_PROCESSINFO

```
typedef struct _AUX_PERF_PROCESSINFO {
      unsigned short ProcessID;
      GUID ProcessGuid;
      unsigned short ProcessNameOffset;
} AUX_PERF_PROCESSINFO;
```

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

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

ProcessName (2 bytes): Offset relative to the beginning of the AUX_HEADER structure that MUST exist prior to this structure which points to a null-terminated Unicode string which contains the client process name.

2.2.2.9 AUX_PERF_DEFMDB_SUCCESS

```
typedef struct _AUX_PERF_DEFMDB_SUCCESS {
      unsigned long TimeSinceRequest;
      unsigned long TimeToCompleteRequest;
      unsigned short RequestID;
} AUX_PERF_DEFMDB_SUCCESS;
```

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.

2.2.2.10 AUX_PERF_DEFGC_SUCCESS

```
typedef struct _AUX_PERF_DEFGC_SUCCESS {
      unsigned short ServerID;
      unsigned short SessionID;
      unsigned long TimeSinceRequest;
      unsigned long TimeToCompleteRequest;
      unsigned char RequestOperation;
} AUX_PERF_DEFGC_SUCCESS;
```

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 which was successful.

2.2.2.11 AUX_PERF_MDB_SUCCESS

```
typedef struct _AUX_PERF_MDB_SUCCESS {
      unsigned short ClientID;
      unsigned short ServerID;
      unsigned short SessionID;
      unsigned short RequestID;
      unsigned long TimeSinceRequest;
      unsigned long TimeToCompleteRequest;
} AUX_PERF_MDB_SUCCESS;
```

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

```
typedef struct _AUX_PERF_MDB_SUCCESS_V2 {
      unsigned short ProcessID;
      unsigned short ClientID;
      unsigned short ServerID;
      unsigned short SessionID;
      unsigned short RequestID;
      unsigned long TimeSinceRequest;
      unsigned long TimeToCompleteRequest;
} AUX_PERF_MDB_SUCCESS_V2;
```

PrcoessID (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.

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

```
typedef struct _AUX_PERF_GC_SUCCESS {
      unsigned short ClientID;
      unsigned short ServerID;
      unsigned short SessionID;
      unsigned long TimeSinceRequest;
      unsigned long TimeToCompleteRequest;
      unsigned char RequestOperation;
} AUX_PERF_GC_SUCCESS;
```

ClientID (2 bytes): Client identification number.

ServerID (2 bytes): Server identification number.

SessionID (2 bytes): Session identification number.

TimeSinceRequest (2 bytes): Number of milliseconds since successful request occurred.

TimeToCompleteRequest (2 bytes): Number of milliseconds the successful request took to complete.

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

2.2.2.14 AUX_PERF_GC_SUCCESS_V2

```
typedef struct _AUX_PERF_GC_SUCCESS_V2 {
      unsigned short ProcessID;
      unsigned short ClientID;
      unsigned short ServerID;
      unsigned short SessionID;
      unsigned long TimeSinceRequest;
      unsigned long TimeToCompleteRequest;
      unsigned char RequestOperation;
} AUX_PERF_GC_SUCCESS_V2;
```

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 which was successful.

2.2.2.15 AUX_PERF_FAILURE

```
typedef struct _AUX_PERF_FAILURE {
      unsigned short ClientID;
      unsigned short ServerID;
      unsigned short SessionID;
      unsigned short RequestID;
      unsigned long TimeSinceRequest;
      unsigned long TimeToFailRequest;
      unsigned long ResultCode;
      unsigned char RequestOperation;
} AUX_PERF_FAILURE;
```

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.

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

ResultCode (4 bytes): Error code return of failed request.

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

2.2.2.16 AUX_PERF_FAILURE_V2

```
typedef struct _AUX_PERF_FAILURE_V2 {
      unsigned short ProcessID;
      unsigned short ClientID;
      unsigned short ServerID;
      unsigned short SessionID;
      unsigned short RequestID;
      unsigned long TimeSinceRequest;
      unsigned long TimeToFailRequest;
      unsigned long ResultCode;
      unsigned char RequestOperation;
} AUX_PERF_FAILURE_V2;
```

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.

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

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

ResultCode (4 bytes): Error code return of failed request.

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

2.2.2.17 AUX_CLIENT_CONTROL

```
typedef struct _AUX_CLIENT_CONTROL {
      unsigned long EnableFlags;
      unsigned long ExpiryTime;
} AUX_PERF_CONTROL;
```

EnableFlags (4 bytes): The following table describes the flags that instruct the client to either enable or disable behavior.

Flag	Value	Description
ENABLE_PERF_SENDTOSERVER	0x00000001	Client MUST start sending performance information to server.
ENABLE_PERF_SENDTOMAILBOX	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 normally 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_SENDGCDATA	0x00000010	Client MUST include performance data of the client communicating with the directory service.

