[MS-PSOM]:

PSOM Shared Object Messaging Protocol

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation for protocols, file formats, languages, standards as well as overviews of the interaction among each of these technologies.

- **Copyrights**. This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the technologies described in the Open Specifications and may distribute portions of it in your implementations using these technologies or your documentation as necessary to properly document the implementation. You may also distribute in your implementation, with or without modification, any schema, IDL's, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications.

- **No Trade Secrets**. Microsoft does not claim any trade secret rights in this documentation.

- **Patents**. Microsoft has patents that may cover your implementations of the technologies described in the Open Specifications. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. However, a given Open Specification may be covered by Microsoft Open Specification Promise or the Community Promise. If you would prefer a written license, or if the technologies described in the Open Specifications are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.

- **Trademarks**. The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights. For a list of Microsoft trademarks, visit www.microsoft.com/trademarks.

- **Fictitious Names**. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

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

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

Revision Summary

Date	Revision History	Revision Class	Comments
3/31/2010	0.1	Major	Initial Availability
4/30/2010	0.2	Editorial	Revised and edited the technical content
6/7/2010	0.3	Editorial	Revised and edited the technical content
6/29/2010	0.4	Editorial	Changed language and formatting in the technical content.
7/23/2010	0.4	No Change	No changes to the meaning, language, or formatting of the technical content.
9/27/2010	1.0	Major	Significantly changed the technical content.
11/15/2010	1.0	No Change	No changes to the meaning, language, or formatting of the technical content.
12/17/2010	1.0	No Change	No changes to the meaning, language, or formatting of the technical content.
3/18/2011	1.0	No Change	No changes to the meaning, language, or formatting of the technical content.
6/10/2011	1.0	No Change	No changes to the meaning, language, or formatting of the technical content.
1/20/2012	2.0	Major	Significantly changed the technical content.
4/11/2012	2.0	No Change	No changes to the meaning, language, or formatting of the technical content.
7/16/2012	2.0	No Change	No changes to the meaning, language, or formatting of the technical content.
10/8/2012	3.0	Major	Significantly changed the technical content.
2/11/2013	3.0	No Change	No changes to the meaning, language, or formatting of the technical content.
7/30/2013	3.1	Minor	Clarified the meaning of the technical content.
11/18/2013	3.1	No Change	No changes to the meaning, language, or formatting of the technical content.
2/10/2014	3.1	No Change	No changes to the meaning, language, or formatting of the technical content.
4/30/2014	3.2	Minor	Clarified the meaning of the technical content.
7/31/2014	3.2	No Change	No changes to the meaning, language, or formatting of the technical content.
10/30/2014	3.2	No Change	No changes to the meaning, language, or formatting of the technical content.
9/4/2015	3.2	No Change	No changes to the meaning, language, or formatting of the technical content.

Table of Contents

1   Introduction

The PSOM Shared Object Messaging Protocol is used to exchange messages between the client and server. A message typically represents a method invocation of a remote object, with a sequence of understood parameters.

This protocol can be divided into three areas:

▪ **Connection:** Establish and negotiate interfaces between a client and a server.

▪ **Distributed object primitives:** Detail the format in which messages are sent and received.

▪ **Application-specific calls:** Explain the sequence of messages required to perform an operation.

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

1.1   Glossary

The following terms are specific to this document:

Advanced Encryption Standard (AES): A block cipher that supersedes the Data Encryption Standard (DES). AES can be used to protect electronic data. The AES algorithm can be used to encrypt (encipher) and decrypt (decipher) information. **Encryption** converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext. AES is used in symmetric-key cryptography, meaning that the same key is used for the **encryption** and decryption operations. It is also a block cipher, meaning that it operates on fixed-size blocks of plaintext and ciphertext, and requires the size of the plaintext as well as the ciphertext to be an exact multiple of this block size. AES is also known as the Rijndael symmetric encryption algorithm [FIPS197].

ASCII: The American Standard Code for Information Interchange (ASCII) is an 8-bit character-encoding scheme based on the English alphabet. ASCII codes represent text in computers, communications equipment, and other devices that work with text. ASCII refers to a single 8-bit ASCII character or an array of 8-bit ASCII characters with the high bit of each character set to zero.

authentication: The act of proving an identity to a server while providing key material that binds the identity to subsequent communications.

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

certificate: When referring to X.509v3 certificates, that information consists of a public key, a distinguished name (DN) (3) of some entity assumed to have control over the private key corresponding to the public key in the certificate, and some number of other attributes and extensions assumed to relate to the entity thus referenced. Other forms of certificates can bind other pieces of information.

client: (1) A computer on which the remote procedure call (RPC) client is executing.

(2) In DFS-R, a replicating machine acts as a client when it receives replicated files from its upstream partner. Use of the terminology **client** stipulates that the machine contact its upstream server, and is responsible for initiating communication related to receiving replicated files. It does not imply anything about the operating system version or the function of the machine.

cookie: A small data file that is stored on a user's computer and carries state information between participating protocol servers and protocol clients.

Coordinated Universal Time (UTC): A high-precision atomic time standard that approximately tracks Universal Time (UT). It is the basis for legal, civil time all over the Earth. Time zones around the world are expressed as positive and negative offsets from UTC. In this role, it is also referred to as Zulu time (Z) and Greenwich Mean Time (GMT). In these specifications, all references to UTC refer to the time at UTC-0 (or GMT).

decryption: In cryptography, the process of transforming encrypted information to its original clear text form.

distributed object: A collection of interfaces that enable a protocol client and a protocol **server** to exchange messages with each other, and to use those messages to connect or disconnect from distributed objects and to call remote methods that have a predefined set of parameters. Each instance of a distributed object has a unique identifier, which ensures that messages are routed to the correct object.

Dynamic Hypertext Markup Language (DHTML): An extension of the Hypertext Markup Language (HTML) that allows applications to change styles and attributes of page elements (objects) and to replace existing elements (objects) with new ones.

encryption: In cryptography, the process of obscuring information to make it unreadable without special knowledge.

fully qualified domain name (FQDN): An unambiguous domain name (2) that gives an absolute location in the Domain Name System's (DNS) hierarchy tree, as defined in [RFC1035] section 3.1 and [RFC2181] section 11.

hash: A fixed-size result that is obtained by applying a one-way mathematical function, which is sometimes referred to as a hash algorithm, to an arbitrary amount of data. If the input data changes, the hash also changes. The hash can be used in many operations, including **authentication** and digital signing.

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

Hypertext Transfer Protocol Secure (HTTPS): An extension of HTTP that securely encrypts and decrypts web page requests. In some older protocols, "Hypertext Transfer Protocol over Secure Sockets Layer" is still used (Secure Sockets Layer has been deprecated). For more information, see [SSL3] and [RFC5246].

keepalive message: A protocol message that is sent between a protocol client and a protocol server to help ensure that a connection is considered active by all endpoints (5). Inactive connections are considered idle and are likely to be closed by either endpoint (5) to conserve resources.

network byte order: The order in which the bytes of a multiple-byte number are transmitted on a network, most significant byte first (in **big-endian** storage). This may or may not match the order in which numbers are normally stored in memory for a particular processor.

notification: A process in which a subscribing Session Initiation Protocol (SIP) client is notified of the state of a subscribed resource by sending a NOTIFY message to the subscriber.

proxy: A computer, or the software that runs on it, that acts as a barrier between a network and the Internet by presenting only a single network address to external sites. By acting as a go-between that represents all internal computers, the proxy helps protects network identities while also providing access to the Internet.

PSOM channel: A packet, datagram, octet stream connection, or sequence of logical connections that exists between endpoints (5) that are not unique. The channel defines a unique identity for each endpoint (5) and helps secure communications between them. It uses a **root distributed object** to enable both logical connections between child **distributed objects** and the exchange of messages between peers. A single PSOM connection can contain multiple PSOM channels.

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

root distributed object: The top-level **distributed object** to which a protocol client or protocol **server** connects immediately after a channel is created. After a connection is established with a root distributed object, all other distributed objects on the same channel are connected.

server: A replicating machine that sends replicated files to a partner (client). The term "server" refers to the machine acting in response to requests from partners that want to receive replicated files.

SHA-1: An algorithm that generates a 160-bit hash value from an arbitrary amount of input data, as described in [RFC3174]. SHA-1 is used with the Digital Signature Algorithm (DSA) in the Digital Signature Standard (DSS), in addition to other algorithms and standards.

TCP/IP: A set of networking protocols that is widely used on the Internet and provides communications across interconnected networks of computers with diverse hardware architectures and various operating systems. It includes standards for how computers communicate and conventions for connecting networks and routing traffic.

token: A word in an item or a search query that translates into a meaningful word or number in written text. A token is the smallest textual unit that can be matched in a search query. Examples include "cat", "AB14", or "42".

Transmission Control Protocol (TCP): A protocol used with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. TCP handles keeping track of the individual units of data (called packets) that a message is divided into for efficient routing through the Internet.

Transport Layer Security (TLS): A security protocol that supports confidentiality and integrity of messages in client and server applications communicating over open networks. **TLS** supports server and, optionally, client authentication by using X.509 certificates (as specified in [X509]). **TLS** is standardized in the IETF TLS working group. See [RFC4346].

Uniform Resource Identifier (URI): A string that identifies a resource. The URI is an addressing mechanism defined in Internet Engineering Task Force (IETF) Uniform Resource Identifier (URI): Generic Syntax [RFC3986].

Uniform Resource Locator (URL): A string of characters in a standardized format that identifies a document or resource on the World Wide Web. The format is as specified in [RFC1738].

UTF-8: A byte-oriented standard for encoding Unicode characters, defined in the Unicode standard. Unless specified otherwise, this term refers to the UTF-8 encoding form specified in [UNICODE5.0.0/2007] section 3.9.

web server: A server computer that hosts websites and responds to requests from applications.

X.509: An ITU-T standard for public key infrastructure subsequently adapted by the IETF, as specified in [RFC3280].

XML fragment: Lines of text that adhere to XML tag rules, as described in [XML], but do not have a Document Type Definition (DTD) or schema, processing instructions, or any other header information.

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

1.2   References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the Errata.

1.2.1   Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information.

[ECMA-376] ECMA International, "Office Open XML File Formats", 1st Edition, ECMA-376, December 2006, http://www.ecma-international.org/publications/standards/Ecma-376.htm

[FIPS197] FIPS PUBS, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[IEEE754] IEEE, "IEEE Standard for Binary Floating-Point Arithmetic", IEEE 754-1985, October 1985, http://ieeexplore.ieee.org/servlet/opac?punumber=2355

[ISO/IEC29500:2011] ISO/IEC, "Information technology -- Document description and processing languages -- Office Open XML File Formats -- Parts 1-4", ISO/IEC 29500-1:2011, 2011, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=59575

[MS-CONFBAS] Microsoft Corporation, "Centralized Conference Control Protocol: Basic Architecture and Signaling".

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., et al., "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999, http://www.rfc-editor.org/rfc/rfc2616.txt

1.2.2   Informative References

None.

1.3   Overview

This protocol is designed to facilitate communications for data collaboration and Web conferencing applications. The two roles understood by this protocol are client and **server**. This protocol could be used symmetrically across both roles. However, a small number of messages and actions are performed by a specific role, but need to be understood by both. A server (2) can be connected to multiple clients at a time, but clients do not have a direct way to exchange messages, unless the

server (2) decides to communicate with them. A server (2) differentiates from different clients based on their transport connection.

This protocol provides a set of strongly-typed interfaces to be defined for both a server and client. These interfaces can be called remotely, as this protocol takes care of all parameter marshaling and method invocation. This protocol also provides a way to create an object hierarchy. These interfaces are allowed to connect and disconnect a child object, known as a **distributed object**. A distributed object is composed of the following parts:

▪ The client-side distributed object peer.

▪ Server-side distributed object peer components.

A concrete instantiation of a distributed object is allowed to send and receive messages to and from its peer when it is connected. There can be multiple instances of a distributed object at any time; it is analogous to a class definition and instance relationship. Each distributed object instance has a unique identifier that allows PSOM operations, as described in section 2.2.2, to be routed to them. This is known as a "**proxy** identifier".

A distributed object can exist without the distributed object that created it, and is allowed to exchange messages with its peer. Components on each individual client are allowed to communicate with each other, and all components on the server (2) can communicate amongst each other directly. This allows the creation of an object model where clients and the server (2) can exchange messages, and the server (2) can record the state of the conference and broadcast any changes to other connected clients via the distributed objects.

When a client and server connection is authenticated, distributed objects are logically connected so that they can exchange messages. Some distributed objects are connected when a user creates content; others exist in a given conference and have specific responsibilities. The connected distributed objects are allowed to exchange messages, which are defined by server and client interfaces for each distributed object. An example interface could be the existence of an **sBroadcast** message on the server interface and a **cReceiveBroadcast** on each client interface. When multiple clients connect to the server and logically connect the distributed object with this set of messages, a client could send an **sBroadcast** message to the server's distributed object peer. When the server receives this message with a payload of data to broadcast, it could then call **cReceiveBroadcast** with that data on all clients' distributed object for that given peer. This enables the construction of a distributed object model where clients and the server can exchange messages.

An interface is composed of a set of methods, which are also referred to as messages, and each method is allowed to have any number of parameters. The supported parameter types include:

▪ String.

▪ 16/32/64-bit integers.

▪ Floating point values, byte.

▪ References to other connected distributed objects.

▪ Arrays of basic parameter types and arrays of arrays.

For the full list of supported parameters, see section 2.2.

On the network, all byte orders are **big-endian**, unless otherwise indicated.

At a high level, messages sent as part of this protocol can be split into two parts:

▪ **Kernel:** Handles connection, versioning, **keepalive messages**, and preparation to connect the application distributed objects.

▪ **Application distributed object:** Allows Web conferencing components to exchange messages to perform user-related activities, such as sharing a document.

This abstraction extends into the way channels are used. A channel extended by this abstraction is a **PSOM channel**. PSOM channels are created by the client and have unique integer identifiers. PSOM channel zero, which is the initial PSOM channel, is created implicitly after connection. A PSOM channel provides a way for distributed objects to route messages to each other. Each PSOM channel has a **root distributed object** that provides a way to connect child distributed objects and exchange messages between peers. The following types of messages help route information:

▪ **Record messages**: Deal with PSOM channel management and routing of inside messages to the appropriate PSOM channel.

▪ **PSOM operation messages**: Enclosed in a record message. Contain the proxy identifier to route distributed object method invocation and connection operations.

The other PSOM channel used is PSOM channel 2, where all application distributed object messages are exchanged, as shown in the following diagram.

Kernel Distributed Object	Application Distributed
Root: ConnMgr	Objects
	Root: Meeting
Channel 0	**Channel 2**
Kernel	Application
Connection	
Network	
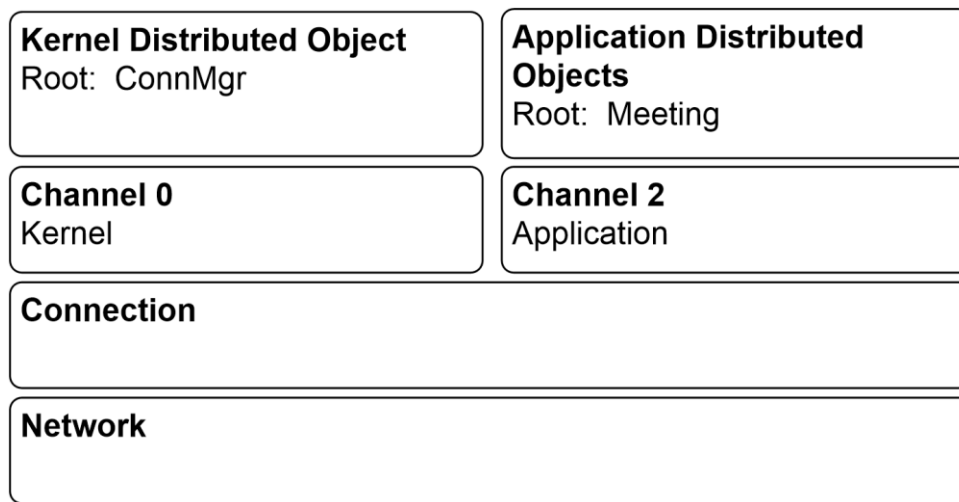
Figure 1: Overview of kernel and application channels

The protocol does not specify any explicitly-defined acknowledgement components. If an application-level distributed object chooses to add this functionality, it is outside the scope of the kernel and message protocol. Generally, low-level failures are indicated by a server (2) or client sending a special termination message and disconnecting. In some cases, either might disconnect in response to a protocol violation.

As previously stated, there can be multiple instances of a distributed object. The following figure shows a sample distributed object structure on PSOM channel 2. The root distributed object has only one instance, and two child objects of the same type. Each child can only exchange messages with the corresponding child peer.
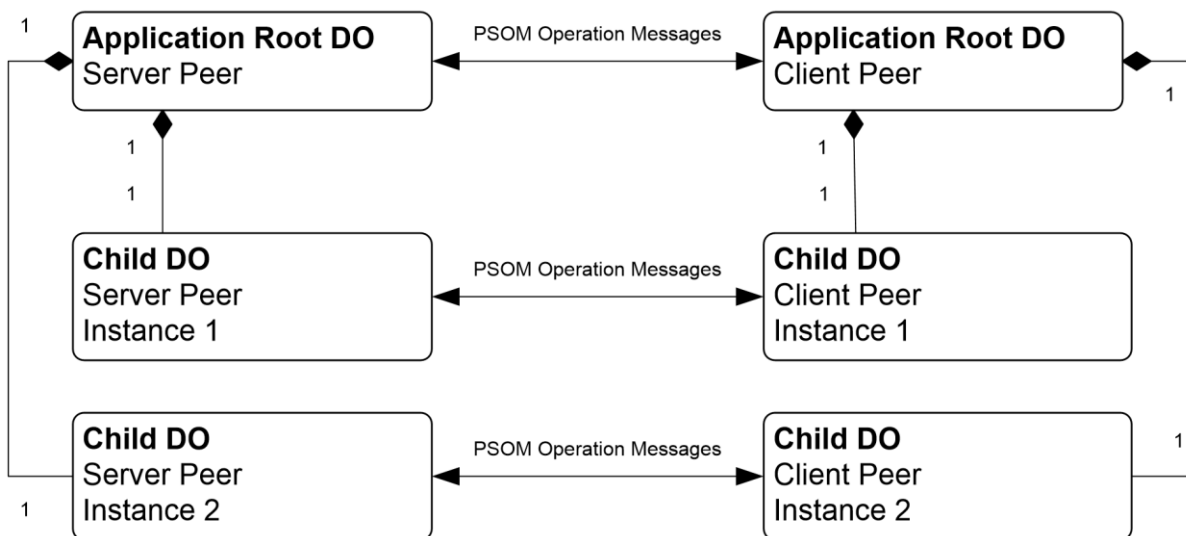
Figure 2: Sample Application distributed object connection

1.3.1 General Data Flow

Connections are initiated by a client. The transport, as long as it is agreed on by client and server components implicitly, can be any reliable transport. In general, **Transport Layer Security (TLS)** is the required transport. Once the connection is initiated and **encryption** negotiated, the client sends **authentication** information. If the server determines that the client is valid, it responds with an authentication acknowledgement; otherwise it terminates the connection.

When authentication is completed, a capability and versioning sequence occurs. In this sequence, the client sends **hash** codes and versions that are understood by both sides. These hash codes are the sum of the client interface and the server interface hash codes. When completed, the server does the same, except that it communicates with each distributed object. During negotiation both sides choose the highest compatible version and setup the methods and events corresponding to that. Any disconnects as a result of versioning conflicts occur when a connect **PSOM operation** message is received. If the server or client receives an unexpected hash code, such as when a different value is received than was expected, it sends a **break** or **close** message immediately and terminates the connection.

The definition of messages depends on the interface definition, which is summarized in a hash code that changes if the interface changes. This allows the client and server to ensure that they understand the order of bytes in a **PSOM operation** message. Therefore, when versioning is completed, the server and client can assume that the corresponding role has the same interface definitions.

The client then sends a record message to create PSOM channel 2 for the root distributed object. At that point, the root distributed object is responsible for connecting child distributed objects and sending any other messages that are needed for the conference session. When the connection is completed, the server and client distributed objects exchange messages based on the actions users take in a Web conference. When a user disconnects, **PSOM channel record** messages are exchanged to shut down the appropriate peers, and the client and server terminate the connection, as shown in the following diagram.
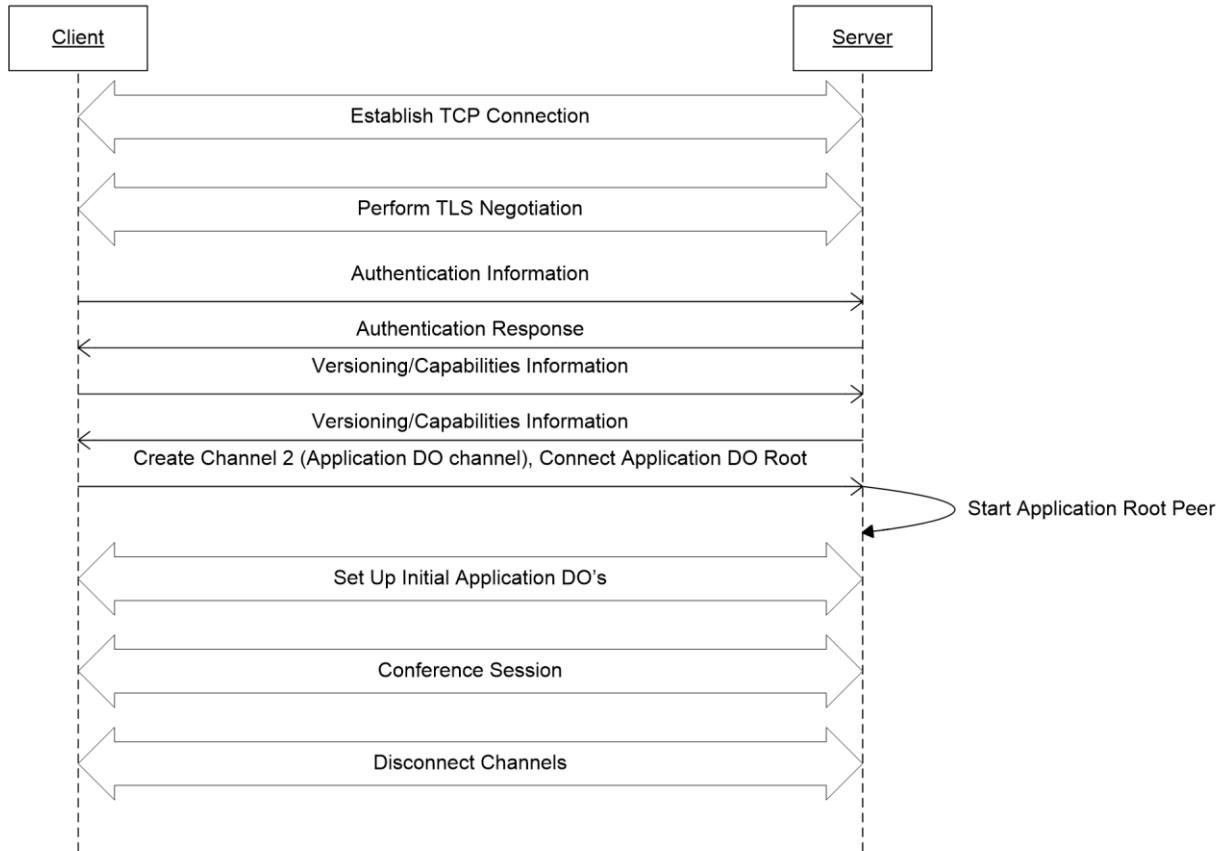
Figure 3: High–level protocol data flow

1.3.2 Message Flow

The understanding of a particular distributed object interface is critical in understanding both version negotiation and application communications. For details about distributed object interfaces, see section 3.1.4.1.2.

The message flow diagramed in this section has the following distributed objects defined:

▪ **ExampleDO**

▪ **ChildDO**

ChildDO is the child of **ExampleDO**. **ExampleDO** has a single method defined on the server interface, which is named **HookUpChild**. This method can be called by any client and takes no parameters. **ChildDO** has two methods. The server piece defines **ExampleServerMethod(string str)** and the client interface defines **ExampleClientMethod(string str)**. Both interfaces accept an argument of type string.

The sequence of calls is as follows:

1. **Client::ExampleDO** calls **Server::ExampleDO**'s **HookUpChild()** method. This sample method causes the server to hook up a child distributed object of type **ChildDO**.

2. Server creates an object of type **ChildDO** to handle messages for that distributed object instance.

3. Server sends a special protocol message asking the **Client::ExampleDO** parent to create and allow the **ChildDO** instance to send and receive messages.

4. The client creates an instance of **ChildDO**.

5. When the **ChildDO** sample is initialized, it could call a server-side method on its peer, **Server::ChildDO**. It does so, and passes a string to be marshaled to the server (2). This protocol passes this message on.

6. When the server-side instance of **ChildDO** receives the **ExampleServerMethod(string)** call, it could call back with another method call, if the implementer chooses to do so.
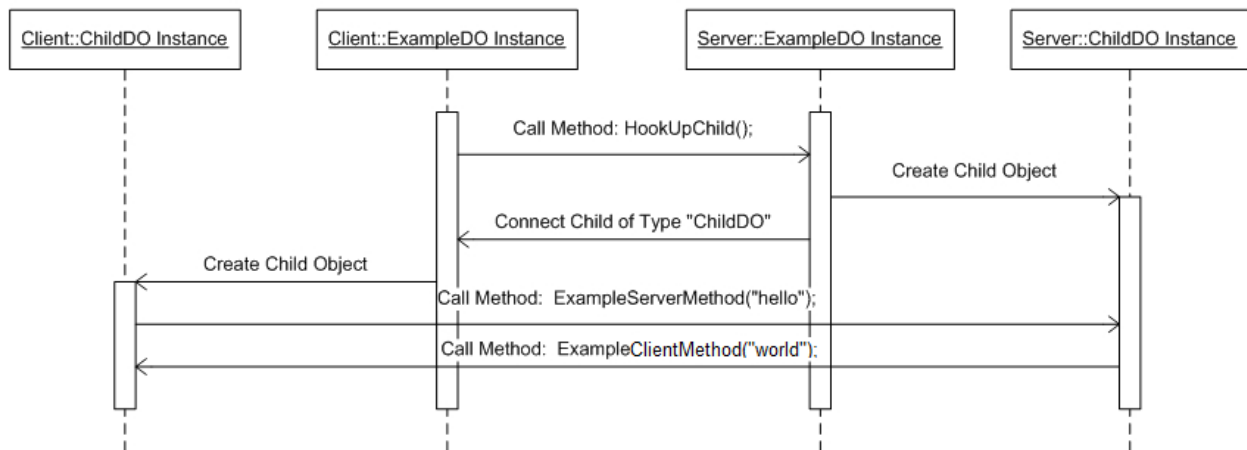
Figure 4: Example message flow between server and client distributed objects

1.3.3 Channels and Channel Distributed Object Roots

PSOM channels provide a way to divide kernel and application distributed objects easily. Every PSOM channel has a single root distributed object that is implicitly connected as soon as the PSOM channel is created. Messages exchanged by distributed object peers happen on the appropriate PSOM channel, as the peer and proxy identifiers used to route messages are unique on a per PSOM channel basis.

Only PSOM channels 0 and 2 are used by this protocol. Both are created by the client. PSOM channel zero is used for all kernel communications. This PSOM channel is mainly used for exchanging distributed object version information. PSOM channel 2 is used for all application-related messages. These messages allow the Web conferencing application to define and carry out a Web conference. Both PSOM channels can be connected at the same time, as they provide a way to multiplex messages on a single connection.

1.4 Relationship to Other Protocols

A few values and details used during protocol initialization depend on the [MS-CONFBAS] protocol. See sections 3.2.3 and 3.3.3 for details. However, there is no direct interaction with [MS-CONFBAS].

1.5 Prerequisites/Preconditions

This protocol is used on top of a reliable transport such as **TCP/IP** with TLS encryption, as the protocol assumes that messages are not dropped and that the integrity of each message is maintained. TLS is required to connect directly to the server.

A proxy server can be used to balance loads to the server. The only difference between the proxy and the server is that an extra authentication **token** is required upon connection. All other interactions are identical.

1.6 Applicability Statement

This protocol is used in Web conferencing scenarios, where a strongly-typed messaging infrastructure, with Web conferencing-related functionality, is required.

1.7 Versioning and Capability Negotiation

This protocol provides versioning and capability negotiation at the level of each distributed object. Exchange of interface versions and hashes happens before the application distributed objects are connected, so implementations can make decisions about what methods are supported for each distributed object. During this exchange the Server/**Client** must present all the versions of the distributed objects it supports to the other side. On completion of the exchange both sides validate and set the highest compatible versions as the prevalent ones for each distributed object. The Client(2) and Server then must honor the version and call only methods which are compatible with the negotiated version on a given distributed object.

This document covers versioning issues in the following areas:

- **Supported transports:** This protocol can be implemented on top of **Transmission Control Protocol (TCP)** with TLS encryption, as discussed in section 2.1.

- **Protocol versions**

- **Security and authentication methods:** This protocol is required to be used on top of TCP with TLS encryption.

- **Localization:** There is no localization-dependent protocol behavior. **Break** messages are to be treated as error codes, rather than strings to be shown to a user.

- **Capability negotiation:** See previous discussion in this section.

1.8 Vendor-Extensible Fields

None.

1.9 Standards Assignments

None.

2 Messages

2.1 Transport

This protocol MUST be used over a TLS channel. The port and address are configurable, and are passed in from an external source, as described in section 3.2.3.1.

2.2 Message Syntax

2.2.1 Records

A record is designed to encapsulate a **PSOM channel** message, as defined in section 2.2.2. At a high level, protocol messages contain a record, and optionally contain a **PSOM channel** message. Each record is encapsulated in a data structure that contains a message type and a body and has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
Record message type								Message body (variable)																							
...																															

Record message type (1 byte): The type of the record message.

Message body (variable): The message body length is determined both by type and embedded lengths for some types.

2.2.1.1 Record Types

The body lengths for all record types are unsigned, 32-bit, big-endian encoded integers.

2.2.1.1.1 Close Message

The **Close** message is represented by "0x00".

No data is contained in this message other than its type. This message closes the PSOM channel. This message MUST be sent during the disconnection phase, and results in all connected distributed objects transitioning into the disconnected state without sending messages. All other PSOM channels MUST be closed before the base PSOM channel, channel zero, is closed. This message has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
Close																															

Close (1 byte): Header; defined as "0x00".

2.2.1.1.2 SetChannel Message

The **SetChannel** message is represented by "0x04".

SetChannel contains a 32-bit message body and indicates that all future records SHOULD be directed to the set channel, which is communicated as the 32-bit message body. The body is represented in

network order, big-endian. For example, if the application channel is represented with a channel identifier of 2, this is represented as "0x00000002". This message has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
SetChannel								ChannelId																							
...																															

SetChannel (1 byte): Header for message type; defined as "0x04".

ChannelId (4 bytes): The representation of the channel identifier to be set. This PSOM channel is used for all following messages, with the exception of other **SetChannel** messages.

2.2.1.1.3 Break Message

The **Break** message is represented by "0x06".

A **Break** message results in the same actions as a **Close** message but indicates that an error happened. A **Break** message contains a reason string that is not a PSOM-encoded string or a string provided as an argument to a method. It is an **ASCII** encoding of an error reason code, which is an English representation of a reason that can be used for debugging purposes. The length of the string precedes the byte representation of the characters. This message has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
BreakChannel								ReasonLength																							
...								ReasonString (variable)																							
...																															

BreakChannel (1 byte): Header for the message type; defined as "0x06".

ReasonLength (4 bytes): The unsigned length of the following reason string.

ReasonString (variable): The ASCII byte representation of the reason code string. Each byte represents a single character.

For example, a **Break** message with the reason string "bye" is represented as follows:

```
0x06; 0x00000003; 0x62; 0x79; 0x65;
```

2.2.1.1.4 RpcMessage Message

The **RpcMessage** message is represented by "0x16".

This message represents a message to a PSOM channel for a PSOM operation, as described in the following section. The message body length varies, and depends on the contents of the message. The first 4 bytes following the message type MUST be the big-endian encoding of the body length in bytes. The body of the message is described in section 2.2.2. This message has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
RpcMessage								BodyLength																							
...								Body (variable)																							
...																															

RpcMessage (1 byte): Header for the message type; defined as "0x16".

BodyLength (4 bytes): The unsigned length of the following body.

Body (variable): The payload of the PSOM operation message. This message is described in section 2.2.2.

2.2.1.1.5 RPCOpen Message

The **RPCOpen** message is represented by "0x37".

This is a **PSOM channel open** message and instructs the implementation to begin accepting messages for the specified PSOM channel. The record body MUST include the PSOM channel identifier, which is an unsigned 32-bit integer, and a special **ConnMgr** distributed object. **RpcMessage** MUST follow this identifier. For more information about the special **RpcMessage**, see section 2.2.2.1. This message MUST only be used to create the root distributed object PSOM channel, PSOM channel 2, and not the connection PSOM channel, PSOM channel zero. This message has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
RpcOpen								ChannelId																							
...								RpcMsgBodyLen																							
...								RpcMessageBody																							
...																															

RpcOpen header (1 byte): The header to denote the message type.

ChannelId (4 bytes): The unsigned integer value of the PSOM channel to open.

RpcMessageBodyLen (4 bytes): The unsigned integer value of the length of the message body.

RpcMessageBody (variable): The contents of the message, intended to be treated as an **RpcMessage** body for a distributed object proxy object on the previous PSOM channel.

2.2.2 PSOM Operation Channel Messages (RpcMessage)

PSOM channel messages represent a PSOM operation, which is an action to take or method to call that involves a particular distributed object. PSOM channel messages are exchanged between the two peers of a distributed object, client and server, and represent one of the following operations:

- Connect child.

- Disconnect child.

- Call a method.

These operations are described in additional detail in section 2.2.2.1 and section 2.2.2.2.

The term "RPC message" refers to a PSOM operation that requests a call of a given method on a distributed object interface.

Each PSOM operation is sent on a given PSOM channel. Any PSOM message sent after a **SetChannel** record is on that PSOM channel. The length of these messages on the network can be determined by the length header in the record frame.

Each connected distributed object instance is assigned a proxy identifier, which is a signed 32-bit integer, but represented as a **GenericInt** on the network. When a distributed object is first connected, it MUST be assigned a unique identifier that is understood by the server and client.

When a distributed object is connected, the peer that is connecting the distributed object MUST assign any outgoing connect message a monotonically increasing integer, starting at 1, while the receiving distributed object MUST assign it a monotonically decreasing value starting at -1. When a server (2) sends a connect message for a particular distributed object, that peer MUST be assigned a proxy identifier of +n, where n is any positive integer greater than zero. The receiving distributed object MUST assign the value −n to the peer for its **ProxyId**. When a server or client receives a connect request, it MUST assign the negative value of the **ProxyId**. For example, the first child distributed object of the **Meeting** distributed object, which is the PSOM channel root distributed object, which is connected by the server, receives a **ProxyId** of -1 on the client side, and a value of 1 on the server side. Therefore, both the client and server can uniquely refer to this instance when sending PSOM operation messages.

PSOM operation messages are routed to an existing distributed object peer with a valid **ProxyId**. The receiving peer is the one whose **ProxyId** is the received value, negated. Therefore, any time a **ProxyId** is read from the network, it MUST be immediately negated, causing a negative to become positive and a positive to become negative, to determine which peer instance receives the message.

The root distributed object has a **ProxyId** of zero.

In general, a PSOM operation frame is formatted as follows.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
OpenClose								ProxyId (variable)																							
...																															
Payload (variable)																															
...																															

OpenClose (1 byte): Either "0x84" (open) or "0x86" (close).

ProxyId (variable): Signed integer encoded as a **GenericInt**.

Payload (variable): The body for an RPC message, as defined in section 2.2.2.2. The length can be determined by the length of the current record body. **Payload** MUST be present if the **OpenClose** byte header is present.

2.2.2.1 Connect/Disconnect Child

These operations are symmetric, and provide the ability for a parent distributed object to connect and disconnect child distributed objects. Child objects are useful from a design perspective, and help segment functionality within objects. The only distributed object connected implicitly is the root distributed object of a PSOM channel, which is either the **Meeting** distributed object or the **ConnMgr** distributed object, depending on the PSOM channel. Any other distributed objects MUST be explicitly connected. Distributed object connect and disconnect messages can be sent by either the client or server.

Note that proxy identifiers for connecting and identifying distributed objects are determined by negating any incoming value. When any given distributed object is connected, it has the value "+n", where n is greater than zero, on the initiating side, while the proxy has the value "−n" on the receiver's side. Therefore, to parse an incoming message, the incoming value MUST be negated and the resulting negated value is looked up in a local table with **proxyId** to distributed object mappings.

2.2.2.1.1 Connect (OP_CONNECT)

All connect requests include a **PartName** that is used to identify which child to connect. This part name is understood explicitly by both the server and client components. The part name can be any string that is understood by both the client and server distributed object peers. The string communicates what object to connect that is eligible to receive messages. When an **OP_CONNECT** message is sent, the peer MUST assume the corresponding peer is eligible to receive messages. Failure does not have to be indicated but, if it is, it can be done through disconnection or an RPC message to another distributed object by the application. This message is formatted as follows.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
OP_CONNECT									ParentProxyId (variable)																								
...																																
PartName (variable)																																
...																																
Hash (variable)																																
...																																

OP_CONNECT (1 byte): The constant header for a **Connect** operation within the record frame. The value MUST be set to 0x84.

ParentProxyId (variable): A 32-bit signed integer, encoded as a **GenericInt**. Represents the parent of the child distributed object to be connected.

PartName (variable): A PSOM-encoded string that details the name of the child to connect.

Hash (variable): A 64-bit signed integer encoded as a **GenericInt**. This represents the hash of the initiating peer's child distributed object. For example, if the client is sending the **OP_CONNECT** message, it includes the hash of the **ClientInterface** portion of the distributed object. If the server is initiating the connection, it includes the **ServerInterface** hash for this parameter. The incoming hash MUST match the distributed object that is to be connected.

2.2.2.1.2 Disconnect (OP_CLOSE)

Disconnecting a distributed object means that a distributed object peer can no longer receive **PSOM operation** messages. A special header is sent, followed by the **GenericInt** encoding of the proxy identifier to disconnect. Any child objects of a connected distributed object can still receive messages, and are therefore still connected, unless they are disconnected through disconnect **PSOM operation** messages sent to them. A disconnect message can be sent as a response to a received disconnect for a given proxy as an acknowledgment. This message is formatted as follows.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
OP_CLOSE								ProxyId (variable)																							
...																															

OP_CLOSE (1 byte): The constant header for a **Close** operation within the record frame. The value MUST be set to 0x86.

ProxyId (variable): A 32-bit signed integer, encoded as a **GenericInt**. Represents the distributed object peer to be closed.

2.2.2.2 RPC Message (Call Method) (OP_DATA)

If the first byte is not "0x84" or "0x86", **OP_CONNECT** or **OP_CLOSE**, the PSOM operation refers to an RPC message. Because of the structure of the **GenericInt** encoding algorithm, "0x84" or "0x86" are never the first byte in a **GenericInt**, so this is a safe assumption.

The RPC message type indicates the message is meant to call a given interface method with a pre-defined set of parameters. The structure of an RPC message is as follows.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
ProxyId (variable)																															
...																															
MethodIndex								Payload (variable)																							
...																															

ProxyId (variable): The sending proxy's identifier. Encoded as a **GenericInt**.

MethodIndex (1 byte): A signed byte indicating the method to call. See sections 3.1.4.1.1 and 3.1.4.1.2.1 for more details.

Payload (variable): The individual parameters required to call the method that corresponds to **MethodIndex**. If no parameters are expected, this MUST be omitted.

The ordering of **Payload** MUST match the ordering of parameters in the distributed object definition. For example, if a method is declared as follows:

```
Method(Int32 param1, String param2)
```

The encoded bytes in the parameters section are, in order:

param1 (variable): Encoded as a **GenericInt**.

param2 (variable): Encoded as a PSOM string type. This is a 2-byte length header followed by a specially encoded payload of byte representations of characters.

RPC messages do not require any response. The receiver MUST invert the sign of **ProxyId** to find the appropriate distributed object peer instance to handle the message.

3   Protocol Details

3.1   Common Details

In general, the server and client roles are symmetric. However, there are some actions that a server cannot take. For more information about these restricted actions, see section 3.3.

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.

This protocol is a highly symmetric protocol at the connection/channel/RPC layer. The differences between the server and client exist at a higher layer, where a set of message interfaces are defined. At the RPC layer, there is a set of objects. Each object in the set is a distributed object. Each distributed object has a server (2) and client component. Each component is able to exchange messages, disconnect from the other, or connect to another distributed object.

Each RPC message is a method on each server (2) and client interface. This method never returns a value, but it does take parameters. These parameters are listed in the following section.

3.1.1.1 PSOM types

This section contains a list of all of the supported PSOM types, which are used as part of the record message **RpcMessage** and distributed object method parameters. When some types, such as **Int32**, are listed in the following sections, those types only apply to the parameters in the interface definitions.

3.1.1.1.1 Arrays

Arrays are not a type. They modify other primitive PSOM types to create collections. Each array is encoded as the number of elements as a **GenericInt**, followed by each element in sequence. Because all elements are of the same type, the length of the array can be determined based on the length of each type. It is possible to create an array of arrays. This is represented by treating each contained array as its own type. For example, an array of 2 arrays of type **byte** with 1 element each would look as follows:

```
[GenericInt(2)] [GenericInt(1)] [value of byte in array 0] [GenericInt(1)] [value of byte in
array 1]
```

3.1.1.1.2 Boolean

Encoded as "0x01" for true; "0x00" for false.

3.1.1.1.3 Byte

An unsigned byte. This type is encoded exactly as represented over the network.

3.1.1.1.4 DistributedObject Reference

A distributed object reference is a reference to a distributed object peer. If the object is null, it is encoded as a single byte, "0x8C". To refer to a non-null distributed object, this is encoded as a **GenericInt** containing the value of a proxy identifier.

3.1.1.1.5 GenericInt

GenericInt is a specially formatted value used to represent any signed integer that is 64 bytes or less. The encoding algorithm results in a variable size based on the original value. For algorithm details, see section 6. Note that interface details can reference **Int32** and **Int64**. These SHOULD be encoded as **GenericInt**.

A **GenericInt** is never a directly referenced type; instead, it is an encoding for **Int32**, **Int64**, **UInt32**, **UInt64** types. They are all encoded the same way, but the given integer types determine the appropriate bounds and sign for the number.

3.1.1.1.6 Int32

A 32-bit signed integer encoded as a **GenericInt**.

3.1.1.1.7 Int64

A 64-bit signed integer encoded as a **GenericInt**.

3.1.1.1.8 String

String represents a specially-encoded string of a **UTF-8** string.

For algorithm details, see section 6.

3.1.1.1.9 Double

A double-precision (64-bit) floating point number, big-endian encoded with the bytes in the sequence specified in [IEEE754] section 3.

3.1.2 Timers

There is a **ping** method called with no parameters used as a keepalive message sent every 30 seconds by both the client and the server on the **ConnMgr** distributed object. For the **ConnMgr** definitions, see section 3.1.3.1. When the timer expires, a new one is set at the same interval period.

3.1.3 Initialization

When this protocol is initialized, it MUST begin connecting to a server or client or wait for incoming connections from a server. The protocol has no state at this point, and no distributed objects are connected. At a high level, connection is established, and is followed by authentication, versioning, and then the root distributed object connection. For more information about these steps, see section 3.2 and section 3.3.

3.1.3.1 ConnMgr Distributed Object

The **ConnMgr** distributed object is defined in its corresponding client and server sections. This distributed object is responsible for exchanging versioning information about supported distributed objects between the client and server. At this point, there are no additional versioning requirements because there is only one version.

3.1.4 Higher-Layer Triggered Events

The higher-layer objects consist of a set of distributed objects that represent a conference. The root distributed object is the **Meeting** distributed object. It is the first distributed object connected on that PSOM channel. The following sections describe how distributed objects are connected, and all possible methods on each interface. Unexpected calls, dictated by business logic, can result in other distributed object calls in the reverse direction, or result in a **Break** or **Close** message being sent.

3.1.4.1 Distributed Objects

All interface parameters are encoded as PSOM arguments as specified in section 3.1.1.1.

3.1.4.1.1 Distributed Object Interface Definition

The understanding of particular distributed object interfaces is critical in understanding both version negotiation and application communications. This section describes a way of representing client and server distributed object interfaces.

```
[Name="ProtocolName", Version=1]
DOInterface DOName
{
    [Hash=1234567890123456L]
    ServerInterface
    {
        SampleServerMethod1(Int64 parameter1, Int32 parameter2);
    }

    [Hash=-6543210987654321L]
    ClientInterface
    {
        SampleClientMethod1();
    }

    Children
    {
        (DOChildName1, DOChildType1)
        (DOChildName2, DOChildType2)
    }
}
```

Attributes are denoted by square brackets ([or]) and are intended to add information to the object that follows them. Each **DOInterface** entry has a **ServerInterface**, **ClientInterface**, and **Children** as sub-components. For detail about the PSOM operation methods that can be called by the opposite peer, see section 3.2.4.1 and section 3.3.4.1. The ordering is important. The first method in each interface type is assigned a method index of 1, and each subsequent method declaration is given a monotonically increasing integer value. This value is used as a reference in certain **PSOM operation** messages.

In some cases, the child objects might be represented differently. For example, the name might be used to indicate alternative usage. If this is the case, **DOChildNameN** is referenced appropriately in the text that describes the child elements. In general, **DOChildNameN** refers to a specific single child that has the distributed object type **DOChildType**. **DOChildNameN** is the text of the string sent as part of the **OP_CONNECT** message.

3.1.4.1.1.1   DOInterface Attributes

Name (string): The string representation of the protocol name. Used to identify which **DOInterface** is referenced.

Version (integer): An integer value greater than zero. See section 3.1.4.1.3 for details.

3.1.4.1.1.2 Server/Client Interface Attributes

Hash (64-bit signed integer): A pre-defined value used to ensure interfaces are the same. This is provided for future extensions to use in testing for conflicts.

3.1.4.1.1.3 Method Declarations

Each method declaration MUST be in the following format:

```
MethodName(ParameterType1 parameterName1, …);
```

Method ordering is important, as specified in section 3.1.4.1.1, because it reflects the byte identifier used on the wire. The **ParameterType** MUST be a PSOM supported type, as defined in section 3.1.1.1. For example, "Int32" represents a signed, 32-bit integer that is encoded as a **GenericInt** on the wire.

Note that because messages do not have return values, the previous method definitions do not specify them.

3.1.4.1.1.4 Children

Children is not exclusive. Other names can be sent without declaring it. For example, to connect to a specific instance of an object, a connection message can be sent with a custom string such as "objectInstance123". This attribute details child objects for the reader. The previous interface definition includes a distributed object with two well-defined child objects with a given name and type. Because distributed object connection routines allow any string to be sent on the wire, the Children section will detail any variants that can be sent as part of the **OP_CONNECT** message.

3.1.4.1.2 Sample Distributed Object

This section illustrates a sample distributed object to help the reader understand the format for these distributed objects. Server and client method descriptions can be found in section 3.3 and section 3.2.