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

2.2.2.18 AUX_OSVERSIONINFO

```
typedef struct _AUX_OSVERSIONINFO {
      unsigned long OSVersionInfoSize;
      unsigned long MajorVersion;
      unsigned long MinorVersion;
      unsigned long BuildNumber;
      unsigned long Reserved1;
      unsigned char Reserverd2[ 128 ];
      unsigned short ServicePackMajor;
      unsigned short ServicePackMinor;
      unsigned short Reserved3;
      unsigned short Reserved4;
      unsigned char Reserved5;
 } AUX_OSVERSIONINFO;
```

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

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

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

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

Reserved1 (4 bytes): Reserved. Content MUST be ignored by client.

Reserved2 (128 bytes): Reserved. Content MUST be ignored by client.

ServicePackMajor (2 bytes): Major version number of latest OS Service Pack installed on server.

ServicePackMinor (2 bytes): Minor version number of latest OS Service Pack installed on server.

Reserved3 (2 bytes): Reserved. Content MUST be ignored by client.

Reserved4 (2 bytes): Reserved. Content MUST be ignored by client.

Reserved5 (1 byte): Reserved. Content MUST be ignored by client.

2.2.2.19 AUX_EXORGINFO

```
typedef struct  _AUX_EXORGINFO {
      unsigned long OrgFlags;
} AUX_EXORGINFO;
```

OrgFlags (4 bytes): Flags indicating the server organizational information.

Flag	Value	Description
PUBLIC_FOLDERS_ENABLED	0x00000001	Organization has public folders.

3 Protocol Details

There are two RPC interfaces which comprise the Office-Exchange Protocol: 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. In order to establish a CXH, a call to EcDoConnectEx MUST be successful. The following table lists all method calls which require a valid CXH:

CXH Based Methods	Interface
EcDoDisconnect	EMSMDB
EcRRegisterPushNotification	EMSMDB
EcDoConnectEx	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. In order to establish an ACXH a call to EcDoAsyncConnectEx on the EMSMDB interface MUST be successful. The following table lists all method calls which 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 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. Once 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 Handle (ACXH) 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 lost, the Session Context and all Session Context information MUST be destroyed, all open resources MUST be closed, and all server objects 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 the server will allow clients to communicate over. This is done by calling RPC function RpcServerUseProtseqEp. See [MS-RPCE] for additional information about this function and protocol sequences. The supported protocol sequences are specified in section 2.1. Please note some protocol sequences use named endpoints which are also specified in section 2.1.

The server then MUST register the different authentication methods allowed on the EMSMDB interface. This is done by calling RPC function RpcServerRegisterAuthInfo. See [MS-RPCE] for more information about this function and authentication methods.

The server then MUST start listening for RPC calls by calling RPC function RpcServerListen. See [MS-RPCE] for more information about this function.

The server then MUST register the EMSMDB interface. This is done by calling RPC function RpcServerRegisterIfEx. See [MS-RPCE] for more information about this function.

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. See [MS-RPCE] for more information about these functions.

3.1.4 Message Processing Events and Sequencing Rules

This protocol MUST indicate to the RPC runtime that it is to perform a strict NDR data consistency check at target level 5.0, as specified in [MS-RPCE] Section 3.

In the table below, the term “Reserved” means that the client SHOULD NOT send the opnum.

This interface includes the following methods:

Method	Opnum	Description
Opnum0Reserved	0	Reserved.
EcDoDisconnect	1	Closes a Session Context with the server. The Session Context is destroyed and all associated server state, objects, and resources associated with the Session Context are released. The method requires an active Session Context Handle returned from EcDoConnectEx.
Opnum2Reserved	2	Reserved.
Opnum3Reserved	3	Reserved.
EcRRegisterPushNotification	4	Registers a callback address with the server for a Session Context. The callback address is used to notify the client of a pending event on the server. The method requires an active Session Context Handle returned from EcDoConnectEx.

Method	Opnum	Description
Opnum5Reserved	5	Reserved.
EcDummyRpc	6	This call does nothing. A client can use it to determine if it can communicate with the server.
Opnum7Reserved	7	Reserved.
Opnum8Reserved	8	Reserved.
Opnum9Reserved	9	Reserved.
EcDoConnectEx	10	Creates a Session Context Handle on the server to be used in subsequent calls to EcDoDisconnect, EcDoRpcExt2 and EcDoAsyncConnectEx.
EcDoRpcExt2	11	Passes generic remote operation (ROP) commands to the server for processing within a Session Context. The method requires an active Session Context Handle returned from EcDoConnectEx.
Opnum12Reserved	12	Reserved.
Opnum13Reserved	13	Reserved.
EcDoAsyncConnectEx	14	Binds a Session Context Handle returned in EcDoConnectEx to a new Asynchronous Context Handle which can be used in calls to EcDoAsyncWaitEx in interface AsyncEMSMDB. The method requires an active Session Context Handle returned from EcDoConnectEx.