3.1.4.1.2.1 Interface

The following is a sample interface definition for a distributed object.

```
[Name="ProtocolName", Version=1]
DOInterface DOName
{
    [Hash=1234567890123456]
    ServerInterface
    {
        sSampleServerMethod1(Int64 parameter1, Int32 parameter2);
    }

    [Hash=-6543210987654321]
    ClientInterface
    {
        cSampleClientMethod1();
    }

    Children
    {
        (DOChildName1, DOChildType1)
        (DOChildName2, DOChildType2)
    }
}
```

The bracketed fields indicate the names that are used as part of the name in the **Connect** messages, and the hashes are exchanged during connection and versioning. The order of the messages indicates their base **MethodIndex**, which is defined by the alphabetic order, compared ordinally, relative to other methods on that interface.

Children is defined as the set of distributed objects that the **DOName** distributed object is able to send **RPC connect** messages for. The directionality of those connect messages is limited to one way, either server to client or client to server, but that is defined further for each distributed object.

3.1.4.1.2.2   Sample Server Method

<< The description of a method goes here. The server is able to receive a message with this name >>

parameter1 (Int64): << Description of parameter >>

parameter2 (Int32): << Description of parameter >>

3.1.4.1.2.3   Sample Client Method

See section 3.1.4.1.2.2. The only difference between a client and a server method is that the former contains descriptions for a message that the client, rather than the server, can receive.

3.1.4.1.2.4   Children

Anything that comes in a **Connect** message is detailed under **Children**. Because the **OP_CONNECT** message is sent with a string for the name of the distributed object or a string value to connect, each server or client portion of the distributed object MUST be aware of the distributed objects it is allowed to connect. The Children section will describe the direction for this **OP_CONNECT** message. Either the client or the server can send an **OP_DISCONNECT** message to its peer at any time.

3.1.4.1.3 Versioning

Each distributed object has a version on the client and the server (2). Some implementations MAY send multiple versions via the **addProtocol** message; only the version that matches the values in the **DOInterface** attributes SHOULD be respected, and other values MUST be ignored.

3.1.4.1.4 ContentManager

ContentManager is a distributed object that maintains a collection of **Content** distributed objects that are currently present in the **Meeting**. It provides methods for deleting **Content** distributed objects. It notifies the client of **Content** distributed objects that have been added through **UploadManager** or removed. It also provides support for reserving the title before adding it to ensure uniqueness of all content titles within a **Meeting**. Additionally, it allows a user to become the active presenter and to give up the active presenter baton, and gives **notifications** when the active presenter changes.

The following table lists the **TitleReservationStatus** values. For more information, see section 3.3.4.1.3.1.

Value	Numeric value	Description
ReservedForCreation	1	The requested title was reserved successfully.
ReservedForUpgrade	2	The requested title was reserved for upgrade successfully.
FailedReservedForCreation	3	The requested title could not be reserved because it is already reserved by someone else for creation.

Value	Numeric value	Description
FailedReservedForUpgrade	4	The requested title could not be reserved because it is already reserved by someone else for upgrade.
FailedExternalIdLockedForCreate	5	The requested external identifier could not be reserved because it is already reserved by someone else for creation.
FailedExternalIdLockedForUpgrade	6	The requested external identifier could not be reserved because it is already reserved by someone else for upgrade.
FailedReservationMaxExceeded	7	The user exceeded the maximum number of allowed reservations.<1>
FailedCookieInUse	8	The supplied **cookie** is already in use for an existing reservation.
FailedNotAuthorized	9	The user does not have the proper credentials to complete the operation.
FailedInvalidExtension	10	The extension of the provided title is not allowed for upload.
FailedInvalidTitle	11	The provided title was not in a valid format. Reasons for this error include that the character length is longer than a chosen value, or contains characters that are not supported. This is used for the server to enforce well-named restrictions on the title if it is used as a file name.

3.1.4.1.4.1   Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.ContentManager", Version=2]
DOInterface ContentManager
{
    [Hash=3800622354142801969]
    ServerInterface
    {
        void sDeleteContent(Int64 contentId);

        void sPresent();

        void sReleaseTitle(Int32 cookie);

        void sReserveTitle(String title, Int32 cookie);

        void sReserveTitle(String title, Int32 cookie, String externalId);

        void sStopPresenting();
    }

    [Hash=-8255121175073997388]
    ClientInterface
    {
        void cContentAdded(Int64 contentId, String type);

        void cContentCreated(Int64 contentId, Int32 cookie);

        void cContentCreationFailed(Int32 cookie, Int32 reason);

        void cContentRemoved(Int64 contentId);

        void cReserveTitleCompleted(Int32 /* TitleReservationStatus */ status, Int32 cookie,
Int64 contentId, Int64 owningUserId);
```

```
            void cSetActiveContent(Int64 activeContentId);

            void cSetActivePresenter(Int64 activePresenterId);
            void cTitleReleased(Int32 cookie);

        }

        Children
        {
            (UploadManager, Microsoft.Rtc.Server.DataMCU.Meeting.UploadManager)
            ("content.X", Microsoft.Rtc.Server.DataMCU.Meeting.Content)
        }
    }
```

3.1.4.1.4.2   Children

The **UploadManager** distributed object is connected by the server and is a required distributed object.

The **Content** distributed objects are hooked up by the client by sending an **rpcConnect**. The part name used for the **rpcConnect** is a string concatenation of the string "content." and the **contentId** in base-10 format, with no leading zeros. As an example, if a client is attempting to hook up the content with the **contentId** of "2", the string passed in **rpcConnect** is "content.2".

3.1.4.1.5 Content

Content is a distributed object that handles generic content methods, such as presenting a piece of content, renaming a piece of content, setting content's visibility in the **Meeting**, or being told properties about the content. The content-specific methods and properties, such as the number of slides in a PowerPoint presentation, or which slide to navigate to, is handled by the **extendedContent** child object.

The following table lists the **SetTitleStatus** values and descriptions.

Value	Numeric value	Description
Success	0	The operation completed successfully.
FailedTitleExists	1	Another piece of content has this title.
FailedReservedForCreation	2	The requested title is currently reserved.
UserNotAuthorized	3	The user is not allowed to modify the title.
FailedInvalidExtension	4	The extension is not on the approved list.
FailedInvalidTitle	5	The provided title was not in a valid format.

The following table lists the **ContentVisibility** values and descriptions.

Value	Numeric value	Description
MeetingOrganizer	0	Only the meeting organizer can see this content.
Presenters	1	Only presenters can see this content.
Everyone	2	Everyone can see this content.

3.1.4.1.5.1 Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.Content", Version=1]
DOInterface Content
{
    [Hash=-6470662138903778586]
    ServerInterface
    {
        void sForceSync ();

        void sMakeHighestPresentationOrder ();

        void sPresent ();

        void sSetTitle (string title);

        void sSetVisibility (Int32 /* ContentVisibility */ visibility);

        void sStopPresenting ();
    }

    [Hash=1113513223610002283]
    ClientInterface
    {
        void cConnectCompleted ();

        void cForceSync ();

        void cSetCreationTime (string creationTime);

        void cSetLastUsedTime (string lastUsedTime);

        void cSetNativeFileInfo (string fileName, byte[] key, byte[] iv, byte[]
hash, Int64 fileSize);

        void cSetOwnerId (Int64 id);

        void cSetPresentInfo (Boolean presented, Int64 presenterId);

        void cSetPresentationOrder (Int64 presentationOrder);

        void cSetTitle(string title);

        void cSetTitleComplete(Int32 /* SetTitleStatus */ status, string title);

        void cSetVisibility (Int32 /* ContentVisibility */ visibility);
    }

    Children
    {
        ("extendedContent", content-specific-DO)
    }
}
```

The following is the interface definition of the **Content** distributed object for version 10<u><2></u>.

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.Content", Version=10]
DOInterface Content
```

```
{
    [Hash=-2530343413165516885]
    ServerInterface
    {
        void sForceSync ();

        void sMakeHighestPresentationOrder ();

        void sPresent ();

        void sSetTitle (string title);

        void sSetVisibility (Int32 /* ContentVisibility */ visibility);

        void sStopPresenting ();
    }

    [Hash=974079596268293062]
    ClientInterface
    {
        void cConnectCompleted ();

        void cForceSync ();

        void cSetCreationTime (string creationTime);

        void cSetLastUsedTime (string lastUsedTime);

        void cSetNativeFileInfo (string fileName, byte[] key, byte[] iv, byte[]
hash, Int64 fileSize);

        void cSetOwnerId (Int64 id);

        void cSetPresentInfo (Boolean presented, Int64 presenterId);

        void cSetPresentationOrder (Int64 presentationOrder);

        void cSetTitle(string title);

        void cSetTitleComplete(Int32 /* SetTitleStatus */ status, string title);

        void cSetVisibility (Int32 /* ContentVisibility */ visibility);

        void cSetOriginalFileUrl (string originalFileUrl);

        void cSetViewingUrl (string viewingUrl);

        void cSetRecordingUrl (string recordingUrl);

    }

    Children
    {
        ("extendedContent", content-specific-DO)
    }
}
```

3.1.4.1.5.2   Children

The **Content** distributed object has one client-hooked-up child distributed object, called the **extendedContent** distributed object. This is the implementation of the content's specific features, such as PowerPoint. To connect the **extendedContent** distributed object, the client sends an **rpcConnect** with the string "extendedContent".

3.1.4.1.6 Meeting

Meeting is the root distributed object for all the application distributed objects. It consists of the **ContentManager**, which keeps track of all content in the **Meeting**, and the **ContentUserManager**, which keeps track of all relevant users in a **Meeting**. It also communicates the **Uniform Resource Locator (URL)** base for the **Meeting**.

3.1.4.1.6.1   Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.Meeting", Version=2]
DOInterface Meeting
{
    [Hash=7811924786664530844]
    ServerInterface
    {
        void sSetInfo(String info);
    }

    [Hash=2106930589629680263]
    ClientInterface
    {
        void cMeetingReady();

        void cSetInfo(String info);

        void cSetServerTime(String serverTime);

        void cSetUrlBase(String urlBase);
    }

    Children
    {
        (ContentManager, Microsoft.Rtc.Server.DataMCU.Meeting.ContentManager)
        (ContentUserManager, Microsoft.Rtc.Server.DataMCU.Meeting.ContentUserManager)
    }
}
```

3.1.4.1.6.2   Children

The **ContentManager** and **ContentUserManager** MUST be connected as **child** distributed objects of the **Meeting** before **cMeetingReady** is called.

3.1.4.1.7 ContentUserManager

The **ContentUserManager** is a one-way notification mechanism to tell each client which **Uniform Resource Identifier (URI)** and display name map to a given user identifier. This way, the URI and display name can be sent once, and each content and annotation can just use an identifier to indicate ownership or that an action happened.

3.1.4.1.7.1   Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.ContentUserManager", Version=1]
DOInterface ContentUserManager
{
    [Hash=5320330165687787020]
    ServerInterface
    {
```

```
        }

        [Hash=5320330165687787020]
        ClientInterface
        {
            void cUsersAdded(Int64[] ids, String[] uris, String[] displayNames);

            void cUsersRemoved(Int64[] ids);
        }
    }
```

3.1.4.1.8 UploadManager

The **UploadManager** distributed object is responsible for uploading OC Package files to the server.
See section 3.2.4.1.4 for details related to the OC Package. The OC Package is a single file for the
purposes of uploading. The upload traffic is through the PSOM channel, rather than **Hypertext
Transfer Protocol (HTTP)**, for the download.

3.1.4.1.8.1   Interface

The following is the interface definition of the **UploadManager** distributed object for version 1.

```
    [Name="Microsoft.Rtc.Server.DataMCU.Meeting.UploadManager", Version=1]
    DOInterface UploadManager
    {
        [Hash=4004400404121921234]
        ServerInterface
        {
            void sRequestUpload(Int64 fileLength, Int32 cookie, String manifestXml);

            void sRequestUpload(Int64 packedLength, Int64 unpackedLength, Int32 cookie);

            void sUploadFinished(Int32 cookie, Boolean cancel);

        }

        [Hash=-8511879503649873756]
        ClientInterface
        {
            void cAcceptUpload(Int32 cookie, DistributedObject stream);

            void cRejectUpload(Int32 cookie, Int32 reason);

            void cSetAvailableSpace(Int64 size);

            void cUploadFinished(Int32 cookie, Int32 reason);

        }

        Children
        {
            (UploadStreams, UploadStream)
        }
    }
```

The following is the interface definition of the **UploadManager** distributed object for version 2<3>.

```
    [Name="Microsoft.Rtc.Server.DataMCU.Meeting.UploadManager", Version=2]
    DOInterface UploadManager
    {
        [Hash=2601804729028999169]
        ServerInterface
```

```
    {
        void sRequestUpload(Int64 fileLength, Int32 cookie, String manifestXml);

        void sRequestUpload(Int64 packedLength, Int64 unpackedLength, Int32 cookie);

        void sUploadFinished(Int32 cookie, Boolean cancel);

        void sRequestWebUploadBlob(Int32 cookie, String manifestXml);

        void sRequestWebDownloadBlob(Int32 cookie, String obfuscatedName, Boolean
isAttachment);

    }

    [Hash=1079790469891676391]
    ClientInterface
    {
        void cAcceptUpload(Int32 cookie, DistributedObject stream);

        void cRejectUpload(Int32 cookie, Int32 reason);

        void cSetAvailableSpace(Int64 size);

        void cUploadFinished(Int32 cookie, Int32 reason);

        void cAcceptWebUpload(Int32 cookie, String uploadUrl);

        void cAcceptWebDownload(Int32 cookie, String downloadUrl);

        void cRejectWebDownload(Int32 cookie, Int32 reason);

    }


    Children
    {
        (UploadStreams, UploadStream)
    }
}
```

3.1.4.1.8.2   Children

UploadManager has one **Child**. The **Child** MUST be connected by the server sending a **Connect** message to the client that requested an upload to start.

3.1.4.1.8.2.1 UploadStreams

UploadStreams is the **UploadStream** distributed object that is currently uploading data to the server.

3.1.4.1.9 UploadStream

The **UploadStream** distributed object is responsible for uploading content package files to the server. Note that the names **UploadStream** and **IRCStream** are used interchangeably.

3.1.4.1.9.1   Interface

```
Name="Microsoft.Rtc.Server.DataMCU.Meeting.Parts.IRCStream", Version=1]
DOInterface UploadStream
{
    [Hash=-6716385024907738156]
    ServerInterface
    {
```

```
            void sDisconnect();

            void sWrite(byte[] data, Int32 packetNum);
        }

        [Hash=5963839780483567246]
        ClientInterface
        {
            void cDisconnect();

            void cWriteComplete(Int32 nBytes);
        }
    }
```

3.1.4.1.10   NativeFileOnlyContent

The **NativeFileOnlyContent** distributed object is a content distributed object that is used to share a file with meeting attendees. A presenter uploads a file and creates a **NativeFileOnlyContent** distributed object, and the attendees can choose to download the file. The **NativeFileOnlyContent** distributed object is a wrapper of a basic content type, and does not require other specific content operations.

There is a one-to-one mapping between **NativeFileOnlyContent** and **Content**. If **Content**'s type is **NativeFileOnly**, a **NativeFileOnlyContent** instance, or proxy, MUST be created.

3.1.4.1.10.1 Interface

```
    [Name="Microsoft.Rtc.Server.DataMCU.Meeting.NativeFileOnlyContent", Version=1]
    DOInterface NativeFileOnlyContent
    {
        [Hash=6421877628186475469]
        ServerInterface
        {
            /* No methods exist */
        }

        [Hash=5585496037459248534]
        ClientInterface
        {
            void cConnectCompleted();
        }
    }
```

3.1.4.1.11   PptContent

The **PptContent** distributed object represents an instance of PowerPoint content shared in the session. This distributed object allows the server to signal content resource location, content source encryption information, and changes to the navigation state of the content made by other clients.

The **ResourceFormat** enumeration represents the different resources that can be associated with a **PptContent**. The following table lists the values of the **ResourceFormat** enumeration.

Value	Numeric value	Description
None	0	No resource format
PartialPpt	1	Partial PowerPoint resource format
FullPpt	2	Full PowerPoint resource format
NativeFile	3	Native PowerPoint file resource

Value	Numeric value	Description
		format
PartialDhtml_Deprecated	4	Deprecated
FullDhtml_Deprecated	5	Deprecated
PartialJpeg	6	Partial Jpeg resource format
FullJpeg	7	Full Jpeg resource format
PreviewThumbnail	8	Preview Thumbnail resource format
SlideThumbnails	9	Slide Thumbnails resource format
Notes	10	Notes resource format
Dhtml	11	Dhtml resource format

The **ResourceErrorCode** enumeration enumerates the reasons that a resource is not available. The following table lists the values of the **ResourceErrorCode** enumeration.

Value	Numeric value	Description
ClientConversionFailed	1	Failure during conversion on the client
ClientUploadFailed	2	Failure during upload on the client
ClientDependentResourceFailed	3	Failure for a dependent resource on the client
UploaderClientDisconnected	100	Uploading client was disconnected before completing all the resource uploads
UploaderClientDemoted	101	Uploading client was demoted before completing all the resource uploads
UnsupportedResourceFormat	102	Unsupported resource format was specified

PptLocation: The following **Location** schema is used as the location argument for the methods **sSetLocation** and **cSetLocation**.

```
<xs:schema
  version="1.0"
  targetNamespace="http://schemas.microsoft.com/2008/08/datamcu-content"
  xmlns:tns="http://schemas.microsoft.com/2008/08/datamcu-content"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ms="urn:microsoft-cpp-xml-serializer"

  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  >

  <xs:annotation>
    <xs:documentation>Schema for PowerPoint Content Location.</xs:documentation>
  </xs:annotation>
```

```
        <xs:element name="PptLocation" type="tns:pptloc_type"
ms:className="CXmlPptContentLocationRoot" />

  <!-- Complex type for PptLocation -->
  <xs:complexType name="pptloc_type" ms:className="CXmlPptContentLocation" >
    <xs:sequence>
       <xs:element name="SlideNumber" type="xs:nonNegativeInteger"
ms:PropertyName="SlideNumber"/>
       <xs:element name="Click" type="xs:nonNegativeInteger"  ms:PropertyName="Click" />

     <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded"></xs:any>
    </xs:sequence>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>

</xs:schema>
```

SlideNumber: MUST be greater than or equal to zero and less than the slide count of the **PptContent**.

Click: SHOULD be greater than or equal to zero and less than or equal to the number of clicks in the slide referenced by **SlideNumber**.

3.1.4.1.11.1 Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.PptContent", Version=2]
DOInterface PptContent
{
    [Hash=5075721282249770644]
    ServerInterface
    {
        void sSetLocation(String location);

        void sSetResourceErrorCode(
            Int32 beginSlide,
            Int32 endSlide,
            Int32 format,
            Int32 errorCode);
    }

    [Hash=-3775280164302319261]
    ClientInterface
    {
        void cConnectCompleted();

        void cErrorAddingResource(
            Int32 resourceId,
            Int32 errorCode);

        void cSetFullPptFileInfo(
            String url,
            Byte[] key,
            Byte[] iv,
            Byte[] hash);

        void cSetLocation(String location);

        void cSetNotesInfo(
            String url,
            Int32[] slideNumbers,
            Byte[] key,
            Byte[] iv,
            Byte[] hash);

        void cSetPartialPptFileInfo(
```

```
            String url,
            Int32[] slideNumbers,
            Byte[] key,
            Byte[] iv,
            Byte[] hash);

        void cSetPreviewThumbnailFileInfo(
            String url,
            Byte[] key,
            Byte[] iv,
            Byte[] hash);

        void cSetResourceErrorCode(
            Int32 beginSlide,
            Int32 endSlide,
            Int32 format,
            Int32 errorCode);

        void cSetSlideCount(Int32 slideCount);

        void cSetSlideDhtmlMarkupInfo(
            Int32 slideNumber,
            Int32 clickCount,
            String markupUrl,
            String[] resourceOriginalFileNames,
            Byte[] key,
            Byte[] iv,
            Byte[] hash);

        void cSetSlideDhtmlResourceInfo(
            String resourceOriginalFileName,
            String url,
            Byte[] key,
            Byte[] iv,
            Byte[] hash);

        void cSetSlideJpegInfo(
            Int32 slideNumber,
            String url,
            Byte[] key,
            Byte[] iv,
            Byte[] hash);

        void cSetThumbnailFileInfo(
            String[] files,
            String url,
            Byte[] key,
            Byte[] iv,
            Byte[] hash);
    }

    Children
    {
        (annotationContainer, AnnotationContainer)
    }

}
```

The following is the interface definition of the **PptContent** distributed object for version 11<4>. Note that there are no method changes between version 2 and version 11. The only change is in the version number.

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.PptContent", Version=11]
DOInterface PptContent
{
```

```
            [Hash=5075721282249770644]
            ServerInterface
            {
                void sSetLocation(String location);

                void sSetResourceErrorCode(
                    Int32 beginSlide,
                    Int32 endSlide,
                    Int32 format,
                    Int32 errorCode);
            }

            [Hash=-3775280164302319261]
            ClientInterface
            {
                void cConnectCompleted();

                void cErrorAddingResource(
                    Int32 resourceId,
                    Int32 errorCode);

                void cSetFullPptFileInfo(
                    String url,
                    Byte[] key,
                    Byte[] iv,
                    Byte[] hash);

                void cSetLocation(String location);

                void cSetNotesInfo(
                    String url,
                    Int32[] slideNumbers,
                    Byte[] key,
                    Byte[] iv,
                    Byte[] hash);

                void cSetPartialPptFileInfo(
                    String url,
                    Int32[] slideNumbers,
                    Byte[] key,
                    Byte[] iv,
                    Byte[] hash);

                void cSetPreviewThumbnailFileInfo(
                    String url,
                    Byte[] key,
                    Byte[] iv,
                    Byte[] hash);

                void cSetResourceErrorCode(
                    Int32 beginSlide,
                    Int32 endSlide,
                    Int32 format,
                    Int32 errorCode);

                void cSetSlideCount(Int32 slideCount);

                void cSetSlideDhtmlMarkupInfo(
                    Int32 slideNumber,
                    Int32 clickCount,
                    String markupUrl,
                    String[] resourceOriginalFileNames,
                    Byte[] key,
                    Byte[] iv,
                    Byte[] hash);

                void cSetSlideDhtmlResourceInfo(
                    String resourceOriginalFileName,
                    String url,
```

```
                    Byte[] key,
                    Byte[] iv,
                    Byte[] hash);

            void cSetSlideJpegInfo(
                    Int32 slideNumber,
                    String url,
                    Byte[] key,
                    Byte[] iv,
                    Byte[] hash);

            void cSetThumbnailFileInfo(
                    String[] files,
                    String url,
                    Byte[] key,
                    Byte[] iv,
                    Byte[] hash);

        }

        Children
        {
            (annotationContainer, AnnotationContainer)
        }

    }
```

3.1.4.1.11.2 Children

The **AnnotationContainer** child distributed object is responsible for managing the annotations of the **PptContent**. Connection to this child is initiated by the server.

3.1.4.1.12   AnnotationContainer

The **AnnotationContainer** distributed object is responsible for communicating operations involving **Annotations**. These operations include adding and removing annotations, and changing annotation properties. Any component that needs **Annotation** functionality SHOULD include this distributed object as a child.

AnnotationConstraint: Used in some distributed object messages. These represent various constraints on annotations referenced in server and client message sections. The following table lists the values for **AnnotationConstraint**.

Value	Numeric value	Description
MaxNumDrawingAnnotations	1	The maximum number of drawing annotations that are allowed by the server
MaxNumTextAnnotations	2	The maximum number of text annotations that are allowed by the server
MaxNumImageAnnotations	3	The maximum number of image annotations that are allowed by the server
MaxNumStampAnnotations	4	The maximum number of stamp annotations that are allowed by the server
MaxDrawingPathDataLength	5	The maximum length of PathData property that is allowed by the

Value	Numeric value	Description
		server
MaxDrawingStrokeThickness	6	The maximum value of StrokeThickness property that is allowed by the server
MaxTextLength	7	The maximum length of Text property that is allowed by the server
MaxTextFontSize	8	The maximum value of Font property that is allowed by the server
MaxImageFileSize	9	The maximum file size of an image that is allowed by the server
MaxImageWidth	10	The maximum width of an image that is allowed by the server
MaxImageHeight	11	The maximum height of an image that is allowed by the server

Annotation Properties: Used in some distributed object messages. These represent various properties for annotations referenced in server and client message sections. The following table lists the property names, and describes the property values.

Property name	Property value (String)
"LOCALID"	Annotation Id assigned on the client that requested the add operation
"ANCHOR"	Annotation location
"EXTENT"	Drawing/Image Annotation size
"DRAWINGTYPE"	Drawing annotation type
"STROKE"	Drawing annotation stroke color
"STROKETHICKNESS"	Drawing annotation stroke thickness
"FILL"	Drawing annotation fill color
"DATA"	Drawing annotation path data
"IMAGETYPE"	Image annotation type
"TEXT"	Text annotation text
"WIDTH"	Text annotation width
"FONTFACE"	Text annotation font face
"FONTSIZE"	Text annotation font size
"FONTCOLOR"	Text annotation font color
"TEXTDIRECTION"	Text annotation text direction

3.1.4.1.12.1 Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.AnnotationContainer", Version=1]
DOInterface AnnotationContainer
{
    [Hash=-5714708003270970775]
    ServerInterface
    {
        void sAddAnnotation(Int32 type, String[][] properties);

        void sChangeProperties(Int32 id, Int32 gen, String[][] properties);

        void sChangePropertyForGroup(Int32[] ids, Int32[] gens, String property, String
value);

        void sChangePropertyForGroup(Int32[] ids, Int32[] gens, String property, String[]
values);

        void sChangeText(Int32 id, Int32 gen, Int32 textVersion, Int32[] begins, Int32[]
ends, String[] characters);

        void sClearAnnotations();

        void sRemoveAnnotation(Int32 id);

        void sRemoveAnnotations(Int32[] ids, Int32 cookie);

        void sSetTelepointer(String anchor, Boolean visible);
    }

    [Hash=2571074477103256610]
    ClientInterface
    {
        void cAddAnnotationBatch(Int32[] ids, Int32[] gens, Int32[] types, Int64[] ownerIds,
Int64[] modifierIds, Int32[] propertyCounts, String[] properties, String[] values);

        void cChangePropertiesBatch(Int32[] ids, Int32[] gens, Int64[] modifierIds, Int32[]
propertyCounts, String[] properties, String[] values);

        void cChangeTextBatch(Int32[] ids, Int32[] gens, Int64[] modifierIds, Int32[]
textVersions, Int32[] deltaCounts, Int32[] begins, Int32[] ends, String[] characters);

        void cClearAnnotations(Int64 removerId);

        void cErrorAddAnnotation(Int32 type, String[][] properties, String errorCode);

        void cErrorChangeProperties(Int32 id, Int32 gen, Int64 modifierId, String[][]
properties, String errorCode);

        void cErrorChangePropertyForGroup(Int32[] ids, Int32[] gens, Int64[] modifierIds,
String property, String[] values, String errorCode);

        void cErrorChangeText(Int32 id, Int32 gen, Int64 modifierId, String errorCode);

        void cErrorClearAnnotations(String errorCode);

        void cErrorRemoveAnnotation(Int32 id, String errorCode);

        void cErrorRemoveAnnotations(Int32[] ids, String errorCode, Int32 cookie);

        void cErrorSetTelepointer(String errorCode);

        void cRemoveAnnotation(Int32 id, Int64 removerId);

        void cRemoveAnnotations(Int32[] ids, Int64 removerId, Int32 cookie);

        void cSetAnnotationConstraints(Int32[] constraints, Int32[] values);
```

```
        void cSetImageFileInfo(Int32 id, String url, Byte[] key, Byte[] iv, Byte[] hash);

        void cSetTelepointer(String anchor, Int64 ownerId, Boolean visible);
    }

    Children
    {
    }
}
```

3.1.4.1.12.2 Children

This distributed object MUST NOT contain any child distributed objects.

3.1.4.1.13   WhiteboardContent

The **WhiteboardContent** distributed object is responsible for operations on whiteboard content. It represents an instance of a **Whiteboard** shared in the content session. Currently, there is only one operation. This distributed object is also a parent for the **AnnotationContainer** distributed object, which is responsible for the annotation functionality on the **Whiteboard**.

3.1.4.1.13.1 Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.WhiteboardContent", Version=1]
DOInterface WhiteboardContent
{
    [Hash=4720625287907297465]
    ServerInterface
    {
    }

    [Hash=5909677840878629841]
    ClientInterface
    {
        void cConnectCompleted();
    }

    Children
    {
        (annotationContainer, AnnotationContainer)
    }
}
```

3.1.4.1.13.2 Children

The **WhiteboardContent** distributed object has one child distributed object, the **AnnotationContainer** distributed object. The **AnnotationContainer** distributed object encapsulates the annotation functionality for whiteboard content. The server MUST initiate the connect operation for the child distributed object, supplying the distributed object name "annotationContainer". Exactly one **AnnotationContainer** child distributed object MUST be connected.

3.1.4.1.14   PollContent

The **PollContent** distributed object is responsible for operations on poll content. It represents an instance of a real-time collaborative poll, or survey, with a single question and a list of up to 7 multiple choice answers. The creator includes initial poll information, both question and choices, during the package upload. Clients are allowed to vote for any given choice, where the values zero

("0") to "6" represent a given choice. A value of "-1" corresponds to no vote. When a given client votes, the sum of the votes for a given choice are tabulated and sent out to all clients that are allowed to view poll results.

The **open** state represents whether or not users can vote. The **results** state allows the client to show or hide results.

3.1.4.1.14.1 Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.PollContent", Version=1]
DOInterface PollContent
{
    [Hash=7882912516919930871]
    ServerInterface
    {
        void sClearVotes();

        void sModify(String question, String[] choices, Boolean rememberPastUsers, String
customizationXml);

        void sSetOpenState(Boolean open);

        void sSetResultsState(Boolean visibleToAll);

        void sVote(Int32 choice);
    }

    [Hash=-1572291151947590318]
    ClientInterface
    {
        void cConnectCompleted();

        void cSetLocalVote(Int32 choice);

        void cSetOpenState(Boolean open);

        void cSetQuestion(String question, String[] choices, Boolean rememberPastUsers,
String customizationXml);

        void cSetResults(Int32[] results);

        void cSetResultsState(Boolean visibleToAll);
    }

    Children
    {
    }
}
```

3.1.4.1.14.2 Children

The **PollContent** distributed object MUST NOT contain any child distributed objects.

3.1.4.1.15   SharedNotesContent

This section follows the product behavior described in endnote<5>.

The **SharedNotesContent** distributed object represents an instance of a Notes page shared in the session. A presenter shares a Notes page, and includes the Notes URLs and Notes titles during the package upload. The attendees can choose to view the page using the URLs.

A **SharedNotesContent** package upload MUST NOT be attempted unless a **SharedNotesContent** interface has been negotiated successfully between client and server.

3.1.4.1.15.1 Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.SharedNotesContent", Version=3]
DOInterface SharedNotesContent
{
    [Hash=6726721969813329838]
    ServerInterface
    {
        void sSetErrorCode(Int32 code);
    }

    [Hash=8721147891391263529]
    ClientInterface
    {
        void cConnectCompleted();

        void cSetNotesId(String notesId);

        void cSetNotesTitles(String notebookTitle, String sectionTitle, String pageTitle);

        void cSetErrorCode(Int32 code);
    }

    Children
    {
    }

}
```

3.1.4.1.15.2 Children

The **SharedNotesContent** distributed object MUST NOT contain any child distributed objects.

3.1.4.1.16   QnaContent

This section follows the product behavior described in endnote<6>.

The **QnaContent** distributed object represents an instance of a questions and answers shared in the session. A presenter initiates sharing of the questions and answers content. The content does not have any questions right after upload.

A **QnaContent** package upload MUST NOT be attempted unless a **QnaContent** interface has been negotiated successfully between client and server.

The **QnaOpenState** enumeration enumerates the supported values for questions and answers content open state. The following table lists the values of the **QnaOpenState** enumeration.

Value	Numeric value	Description
None	0	The questions and answers content was not initialized
Open	1	The questions and answers content is available and the meetings participant can ask new questions or answer existing questions
Suspended	2	The questions and answers content exists, but not available for client interaction

3.1.4.1.16.1 Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.QnaContent", Version=1]
DOInterface QnaContent
{
    [Hash=-210597168530409383]
    ServerInterface
    {
      void sSetOpenState(QnaContentConstants.QnaOpenState openState);

      void sPutBlob(string blob);
    }

    [Hash=473785100728654906]
    ClientInterface
    {
      void cSetOpenState(QnaContentConstants.QnaOpenState openState);

      void cSetQuestionsCount(int count);

      void cPutBlob(string blob);

    /* General Content Calls */
      void cConnectCompleted();
    }

    Children
    {
    }
}
```

3.1.4.1.16.2 Children

The **QnaContent** distributed object MUST NOT contain any child distributed objects.

3.1.4.1.17   SharedLinksContent

This section follows the product behavior described in endnote<7>.

The **SharedLinksContent** distributed object represent an instance of a document that is intended to be viewed or edited simultaneously on multiple end points. It also provides a mechanism to specify the current active document.

3.1.4.1.17.1 Interface

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.SharedLinksContent", Version=2]
DOInterface SharedLinksContent
{
    [Hash=7904618883769218068]
    ServerInterface
    {
        void sSetActive();
        void sSetInactive();
    }

    [Hash=-988586141242916940]
    ClientInterface
    {
        void cConnectCompleted();
        void cSetViewURL(string viewURL);
        void cSetEditURL(string editURL);
        void cSetActive();
        void cSetInactive();
```

```
        void cSetUniqueId(string uniqueId);
        void cSetState(string contentTitle, string viewURL, string editURL, string
uniqueId, bool active, int32[] msDiagIds);
    }

    Children
    {
    }
}
```

3.1.4.1.17.2 Children

The **SharedLinksContent** distributed object MUST NOT contain any child distributed objects.

3.1.5 Message Processing Events and Sequencing Rules

Each message MUST be processed in the order received. If an unrecognized message is ever received, the connection MUST be terminated immediately.

3.1.6 Timer Events

When the **ping** timer expires, a **ping** message is sent with no data on PSOM channel zero. This serves as a keepalive message on the transport.

3.1.7 Other Local Events

When a connection is lost, no recovery action is taken. To reconnect, a full connect sequence MUST be repeated. All timers SHOULD be reset, and the state SHOULD mirror that of the starting state prior to connection.

3.2 Client Details

The client and server have the life-cycle stages shown in the following diagram.
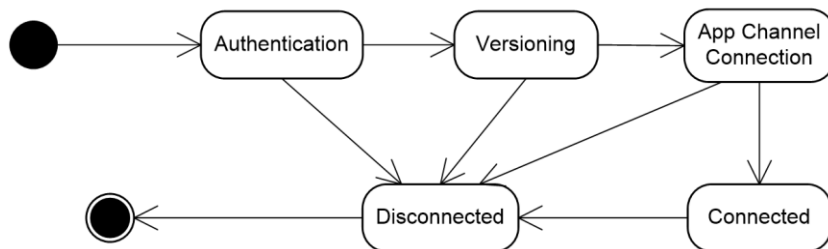
Figure 1: Client and server life-cycle stages

Each state requires mutual operations by both the client and the server (2) to get to the next state. If there is a critical failure, the connection MUST be terminated.

3.2.1 Abstract Data Model

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

The abstract data model for the client is covered in section 3.1.1 because this protocol is a symmetric protocol at the lower layer. At a higher layer, the client has a set of interfaces and objects that are created. Generally, the server sends **RPC connect** messages, except in the case of the content-related distributed objects. These objects are generally connected by the client sending the connect message.

3.2.2 Timers

There is a **ping** method called with no parameters used as a keepalive message sent every 30 seconds by the client on the **ConnMgr** distributed object. For the **ConnMgr** definition, see section 3.2.3.1.3. When the timer expires, a new one is set at the same interval period.

3.2.3 Initialization

When this protocol is initialized, it MUST begin connecting to a client or server or wait for incoming connections from a server. The protocol has no state at this point, and no distributed objects are connected. At a high level, connection is established, and is followed by authentication, versioning, and then the root distributed object connection.

3.2.3.1 Connections

This section describes the steps required for a client to connect and successfully negotiate a connection with a server. Connection entails the following main components:

▪ Authentication

▪ Interface versioning

▪ Root distributed object PSOM channel negotiation

3.2.3.1.1 Authentication

Authentication involves two steps:

1. Obtain the **AuthenticationToken**, a string used to identify users to the **DataMCU**. This step is done via a different PSOM channel than the next step.

2. PSOM connection join.

3.2.3.1.1.1 Obtain the Authentication Token

For references to details such as **sAuthId** and **pwrpc.modes**, see [MS-CONFBAS] section 2.2.3.17 and [MS-CONFBAS] section 2.2.3.18. These references describe the **AddUser** request and response mechanism where these values are obtained.

The client needs the **sAuthId** from the **AddUser** response from the server to authenticate. Additionally, if a proxy server is used to allow for load balancing of the server, the client needs the **proxyHeader** field from the **AddUser** response.

If the **pwrpc.modes** field is "tls", only **sAuthId** is needed. If it is "fwdtls", both **sAuthId** and **proxyHeader** MUST be sent to the server as follows.

To determine the **fully qualified domain name (FQDN)** to connect to, if **pwrpc.modes** is "tls", the client MUST establish a TLS connection to the server and port given in **pwrpc.pwsURI**. This MUST be in the form *"[fqdn]:[port]"*, where *[fqdn]* is the FQDN of the server, and *[port]* is the open port that will accept connections. If **pwrpc.modes** is "fwdtls", the client MUST attempt to establish a connection to a proxy with a given FQDN and port at proxy[*i*].FQDN and proxy[*i*].Port where *i* is an

integer that starts from zero and goes to N. The client MUST try to connect to a random value of *i*, and if that fails, attempt to connect to another possible FQDN and port.

In establishing a connection, if **alternativeName** is present in the **AddUser** and **pwrpc.modes** is "tls", **alternativeName** MUST be used in place of the **X.509 certificate** subject to the validation of the FQDN to complete the TLS negotiation. When the connection is established through TLS, authentication begins.

Authorization is only sent by the client and validated by the server (2). The server (2) MUST immediately terminate any client connection that does not provide this information correctly.

Immediately after establishing a connection to the server, if applicable, the client MUST send the length of **proxyHeader** in **network byte order**. It MUST follow that with the value it has for **proxyHeader**.

Regardless of whether a proxy is present, the client MUST then send the following in network byte order:

```
[0x70 77 32 00][0x 00 00 00 00] [length of sAuthId in bytes][sAuthId].
```

See section 3.2.3.1.1.2 for additional details.

To complete authentication of a valid **sAuthId** and **proxyHeader**, the server MUST respond with `[0x 70 77 32 00]`. When this sequence is completed, the versioning stage begins.

3.2.3.1.1.2   PSOM Connection Join

When a transport connection is established, the client MUST place a special signature on the wire, followed by the **AuthenticationVersion**. Then it MUST place an encoded version of the **AuthenticationToken**, which was obtained in section 3.3.3.1.1, on the wire. At this point, the client can continue to send data. The data that the client MUST place on the wire for the PSOM Connection Join has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
Signature																															
AuthenticationVersion																															
AuthenticationTokenLen																															
AuthenticationToken (variable)																															
...																															

Signature (4 bytes): Defined as "0x70773200". This field is constant.

AuthenticationVersion (4 bytes): Defined as "0x00000000". This field is constant.

AuthenticationTokenLen (4 bytes): The length of the following **AuthenticationToken**, in bytes.

AuthenticationToken (variable): The byte representation of each character in the **AuthenticationToken** string.

3.2.3.1.2 Interface Versioning

After authentication is complete, the client MUST exchange hashes of each supported distributed object version with the server. Because the root PSOM channel, channel zero, is set up implicitly, a special distributed object is used to exchange versioning information messages. This distributed object is referred to as **ConnMgr** and handles exchanging the supported list of distributed objects, which are to be connected on PSOM channel 2 at a later time. If at any time version hashes do not match, the client MUST terminate the connection immediately. A **break** message can be sent prior to closing the connection.

Prior to interface versioning, the client MUST send a **setChannel** record message to the server, and set the **channelId** to "0x00000000". After that, it is free to send any RPC messages to its **ConnMgr** peer on the server, as the root distributed object is implicitly connected.

At a high level, **ConnMgr** negotiation has the client send the following RPC messages to the server:

- A **version** message.

- An **addProtocol** message for each supported protocol on the client.

- A **doneProtocols** message.

The payload of **version** SHOULD be the distributed object hash for **ConnMgr**'s client interface. An **addProtocol** call MUST be made for each supported distributed object on the client side. Each **addProtocol** call MUST contain the **DOInterface** name attribute, the set of supported versions, which is also declared in the **DOInterface** definition, and each version's corresponding client hash code. When **doneProtocols** is sent, **version**, **addProtocol**, and **doneProtocols** MUST NOT be sent by the client again.

When the client receives **doneProtocols** from the server, and has sent **doneProtocols** to the server (2) after exchanging the hashes, the versioning phase is complete. At this point, the connection is established, and the client validates that it has compatible protocol versions with the server (2).

The **log** method MUST NOT be called by the client.

3.2.3.1.3 ConnMgr Distributed Object Interface Definition

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.Pod.ConnMgr", Version=1]
DOInterface ConnMgr
{
    [Hash=8322047979521208965L]
    ClientInterface
    {
        void version(Int64 stubHash);

        void addProtocol(String name, Int32[] versions, Int64[] hashes);

        void doneProtocols();

        void ping();
    }

    Children
    {
    }
}
```

3.2.3.1.3.1   ConnMgr Client Methods

ConnMgr client methods MUST be used to complete connection.

3.2.3.1.3.1.1 version

Allows the server to send its **ConnMgr** hash code to the client.

stubHash (Int64): The hash code of the **ConnMgr ServerInterface**.

3.2.3.1.3.1.2 addProtocol

Allows the server to communicate the name, versions, and hashes for a given distributed object interface. Some implementations MAY send multiple versions/hashes. Only the corresponding versions/hashes from the **DOInterface** are required. It MUST never be called after **doneProtocols** is sent.

name (String): The name of the **DOInterface**, acquired from the attribute **Name**.

versions (Int32[]): A list of versions.

hashes (Int64[]): A list of summed hashes for the corresponding client and server interface versions, or client interface hash value plus server interface hash value.

3.2.3.1.3.1.3 doneProtocols

The **doneProtocols** method is called once all **addProtocol** calls are completed. This call signals that the server is done sending protocol versions with **addProtocol**.

3.2.3.1.3.1.4 ping

The **ping** method is used as a keepalive mechanism. It is a no-op. For more information about this method, see section 3.2.2.

3.2.3.1.4 Root Distributed Object Channel Negotiation

After versioning is complete, the client MUST initiate creation of PSOM channel 2. This PSOM channel is used to send all application messages, unlike PSOM channel zero that is exclusively used for protocol level negotiations and messages. Any message sent on PSOM channel 2 has to do with application logic.

To do this, the client MUST send an **rpcOpenMessage** with a 32-bit integer payload of "0x00000002". Per **rpcOpenMessage** in section 2.2.1.1.5, this message MUST be followed by an RPC message that references the **ConnMgr** proxy, because the current PSOM channel is still zero, and SHOULD call the **lookup** method. The values of **name** and **protocol** are ignored, but MUST be valid PSOM string payloads. The value of **proxyHash** is ignored, but MUST be a valid PSOM 64-bit integer payload. At this point, the message MUST be sent.

When the **lookup** message is sent, it MUST be followed by a **setChannel** record. The payload of this record is a 32-bit big-endian unsigned integer with the value "0x00000002". At this point, the root distributed object PSOM channel was negotiated successfully. Any following messages are sent on PSOM channel 2, unless another **setChannel** message is sent to modify this. For more information about the root distributed object connection and connection of the subsequent application distributed objects, see section 3.1.4.1.

3.2.4 Higher-Layer Triggered Events

This section details application PSOM channel events. Any application PSOM channel actions are driven by user action.

3.2.4.1 Distributed Objects

This section details all **Application** distributed objects and their client-side methods. When the connection and versioning phase is complete, the root distributed object, **Meeting**, is connected. This contains all valid server to client RPC calls.

3.2.4.1.1 Meeting

The client component of the **Meeting** distributed object is the root distributed object for the application distributed objects. Its sole responsibility is to receive distributed object **connect** messages and the messages in the following subsections.

3.2.4.1.1.1   Methods

The methods for the **Meeting** distributed object are as follows:

cSetInfo: This method can be called by the server. If a client receives this, it SHOULD ignore it.

- **info (String):** Not defined.

cSetUrlBase: This method communicates information about the base URL for all downloadable files. It MUST be called by the server before **cMeetingReady** is called.

- **urlBase (String):** The base URL used for all downloadable files. This MUST be a valid URL path.

cMeetingReady: This method signals that all **Meeting** distributed object states were communicated. Logically, the **Meeting** distributed object has nothing else to do once this is called.

cSetServerTime: This method communicates the current time on the server in **Coordinated Universal Time (UTC)**. The format MUST be: *yyyy'-'MM'-'dd'T'HH':'mm':'ss*. It MUST be called by the server before **cMeetingReady** is called.

- **serverTime (String):** The current time in UTC on the server when this message is sent.

3.2.4.1.2 ContentUserManager

3.2.4.1.2.1   Methods

cUsersAdded: This method is called to tell the user about new user mappings. An **id**, **uri**, and **displayName** are grouped together. If the arrays each contain two elements, the first element of each array is the first user entry, and the second element of each array is the second user entry.

- **ids (Int64[]):** Array of server-generated user identifiers for this mapping.

- **uris (string []):** Array of URIs.

- **displayNames (string[]):** Array of **displayNames**.

cUsersRemoved: This method indicates that a user mapping is no longer valid.

  - **ids (Int64[]):** Array of server-generated user IDs that are no longer valid.

3.2.4.1.3 ContentManager

The server MUST call **cContentAdded** or **cContentRemoved** to inform clients when a piece of content is available or no longer available to that client. Additionally, when a client requests a new piece of content and it is created, the server MUST call **cContentCreated** to let the client know that the content was created.

Prior to creating any content, the title MUST be reserved by the client. This enforces title uniqueness among all content.

The active presenter for the meeting is managed through the Content Manager. This is managed via **sPresent** and **sStopPresenting** and communicated to the clients via **cSetActivePresenter**.

3.2.4.1.3.1   Methods

The methods for the **ContentManager** distributed object are as follows:

cContentAdded: This method informs the client that a new piece of content is available.

- **contentId (Int64):** A server-generated integer that uniquely identifies the content for this meeting.

- **type (String):** The type of the content object. This MUST be one of the following: "Content.Ppt", "Content.Whiteboard", "Content.NativeFileOnly", "Content.Poll", "Content.SharedNotes"<u><8></u> and "Content.PptWac"<u><9></u>. These correspond to the content's implementation distributed object.

cContentCreated: This method is called only on the client that originated the content creation request. It indicates to the creating client that a new piece of content was successfully created. This gives the new **contentId** and the cookie that was sent during the creation request so that the client can map the request to the **Content** distributed object.

- **contentId (Int64):** A server-generated integer that uniquely identifies the content for this **Meeting**.

- **cookie (Int32):** A client-generated integer that the client put in the content creation request via the **UploadManager**.

cContentCreationFailed: This method was deprecated.

- **cookie (Int32):** Deprecated.

- **reason (Int32):** Deprecated.

cContentRemoved: This method informs a client that the content is no longer available to it.

- **contentId (Int64):** A server-generated integer that uniquely identifies the content for this meeting.

cReserveTitleCompleted: This method informs the client about the success or failure of an attempt to reserve a title.