3.1.4.1 Opnum0Reserved (opnum 0)

This method is reserved and SHOULD NOT be used.

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 associated with the Session

Context are released. This call requires an active Session Context Handle (CXH) returned from method EcDoConnectEx.

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

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

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

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

3.1.4.3 Opnum2Reserved (opnum 2)

This method is reserved and SHOULD NOT be used.

3.1.4.4 Opnum3Reserved (opnum 3)

This method is reserved and SHOULD NOT be used.

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 returned from method EcDoConnectEx.

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 should send a packet containing just the opaque context data to the callback address. The callback address specifies which network transport should be used to send the data packet.

See [MS-OXCNOTIF] for additional information on notification handling.

```
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 Session Context Handle which was created by calling EcDoConnectEx. The server uses the Session Context Handle to identify the Session Context to use for this call. On output the server MUST return the same Session Context Handle on success.

The server can destroy the Session Context Handle by returning a zero Session Context Handle. The server might wish to destroy the Session Context Handle for the following reasons:

1. The Session Context Handle passed in is invalid.
2. Attempting 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 used when sending a notification to the client.

cbContext: This parameter contains the size of the opaque client context data 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. See [sockaddr] informative reference for further information.

The server should support the address families AF_INET and AF_INET6 for a callback address corresponding to the protocol sequence types specified in section 2.1.

If an address family is requested which is not supported, the server MUST return error code ecInvalidParam. If the address family is supported, but the servers communications stack 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 following error codes are returned. Additional implementation specific error codes could be returned.

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

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

3.1.4.6 Opnum5Reserved (opnum 5)

This method is reserved and SHOULD NOT be used.

3.1.4.7 EcDummyRpc (opnum 6)

The method EcDummyRpc does nothing. 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 Opnum7Reserved (opnum 7)

This method is reserved and SHOULD NOT be used.

3.1.4.9 Opnum8Reserved (opnum 8)

This method is reserved and SHOULD NOT be used.

3.1.4.10 Opnum9Reserved (opnum 9)

This method is reserved and SHOULD NOT be used.

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 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 **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, range(0x0, 0x1008)] unsigned long *pcbAuxOut
);
```

hBinding: A valid RPC binding handle.

pcxh: On success the server MUST return a unique value to be used as a Session Context Handle (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.

ulFlags: For normal client calls this value MUST be 0x00000000.

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

ulConMod: The connection modulus is a client derived 32-bit hash value of the distinguished name 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 should be sent if Unicode format is not requested by the client on subsequent calls using this Session Context.

ulLcidString: The locale 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, this value will be 0xFFFFFFFF. To link to an existing Session Context, this value should be 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 found, the server MUST link the Session Context created by this call with the one found.

A server allows Session Context linking for the following reasons:

1. To consume a single CAL (Client Access License) for all the connections made from a single client machine. 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 machine. See RopPending in [MS-OXCNOTIF] for more information.

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

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

pcmsPollsMax: The server returns the number of milliseconds a client should wait 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 if any events are pending for the client. See [MS-OXNOTIF] for more information on Notifications.

pcRetry: The server returns the number of times a client should retry 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 SHOULD NOT be enforced by the server.

pcmsRetryDelay: The server returns the number of milliseconds a client should wait 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 should wait the number of milliseconds specified in this output parameter before retrying the call. The number of times a client should retry is returned in parameter pcRetry. This is a suggested delay for the client and SHOULD NOT be enforced by the server.

piCxr: The server returns a session index value 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 wishes 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. See [MS-OXCROPS] and [MS-OXCNOTIF] for more information on RopPending.

szDNPrefix: The server returns the server's distinguished name.

szDisplayName: The server returns the server's display name.

rgwClientVersion: The client passes the client protocol version the server should use 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 should use to determine what protocol functionality the server supports. For more information 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 SHOULD return the value passed in rgwClientVersion by the client. There is no client protocol version negotiation the server can do. 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 supported by the server.

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 ulIcxrLink and piCxr parameters for additional details). If the server supports Session Context linking, the server should verify 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 SHOULD NOT fail the EcDoConnectEx call, but simply not do linking.

On output, the server needs to return a time stamp in which the new Session Context was created. The server should save 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 including the RPC_HEADER_EXT header is contained in parameter cbAuxIn.

See section 3.1.7 for further information on how to access the embedded auxiliary payload buffer. See section 3.1.8 for further information on 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 error code ecRpcFormat if the request buffer is larger than 0x00001008 bytes in size.

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 further information on how to access the embedded auxiliary payload buffer. See section 3.1.8 for further information on 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 error code ecRpcFormat if this value is larger than 0x00001008.

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

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

Name	Value	Meaning
ecRpcAuthentication	0x000004B6	The szUserDN 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, hBinding, authentication is not set with RPC_C_AUTHN_LEVEL_PKT_PRIVACY. For more information about setting the authentication and authorization, see RpcBindingSetAuthInfoEx. The client should attempt the call again with new binding handle which is encrypted.
ecClientVerDisallowed	0x000004DF	The server requires encryption, but the client is not encrypted and the client does not support receiving error code ecNotEncrypted being returned by the server. See section 3.1.9 for which client versions do not support receiving error code ecNotEncrypted.
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.
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 protocol version is below the minimum required by the server.
ecCachedModeRequired	0x000004E1	The server requires the client to be running in cache mode. See section 3.1.9 for 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 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

Name	Value	Meaning
		their mailbox information through a different means (for example, webmail, POP, IMAP, and so on). See section 3.1.9 for 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, range(0x0, 0x40000)] unsigned long *pcbOut,
   [in, size_is(cbAuxIn)] unsigned char rgbAuxIn[],
   [in] unsigned long cbAuxIn,
   [out, length_is(*pcbAuxOut), size_is(*pcbAuxOut)] unsigned char
rgbAuxOut[],
   [in, out, range(0x0, 0x1008)] unsigned long *pcbAuxOut,
   [out] unsigned long *pulTransTime
);
```

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

The server can destroy the Session Context Handle by returning a zero Session Context Handle. The server might wish to destroy the Session Context Handle for the following reasons:

1. Determines the ROP request payload in the rgbIn buffer is malformed or length parameters are invalid.
2. The Session Context Handle passed in is invalid.
3. Attempting to access a mailbox that is in the process of being moved.

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

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

See section 3.1.7 for information on 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 Remote Operation (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 section 3.1.7 for further information on 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 on 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 0x00008008 bytes in size. The server MUST fail with error code ecRpcFormat if the request buffer is smaller than 0x00000008 bytes in size. See the [MS-OXCROPS] normative reference for further details.

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. See section 3.1.7 for further information on how to format the ROP response payload. The size of the ROP response payload plus the RPC_HEADER_EXT header is returned in pcbOut.

For more information on 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 0x00008008. The server MUST fail with error code ecRpcFormat if the value in pcbOut on input is larger than 0x00040000.

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 SHOULD return 0x00000000 on failure as there is no ROP response payload. The client SHOULD ignore 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 further information on how to access the embedded auxiliary payload buffer. See section 3.1.8 for further information on 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 error code ecRpcFormat if the request buffer is larger than 0x00001008 bytes in size.

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 further information on how to access the embedded auxiliary payload buffer. See section 3.1.8 for further information on 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 error code ecRpcFormat if this value is larger than 0x00001008.

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

pulTransTime: On output, the server should store 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 if time was spent transmitting the request/response on the network on processing time on the server.

Error Values: If the method succeeds, the return value is 0. If the method fails, the following error codes are returned. Additional implementation specific error codes could be returned.

Name	Value	Meaning
ecRpcFormat	0x000004B6	The format of the request was found to be invalid. This is a generic error which means something about the request whether it be a length or 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 Opnum12Reserved (opnum 12)

This method is reserved and SHOULD NOT be used.

3.1.4.14 Opnum13Reserved (opnum 13)

This method is reserved and SHOULD NOT be used.

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) which can be used in calls to EcDoAsyncWaitEx in interface AsyncEMSMDB. This call requires an active Session Context Handle (CXH) returned from method EcDoConnectEx.

This method is part of Notification handling. See [MS-OXCNOTIF] for additional information on the topic of Notifications.