- **status (Int32 - TitleReservationStatus):** Result of the attempt to reserve a title.

- **cookie (Int32):** A client-generated integer that clients use to track the reservation request.

- **contentId (Int64):** The **contentId** associated with the reservation request. For a creation reservation, this is zero ("0"). For an upgrade reservation, this is the **contentId** of the content to be upgraded (currently not supported). Otherwise, this contains the **contentId** of the content that already owns the reservation.

- **owningUserId (Int64):** The **ContentUser** identifier associated with the client that owns the reservation.

cSetActiveContent: This method informs the client what the active content is for the **Meeting**. This can be zero ("0") to indicate there is no active content.

- **activeContentId (Int64):** A server-generated integer that represents the **contentId** of the content that is the actively presented content.

cSetActivePresenter: This method informs the client who the current active presenter is for the **Meeting**. This can be zero ("0") to indicate there is no active presenter.

- **activePresenterId (Int64):** A server-generated integer that represents the **ContentUser** Id of the user who is the active presenter.

cTitleReleased: This method informs the client that the reservation was released. This can be because of a client request or other conditions, such as a demotion that prevents the client's use of title reservations.

- **cookie (Int32):** The client-generated integer that was sent as part of the **sReserveTitle** request.

3.2.4.1.4 UploadManager

The file uploaded through **UploadManager** as part of the process of sending a file from the client to the server is designated as an "upload package." This package MUST be in the format of an Open Packaging Convention (OPC) container file. See section 7 for a sample package.

OPC is defined in [ISO/IEC29500:2011] Part 2 and in [ECMA-376].

The upload package, known as an OC package, MUST contain a manifest file called "OcpManifest.xml." The client can include additional files if they are referenced in the manifest file. This manifest file contains instructions for the server to use to create a content instance or an image annotation, to upgrade a content instance by replacing it with newer data or files, or to update a **PptContent** in data groups. Upgrading a content instance is currently not supported, and is reserved for future use. A server implementation MAY ignore this request. Any file referred to in the manifest MUST be present in the root level of the OC package. See the schema in section 3.2.4.1.4.2 for details of the format of the manifest.

To upload an OC package, the client MUST initiate the upload by sending an **sRequestUpload** message to the server.

The server can accept or deny the upload. If the server accepts the upload, the client can proceed with the upload using the upload stream distributed object provided by the server. When the upload is complete, the client MUST call the **sUploadFinished** method to notify the server that it sent all the data.

The following table lists the **UploadFinishReason** response codes.

Response code	Numeric value	Description
Ok	0	Success.
UserCancel	1	The user cancelled the upload.
MaxPackageSizeExceeded	2	The size of the package was too large as determined by the server (2).
CapacityExceeded	3	There is not enough space on the server (2) to complete the operation.
UnknownFailure	4	Unknown failure.
AlreadyUploading	5	The client is already uploading with that cookie.
VerifyFailed	6	There was an error verifying the package on the server (2).
VirusScanTimeout	7	The virus scanning engine took an excessive amount of time.
NotUploading	8	The cookie provided was invalid.

Response code	Numeric value	Description
TooManyUploads	9	The client attempted to initiate too many uploads at once.<10>
ArchiveFailed	10	There was a failure archiving important information.
TooManyContents	11	There are more than a predefined number of **Content** objects in the conference.<11>
TooManySlides	12	A given PowerPoint content upload contained more than a predefined number of slides.<12>

3.2.4.1.4.1   Methods

cAcceptUpload: This method is called by the server to notify the client that the server is ready, and the client can proceed with the upload for the content that corresponds to the specified cookie. The client can proceed to upload the package when it receives the call.

- **cookie (Int32):** The unique cookie value that corresponds to the content.

- **stream (DistributedObject):** The **UploadStream** distributed object that the client uses to upload the content package, where **DistributedObject** is defined in section 3.1.1.1.4.

cRejectUpload: This method is called by the server to notify the client that the content upload request with the specific cookie is rejected and any further calls with the given cookie MUST NOT be made. After the client calls **sRequestUpload**, the server MUST respond with **cRejectUpload** or **cAcceptUpload**.

- **cookie(int32):** The unique cookie value that corresponds to the content.

- **reason (Int32):** A reason code for rejecting the upload. See the preceding table for the response codes.

cUploadFinished: This method is called by the server to notify the upload client that the server finished processing the content upload that corresponds to the specific cookie. The function is called in both success and failure cases.

- **cookie (Int32):** The unique cookie value that corresponds to the content.

- **reason (Int32):** The reason that the upload finished. In failure cases, the parameter specifies the reason of the failure.

cSetAvailableSpace: This method is called by the server to notify the client that the storage space is available on the server. It is currently deprecated.

- **size (Int64):** Specifies the number of bytes available on the server.

cAcceptWebUpload<13>: This method is called by the server to notify the client that the server **sRequestWebUploadBlob** method has succeeded. The client responds by starting to upload a file using the provided **uploadUrl** value. It is guaranteed that at least one client attempt will be allowed for uploading a file content using the browser **POST** form upload **HTTPS** request.

Change made to resolve document build error.

cAcceptWebDownload<14>: This method is called by the server to notify the client that the server **sRequestWebDownloadBlob** method has succeeded. The client should respond by starting to download a file using the provided **downloadUrl** value. It is guaranteed that at least one client attempt will be allowed for downloading a file using the browser **GET** HTTPS request

- **Cookie (Int32):** The unique cookie value that corresponds to the original client request.

▪ **downloadUrl(String):** A Url to use for download an unencrypted file using GET HTTPS request.

cRejectWebDownload<15>: This method is called by the server to notify the client that the server **sRequestWebDownloadBlob** method has failed.

▪ **Cookie (Int32):** The unique cookie value that corresponds to the original client request.

▪ **reason (Int32):** The reason why **sRequestDownloadBlobWeb** request has failed.

The following table lists the **DownloadFinishReason<16>** response codes.

Response code	Numeric value	Description
Ok	0	Success (reserved and currently is never used).
NotAuthorized	1	The user was not authorized to download this file.
NotFound	2	The specified filename is not found.
NotSupported	3	This conference does not support decrypted files download using the HTTPS **GET** request.
UnknownFailure	4	Unknown failure.

3.2.4.1.4.2   Schema

The schema for **Package** is as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema
  version="1.0"
  targetNamespace="http://schemas.microsoft.com/2008/12/ocp"
  xmlns:tns="http://schemas.microsoft.com/2008/12/ocp"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ms="urn:microsoft-cpp-xml-serializer"

  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  >

  <xs:annotation>
    <xs:documentation>Schema file for the OC Package manifest file between client and server
for Data MCU.</xs:documentation>
  </xs:annotation>

  <!-- Ocp root element definition -->
  <xs:element name="ocp" type="tns:ocp-type" ms:className="CXmlOcpDocument"/>

  <!-- ocp-type definition for root element -->
  <xs:complexType name="ocp-type" ms:className="CXmlOcp">
    <xs:sequence>
      <xs:choice minOccurs="1" maxOccurs="1">
        <xs:element name="createContent" type="tns:create-content-type"
ms:propertyName="CreateContent"/>
        <xs:element name="createAnnotation" type="tns:create-annotation-type"
ms:propertyName="CreateAnnotation"/>
        <xs:element name="createPptContentResource" type="tns:create-ppt-content-resource-
type" ms:propertyName="CreatePptContentResource"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <!-- content-creation-type definition for contentCreation node -->
```

```xml
  <xs:complexType name="create-content-type" ms:className="CXmlCreateContent">
    <xs:sequence>
      <xs:element name="upgrade" type="tns:upgrade-type" ms:propertyName="Upgrade"
minOccurs="0" maxOccurs="1" />
      <xs:element name="common" type="tns:common-type" ms:propertyName="Common" />
      <xs:element name="contentDetail" type="tns:content-detail-type"
ms:propertyName="ContentDetail" />
    </xs:sequence>
  </xs:complexType>

  <!-- common-type definition for contentCreation node -->
  <xs:complexType name="common-type" ms:className="CXmlCommon">
    <xs:sequence>
      <xs:element name="title" type="xs:string" ms:propertyName="Title" />
      <xs:element name="nativeFile" type="xs:string" ms:propertyName="NativeFile"
minOccurs="0" />
      <xs:element name="visibility" type="tns:visibility-type" ms:propertyName="Visibility"
minOccurs="0" />
      <xs:element name="presented" type="xs:boolean" ms:propertyName="Presented"
minOccurs="0" />
      <xs:element name="originalFileUrl" type="xs:string" ms:propertyName="OriginalFileUrl"
minOccurs="0" />
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="visibility-type" ms:className="CXmlVisibilityType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="MeetingOrganizer"/>
      <xs:enumeration value="Presenters"/>
      <xs:enumeration value="Everyone"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- upgrade-type definition for contentCreation node -->
  <xs:complexType name="upgrade-type" ms:className="CXmlUpgradeType">
    <xs:attribute name="contentId" type="xs:long" ms:propertyName="ContentId" />
  </xs:complexType>

  <!-- content-detail-type definition for contentCreation node -->
  <xs:complexType name="content-detail-type" ms:className="CXmlContentDetail">
    <xs:sequence>
      <xs:any namespace="##other" processContents="lax" minOccurs="0"></xs:any>
    </xs:sequence>
    <xs:attribute name="type" type="tns:content-type-type" ms:propertyName="Type"/>
  </xs:complexType>

  <xs:simpleType name="content-type-type" ms:className="CXmlContentType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Content.Ppt"/>
      <xs:enumeration value="Content.Whiteboard"/>
      <xs:enumeration value="Content.NativeFileOnly"/>
      <xs:enumeration value="Content.Poll"/>
      <xs:enumeration value="Content.SharedNotes"/>
      <xs:enumeration value="Content.PptWac"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- annotation-creation-type definition for createAnnotation node -->
  <xs:complexType name="create-annotation-type" ms:className="CXmlCreateAnnotation">
    <xs:sequence>
      <xs:element name="creationProperties" type="tns:creationProperties-type"
ms:propertyName="CreationProperties" />
      <xs:element name="annotationDetail" type="tns:annotation-detail-type"
ms:propertyName="AnnotationDetail" />
    </xs:sequence>
  </xs:complexType>

  <!-- creationProperties-type definition for createAnnotation node -->
  <xs:complexType name="creationProperties-type" ms:className="CXmlCreationProperties">
```

```
    <xs:sequence>
      <xs:element name="parentId" type="xs:string" ms:propertyName="ParentId" />
    </xs:sequence>
  </xs:complexType>

  <!-- annotation-detail-type definition for createAnnotation node -->
  <xs:complexType name="annotation-detail-type" ms:className="CXmlAnnotationDetail">
    <xs:sequence>
      <xs:any namespace="##other" processContents="lax" minOccurs="0"></xs:any>
    </xs:sequence>
  </xs:complexType>

  <!-- create-ppt-content-resource-type definition -->
  <xs:complexType name="create-ppt-content-resource-type"
ms:className="CXmlCreatePptContentResource">
    <xs:sequence>
      <xs:element name="contentId" type="xs:long" ms:propertyName="ContentId" />
      <xs:element name="resourceDetail" type="tns:ppt-content-resource-detail-type"
ms:propertyName="ResourceDetail" />
    </xs:sequence>
  </xs:complexType>

  <!-- annotation-detail-type definition for createAnnotation node -->
  <xs:complexType name="ppt-content-resource-detail-type"
ms:className="CXmlPptContentResourceDetail">
    <xs:sequence>
      <xs:any namespace="##other" processContents="lax" minOccurs="0"></xs:any>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

The schema for content-specific portions of the OC Package manifest is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema
  version="1.0"
  targetNamespace="http://schemas.microsoft.com/2008/12/ocp-content-detail"
  xmlns:tns="http://schemas.microsoft.com/2008/12/ocp-content-detail"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ms="urn:microsoft-cpp-xml-serializer"

  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  >

  <xs:annotation>
    <xs:documentation>Schema for content-specific portions of the OC Package
manifest.</xs:documentation>
  </xs:annotation>

  <xs:element name="chunkedPptContent" type="tns:chunked-ppt-content-detail-type"
ms:className="CXmlChunkedPptContentDetailDocument" />

  <xs:complexType name="chunked-ppt-content-detail-type"
ms:className="CXmlChunkedPptContentDetail">
    <xs:sequence>
      <xs:element name="numSlides" type="xs:nonNegativeInteger" minOccurs="0" maxOccurs="1"
ms:propertyName="NumSlides" />
      <xs:element name="previewThumbnailFile" type="xs:string" minOccurs="0" maxOccurs="1"
ms:propertyName="PreviewThumbnailFile" />

      <!-- The initial set of resources that can be optionally sent during content creation
-->
      <xs:element name="pptContentResources" type="tns:ppt-content-resources-type"
minOccurs="0" maxOccurs="1" ms:className="CXmlPptContentResourcesDocument" />
```

```
      </xs:sequence>
    </xs:complexType>


    <xs:element name="pptContentResources" type="tns:ppt-content-resources-type"
ms:className="CXmlPptContentResourcesDocument" />

    <xs:complexType name="ppt-content-resources-type" ms:className="CXmlPptContentResources">
      <xs:sequence>
        <xs:element name="pptContentResource" type="tns:ppt-content-resource-type"
ms:propertyName="PptContentResourceList" minOccurs="1" maxOccurs="unbounded"
ms:className="CXmlPptContentResource">
        </xs:element>
      </xs:sequence>
    </xs:complexType>

  <xs:complexType name="ppt-content-resource-type" ms:className="CXmlPptContentResource">
      <xs:choice>
          <xs:element name="pptThumbnailPackage" type="tns:ppt-thumbnail-package-type"
ms:propertyName="PptThumbnailPackage" />
          <xs:element name="pptNotesPackage" type="tns:ppt-notes-package-type"
ms:propertyName="PptNotesPackage" />
          <xs:element name="pptDhtmlMarkupPackage" type="tns:ppt-dhtml-markup-package-type"
ms:propertyName="PptDhtmlMarkupPackage"/>
          <xs:element name="pptDhtmlResourcesPackage" type="tns:ppt-dhtml-resources-package-
type" ms:propertyName="PptDhtmlResourcesPackage" />
          <xs:element name="pptJpegPackage" type="tns:ppt-jpeg-package-type"
ms:propertyName="PptJpegPackage" />
          <xs:element name="partialPptFile" type="tns:ppt-partial-ppt-package-type"
ms:propertyName="PptPartialPptPackage" />
          <xs:element name="fullPptFile" type="tns:ppt-full-ppt-package-type"
ms:propertyName="PptFullPptPackage" />
          <xs:element name="nativeFile" type="xs:string"   ms:propertyName="NativeFile" />
      </xs:choice>
      <xs:attribute name="resourceId" type="xs:long" ms:propertyName="ResourceId" />
    </xs:complexType>

    <xs:complexType name="ppt-thumbnail-package-type" ms:className="CXmlPptThumbnailPackage">
      <xs:sequence>
          <xs:element name="thumbnailPackageFile"  type="xs:string"
ms:propertyName="ThumbnailPackageFile" />
          <xs:element name="pptThumbnailSlides" type="tns:ppt-thumbnail-slide-type"
ms:propertyName="PptThumbnailSlideList" minOccurs="1" maxOccurs="unbounded"
ms:className="CXmlPptThumbnailSlide" />
      </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ppt-thumbnail-slide-type"  ms:className="CXmlPptThumbnailSlide">
      <xs:attribute name="slideNumber"    type="xs:nonNegativeInteger"
ms:propertyName="SlideNumber"/>
      <xs:attribute name="thumbnailFile"  type="xs:string"  ms:propertyName="ThumbnailFile"
/>
    </xs:complexType>

    <xs:complexType name="ppt-notes-package-type" ms:className="CXmlPptNotesPackage">
      <xs:sequence>
          <xs:element name="pptNotesSlides" type="tns:ppt-notes-slide-type"
ms:propertyName="PptNotesSlideList" minOccurs="1" maxOccurs="unbounded"
ms:className="CXmlPptNotesSlide"/>
      </xs:sequence>
      <xs:attribute name="notesPackageFile"  type="xs:string"
ms:propertyName="NotesPackageFile" />
    </xs:complexType>

    <xs:complexType name="ppt-notes-slide-type"  ms:className="CXmlPptNotesSlide">
      <xs:attribute name="slideNumber"    type="xs:nonNegativeInteger"
ms:propertyName="SlideNumber"/>
      <xs:attribute name="notesFile"  type="xs:string"  ms:propertyName="NotesFile" />
    </xs:complexType>
```

```xml
    <xs:complexType name="ppt-dhtml-markup-package-type"
ms:className="CXmlPptDhtmlMarkupPackage">
        <xs:sequence>
            <xs:element name="pptSlideMarkup" type="tns:ppt-dhtml-slide-markup-type"
ms:propertyName="PptDhtmlMarkupList" minOccurs="1" maxOccurs="unbounded"
ms:className="CXmlPptDhtmlMarkup"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ppt-dhtml-slide-markup-type" ms:className= "CXmlPptDhtmlMarkup">
        <xs:sequence>
            <xs:element name="pptDhtmlResource" type="tns:ppt-dhtml-resource"
ms:propertyName="PptDhtmlResourceList"  minOccurs="0" maxOccurs="unbounded"
ms:className="CXmlDhtmlResource"/>
        </xs:sequence>
        <xs:attribute name="slideNumber" type="xs:nonNegativeInteger"
ms:propertyName="SlideNumber" />
        <xs:attribute name="numClicks" type="xs:nonNegativeInteger" ms:propertyName="NumClicks"
/>
        <xs:attribute name="dhtmlMarkupPackageFile" type="xs:string"
ms:propertyName="DhtmlMarkupPackageFile" />
    </xs:complexType>

    <xs:complexType name="ppt-dhtml-resource" ms:className="CXmlPptDhtmlResource">
        <xs:attribute name="resourceFileName" type="xs:string"
ms:propertyName="ResourceFileName" />
    </xs:complexType>

    <xs:complexType name="ppt-dhtml-resources-package-type"
ms:className="CXmlPptDhtmlResourcesPackage">
        <xs:sequence>
            <xs:element name="pptDhtmlResource" type="tns:ppt-dhtml-resource"
ms:propertyName="PptDhtmlResourceList"  minOccurs="1" maxOccurs="unbounded"
ms:className="CXmlDhtmlResource"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ppt-jpeg-package-type" ms:className="CXmlPptJpegPackage">
        <xs:sequence>
            <xs:element name="pptJpegSlides" type="tns:ppt-jpeg-slide-type"
ms:propertyName="PptJpegSlideList" minOccurs="1" maxOccurs="unbounded"
ms:className="CXmlPptJpegSlide" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="ppt-jpeg-slide-type" ms:className="CXmlPptJpegSlide">
        <xs:attribute name="slideNumber"   type="xs:nonNegativeInteger"
ms:propertyName="SlideNumber"/>
        <xs:attribute name="jpegFile"  type="xs:string"  ms:propertyName="JpegFile" />
    </xs:complexType>

    <xs:complexType name="ppt-partial-ppt-package-type"
ms:className="CXmlPptPartialPptPackage">
        <xs:sequence>
            <xs:element name="slideNumber" type="xs:nonNegativeInteger"
ms:propertyName="SlideNumberList" minOccurs="1" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="partialPptFile"  type="xs:string"
ms:propertyName="PartialPptFile" />
    </xs:complexType>

    <xs:complexType name="ppt-full-ppt-package-type" ms:className="CXmlPptFullPptPackage">
        <xs:attribute name="fullPptFile"  type="xs:string"   ms:propertyName="FullPptFile" />
        <xs:attribute name="useAsNativeFileAlso" type="xs:boolean"
ms:propertyName="UseAsNativeFileAlso" />
    </xs:complexType>

    <!-- WhiteboardContent-specific schema -->
```

```
    <xs:element name="whiteboardContent" type="tns:whiteboard-content-detail-type"
ms:className="CXmlWhiteboardContentDetailDocument" />

    <xs:complexType name="whiteboard-content-detail-type"
ms:className="CXmlWhiteboardContentDetail">
      <xs:sequence>
        <xs:element name="whiteboardType" type="tns:whiteboard-type"
ms:propertyName="WhiteboardType" />
      </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="whiteboard-type" ms:className="CXmlWhiteboardType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="empty"/>
      </xs:restriction>
    </xs:simpleType>

    <!-- SharedNotesContent-specific schema -->
    <xs:element name="sharedNotesContent" type="tns:sharedNotes-content-detail-type"
ms:className="CXmlSharedNotesContentDetailDocument" />

    <xs:complexType name="sharedNotes-content-detail-type"
ms:className="CXmlSharedNotesContentDetail">
      <xs:sequence>
        <xs:element name="notesURL" type="xs:string" ms:propertyName="notesURL" minOccurs="0"
maxOccurs="1"/>
        <xs:element name="notesId" type="xs:string" ms:propertyName="notesId" minOccurs="0"
maxOccurs="1" />
        <xs:element name="notebookTitle" type="xs:string" ms:propertyName="notebookTitle"
minOccurs="0" maxOccurs="1"/>
        <xs:element name="sectionTitle" type="xs:string" ms:propertyName="sectionTitle"
minOccurs="0" maxOccurs="1"/>
        <xs:element name="pageTitle" type="xs:string" ms:propertyName="pageTitle" minOccurs="0"
maxOccurs="1"/>
      </xs:sequence>


    </xs:complexType>


    <!-- NativeFileOnlyContent-specific schema -->
    <xs:element name="nativeFileOnlyContent" type="tns:nativeFileOnly-content-detail-type"
ms:className="CXmlNativeFileOnlyContentDetailDocument" />

    <xs:complexType name="nativeFileOnly-content-detail-type"
ms:className="CXmlNativeFileOnlyContentDetail">
      <xs:sequence>
        <xs:element name="nativeFileOnlyType" type="tns:nativeFileOnly-type"
ms:propertyName="NativeFileOnlyType" />
      </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="nativeFileOnly-type" ms:className="CXmlNativeFileOnlyType">
      <xs:restriction base="xs:string">
        <xs:enumeration value="empty"/>
      </xs:restriction>
    </xs:simpleType>

    <!-- PollContent-specific schema -->
    <xs:element name="pollContent" type="tns:poll-content-detail-type"
ms:className="CXmlPollContentDetailDocument" />

    <xs:complexType name="poll-content-detail-type" ms:className="CXmlPollContentDetail">
      <xs:sequence>
        <xs:element name="question" type="xs:string" ms:propertyName="Question" />
        <xs:element name="choices" type="tns:poll-content-choices-type"
ms:propertyName="Choices" />
        <xs:element name="rememberPastUsers" type="xs:boolean"
ms:propertyName="RememberPastUsers" />
```

```
            <xs:element name="customizationXml" type="xs:string" ms:propertyName="CustomizationXml"
/>
      </xs:sequence>
    </xs:complexType>

    <xs:element name="choices" type="tns:poll-content-choices-type"
ms:className="CXmlPollContentChoicesDocument" />

    <xs:complexType name="poll-content-choices-type" ms:className="CXmlPollContentChoices">
      <xs:sequence>
        <xs:element name="choice" type="xs:string" ms:propertyName="PollContentChoiceList"
minOccurs="1" maxOccurs="unbounded" ms:className="CXmlPollContentChoices">
        </xs:element>
      </xs:sequence>
    </xs:complexType>

    <!-- QnaContent-specific schema -->
    <xs:element name="qnaContent" type="tns:qna-content-detail-type"
ms:className="CXmlQnaContentDetailDocument" />

    <xs:complexType name="qna-content-detail-type" ms:className="CXmlQnaContentDetail">
      <xs:sequence>
        <!-- There's no specific content needs to be specified here so far, will add if needed-
->
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

3.2.4.1.5 UploadStream

The **UploadStream** distributed object client uploads an OC package to the server. The client MUST call the **sWrite** method to send a data group of the OC package to the server. It MUST provide the byte array data, and a sequence number that denotes the data sent this time. The sequence number MUST be an incrementing number. The server can disconnect the client by the **cDisconnect** call.