```
long __stdcall EcDoAsyncConnectEx(

    [in] CXH cxh,

    [out, ref] ACXH * pacxh

);
```

cxh: Client MUST pass a valid CXH which 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 Asynchronous Context Handle which is associated with the Session Context passed in parameter cxh. This Asynchronous Context Handle can be used to make a call to EcDoAsyncWaitEx on interface AsyncEMSMDB.

Error Values: If the method succeeds, the return value is 0. If the method fails, the following error codes are returned. Additional implementation specific error codes could be returned.

Name	Value	Meaning
ecRejected	0x000007EE	Server has asynchronous RPC notifications disabled. Client should either poll for notifications or call 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 which use an extended buffer mechanism where the payload is preceded by a header. The header contains flags which determine whether or not the payload has been compressed, obfuscated, or another extended buffer and payload exist after the current payload. A single payload MUST NOT exceed 32 KB in size.

An extended buffer is used in 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 definition of [RPC_HEADER_EXT] for further information on the structure and individual fields.

The client or server MAY choose not to compress the payload if the payload is small. The client or server MAY choose to not obfuscate the payload if the payload has already been compressed. The client or server MAY 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 below 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 additional information on 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 additional information on 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 more information on 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 for additional information on 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 additional information on 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 information on 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 SHOULD honor the client request. The client MUST NOT assume a response will compressed or obfuscated if requested and should have 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 additional information on 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 additional information on 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 additional information on 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 contain 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 additional information on 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 additional information on 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 additional information on 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 additional information on 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 additional information on 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 more information on 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 additional information on 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 additional information on 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 information on 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 Office Exchange 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: Indicates the number of bytes from the coding position backward. A window of size W contains the last W processed bytes.

Pointer: Points to 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 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 may 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 Office Exchange 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 recommended to not attempt further compressing 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

The Office Exchange 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'0000000000011', 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 10, which is included in the high-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 high-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 high-order nibble is extracted and used to compute length for this metadata instance. The low-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 high-order nibble used for this length and the low-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-order nibble of the shared byte)
25	b'111' (three bits in the original two bytes of metadata) + b'1111' (in the high-order nibble of the shared byte) + b'00000000' (in the next byte)
26	b'111' (three bits in the original two bytes of metadata) + b'1111' (in the high-order nibble 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-order nibble 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-order nibble 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-order nibble 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 wishes to have secure communications with the server it MUST use RPC-level packet encryption.

The algorithm used to obscure data is straight-forward and simple. Every byte of the data to be obfuscated should be XORed 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 which support “packing” are RopQueryRows, RopReadStream, and RopFastTransferSourceGetBuffer. See [MS-OXCROPS] for more information 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.

In order for the server to produce additional response data, the server 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 producing 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 KBfor RopQueryRows. The server MUST NOT place more than 96 individual payloads into a single rgbOut response buffer.

When adding 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 SHOULD request the server return “as much” data as possible. See [MS-OXCROPS] for additional information on 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 which 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 message 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 MAY 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 more information on the AUX_HEADER and how it's 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 information on 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 SHOULD NOT throw an error if there is an auxiliary block it does not identify. This will allow for future expansion to the auxiliary blocks without impacting 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 (2.2.2.6)	Sent to the server as diagnostic information about the client for more robust reporting of networking issues. 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.
AUX_PERF_PROCESSINFO (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 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.
AUX_PERF_SESSIONINFO (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. If writing a client, it is recommended that AUX_PERF_SESSIONINFO_V2 be used instead. A server SHOULD still support this older session information auxiliary block. This block can also be passed in the EcDoRpcExt2 auxiliary input buffer.
AUX_PERF_SESSIONINFO_V2 (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. 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 (2.2.2.2).

Sent by client to server in EcDoRpcExt2:

Block	Description
AUX_PERF_SESSIONINFO (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. If writing a client it is recommended that AUX_PERF_SESSIONINFO_V2 be used instead. A server SHOULD still support this older session information auxiliary block. This block can also be passed in the EcDoConnectEx auxiliary input buffer.
AUX_PERF_SESSIONINFO_V2 (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. This block can also be passed in the EcDoConnectEx auxiliary input buffer.
AUX_PERF_SERVERINFO (2.2.2.7)	Sent to the server as diagnostic information about the server in which 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 (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_REQUESTINFO block sent to the server. The RequestID is also used in other performance blocks to identify which request to associate the performance data. 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 (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 (2.2.2.10)	Sent to the server as diagnostic information to report a previously success call to the Active Directory. 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 (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 SHOULD still support this older session information auxiliary block.
AUX_PERF_MDB_SUCCESS_V2 (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	Sent to the server as diagnostic information to report a

(2.2.2.13)	previously success call to the Active Directory. 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 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 SHOULD still support this older session information auxiliary block.
AUX_PERF_GC_SUCCESS_V2 (2.2.2.14)	Sent to the server as diagnostic information to report a previously success 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 (2.2.2.15)	Sent to the server as diagnostic information to report a previously FAILED call to the messaging server or Active Directory. 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 SHOULD still support this older session information auxiliary block.
AUX_PERF_FAILURE_V2 (2.2.2.16)	Sent to the server as diagnostic information to report a previously FAILED call to the messaging server or Active Directory. 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 (2.2.2.2).

Sent by server to client in EcDoConnectEx:

Block	Description
AUX_CLIENT_CONTROL (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 additional information on what client behavior the server can adjust. The client SHOULD alter its behavior based on this request.
AUX_OSVERSIONINFO (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 OS version and OS service pack information.
AUX_EXORGINFO (2.2.2.19)	Sent to the client as informational data to help the client decided 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 attempt to open a Public Store if the server informs the client that they're not present or disabled. If this block is not returned to the client, the client SHOULD assume 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 (2.2.2.2).

Sent by server to client in EcDoRpcExt2:

Block	Description
AUX_CLIENT_CONTROL (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 additional information on what client behavior the server can adjust. The client SHOULD alter 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 How to Compares Version Numbers

On the wire, client and server versions numbers are passed as three WORD values. See section 3.1.4.11 for additional information about EcDoConnectEx. 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 build of Exchange 2007 with Service Pack 1 applied. 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 which 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 convers 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 following table.

Server version	Description
6.0.6755.0	The server supports passing the sentinel value 0xBABE in the BufferSize field of a RopFastTransferSourceGetBuffer request. See [MS-OXCROPS]
8.0.295.0	The server supports passing the sentinel value 0xBABE in the ByteCount field of a RopReadStream request. See [MS-OXCROPS]
8.0.324.0	The server supports the flag CLI_WITH_PER_MDB_FIX in the OpenFlags field of a RopLogon request. See [MS-OXCROPS] and [MS-OXCSTOR]
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 corresponding to that support. A server cannot mix and match functionality. In order to support functionality at one server version level, it 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 following 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. See [MS-OXCROPS]
11.0.0.4920	The client supports receiving ecServerBusy in the ReturnValue field of the RopFastTransferSourceGetBuffer response. The client also assumes that the BackoffTime field will be present when the ReturnValue is ecServerBusy. If ReturnValue is not ecServerBusy, the BackoffTime field is not present. See [MS-OXCROPS] and [MS-OXCFXICS]
12.0.0.0	The client supports receiving the errors ecCachedModeRequired, ecRpcHttpDisallowed, and ecProtocolDisabled on the EcDoConnectEx call; otherwise, the client will get back ecClientVerDisallowed instead.
12.0.3118.0	The client supports receiving an AUX_EXORGINFO block in the rgbAuxOut buffer on the EcDoConnectEx call.
12.0.3619.0	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 attempting to connect on a non-encrypted connection.
12.0.3730.0	The client supports send optimization for Incremental Change Synchronization using PidTagTargetEntryId. See [MS-OXCFXICS] for more details.
12.0.4207.0	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 more information on extended buffer "packing".

Client version	Description
12.0.4228.0	The client supports receiving RopBackoff in the ROP response buffer of the EcDoRpcExt2 call. See [MS-OXCROPS]

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 corresponding to that support. A client cannot mix and match functionality. In order 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 which require a CXH context handle.

3.2.2 Timers

No protocol timers are required beyond those internal ones used in RPC to implement resiliency to network outages. For more information, see RPC Protocols Extensions [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. See information detailed in section 3overview for method dependencies.

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. Below is a list of parameters which the client should have specific handling:

hBinding: A valid RPC binding handle which MUST have a server name, protocol sequence and authentication method defined. Some protocol sequences have named endpoints which MUST be used. See section 2.1 for more information on how to create a binding handle.

pcxh: On success, this field will contain the Session Context Handle (CXH). The CXH MUST be stored on the client and used in subsequent calls on the EMSMDB interface which 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 SHOULD ensure that for a particular distinguished name passed in field szUserDN, the hash value should always be 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 created by this call with an existing Session Context on the server created from a previous call to EcDoConnectEx.

A client may want to link two Session Contexts for the following reasons:

1. To consume a single CAL (Client Access License) for all the connections made from a single client machine. 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 machine. See RopPending in [MS-OXCNOTIF] for more information.

If a client does not wish 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 SHOULD only pass 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 value 0x01.

pcmsPollsMax: On success, this value is the number of milliseconds the client SHOULD wait before polling the server for notification information. There are other more dynamic options available to the client for receiving notifications from the server. See [MS-OXCNOTIF] for more information on working with Notifications. The client should save this value and associated it with the Session Context Handle (CXH).

pcRetry: On success, this value is the number of times the client SHOULD retry a subsequent EMSMDB method call which uses the Session Context Handle (CXH) returned in field pcxh. See section 3.2.4.3 for more information on retrying RPC calls. This value should be save and associated with the Session Context Handle (CXH).

pcmsRetryDelay: On success, this value is the number of milliseconds a client should wait before retrying a subsequent EMSMDB method call which uses the Session Context Handle (CXH) returned in field pcxh. See section 3.2.4.3 for more information on retrying RPC calls. This value should be saved and associated with the Session Context Handle (CXH).

piCxr: On success, this value is a 16-bit session index which can be used in conjunction with the value returned in pulTimeStamp to link two Session Contexts on the server. See field ulIcxrLink for more information on how to link Session Contexts and the reason why a client might want to.

This value should be saved and associated with the Session Context Handle (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] and [MS-OXCNOTIF] for more information on RopPending.

rgwClientVersion: The client MUST pass the version number of the highest client protocol version it supports. This value will information the server of what protocol functionality the client supports. For more information 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 which the client SHOULD use to determine what protocol functionality the server supports. For more information about how version numbers are interpreted from the wire data and the expected server behavior, see section 3.1.9. This value should be saved and associated with the Session Context Handle (CXH).

pulTimeStamp: If a client wishes to link the Session Context 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 wish to link Session Contexts, the client SHOULD pass value 0x00000000.

On success, this value is the Session Context creation time stamp. The server should save the Session Context creation time stamp and associate it with the Session Context Handle (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. Below is a list of parameters for which the client SHOULD have specific handling:

pcxh: The client MUST pass a valid Session Context Handle (CXH) which was created by calling EcDoConnectEx. On output, the server might have prematurely closed the client's session by clearing the Session Context Handle 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 which require a valid Session Context Handle (CXH) SHOULD be retried if the call fails with RPC status RPC_S_SERVER_TOO_BUSY. The number of times the client should retry and the amount of time the client should wait before retrying is based on fields pcRetry and pcmsRetryDelay returned on EcDoConnectEx. Method EcDoConnectEx is the only method which creates a CXH, so it is a prerequisite for any method which 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 SHOULD clean up any saved session state information and close the Session Context Handle (CXH) if not already set to zero. The sessions binding handle SHOULD also be closed.