3.2.4.1.5.1   Methods

cDisconnect: This method is called by the server to disconnect the upload stream distributed object.

cWriteComplete: This method is called by the server (2) to notify the client that a previously uploaded data group is received.

  ▪   **nBytes (Int32):** The number of bytes for the uploaded data group.

3.2.4.1.6 Content

3.2.4.1.6.1   Methods

The methods for the **Content** distributed object are as follows:

cConnectCompleted: This method is called when the server finishes setting up the initial state for the content. It does this by calling various other client-side calls to set initial properties.

cForceSync: This method is not used.

cSetCreationTime: This method tells the client what time the content was created on the server (2).

  ▪   **creationTime (string):** The time the content was created, in UTC time in string format.

cSetLastUsedTime: This method tells the client the time when the content was last used or presented on the server (2).

- **lastUsedtime (string):** The time the content was last used, in UTC time in string format.

cSetNativeFileInfo: If the content has a "native file", such as the original PowerPoint document, attached to it, this method is used to communicate the file name, **decryption** keys, hash, and file size.

- **fileName (string):** Name of the native file.

- **key (byte[]):** Byte array containing the decryption key.

- **iv (byte[]):** Byte array containing the initialization vector for decryption.

- **hash (byte[]):** Byte array containing the hash for this file.

- **fileSize (Int64):** The size of the file.

cSetOwnerId: This method informs the client of the **ContentUser** identifier of the owner, or the person who created and uploaded this content.

- **id (Int64):** The server-generated integer of the **ContentUser** identifier.

cSetPresentInfo: This method informs the client who the current active presenter is for this content, if any, and the current presented state. **ContentManager**'s **cSetActiveContent** MUST be sent in addition to this because it is more accurate.

- **presented (Boolean):** Whether or not this content is in a presented state.

- **presenterId (Int64):** A server-generated integer that represents the **ContentUser** identifier of the user who is the active presenter of this content.

cSetPresentationOrder: This method informs the client what the current order is for showing content. The highest number is the most recently presented content.

- **presentationOrder (Int64):** Presentation order. Zero ("0") if the content is not presented.

cSetTitle: This method informs the client what the title of the content is.

- **title (string):** Title of the content.

cSetTitleComplete: This method informs the client about the success or failure of an **sSetTitle** call.

- **status (Int32 - SetTitleStatus):** Status, either success or failure, of the **sSetTitle** call.

- **title (string):** Title that was requested for the content.

cSetVisibility: This method informs the client what the current visibility of this content is.

- **visibility (Int32 - ContentVisibility):** Value representing the visibility.

cSetOriginalFileUrl<17>: This method informs the client what the original file URL of the content is, if it is available. This MAY be used when the content file is on a shared location.

- **originalFileUrl (string):** Original File URL of the content.

cSetViewingUrl<18>: This method informs the client what the viewing URL of the content is, if it is available. This MAY be used when the content can be viewed via an URL.

- **viewingUrl (string):** Viewing URL of the content.

cSetRecordingUrl<19>: This method informs the client what the recording URL of the content is, if it is available. This MAY be used when the content can be recorded via an URL.

- **recordingUrl (string):** Recording URL of the content.

3.2.4.1.7 NativeFileOnlyContent

NativeFileOnlyContent's client interface is only called when the associated **Content** distributed object is logically connected.

3.2.4.1.7.1 Methods

cConnectCompleted: This method allows the server to communicate that the logical content distributed object connection is completed and that **cSetNativeFileInfo** was called previously. This method MUST be called once per client after all other **Content** distributed object calls are completed.

3.2.4.1.8 AnnotationContainer

cSetAnnotationConstraints MUST be called on the client indicating all the constraints that are going to be imposed on the server. After this, **cAddAnnotationBatch** MUST be called on the client with all of the pre-existing annotations.

3.2.4.1.8.1 Methods

cAddAnnotationBatch: This method is used by the server to send a batch of added annotations to the client.

- **ids (Int32 []):** Identifiers of the annotations added.

- **gens (Int32 []):** Current generation numbers for the annotations.

- **types (Int32 []):** Types of the annotations added.

- **ownerIds (Int64 []):** Identifiers of the owners of the annotations.

- **modifierIds (Int64[]):** Identifiers of the last modifiers of the annotations.

- **propertyCounts (Int32 [])**: Count of properties per annotation.

- **properties (String []):** Property names for the annotations. These MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12. This array can be larger than the other arrays. The **propertyCounts** MUST be used to figure out how many property names belong to each annotation.

- **values (String []):** Property values for the annotations. This array can be larger than the other arrays. The **propertyCounts** MUST be used to figure out how many property values belong to each annotation.

cChangePropertiesBatch: This method is used by the server (2) to send a batch of changed properties to the client.

- **ids (Int32 []):** Identifiers of the annotations changed.

- **gens (Int32 []):** Current generation numbers for the annotations.

- **modifierIds (Int64 []):** Identifiers of the modifiers of the annotations.

- **propertyCounts (Int32 []):** Count of properties per annotation.

- **properties (String []):** Property names for the annotations. These MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12. This array can be larger than the other arrays. The **propertyCounts** MUST be used to figure out how many property names belong to each annotation.

- **values (String []):** Property values for the annotations. This array can be larger than the other arrays. The **propertyCounts** MUST be used to figure out how many property values belong to each annotation.

cChangeTextBatch: This method is used by the server (2) to send a batch of text changes to the client. This method MUST only be called for text annotations.

- **ids (Int32 []):** Identifiers of the text annotations that have changed.

- **gens (Int32 []):** Current generation numbers for the annotations.

- **modifierIds (Int64 []):** Identifiers of the modifiers of the annotations.

- **textVersions (Int32 []):** Current text versions for the annotations.

- **deltaCounts (Int 32[]):** Count of deltas per annotation.

- **begins (Int 32[]):** Beginning positions of the text deltas. This array can be larger than the other arrays. The **deltaCounts** MUST be used to figure out how many deltas belong to each annotation.

- **ends (Int 32[]):** Ending positions of the text deltas. This array can be larger than the other arrays. The **deltaCounts** MUST be used to figure out how many deltas belong to each annotation.

- **characters (String []):** Characters in the text deltas. An empty string indicates that the delta is for a deleted sequence in the text. This array can be larger than the other arrays. The **deltaCounts** MUST be used to figure out how many deltas belong to each annotation.

cClearAnnotations: This method is used by the server (2) to indicate to the client that all the annotations have been cleared.

- **removerId (Int64):** Identifier of the remover.

cErrorAddAnnotation: This method is used by the server to indicate an error in adding an annotation. This method MUST only be called on the client that requested the **add** operation. The server MUST NOT send the "Telepointer" type or a non-supported type in the type parameter.

- **type (Int32):** Type of annotation.

The following table lists the values of **type.**

Value	Numeric value	Description
Drawing	"0"	Drawing annotation type
Text	"1"	Text annotation type
Image	"2"	Image annotation type
Telepointer	"3"	Telepointer annotation type

- **properties (String [][]):** Initial set of properties and values for the annotation that were sent to the server. These MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12.

- **errorCode (String):** Error code from the server.

cErrorChangeProperties: This method is used by the server (2) to indicate an error in changing properties for an annotation. The current values on the server (2) MUST be sent back to the client. This method MUST only be called on the client that requested the property changes.

- **id (Int32):** Identifier of the annotation.

- **gen (Int32):** Current generation number of the annotation.

- **modifierId (Int64):** Identifier of the last modifier.

- **properties (String [][]):** Current values of properties that were sent to the server. These MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12.

- **errorCode (String):** Error code from the server.

cErrorChangePropertyForGroup: This method is used by the server (2) to indicate an error in changing a property for a group of annotations. The current values on the server (2) for the property MUST be sent back to the client. This method MUST only be called on the client that requested the property changes.

- **ids (Int32 []):** Identifiers of the annotations.

- **gens (Int32 []):** Current generation numbers for the annotations.

- **modifierIds (Int64):** Identifier of the last modifier.

- **property (String):** The property that could not be changed. This MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12.

- **values (String []):** Current values on the server for the property for the group of annotations.

- **errorCode (String):** Error code from the server.

cErrorChangeText: This method is used by the server to indicate an error in changing the text property of a text annotation. This method MUST only be called on the client that requested the text change.

- **id (Int32):** Identifier of the text annotation.

- **gen (Int32):** Current generation number of the annotation.

- **modifierId (Int64):** Identifier of the last modifier.

- **errorCode (String):** Error code from the server.

cErrorClearAnnotations: This method is used by the server to indicate an error in clearing all the annotations to the client. This method MUST only be called on the client that requested the **clear** operation.

- **errorCode (String):** Error code from the server.

cErrorRemoveAnnotation: This method is used by the server to indicate an error in removing the annotation to the client. This method MUST only be called on the client that requested the **remove** operation.

- **id (Int32):** Identifier of the annotation.

- **errorCode (String):** Error code from the server.

cErrorRemoveAnnotations: This method is used by the server to indicate an error in removing a group of annotations to the client. This method MUST only be called on the client that requested the **remove** operation.

- **ids (Int32 []):** Identifiers of the annotations.

- **errorCode (String):** Error code from the server.

- **cookie (Int32):** Cookie that was passed to the server with the **remove** operation. This cookie is opaque to the server.

cErrorSetTelepointer: This method is used by the server to indicate an error in updating a **Telepointer** to the client. This method MUST only be called on the client that requested the **Telepointer** update.

- **errorCode (String):** Error code from the server.

cRemoveAnnotation: This method is used by the server (2) to indicate to the client that an annotation was removed.

- **id (Int32):** Identifier of the annotation that was removed.

- **removerId (Int64):** Identifier of the remover.

cRemoveAnnotations: This method is used by the server (2) to indicate to the client that a group of annotations were removed.

- **ids (Int32 []):** Identifiers of the annotations that were removed.

- **removerId (Int64):** Identifier of the remover.

- **cookie (Int32):** Cookie that was passed to the server with the **remove** operation. This cookie is opaque to the server.

cSetAnnotationConstraints: This method is used by the server (2) to send the constraints that are going to be imposed on the server (2).

- **constraints (Int32 []):** The constraints defined in the enumeration **AnnotationConstraint**.

- **values (Int32 []):** The values of the constraints.

cSetImageFileInfo: This method is used by the server (2) to set the image file information for an image annotation. The image annotation MUST exist on the client. This method MUST be called exactly once per client for each image annotation.

- **id (Int32):** The identifier of the annotation.

- **url (String):** URL of the image file resource. The image file pointed to by this URL MUST be AES256 encrypted, and can be decrypted by the **key** and **iv** parameters.

- **key (Byte []):** The AES256 key of the encrypted file pointed to by **url**.

- **iv (Byte []):** The initialization vector for **key**.

- **hash (Byte []):** The **SHA-1** hash of the file pointed to by **url**. The **hash** MUST be of the unencrypted file.

cSetTelepointer: This method is used by the server to indicate to the client that a **Telepointer** was updated.

- **anchor (String):** Anchor property that describes the location of the **Telepointer**.

- **ownerId (Int64):** Identifier of the owner.

- **visible (Boolean):** Visibility property that describes whether the **Telepointer** is visible.

3.2.4.1.9 WhiteboardContent

The **cConnectCompleted** method MUST be called exactly once per client.

3.2.4.1.9.1   Methods

cConnectCompleted: This method is used by the server to indicate to the client that its child distributed object is done connecting. This method has no parameters. The child distributed object MUST be connected before this call. This method MUST be called once per client.

3.2.4.1.10   PptContent

The client-side **PptContent** distributed object receives slide count, location, and resource information from the server.

3.2.4.1.10.1 Methods

The methods for the **PptContent** distributed object are as follows:

cConnectCompleted: This method allows the server to communicate that the current location, all slide info, and the thumbnail info have been sent. The **cSetSlideCount** and **cSetLocation** methods SHOULD have been called previously. This method SHOULD be called once per client.

cSetSlideCount: This method allows the server to communicate the number of slides in the **PptContent**.

- **slideCount (Int32):** The number of slides in the **PptContent**.

cSetLocation: This method allows the server to communicate a new location value for the **PptContent**. This method SHOULD be called on all clients in response to a successful **sSetLocation** call from a client.

- **location (String):** New location information to be set for **PptContent**. MUST conform to **PptLocation** schema in section 3.1.4.1.11.

cSetPreviewThumbnailFileInfo: This method allows the server to communicate to clients resource information for the preview thumbnail of the **PptContent**. This method SHOULD be called once per client.

- **url (String):** URL of the preview thumbnail. The file that **url** points to MUST be AES256 encrypted and can be decrypted with **key** and **iv**.

- **key (Byte[]):** The AES256 key of the file pointed to by **url**.

- **iv (Byte[]):** The input vector of **key**.

- **hash (Byte[]):** The SHA-1 hash of the file pointed to by **url**. The hash MUST be of the file in its unencrypted form.

cSetThumbnailFileInfo: This method allows the server to communicate to clients the resource information for the slide thumbnails of the **PptContent**. This method SHOULD be called once per client, and only if there are thumbnail resources uploaded for the **PptContent**.

- **files (String[]):** File names of the thumbnail files that are in the Open Package Convention file pointed to by **url**. File names MUST be in the order of the slide that they refer to.

- **url (String):** URL of the Open Package Convention file containing all slide thumbnails for the **PptContent**. The file that **url** points to MUST be AES256 encrypted and can be decrypted with **key** and **iv**.

- **key (Byte[]):** The AES256 key of the file pointed to by **url**.

- **iv (Byte[]):** The input vector of **key**.

- **hash (Byte[]):** The SHA-1 hash of the file pointed to by **url**. The hash MUST be of the file in its unencrypted form.

cSetPartialPptFileInfo: This method allows the server (2) to communicate to clients the resource information for the partial PowerPoint file. A partial PowerPoint file is a PowerPoint 97-2003 Presentation file that includes some or all of the slides in the presentation.

- **url (String):** URL of the Open Package Convention file containing the partial PowerPoint file. The file that **url** points to MUST be AES256 encrypted and can be decrypted with **key** and **iv**.

- **slideNumbers (Int32[]):** The slide numbers that are contained in the partial PowerPoint file.

- **key (Byte[]):** The AES256 key of the file pointed to by **url**.

- **iv (Byte[]):** The input vector of **key**.

- **hash (Byte[]):** The SHA-1 hash of the file pointed to by **url**. The hash MUST be of the file in its unencrypted form.

cSetNotesInfo: This method allows the server to communicate to clients resource information for the slide notes of the **PptContent**.

- **slideNumbers (Int32[]):** The slide numbers that are contained in the Open Package Convention file containing the notes.

- **url (String):** URL of the Open Package Convention file that contains the slide notes. The file that **url** points to MUST be AES256 encrypted and can be decrypted with **key** and **iv**.

- **key (Byte[]):** The AES256 key of the file pointed to by **url**.

- **iv (Byte[]):** The input vector of **key**.

- **hash (Byte[]):** The SHA-1 hash of the file pointed to by **url**. The hash MUST be of the file in its unencrypted form.

cSetSlideDhtmlResourceInfo: This method allows the server to communicate to clients the resource information for the **Dynamic Hypertext Markup Language (DHTML)** resource for a slide. These resources can be shared between markup resources.

- **resourceOriginalFileName (String):** Unique name corresponding to a shared asset or resource. This can be referred to by one or more **cSetSlideDHtmlMarkupInfo** calls.

- **url (String):** URL for the Open Package Convention file that contains the DHTML resource for the resource. The file that **url** points to MUST be AES256 encrypted and can be decrypted with **key** and **iv**.

- **key (Byte[]):** The AES256 key of the file pointed to by **url**.

- **iv (Byte[]):** The input vector of **key**.

- **hash (Byte[]):** The SHA-1 hash of the file pointed to by **url**. The hash MUST be of the file in its unencrypted form.

cSetSlideDhtmlMarkupInfo: This method allows the server (2) to communicate to clients the resource information for the DHTML resource for a slide.

- **slideNumber (Int32):** Slide number that resources in **url** are for.

- **clickCount (Int32):** The number of animations on the slide.

- **markupUrl (String):** URL for the Open Package Convention file that contains the DHTML resources for the slide. The file that **url** points to MUST be AES256 encrypted and can be decrypted with **key** and **iv**.

- **resourceOriginalFileNames (String[]):** Array of **DhtmlResources** that are associated with this slide. Each entry MUST correspond to a corresponding call from the **cSetSlideDhtmlResourceInfo**'s **resourceOriginalFileName** parameter.

- **key (Byte[]):** The AES256 key of the file pointed to by **url**.

- **iv (Byte[]):** The input vector of **key**.

- **hash (Byte[]):** The SHA-1 hash of the file pointed to by **url**. The hash MUST be of the file in its unencrypted form.

cSetFullPptFileInfo: This method allows the server to communicate to clients the resource information for the PowerPoint 97-2003 format file of the presentation containing all slides of the **PptContent**. This method SHOULD be called once per client and only if the resource is available.

- **url (String):** URL of the Open Package Convention file containing the PowerPoint 97-2003 format file that has all slides of the **PptContent**. The file that **url** points to MUST be AES256 encrypted and can be decrypted with **key** and **iv**.

- **key (Byte[]):** The AES256 key of the file pointed to by **url**.

- **iv (Byte[]):** The input vector of **key**.

- **hash (Byte[]):** The SHA-1 hash of the file pointed to by **url**. The hash MUST be of the file in its unencrypted form.

cSetSlideJpegInfo: This method allows the server (2) to communicate to clients the resource information for the Jpeg resource for a slide.

- **slideNumber (Int32):** Slide number that resources in **url** are for.

- **url (String):** URL for the Open Package Convention file that contains the Jpeg resources for the slide. The file that **url** points to MUST be AES256 encrypted and can be decrypted with **key** and **iv**.

- **key (Byte[]):** The AES256 key of the file pointed to by **url**.

- **iv (Byte[]):** The input vector of **key**.

- **hash (Byte[]):** The SHA-1 hash of the file pointed to by **url**. The hash MUST be of the file in its unencrypted form.

cErrorAddingResource: This method allows the server (2) to communicate to the uploading client that there was a server-side failure while adding a resource. This method SHOULD be called on the client that attempted to upload the particular resource.

- **resourceId (Int32):** Resource identifier of the resource for which the error occurred.

- **errorCode (Int32):** Error code for the failure. MUST be one of the values specified in the **ResourceErrorCode** type in section 3.1.4.1.11.

cSetResourceErrorCode: This method allows the server to communicate to clients that there was an error with adding a particular resource and that the resource is not available.

- **beginSlide (Int32):** The beginning index, inclusive, of the slide range for the error. MUST be greater than or equal to zero, less than the number of slides for the **PptContent**, and less than or equal to **endSlide**.

- **endSlide (Int32):** The ending index, inclusive, of the slide range for the error. MUST be greater than or equal to zero, less than the number of slides for the **PptContent**, and greater than or equal to **beginSlide**.

- **format (Int32):** The format of the resource generating the error. MUST be one of the values specified in the **ResourceFormat** enumeration in section 3.1.4.1.11.

- **errorCode (Int32):** Error code for the failure. MUST be one of the values specified in the **ResourceErrorCode** enumeration in section 3.1.4.1.11.

3.2.4.1.11   PollContent

The **PollContent** client distributed object is able to receive state, the local user's vote, and the aggregate sum of responses for the choices. Both the client and server can choose to enforce maximum length limits on any strings or arrays.

3.2.4.1.11.1 Methods

The methods for the **PollContent** distributed object are as follows:

cSetQuestion: This method allows the server to communicate the poll's question and choices.

- **question (String):** The text of the question.

- **choices (String[]):** An array with length greater than 1 and less than or equal to 7. Each element represents the text of a given choice. The ordinal positions of each array element are held consistent across all other calls. For example, position zero corresponds to position zero in the results array in **cSetResults**.

- **rememberPastUsers (Boolean):** This is provided for future extensibility. MUST be "true".

- **customizationXml (String):** This is provided for future extensibility. MUST be an empty string.

cSetResults: This method allows the server (2) to communicate the sum of the votes for a given choice. If the results are not visible to the client, this MUST be an array of all zeros.

- **results (Int32[]):** Array of integer vote sums. The length MUST be either the maximum allowed number of choices, which is 7, or the number of choices used in **cSetQuestion**. If the results are not visible to a given client, this MUST be an array of all zeros; otherwise it MUST be non-negative sums of the number of users that voted for a given choice.

cSetOpenState: This method allows the server to communicate whether or not clients can call **sVote**.

- **open (Boolean):** When "true", all clients can call **sVote**. When "false", clients SHOULD NOT call **sVote**. If they do, the server MUST respond with **cSetLocalVote** with the value of the client's vote that was set when **cSetOpenState**, which means that **open** equals "true".

cSetResultsState: This method allows the server to communicate whether or not results are shown to a subset of attendees.

- **visibleToAll (Boolean):** When "true", all clients can view the results. If "false", only presenters can view the results.

cSetLocalVote: This method communicates the client's local vote. This MUST be persisted by the server (2) if the client, as the same user, ever connects to this distributed object and had a past vote.

- **choice (Int32):** The value of the user's vote. A value of "-1" represents the absence of a vote.

cConnectCompleted: This method allows the server to communicate that the current location, all slide information, and the thumbnail info have been sent. The **cSetQuestion, cSetOpenState, cSetResultsState,** and **cSetLocalVote** methods SHOULD have been called previously. This method SHOULD be called once per client.

3.2.4.1.12   SharedNotesContent

This section follows the product behavior described in endnote[20].

The client-side **SharedNotesContent** distributed object receives notes identifier and title information from the server.

Note that the notes URLs are received from the server through the **Content** distributed object methods **cSetOriginalFileUrl** and **cSetViewingUrl**.

3.2.4.1.12.1 Methods

The methods for the **SharedNotesContent** distributed object are as follows:

cConnectCompleted: This method allows the server to communicate that the notes Id and title information have been sent. The **cSetNotesId** and **cSetNotesTitles** methods SHOULD have been called previously. This method SHOULD be called once per client.

cSetNotesId: This method allows the server to communicate the notes Id.

- **notesId (String):** The notes Id.

cSetNotesTitles: This method allows the server to communicate the notes title information.

- **notebookTitle (String):** The title of the notebook.

- **sectionTitle (String):** The title of the section.

- **pageTitle (String):** The title of the page.

cSetErrorCode: This method allows the server to communicate notes related errors.

- **code (Int32):** The error code. The only allowed value is 1 denoting an unknown error.

3.2.4.1.13   QnaContent

The client-side **QnaContent** distributed object receives number of questions, open state of the questions and answers content, and opaque BLOBs from the server to be handled to the web page that implements the view for the questions and answers content (indicated by cSetViewingUrl method).

3.2.4.1.13.1 Methods

The methods for the **QnaContent** distributed object are as follows:

cConnectCompleted: This method allows the server to communicate that the open state and the number of questions have been sent. The **cSetOpenState** and **cSetQuestionsCount** methods SHOULD have been called previously. This method SHOULD be called once per client.

cSetOpenState: This method allows the server to communicate the open state of the **QnaContent**.

- **openState (Int32):** The new open state of the questions and answers content. MUST conform to **QnaOpenState** schema in section 3.1.4.1.16.

cSetQuestionsCount: This method allows the server to communicate the number of questions in **QnaContent**.

- **count (Int32):** Number of questions

cPutBlob: This method allows the server (2) to communicate the opaque BLOB to the web page.

- **blob (String):** Opaque BLOB to be posted to the web page

3.2.4.1.14   SharedLinksContent

The client-side **SharedLinksContent** distributed object received the URLs and status of the file to be shared between meeting end points.

3.2.4.1.14.1 Methods

The methods for the **SharedLinksContent** distributed object are as follows:

cConnectCompleted: This method allows the server to communicate that the document content state is ready and has been sent. The **cSetState** method SHOULD have been called previously. This method SHOULD be called once per client.

cSetViewURL: Not in use.

cSetEditURL: Not in use.

cSetActive: Not in use.

cSetInActive: Not in use.

cSetUniqueId: Not in use.

cSetState: This method takes the following parameters.

       string **contentTitle**: Document name.

       string **viewURL**: the URL to the document for viewing purpose.

       string **editURL**: the URL to the document for editing purpose

       string **unigueId**: GUID to identify each document.

       bool **active**:   the document active status. True means is it currently in editing session.

       int32[] **msDiagIds**: IDs for diagnostic purpose.

3.2.5 Message Processing Events and Sequencing Rules

This protocol's lower layer protocol description does not require a particular ordering of messages. However, a higher layer application can require this, but it is dependent on the particular message received. This is highlighted for each component. The client MUST process any incoming message in the order that it is received. Sometimes user input is required prior to sending a response message, if required. In other cases, an immediate response is required.

3.2.6 Timer Events

Because this protocol requires a lossless transport, there are no time-outs. No higher-layer timers exist that are specific to the client. See section 3.1.6 for timers that are common between the client and the server.

3.2.7 Other Local Events

In the event that a connection is terminated by the server, the client can choose to re-establish the connection by starting the connection process from the beginning.

3.3 Server Details

The life-cycle stages of the server are specified in section 3.2. The server is responsible for some conference states and authorizing clients. At the messaging layer, there are minimal differences. See section 3.3.4 for details on specific server functionality.

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 server is composed of a set of higher-layer messages built on the PSOM messaging layer. Logically, the server holds all conference states and allows clients to interact with those states, notifying other clients of any changes that take place.

3.3.2 Timers

There is a **ping** method called, with no parameters, that is used as a keepalive message that is sent every 30 seconds by the server on the **ConnMgr** distributed object. For the **ConnMgr** definitions, see section 3.3.3.1.3. When the timer expires, a new one is set at the same interval period.

3.3.3 Initialization

When this protocol is initialized, it MUST begin connecting to a server or a client, or wait for incoming connections from a server. The protocol has no state at this point, and no distributed objects are connected. At a higher layer, connection is established. This is followed by authentication, versioning, and root distributed object connection.

3.3.3.1 Connections

This section describes the steps required for a server to successfully negotiate a connection with a client. Connection entails three main components:

- Authentication.

- Interface versioning.

- Root distributed object PSOM channel negotiation.

See sections 3.2.3.1.1 and 3.2.3.1.2 for additional details about authentication and interface versioning because there are required responses or actions for each message.

3.3.3.1.1 Authentication

The server MUST generate the appropriate **AddUser** response fields to an authorized user, as defined in [MS-CONFBAS] section 2.2.3.18. Using these details, the client establishes a connection to the server. The server MUST ensure that the **sAuthId** (as defined in [MS-CONFBAS] section 2.2.3.18) and any other token presented during this phase match the one presented to the user via **AddUser**. If the

client does not do this within 120 seconds or does it with an invalid value, the server (2) MUST immediately terminate the incoming connection.

3.3.3.1.2 Interface Versioning

After authentication is completed, the server MUST exchange with the client hashes of each distributed object version that it supports. Because the root PSOM channel, channel zero, is set up implicitly, a special distributed object is used to exchange versioning information messages. This distributed object is referred to as **ConnMgr** and handles exchanging the supported list of distributed objects that are to be connected on PSOM channel 2 at a later time. If at any time the version hashes do not match, the server MUST terminate the connection immediately. A **break** message can be sent prior to closing the connection.

As the server receives the **addProtocol** messages from the client, it MUST validate that the hashes match its own definitions. If the server determines that the hashes are not equal, it MUST terminate the connection with a **close** or **break** message. If the server determines that the hashes are equal, it MUST repeat the same sequence of RPC messages to the client, including its own hash information instead. When the server sends **doneProtocols** to the client, and has received **doneProtocols** from the client after exchanging the hashes, the versioning phase is complete. At this point, the connection is established, and the server (2) validates that it has protocol versions compatible with the client.

3.3.3.1.3 ConnMgr Distributed Object Interface Definition

```
[Name="Microsoft.Rtc.Server.DataMCU.Meeting.Pod.ConnMgr", Version=1]
DOInterface ConnMgr
{
    [Hash=-8221414758688209204L]
    ServerInterface
    {
        void version(Int64 stubHash);

        void addProtocol(String name, Int32[] versions, Int64[] hashes);

        void doneProtocols();

        void log(String msg);

        void lookup(String name, String protocol, Int64 proxyHash);

        void ping();
    }
}
```

3.3.3.1.3.1   ConnMgr Server Methods

3.3.3.1.3.1.1 version

The first distributed object method call. This is used to communicate the overall client interface **ConnMgr** hash to the server.

stubHash (Int64): The hash code of the **ConnMgr ClientInterface**.

3.3.3.1.3.1.2 addProtocol

Allows the client to communicate the name, versions, and hashes for a given distributed object interface. Some implementations MAY send multiple versions. Only the versions/hashes specified in the **DOInterface** attributes SHOULD be sent, other values MUST be ignored. This method MUST NOT be called after **doneProtocols** is sent.

name (String): The name of the **DOInterface**, acquired from the attribute **Name**.

versions (Int32[]): A list of versions.

hashes (Int64[]): A list of summed hashes for the corresponding client and server interface versions, or client interface hash value plus server interface hash value.

3.3.3.1.3.1.3 doneProtocols

The **doneProtocols** method is received when all **addProtocol** calls are completed. This signals that the client is done sending protocol versions with **addProtocol**.

3.3.3.1.3.1.4 log

Deprecated. The **log** method does nothing.

msg (String): No meaning.

3.3.3.1.3.1.5 Lookup

Lookup is a special method call. It is used as part of the **rpcOpen record** message to indicate that a new PSOM channel is to be created.

name (String): Deprecated. Value is ignored.

protocol (String): Deprecated. Value is ignored.

proxyHash (Int64): Deprecated. Value is ignored.

3.3.3.1.3.1.6 ping

This is used as a keep-alive mechanism. It is a no-op.

3.3.4 Higher-Layer Triggered Events

This section details application PSOM channel events.

3.3.4.1 Distributed Objects

The following sections specify higher-layer triggered events for distributed objects.

3.3.4.1.1 Meeting

The server piece of the **Meeting** distributed object MUST send distributed object connect messages for its child objects, **ContentUserManager** and **ContentManager**, immediately after connection. It MUST also call **cSetUrlBase** followed by **cMeetingReady** to complete the logical connection sequence.

3.3.4.1.1.1   Methods

sSetInfo: MUST NOT be called. This method is not currently supported.

▪ **info (String):** Not defined.

3.3.4.1.2 ContentUserManager

No methods exist on the server interface.

3.3.4.1.2.1   Methods

None.

3.3.4.1.3 ContentManager

The server MUST respond to **sReserveTitle** with **cReserveTitleCompleted**.

For active presenter calls such as **sPresent**, which allows the client to become the active presenter, or **sStopPresenting**, which gives up the active presenter role, the server only needs to indicate a change via **cSetActivePresenter**. If there is no change, such as when the request is not honored because of permissions or because it would have no effect, no response is required.

3.3.4.1.3.1   Methods

sDeleteContent: This method requests that the server delete the content.

▪   **contentId (Int64):** The server-generated integer that uniquely identifies the content to be deleted.

sPresent: Claims the active presenter role. If a piece of content is currently being presented, that content is no longer presented.

sReleaseTitle: This method is called by clients to release a title reservation.

▪   **cookie (Int32):** The client-generated integer that was sent as part of a previous **sReserveTitle** request.

sReserveTitle (1): This method is called by clients to reserve a content title.

▪   **title (String):** The title to reserve.

▪   **cookie (Int32):** A client-generated integer that the clients can use to track the completion of this request.

sReserveTitle (2): This method is deprecated.

▪   **title (String):** Deprecated.

▪   **cookie (Int32):** Deprecated.

▪   **externalId (String):** Deprecated.

sStopPresenting: If the requesting client is the current active presenter, relinquishes the stage.

3.3.4.1.4 UploadManager

When the server receives a request from the client, it MUST respond with **cRejectUpload** or **cAcceptUpload**. In the **cRejectUpload** method, the server MUST provide the rejection reason. If the server accepts the upload, it MUST call the client's **cAcceptUpload** method, and it MUST provide the upload **cookie** and the distributed object **UploadStream**. The client can then use the **UploadStream** to send bytes that represent the package to the server.

When the server receives **sUploadFinished**, it can parse the contents of the uploaded file and act appropriately.

The client can, at any time during the upload, cancel the upload. To do this, the client MUST call **sUploadFinished** with the **cancel** flag set to "true". If the client did not upload all of the file's bytes, the server MUST call the client's **cUploadFinished** method, with the upload finished reason.

3.3.4.1.4.1   Methods

sRequestUpload: This method is used for server to server interactions as an alternate method for upload initiation, and SHOULD NOT be called by a client.

- **fileLength (Int64):** The size of the content file to be uploaded.

- **cookie (Int32):** The unique cookie value that corresponds to the content to be created.

- **manifestXml (String):** The XML placed in **OcpManifest.xml**.

sRequestUpload: This method is called by a client to initiate the upload of an OC package. The client MUST pass the packed and unpacked size of the package file to be uploaded, and the cookie associated with the content to be created. This cookie is used to identify various uploads. After this call, the client MUST wait for a response from the server, as described in section 3.2.4.1.4.1.

- **packedLength (Int64):** The packed size of the content package file to be uploaded.

- **unpackedLength (Int64):** The unpacked size of the content package file to be uploaded.

- **cookie (Int32):** The unique cookie value that corresponds to the content to be created.

sUploadFinished: This method MUST be called by the client upon completion of the upload and provide the same cookie as provided in **sRequestUpload**. This covers the cases where the client uploaded all the data and where the client cancels the upload.

- **cookie (Int32):** The unique cookie value that corresponds to the content.

- **cancel (Boolean):** When "false", the user uploaded all the data. When "true", the server MUST cancel the upload.

sRequestWebUploadBlob<21>: This method is called by a client to initiate the upload of a single file using HTTPS Form **POST** request. The client MUST pass the OC package manifest content and the cookie associated with the content to be created. This cookie is used to identify various uploads. After this call, the client MUST wait for a response from the server, as described in section 3.2.4.1.4.1.

- **cookie (Int32):** The unique cookie value that corresponds to the content to be created.

- **manifestXml (String):** The XML placed in OcpManifest.xml file of OC package.

sRequestWebDownloadBlob<22>: This method is primarily for web clients to download a single file over HTTPS with decrypted content. Used whenever the client does not support decryption. After this call, the client MUST wait for a response from the server, as described in section 3.2.4.1.4.1

- **cookie (Int32):** The unique cookie value that corresponds to the content.

- **obfuscatedName(String):** The original encrypted file name sent from the server, to the client using the **cSetNativeFileInfo** method, as described in section 3.2.4.1.6.1

- **isAttachment (Boolean):** True if the server should put the following HTTP response headers when delivering the file to the client:

    ContentType: application/x-download

    Content-Disposition: attachment; filename="filename"

3.3.4.1.5 UploadStream

When the server receives a group of data that was sent by the client, it acknowledges the client by calling the **cWriteComplete** method. The server MUST provide the received data size of the **sWrite** call it received.

3.3.4.1.5.1   Methods

sDisconnect: This method is called by the client to disconnect the **UploadStream** distributed object from the server.

sWrite: This method is called by the client to write a group of data to the server (2). The client MUST provide the data group as a byte array and a sequence number for the write, which MUST be an incrementing number.

▪ **Data (byte[]):** The byte array is the data group to be uploaded.

▪ **packetNum (Int32):** The sequence number of the data group to be uploaded.

3.3.4.1.6 Content

The server MUST call **cSetTitleComplete** to inform clients when an attempt to rename the content via **sSetTitle** either succeeded or failed.

For active presenter calls such as **sPresent**, to become the active presenter, or **sStopPresenting**, to give up the active presenter role, the server only needs to indicate a change via **cSetPresentInfo** or ContentManager's **cSetActiveContent**. If there is no change, such as when the request is not honored because of permissions or because it would have no effect, no response is required.

3.3.4.1.6.1   Methods

sForceSync: This method is not used.

sMakeHighestPresentationOrder: This method makes an already-presented content become the highest presentation order.

sPresent: This method claims the active presenter role for this piece of content. If it was not already presented, it becomes presented.

sSetTitle: This method requests the title be changed.

▪ **title (string):** The title to use to rename the content.

sSetVisibility: This method sets the visibility for this content.

▪ **visibility (Int32 - ContentVisibility):** The visibility to set the content.

sStopPresenting: If the requesting client is the active presenter for this content, this method stops presenting the content and removes the client from the active presenter role.

3.3.4.1.7 NativeFileOnlyContent

NativeFileOnlyContent's server interface has no messages defined. Messages MUST NOT be sent.

3.3.4.1.7.1   Methods

There are no methods defined.

3.3.4.1.8 AnnotationContainer

The server MUST respond to client requests for adding, removing and modifying annotations. It MUST respond by broadcasting the **add** or **remove** or **modify** to all the clients, and in the case of errors, it MUST respond with an error response to the originating client.

3.3.4.1.8.1   Methods

sAddAnnotation: This method is a request to the server to add an annotation of the specified type, using the specified properties. The server MUST validate the type and properties, and if valid, it MUST

add this information to a collection for tracking purposes, because other operations involving this annotation can be sent to the server. At this point, a unique identifier MUST be assigned to the annotation that clients can use to refer to the annotation.

If there is an error while processing this operation, an error response MUST be sent to the client using the **cErrorAddAnnotation** method. If the annotation was successfully added, a notification response MUST be sent to all the clients using the **cAddAnnotationBatch** method. The server can batch all such notifications and send them in batches from time to time.

Only "Drawing", "Text", and "Image" type annotations can be added to the container. The client MUST NOT send an **add** operation for a "Telepointer" annotation, or any non-supported type.

- **type (Int32):** Type of annotation to add.

Value	Meaning
0	Drawing
1	Text
2	Image
3	Telepointer

- **properties (String [][]):** Initial set of properties and values for the annotation. These MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12.

sChangeProperties: This method is a request to the server to change an existing annotation's properties. If the annotation with the specified **id** is not found, the server MUST send an error response to the client using the **cErrorChangeProperties** method. If an invalid generation number or properties are sent, the server SHOULD disconnect the client. If valid parameters are received, the generation number MUST be incremented, and the changed properties MUST be sent to all the clients using the **cChangePropertiesBatch** method.

- **id (Int32):** Identifier of the annotation to change.

- **gen (Int32):** Current generation number.

- **properties (String [][]):** Properties to change. This is a two-dimensional array of property names and property values. These MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12.

sChangePropertyForGroup (1): This method is a request to the server to change a property for a group of existing annotations. The specified property has the same value for all the annotations in the group. If any of the parameters are invalid, the server SHOULD disconnect the client. If none of the annotations are found, the server MUST send an error response to the client using the **cErrorChangePropertyForGroup** method. If individual annotations are not found, the server MUST send an error response to the client using the **cErrorChangeProperties** method. If valid parameters are received and the annotations are found, the generation numbers MUST be incremented, and the changed property MUST be sent to all the clients using the **cChangePropertiesBatch** method.

- **ids (Int32 []):** Identifiers of the annotations to change.

- **gens (Int32 []):** Current generation numbers for the annotations.

- **property (String):** The property that needs to be changed. This MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12.

- **value (String):** The new value of the property.

sChangePropertyForGroup (2): This method is a request to the server to change a property for a group of existing annotations. The specified property can have a different value for each of the annotations in the group. If any of the parameters are invalid, the server SHOULD disconnect the client. If none of the annotations are found, the server MUST send an error response to the client using the **cErrorChangePropertyForGroup** method. If individual annotations are not found, the server MUST send an error response to the client using the **cErrorChangeProperties** method. If valid parameters are received and the annotations are found, the generation numbers MUST be incremented, and the changed property MUST be sent to all the clients using the **cChangePropertiesBatch** method.

- **ids (Int32 []):** Identifiers of the annotations to change.

- **gens (Int32 []):** Current generation numbers for the annotations.

- **property (String):** The property that needs to be changed. This MUST be from the property names specified in the Annotation Properties table in section 3.1.4.1.12.

- **values (String []):** The new values of the property.

sChangeText: This method is a request to the server to change the text property of a text annotation. This method MUST only be called for text annotations. The changes MUST be sent in the form of text deltas. If the annotation is not found, the server MUST send an error response to the client using the **cErrorChangeText** method. If the text version is stale, the server MUST reject the change but, because this is not an error, an error response MUST NOT be sent back to the client. If the text deltas are invalid, the server SHOULD disconnect the client. If valid deltas are received and the annotation is found and the text version is not stale, the generation number MUST be incremented, and the deltas MUST be sent to all the clients using the **cChangeTextBatch** method.

- **id (Int32):** Identifier of the text annotation to change.

- **gen (Int32):** Current generation number.

- **textVersion (Int32):** Text version that the change is based on.

- **begins (Int32 []):** Beginning positions of the text deltas.

- **ends (Int32 []):** Ending positions of the text deltas.

- **characters (String []):** Characters in the text deltas. An empty string indicates that the delta is for a deleted sequence in the text.

sClearAnnotations: This method is a request to the server to remove all the annotations. If there are no annotations to clear, an error response MUST be sent to the client using the **cErrorClearAnnotations** method. If the annotations are successfully cleared, the server MUST send the **clear** operation to all the clients using the **cClearAnnotations** method. This method has no parameters.

sRemoveAnnotation: This method is a request to the server to remove an annotation. If the annotation is not found, an error response MUST be sent to the client using the **cErrorRemoveAnnotation** method. If the operation is successful, the server MUST send the **remove** operation to all the clients using the **cRemoveAnnotation** method.

- **id (Int32):** Identifier of the annotation to remove.

sRemoveAnnotations: This method is a request to the server to remove a group of annotations. If there are no annotations to remove, an error response MUST be sent to the client using the **cErrorRemoveAnnotations** method. If any annotations are removed successfully, the server MUST send a **remove** operation with the identifiers of the annotations that were actually removed to all the clients, using the **cRemoveAnnotations** method.

- **ids (Int32 []):** Identifiers of the annotations to remove.

- **cookie (Int32):** Cookie sent by the client. This cookie is opaque to the server.

sSetTelepointer: This method is a request to the server to set the **Telepointer**'s properties. If the **Telepointer** is visible, and if it was previously added to the server's collection, it MUST be updated. Otherwise, a new **Telepointer** MUST be created and added to the server's collection for tracking purposes. This **Telepointer**'s information MUST be sent to all the clients using the **cSetTelepointer** method.

If the **Telepointer** is not visible, it MUST be removed from the server's collection, and the information MUST be sent to all clients using the **cSetTelepointer** method.

- **anchor (String):** Anchor property that describes the location of the **Telepointer**.

- **visible (Boolean):** Visibility property that describes whether the **Telepointer** is visible.

3.3.4.1.9 WhiteboardContent

No methods exist on the server interface.

3.3.4.1.9.1   Methods

None.

3.3.4.1.10   PptContent

The server-side **PptContent** distributed object is responsible for receiving and fanning out changes to the **PptLocation**, and alerting clients of resource availability and errors. Resources that are uploaded through **UploadManager** for this particular content have availability and status communicated out to clients via client-side methods such as **cSetFullPptFileInfo** or **cSetResourceErrorCode**.

3.3.4.1.10.1 Methods

sSetLocation: This method allows a client to attempt to change the location of the **PptContent**.

- **location (String):** New location information to be set for **PptContent**. MUST conform to the **PptLocation** schema in section 3.1.4.1.11 or be the value "INITIAL_LOCATION". When the **PptContent**'s location is successfully changed, the server SHOULD call **cSetLocation** with the new location on all clients.

sSetResourceErrorCode: This method allows the client to communicate to the server that there was an error while adding a particular resource, and that the resource is not available. The server SHOULD call **cSetResourceErrorCode** for each client with the same information, in response to **sSetResourceErrorCode**.

- **beginSlide (Int32):** The beginning index, inclusive, of the slide range for the error. MUST be greater than or equal to zero, less than the number of slides for the **PptContent**, and less than or equal to **endSlide**.

- **endSlide (Int32):** The ending index, inclusive, of the slide range for the error. MUST be greater than or equal to zero, less than the number of slides for the **PptContent**, and greater than or equal to **beginSlide**.

- **format (Int32):** The resource format that the error is for. MUST be one of the values specified in the **ResourceFormat** enumeration in section 3.1.4.1.11.

- **errorCode (Int32):** Error code for the failure. MUST be one of the values specified in the **ResourceErrorCode** enumeration in section 3.1.4.1.11.

3.3.4.1.11   PollContent

The server-side **PollContent** distributed object is responsible for receiving votes, state updates, question changes, and choice changes and fanning out changes to selected clients that are allowed to see that information. The server SHOULD persist votes and result counts across conferences. The server SHOULD enforce certain limits on vote counts and question length. For example, the server can either ignore calls or send back a distributed object call that reflects the state the server determines all clients ought to have.

Methods such as **cSetQuestion** and **cSetOpenState** are shared objects that clients with sufficient permission can modify. They are used to share the value of a given distributed property.

3.3.4.1.11.1 Methods

The methods for the **PollContent** distributed object are as follows:

sClearVotes: This method instructs the server to clear all existing votes for the poll. When the server receives this call, the server MUST send updated results to all users that can receive them, using **cSetResults**.

sModify: This method allows clients with permission to change the existing question and choices. When this is called and the server determines that the client can make changes and that the values are within appropriate ranges, it MUST send **cSetQuestion** messages to all clients.

- **question (String):** The text of the poll's question.

- **choices (String[]):** The choices for the poll. Each position corresponds to the choice value specified in **sVote**.

- **rememberPastUsers (Boolean):** MUST be "true". Provided for future extensibility.

- **customizationXml (String):** MUST be an empty string. Provided for future extensibility.

sSetOpenState: This method allows clients with sufficient permission to modify whether or not the poll is open. **cSetOpenState** MUST be sent to all clients if the value changes.

- **open (Boolean):** "True" if all users are able to vote by calling **sVote**. "False" otherwise.

sSetResultsState: This method allows clients with sufficient permission to modify whether or not attendees can view results.

- **visibleToAll (Boolean):** When "true", all attendees can view results. **cSetResults** MUST be sent to clients with updated results. When "false", only presenters can view results. **cSetResults** MUST be sent to clients that are no longer able to view results and the accompanying results array MUST contain zeros for all values.

sVote: This method allows clients to vote for a given choice.

- **choice (Int32):** A valid vote is a value from zero ("0") up to the length of the **choices** array in **cSetQuestion**, inclusive. A value of "-1" indicates a lack of a vote. When a user votes for choice zero, the value communicated in the **cSetResults** array position zero SHOULD increase by 1. If the poll is closed, this SHOULD NOT change result values. The server MUST acknowledge client votes by sending **cSetLocalVote** back to the client with the recorded vote value. If the poll is closed, it SHOULD send the value stored before **sVote** was called by that given client.

3.3.4.1.12   SharedNotesContent

This section follows the product behavior described in endnote<23>.

The server-side **SharedNotesContent** distributed object is responsible for storing notes Id, Title and URL information.

3.3.4.1.12.1 Methods

The methods for the **SharedNotesContent** distributed object are as follows:

sSetErrorCode: This method allows clients to communicate an error to the server (2).

▪ **code (Int32):** The error code. The only allowed value is 1 denoting an unknown error.

3.3.4.1.13   QnaContent

This section follows the product behavior described in endnote<24>.

The server-side **QnaContent** distributed object is responsible for changing the open state of the questions and answers content and processing requests from the web page that implements the view.

3.3.4.1.13.1 Methods

The methods for the **QnaContent** distributed object are as follows:

sSetOpenState: This method allows clients to communicate the new open state of the questions and answers content to the server (2).