A client might chose to reconnect with the server automatically by creating a new binding handle and calling EcDoConnectEx. This will create a new Session Context on the server. It should be noted that all Server Objects previously opened on the server will no longer exist and the client MUST issue Rop commands if the client wishes 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 additional information on Session Context and Session Context Handles (CXH).

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 Session Context Handle 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 ACXH context handle must map to the Session Context associated with a CXH context handle. There SHOULD only be one ACXH context handle for a Session Context.

All methods on the AsyncEMSMDB interface which use an ACXH context handle must be performed against the Session Context associated with the ACXH context handle.

The server MUST keep a mapping between the ACXH context handle and an active Session Context on the server. Session contexts can be created and destroyed through the EMSMDB interface.

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

3.3.2 Timers

None.

3.3.3 Initialization

The server first MUST register the different protocol sequences the server will allow clients to communicate over. This is done by calling RPC function RpcServerUseProtseqEp. See [MS-RPCE] for additional information about this function and protocol sequences. The supported protocol sequences are specified in section 2.1. Please note some protocol sequences use named endpoints which are also specified in section 2.1.

The server then MUST register the different authentication methods allowed on the AsyncEMSMDB interface. This is done by calling RPC function RpcServerRegisterAuthInfo. See [MS-RPCE] for more information about this function and authentication methods.

The server then MUST start listening for RPC calls by calling RPC function RpcServerListen. See [MS-RPCE] for more information about this function.

The server then MUST start register the AsyncEMSMDB interface. This is done by calling RPC function RpcServerRegisterIfEx. See [MS-RPCE] for more information 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 [MS-RPCE] for more information 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 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 returned from EcDoAsyncConnectEx on interface EMSMDB. The Asynchronous Context Handle is associated with the Session Context.

This method is part of Notification handling. See [MS-OXCNOTIF] for additional information on the top of notifications.

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

acxh: On input, the client MUST pass a valid Asynchronous Context Handle which was created by calling EcDoAsyncConnectEx on interface EMSMDB. The server uses the Asynchronous Context Handle 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 there are pending events for the client on the Session Context on the server. The client should call 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 the client store an asynchronous Context Handle (ACXH) and use it on subsequent interface calls which require an ACXH context handle.

3.4.2 Timers

No protocol timers are required beyond those internal ones used in RPC to implement resiliency to network outages. See RPC Protocols Extensions [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 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. See information detailed in section 3 overview for method dependencies.

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 the Office Exchange 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. Client should initialize CXH to zero.

   szUserDN: String containing the distinguished named of the user making the EcDoConnectEx call in a directory service. Value: "/o=Microsoft/ou=First Administrative Group/cn=Recipients/cn=johndoe".

   ulFlags: Value 0x00000000. Normal 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 directory service object containing distinguished name in field szUserDN. Server creates Session Context and assigns a Session Context Handle (using 0x00001234 for this example). Server returns the following output values:

   pcxh: Value at CXH pointer is 0x00001234. Please 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. Client should poll every 60 seconds.

   pcRetry: Value at unsigned long pointer is 0x00000006. Client should retry six times.

pcmsRetryDelay: Value at unsigned long pointer is 0x00001770. Client should wait 10 seconds between retries.

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=Microsoft/ou=First Administrative Group/cn=Configuration/cn=Servers/cn=MBX-SRV-02". Client must free RPC-allocated memory.

szDisplayName: Value at unsigned char pointer is a pointer to a null-terminated ANSI string with value "MBX-SRV-02". Client must free RPC-allocated memory.

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

rgwBestVersion: Value at unsigned short array contains values: 0x000C, 0x183E and 0x03E8. Server SHOULD mimic rgwClientVersion if client version supported.

pulTimeStamp: Value at unsigned long pointer is a 32-bit value representing 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. See steps 1 through 3 of section 4.1 and ensure that it has a valid Session Context Handle.

2. Client sends ROP commands to server by calling EcDoRpcExt2 using the Session Context Handle 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 more information on 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 Session Context Handle 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. Server MUST return same value as on input unless session termination is requested in which case it would return 0x00000000.

pulFlags: Value at unsigned long is 0x00000000.

rgbOut: Server returns the following extended buffer and payload contain 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 establish a Session Context with the server. See steps 1 through 3 of section 4.1 and ensure that it has a valid Session Context Handle.

2. Client sends ROP commands to server by calling EcDoRpcExt2 using the Session Context Handle 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 more information on 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 Session Context Handle 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. 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.

<table>
<tr><td rowspan="3">RPC_HEADER_EXT
Flags: 0x0000
Size: 0x7FFE</td><td colspan="3">Payload</td><td rowspan="3">RPC_HEADER_EXT
Flags: 0x0004
Size: 0x2008</td><td colspan="3">Payload</td></tr>
<tr><td colspan="3">ROP Response Commands</td><td colspan="3">ROP Response Command</td></tr>
<tr><td>RopSize
0x7FEE</td><td>Rops
. . .</td><td>SOHT
16 bytes</td><td>RopSize
0x1FF8</td><td>Rop
. . .</td><td>SOHT
16 bytes</td></tr>
</table>

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 establish a Session Context with the server and has a valid Session Context Handle. 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 Session Context Handle returned from the EcDoConnectEx call.

   pcxh: Pointer to CXH value which is 0x00001234.

3. Server processes EcDoDisconnect request. Server verifies that Session Context Handle is for a valid Session Context for this user. Server destroys Session Context and invalidates Session Context Handle. 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, anonymous access to the server SHOULD NOT be granted. Only properly authenticated RPC binding handles should be allowed to make method calls on the EMSMDB and AsyncEMSMDB interfaces.

Most of the EMSMDB and AsyncEMSMDB interface methods require a Session Context Handle which can only be created from a successful call to EcDoConnectEx. The server MUST verify 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 named passed in the szUserDN field. The server should verify that the authentication context on the RPC binding handle has ownership permissions to the directory service object associated with the distinguished name. If the authentication context does not have adequate permissions, the server MUST fail the call and not create a Session Context.

Although the protocol allows for data compression and data obfuscation on method call EcDoRpcExt2, this SHOULD 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 below, 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] Sections 2.2.4 and 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;
```

```
 [ 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, range(0x0, 0x1008)] unsigned long *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, range(0x0, 0x40000)] unsigned long *pcbOut,
[in, size_is(cbAuxIn)] unsigned char rgbAuxIn[],
[in] unsigned long cbAuxIn,
[out, length_is(*pcbAuxOut), size_is(*pcbAuxOut)] unsigned char
rgbAuxOut[],
[in, out, range(0x0, 0x1008)] unsigned long *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