▪ **openState (Int32):** The new open state of the questions and answers content. MUST conform to **QnaOpenState** schema in section 3.1.4.1.16.

sPutBlob: This method allows the web page to communicate the opaque BLOB to the server (2).

▪ **blob (String):** Opaque BLOB posted by the web page to the server

3.3.4.1.14   SharedLinksContent

The server-side SharedLinksContent distributed object is responsible for sharing and permissioning files that are currently being hosted on OneDriveForBusiness.

3.3.4.1.14.1 Methods

sSetActive: Not used

sSetInActive: Not used

3.3.4.2 File Download

Files MUST be made available to clients through a file **Web server** over HTTP or Hypertext Transfer Protocol over Secure Sockets Layer (HTTPS), as specified in [RFC2616].

The message **cSetUrlBase** MUST indicate the root of this Web server. When any other message contains a file name as a parameter, the specified **fileName** MUST be appended to download the referenced file. The files MUST be encrypted with the **Advanced Encryption Standard (AES)**, as defined in [FIPS197], when they are made available on the file Web server. The decryption keys are distributed to clients through various messages where terms such as **key** and **iv**, or initialization vector, are used.

3.3.4.3 Decrypted File Download

For a client that cannot process encrypted downloaded files, **UploadManager** distributed object has added the **sRequestWebDownloadBlob** method (see section 3.3.4.1.4.1).

3.3.4.4 Single File Upload using HTTPS request

For a client that cannot compose an OC package and that can only upload single files using the HTTPS Form **POST** request, **UploadManager** distributed object has added the **sRequestWebUploadBlob** method (see section 3.3.4.1.4.1). The sequence of steps is as follows:

1. Client sends a **sRequestWebUploadBlob** message to server over the PSOM protocol.

2. The server sends back to the client either the **cAcceptWebUpload** or the **cRejectUpload** message.

3. If the client has received the **cAcceptWebUpload** message, it SHOULD use the provided **uploadUrl** to send the file content using the HTTPS Form **POST** request implemented by most browsers.

4. When the upload is finished the server sends back the HTTP 200 response with the body of the following format "<html>200</html>". If the file upload request has failed, the server sends back to the client a non-200 HTTP response and the body MAY contain additional information such as "<html>404</html>" if the file upload permission has expired.

5. After step 4 client MUST call the **sUploadFinished** method and provide the same cookie as in the **sRequestWebUploadBlob** request as well as set the *cancel* parameter to true if the operation has failed.

6. After step 5 the client MAY use the **sRequestWebUploadBlob** message to request a new file upload over HTTPS or retry the failed operation for the same file.

3.3.5 Message Processing Events and Sequencing Rules

All messages MUST be processed in the order that they are received. The server MUST respond to any messages that require a response before reading any additional messages from other clients.

3.3.6 Timer Events

The server has the timer mentioned in section 3.1.2 through the **ping** message. Additionally, any **sAuthId** that is not redeemed within 120 seconds SHOULD expire so that the client can no longer connect with that token.

3.3.7 Other Local Events

None.

3.4 Proxy Details

Any non-PSOM proxies that exist MUST relay the data as it is and SHOULD NOT modify the traffic in any way.

The only supported proxy is silent. It MUST make no changes to traffic. The only difference between the proxy and the server is that an extra authentication token is required upon connection to the proxy. All other interactions are identical. See section 3.3.3.1 for details.

3.4.1 Abstract Data Model

None.

3.4.2 Timers

None.

3.4.3 Initialization

None.

3.4.4 Higher-Layer Triggered Events

None.

3.4.5 Message Processing Events and Sequencing Rules

None.

3.4.6 Timer Events

None.

3.4.7 Other Local Events

None.

4   Protocol Examples

This section details a sample session and divides it into three segments:

- Connection until the root distributed object is about to be connected.

- Root distributed object and child object connection.

- Client and server exchange an RPC message.

The examples are all sequential; however, the server authentication response could come at any time after the client sends the authentication token, but before any other data from the server (2).

4.1   Connection of PSOM Channel Zero (Prior to Root Distributed Object)

In the following example, a client with an authorization token of "300000000000000E36032154C544908" is used to authenticate the connection. The following protocols are defined:

- **ConnMgr** as defined in section 3.1.3.1.

- **Meeting** as defined in section 3.1.4.1.6, but no child objects are connected in this example.

4.1.1   Client to Server Authentication

The client sending data to the server through **AuthenticationToken** has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
Signature																															
AuthenticationVersion																															
AuthenticationTokenLen																															
AuthenticationToken (variable) 0x45333630;0x33323135;0x34433534;0x34393038																															
...																															

Signature (4 bytes): Set to "0x70773200".

AuthenticationVersion (4 bytes): Set to "0x00000000".

AuthenticationTokenLen (4 bytes): Set to "0x00000020".

AuthenticationToken (variable): Set to the following:

```
0x33303030;
0x30303030;
0x30303030;
0x30303030
```

4.1.2 Server to Client Authentication Response

The server sending data in response to the action in the previous section for a successful token submission has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
															Signature																

Signature (4 bytes): Set to "0x70773200".

4.1.3 Client to Server Channel Creation

The client sending data to the server in response to successful authentication confirmation to create channel zero has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
			SetChannel											ChannelId																	
			...																												

SetChannel (1 byte): Header for message type; defined as "0x04".

ChannelId (4 bytes): The representation of the channel identifier to be set. This PSOM channel is used for all messages that follow this one, with the exception of other **SetChannel** messages.

4.1.4 Client to Server Versioning

The client to server versioning sequence represents some of the messages sent by the client to the server during the connection negotiation phase. This illustrates the 3 key messages:

- **version**

- **addProtocol**

- **doneProtocols**

The **addProtocol** message is sent multiple times with different parameters for each registered distributed object.

In the non-block sequence diagrams, there is a number in parentheses. This number is the byte count of the payload.

4.1.4.1 version (stubHash)

This version represents the client notifying the server of its **ConnMgr** client interface hash version, which is "8322047979521208965". This has the following format.

0	1	2	3	4	5	6	7	8	9	1 0	1	2	3	4	5	6	7	8	9	2 0	1	2	3	4	5	6	7	8	9	3 0	1
			RpcMessage											MessageLength																	

...	ProxyId	MethodIndex	Parameter 1
...			

RpcMessage (1 byte): Set to "0x16".

MessageLength (4 bytes): Set to "0x0000000b".

ProxyId (1 byte): Set to "0x00".

MethodIndex (1 byte): Set to "0x01".

Parameter 1 (9 bytes): Int64 stubHash 0x (87,73,7d,da,8b,97,1e,72,85)

4.1.4.2 addProtocol (name, versions, hashes)

The **addProtocol** message sends the following over PSOM channel zero, which is the connection PSOM channel:

name: Microsoft.Rtc.Server.DataMCU.Meeting.Pod.ConnMgr

Versions: "1"

Hashes: "100633220832999761". This is the sum of the server and client interface hashes:

-8221414758688209204 + 8322047979521208965 == 100633220832999761

Data flow:

- RpcMessage (1)
  - "0x16"
- Message Length (4)
  - "0x00000040"
- ProxyId (1)
  - "0x00"
- MethodIndex (1)
  - "0x02"
- **Parameter 1:** name
  - Length of string (2)
    - "0x0030"
  - Payload (48)
    - 0x 9d,88,91,71,7b,56,59,21,2c,47,
    - 0x 28,ff,ff,83,ed,aa,92,87,67,61,
    - 0x 0a,71,27,23,09,34,c9,ce,82,f0,
    - 0x ab,ba,84,68,7c,44,1a,15,39,03,

- 0x 56,ca,f5,c5,d2,80,b9,9d,

- **Parameter 2:** versions

  - Length of array ("1")

    - "0x01"

  - Value of only entry in array encoded as Int32 (1)

    - "0x01"

- **Parameter 3:** hashes

  - Length of array (1)

    - "0x01"

  - Value of only entry in array encoded as Int64 (9)

    - 0x 87,01,65,85,61,94,5a,b9,51

4.1.4.3 doneProtocols

The **doneProtocols** is represented as follows.

Data flow:

- RpcMessage (1)

  - "0x16"

- Message Length (4)

  - "0x00000002"

- ProxyId (1)

  - "0x00"

- MethodIndex (1)

  - "0x03"

Because there are no arguments, this is the only data sent.

4.1.5 Server to Client Versioning

Server to client versioning is very similar to client to server versioning messages.

The following is a **version** RPC call (16 bytes).

```
0x 16,00,00,00,0b,00,01,8f,72,18,55,2a,02,c3,b9,34
```

The following is an **addProtocol** message in a **ConnMgr** RPC call (69 bytes).

```
0x16,00,00,00,40,00,02,00,30,9d,88,91,71,7b,56,59,21,2c,47,28,ff,ff,83,ed,aa,92,87,67,61,0a,7
1,27,23,09,34,c9,ce,82,f0,ab,ba,84,68,7c,44,1a,15,39,03,56,ca,f5,c5,d2,80,b9,9d,01,01,01,87,0
1,65,85,61,94,5a,b9,51
```

The following is an **addProtocol** message in a **Meeting** RPC Call (65 bytes).

```
0x16,00,00,00,3c,00,02,00,2c,59,4c,55,35,37,1a,15,ed,e8,83,ec,bb,83,df,51,76,56,43,23,25,46,3
d,eb,ef,cd,f0,8d,8a,de,4c,77,46,40,2c,38,00,56,c4,ff,ce,c8,a4,b0,88,01,01,01,8f,1b,db,fa,2d,b
c,55,32,5a
```

The following is a **doneProtocols** message in an RPC Call (6 bytes).

```
0x 16,00,00,00,02,00,03
```

4.2 PSOM Channel 2 Distributed Object Root Connection

The **Lookup** message is sent from the client to the server when versioning is complete. This is a special message that creates a new logical PSOM channel, with **id**="2", for the application distributed objects to communicate. The total length of this sequence is 49 bytes.

- rpcOpenMessage (1 byte)

    - "0x37"

- Channel Id = 2 (u32Arg) (4 bytes)

    - "0x00 00 00 02"

- Open Message Length (value == 40 decimal) (4 bytes)

    - "0x00 00 00 28"

- ProxyId (1 byte)

    - "0x00"

- MethodIndex (for lookup) (1 byte)

    - "0x05"

- Parameter 1 – Name (string) – Unused (20 bytes)

    - 0x 00,12,9a,90,b4,4e,3f,51,4d,24,38,01,19,a4,e8,ce,d1,a2,a8,8a

- Parameter 2 – Protocol (string) – Unused (9 bytes)

    - 0x 00,07,c7,f5,df,e9,be,bb,8b,

- Parameter 3 – proxyHash (Int64) - Unused (9 bytes)

    - 0x 8f,6e,14,7b,cb,cb,37,8e,f7

4.3 Server to Client RPC Message Exchange

This example demonstrates the server to client flow for setting up the root distributed object, which is **Meeting**. The sequence is as follows:

- **SetChannel**

- **cSetUrlBase**

- Connect **ContentUserManager**

- Note that other **Connect** messages would be sent here. They are omitted for brevity.

- **cMeetingReady**

SetChannel: Represents a PSOM channel-level request to set the current PSOM channel for all following messages.

- SetChannel (1 byte)

    - "0x04"

- ChannelId u32Arg (4 bytes)

    - "0x 00 00 00 02"

cSetUrlBase Message: Represents a server (2) to client message with a string payload of "http://example.com/conference/1015".

- RpcMessage (1 byte)

    - "0x16"

- Length (4 bytes)

    - "0x 00 00 00 26"

- ProxyId (1 byte)

    - "0x00"

- MethodIndex (1 byte)

    - "0x04"

- Parameter 1 – string (34 bytes)

    - String Length

        - "0x 00 22"

    - String payload

    - 0xd6,bb,94,81,38,3c,0b,50,3e,36

    - 0x05,09,e6,fe,82,de,a1,b2,df,62

    - 0x7d,4d,52,20,24,02,16,ea,ff,84

    - 0x8d,fd,ef,da

Connect ContentUserManager

- RpcMessage (1 byte)

    - "0x16"

- Length (4 bytes)

    - "0x 00 00 00 1f"

- RPC message type (Connect) (1 byte)

    - "0x84"

- Destination ProxyId (1 byte)

    - "0x00"

- Connection String

    - Length

        - "0x 00 12"

    - Payload ("ContentUserManager")

        - 0x ad,b0,9e,75,77,4d,40,10,25,02,

        - 0x 0a,c4,fb,c5,dd,aa,bb,9d,

- Server (2) hash code for distributed object to connect (Int64)

    - 0x 87,49,d5,9c,18,ed,9d,9e,0c

cMeetingReady Message: Signals the distributed object connection routine is complete for the **Meeting** distributed object.

- RpcMessage (1 byte)

    - "0x16"

- Length (4 bytes)

    - "0x 00 00 00 02"

- ProxyId (1 byte)

    - "0x00"

- MethodIndex (1 byte) – corresponds to cMeetingReady (no parameters)

    - "0x01"

sReserveTitle Message: This is a sample client to server message requesting a title reservation. This message follows a **ping** message that was sent by the server, which is not included. The client sets the PSOM channel to 2, because pings are received on PSOM channel zero. The title to be reserved is "Hello World".

- SetChannel (1 byte)

    - "0x04"

- ChannelId (4 bytes)

    - "0x 00 00 00 02"

- RpcMessage (1 byte)

    - "0x16"

- Length (4 bytes)

    - "0x 00 00 00 10"

- ProxyId (1 byte) – recall this is encoded as a PSOM Int32 (and value "-2")

    - "0x fe"

- MethodIndex (1 byte)

  - "0x 04"

- Parameter 1 – Title (string) (13 bytes)

  - 0x 00,0b,0d,33,0b,14,e6,ba,fc,d3,bf,b2,8b

- Parameter 2 – Cookie (1 byte)

  - "0x 01"

Possible messages from the server: This is a set of messages that can be received following a title reservation request. This is sent from the server (2) to the client.

The following is a **cContentUserAdded** message.

- RpcMessage

  - "0x 16"

- Length

  - "0x 00 00 00 3b"

- ProxyId

  - "0x 01"

- MethodIndex

  - "0x 01"

- Parameter 1 – Array with Id

  - 0x 01,01

- Parameter 2 – Array with Uris

  - 0x01,00,26,09,e2,ec,97,cc,b6,81,9f,64,23,64,47,32,34,0c,1c,fc,b5,c2,c9,ba,ba,83,75,3c,4e,5d,26,24,08,0b,e6,fc,df,92,ae,b1,82

- Parameter 3 – Array with Name

  - 0x 01,00,0b,17,2f,06,16,b9,ba,ed,dd,bf,b3,df

The following is a **cTitleReservationCompleted** message.

- RpcMessage

  - "0x 16"

- Length

  - "0x 00 00 00 06"

- ProxyId

  - "0x 02"

- MethodIndex

  - "0x 05"

- Parameter 1 – Status (value corresponds to "ReservedForCreation")

  - "0x 01"

- Parameter 2 – Cookie

  - "0x 01"

- Parameter 3 – contentId

  - "0x 00"

- Parameter 4 – owningUserId

  - "0x 01"

5   Security

5.1   Security Considerations for Implementers

SHA-1 is used as a file hash to optimize for downloads, but is not used for security purposes.

AES is used to encrypt and decrypt files stored on the file Web server.

5.2   Index of Security Parameters

Because this protocol requires a secure transport, the only unique piece is the authorization token.

Security parameter	Section
Authorization token (**AuthId**)	Sections 3.2.3.1.1.1 and 3.3.3.1.1

6 Appendix A: Encoding Algorithms

This appendix covers the specialized encoding algorithms used by this protocol to represent several primitive types.

6.1 GenericInt

This algorithm optimizes on length for smaller, meaning closer to zero, numbers. The following rules apply:

- Numbers in the range -112 to 127 are represented by one byte, which is their value.
- Numbers outside of that range are represented in variable length unsigned big-endian notation.
  - The lead byte indicates their length and sign.
  - The lead byte is 0x80 plus (8 if negative or plus 0 otherwise) plus (*nbytes* minus 1), where *nbytes* is the number of data bytes following, and includes the numbers 1, 2, 3, 4, 6, and 8.

The following table lists some examples:

Number	Representation
0	0x00
255	0x80 0xFF
-255	0x88 0xFF
256	0x81 0x01 0x00

The two numbers in the following table are encoded irregularly because of twos-complement arithmetic:

Number	Representation
-(2^31)	0x88 0x00
-(2^63)	0x8d 0x00 0x00 0x00 0x00 0x00 0x00

6.1.1 Pseudo-Code

The following code sample demonstrates how to encode a **GenericInt** into a variable length byte array.

```
byte[] EncodeGenericInt(int x)
{
    if (-112 <= x && x <= 127)
    {
        return x as byte;
    }

    Int64 absX = AbsoluteValue(x);
    Int32 size = (absX <= 0xFF ? 0
                : (absX <= 0xFFFF ? 1
                : (absX <= 0xFFFFFF ? 2
                : (absX <= 0xFFFFFFFF ? 3
                : (absX <= 0xFFFFFFFFFFFF ? 5
                : 7 )))));
```

```
        byte[] baseBytes = new byte[9];
        Int32 basePosition = 0;

        baseBytes[basePosition++] = ((x >= 0 ? 0x80 : 0x88) + size) as byte;
        if (size >= 7) baseBytes[basePosition++] = ((absX >> 0x38) & 0xFF) as byte;
        if (size >= 6) baseBytes[basePosition++] = ((absX >> 0x30) & 0xFF) as byte;
        if (size >= 5) baseBytes[basePosition++] = ((absX >> 0x28) & 0xFF) as byte;
        if (size >= 4) baseBytes[basePosition++] = ((absX >> 0x20) & 0xFF) as byte;
        if (size >= 3) baseBytes[basePosition++] = ((absX >> 0x18) & 0xFF) as byte;
        if (size >= 2) baseBytes[basePosition++] = ((absX >> 0x10) & 0xFF) as byte;
        if (size >= 1) baseBytes[basePosition++] = ((absX >> 0x08) & 0xFF) as byte;
        if (size >= 0) baseBytes[basePosition++] = ((absX >> 0x00) & 0xFF) as byte;

        byte[] toReturn = SubsetOf(baseBytes from 0 to (size + 1));
        return toReturn;
    }
```

6.2   String

This section describes the encoding algorithm for generating PSOM strings as parameters, given a UTF-8 string as input. The first two bytes are reserved for an encoded version of the length, which is not greater than 0xFFFF.

- Define **varIncrement** equals "-17".

- Define Int32 (signed) **varvalue** equals zero ("0").

- For each byte in the UTF-8 encoded string starting at the last position and moving towards position 0 in the UTF-8 encoded string:

    - Increment **varValue** by **varIncrement**.

    - Execute an **XOR** operation on the byte in the string and **varValue**.

    - Place the resultant byte in the same position as the byte that was just included in the **XOR** operation.

- UTF-8 encoding for wide characters:

    - Characters in the range 0x0001 to 0x007F:

        - Encoded as 1 byte, which matches the character's byte value.

    - Characters in the range 0x0080 to 0x07FF:

        - The wide character is divided into two parts, *x* and *y*, as shown in binary 0000 0*xxx xxyy yyyy*.

        - The first byte is represented as binary 110*x xxxx*.

            - This is the equivalent of **OR**'ing bits 6 and 10 with 0xC0.

        - The second byte is represented as 10*yy yyyy*.

            - This is the equivalent of **OR**'ing bits 0 and 5 with 0x80.

    - Characters in the range 0x0800 to 0xFFFF:

        - The wide character is divided into three parts, which can be represented as *xxxx yyyy yyzz zzzz*.

- The first byte is 1110 *xxxx*.

  - This is the equivalent of **OR**'ing bits 12 and 15 with 0xE0.

  - The second byte is 10*yy yyyy*.

    - This is the equivalent of **OR**'ing bits 6 and 11 with 0x80.

  - The third byte is 10*zz zzzz*.

    - This is the equivalent of **OR**'ing bits 0 and 5 with 0x80.

- Because the maximum size of the string is 0xFFFF, the first two bytes of the string segment are the unsigned integer big-endian representation of the length.

  - Example: A string "pptdemo2.pptx" with a length of 13 becomes "0x00 0x0D" in the length header.

Following is an example of a generated PSOM string:

- A string "pptdemo2.pptx" becomes, in hexadecimal:

        00,0d,53,44,31,32,02,15,e6,a8,85,cc,bd,aa,97

7   Appendix B: Sample Upload Package

An OC Package is composed of a manifest, with the name "OcpManifest.xml", and any other files that are referenced by the manifest. In the case of creating content with a single file attached to it, the upload package contains two parts. Note that this is an OPC file, so there might be additional metadata files per the specification. This example includes a jpeg-encoded image named "Ryan's Fun Day.jpg". Note that the contents and type of this file is irrelevant to the example.

OcpManifest.xml:

```
<ocp xmlns="http://schemas.microsoft.com/2008/12/ocp">
    <createContent>
        <common>
            <title>Ryan's Fun Day.jpg</title>
            <nativeFile>native.file</nativeFile>
        </common>
        <contentDetail type="Content.NativeFileOnly">
            <nativeFileOnlyContent xmlns="http://schemas.microsoft.com/2008/12/ocp-content-
detail">
                <nativeFileOnlyType>empty</nativeFileOnlyType>
            </nativeFileOnlyContent>

        </contentDetail>
    </createContent>
</ocp>
```

native.file: Binary payload of any file to share. In this case, the file is a jpeg-encoded image.

Before this is uploaded, the client sends an **sReserveTitle** message and receives a **cTitleReserved** response from the server (2) with a positive confirmation of the title reservation, with a title that matches the title specified in the preceding **XML fragment**.

8   Appendix C: Product Behavior

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

- Microsoft Lync Server 2010
- Microsoft Lync 2010
- Microsoft Lync Server 2013
- Microsoft Lync Client 2013/Skype for Business

- Microsoft Skype for Business 2016

- Microsoft Skype for Business Server 2015

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

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

<1> Section 3.1.4.1.4: Lync Server 2010, Lync 2010, Lync Server 2013, Lync Client 2013/Skype for Business: The maximum number of allowed reservations is 20.

<2> Section 3.1.4.1.5.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<3> Section 3.1.4.1.8.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<4> Section 3.1.4.1.11.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<5> Section 3.1.4.1.15: Lync Server 2010, Lync 2010: This behavior is not supported.

<6> Section 3.1.4.1.16: Lync Server 2010, Lync 2010: This behavior is not supported.

<7> Section 3.1.4.1.17: Lync Server 2010, Lync 2010: This behavior is not supported.

<8> Section 3.2.4.1.3.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<9> Section 3.2.4.1.3.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<10> Section 3.2.4.1.4: Lync Server 2010, Lync 2010, Lync Server 2013, Lync Client 2013/Skype for Business: The maximum number of concurrent upload streams is 5.

<11> Section 3.2.4.1.4: Lync Server 2010, Lync 2010, Lync Server 2013, Lync Client 2013/Skype for Business: The maximum number of **Content** objects is 50.

<12> Section 3.2.4.1.4: Lync Server 2010, Lync 2010, Lync Server 2013, Lync Client 2013/Skype for Business: The maximum number of slides is 1000.

<13> Section 3.2.4.1.4.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<14> Section 3.2.4.1.4.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<15> Section 3.2.4.1.4.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<16> Section 3.2.4.1.4.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<17> Section 3.2.4.1.6.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<18> Section 3.2.4.1.6.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<19> Section 3.2.4.1.6.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<20> Section 3.2.4.1.12: Lync Server 2010, Lync 2010: This behavior is not supported.

<21> Section 3.3.4.1.4.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<22> Section 3.3.4.1.4.1: Lync Server 2010, Lync 2010: This behavior is not supported.

<23> Section 3.3.4.1.12: Lync Server 2010, Lync 2010: This behavior is not supported.

<24> Section 3.3.4.1.13: Lync Server 2010, Lync 2010: This behavior is not supported.

9 Change Tracking

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

10 Index

Upload package

V