```
typedef [context_handle_noserialize] ACXH;

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

7 Appendix B: Office/Exchange Behavior

The information in this specification is applicable to the following versions of Office/Exchange:

- Office 2003
- Exchange 2003
- Office 2003 with Service Pack 1 applied
- Exchange 2003 with Service Pack 1 applied
- Office 2003 with Service Pack 2 applied
- Exchange 2003 with Service Pack 2 applied
- Office 2007
- Exchange 2007
- Office 2007 with Service Pack 1 applied
- Exchange 2007 with Service Pack 1 applied

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Office/Exchange behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies Office/Exchange does not follow the prescription.

7.1 Protocol Sequences

7.1.1 Exchange Server Support

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

Exchange 2007 allows only the following RPC protocol sequences: ncalrpc, ncacn_ip_tcp and ncacn_http.

7.1.2 Office Client Support

Office 2003 uses only the following RPC protocol sequences: ncacn_ip_tcp and ncacn_http.

Office 2007 uses only the following RPC protocol sequences: ncacn_ip_tcp and ncacn_http.

7.2 Authentication Methods

The following table lists the authentication methods supported by Exchange 2003 and Exchange 2007. A client MUST authenticate using one of these authentication methods.

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

Below is a table indicating which RPC methods are supported in which versions of Exchange server.

EMSMDB Interface:

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

AsyncEMSMDB Interface:

Method	Exchange 2003	Exchange 2003 SP1	Exchange 2003 SP2	Exchange 2007	Exchange 2007 SP1
EcDoAsyncWaitEx				✓	✓

7.3.2 Office Client Support

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

7.3.2.1 Accessing Exchange 2003

Below is a table indicating which RPC methods are used by an Officeclient when accessing an Exchange 2003 server.

EMSMDB Interface:

Method	Office 2003	Office 2003 SP1	Office 2003 SP2	Office 2007	Office 2007 SP1
EcDoDisconnect	✓	✓	✓	✓	✓
EcRRegisterPushNotification	✓	✓	✓	✓	✓

EcDummyRpc					
EcDoConnectEx	✓	✓	✓	✓	✓
EcDoRpcExt2	✓	✓	✓	✓	✓
EcDoAsyncConnectEx					

AsyncEMSMDB Interface:

Method	Office 2003	Office 2003 SP1	Office 2003 SP2	Office 2007	Office 2007 SP1
EcDoAsyncWaitEx					

7.3.2.2 Accessing Exchange 2007

Below is a table indicating which RPC methods are used by an Office client when accessing an Exchange 2007 server.

EMSMDB Interface:

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

AsyncEMSMDB Interface:

Method	Office 2003	Office 2003 SP1	Office 2003 SP2	Office 2007	Office 2007 SP1

EcDoAsyncWaitEx				✓	✓

7.4 Client Access Licenses

As of Exchange 2007 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 Index