INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION
ISO/IEC JTC1/SC29/WG11
CODING OF MOVING PICTURES AND ASSOCIATED AUDIO

ISO/IEC JTC1/SC29/WG11 N0950
MPEG 95/
May 1995 (Boston)

Digital Storage Media Command & Control

INFORMATION TECHNOLOGY -

GENERIC CODING OF MOVING PICTURES AND ASSOCIATED AUDIO: Digital Storage Media Command and Control

ISO/IEC 13818-6

Committee Draft

12-June-1995

The Digital Storage Media Command and Control (DSM-CC) specification is a set of protocols intended to provide the control functions and operations specific to managing ISO/IEC 11172 (MPEG-1) and ISO/IEC 13818 (MPEG-2) bitstreams. Informative Annex A of MPEG-2 Systems (ISO/IEC 13818-1) provides a specification of the syntax and semantics for a simple environment of single-user-to-single-DSM applications. MPEG Systems are also deployed, however, in more diverse and heterogeneous network environments for many applications including, for example, video-on-demand and interactive multimedia. This draft document contains an extension of the DSM-CC protocol for supporting such applications in both stand-alone and heterogeneous network environments and is an integral part of the ISO/IEC 13818 (MPEG-2) standards.

Editorial comments in this document are shown in italics, typically within "[ ]".

[This document reflects all the technical agreements to date, unless specifically indicated elsewise. Table and figure labeling and cross referencing have not been completed. Further editing to clean up styling, etc. is also required.]

LIST OF FIGURES

[to be provided when a correct cross reference is established]

LIST OF TABLES

[to be provided when a correct cross reference is established]

1. Introduction

The Digital Storage Media Command and Control (DSM-CC) specification is a set of protocols intended to provide the control functions and operations specific to managing ISO/IEC 11172 (MPEG-1) and ISO/IEC 13818 (MPEG-2) bitstreams. Informative Annex A of MPEG-2 Systems (ISO/IEC 13818-1) provides a specification of the syntax and semantics for a simple environment of single-user-to-single-DSM applications. MPEG Systems are also deployed, however, in more diverse and heterogeneous network environments for many applications including, for example, video-on-demand and interactive video. This draft document contains an extension of the DSM-CC protocol for supporting such applications in both stand-alone and heterogeneous network environments and is an integral part of the ISO/IEC 13818 (MPEG-2) standards.

Functional groups of DSM-CC operations are described in this document. An embodiment using DSM-CC is not required to implement every functional group. However, if an embodiment implements operations of a group type analogous to those described in this section (e.g., Record and Edit Assembly), then the embodiment shall implement the complete syntax and semantics of the corresponding DSM-CC function group. If an embodiment does not implement operations of a group type described in this section, then a function call to an unimplemented group of functions shall return a status of "Function Not Implemented". *[The exact mechanism is for further study.]*

The grouping of required protocol subsets is for further study. The following list shows an example of such grouping.

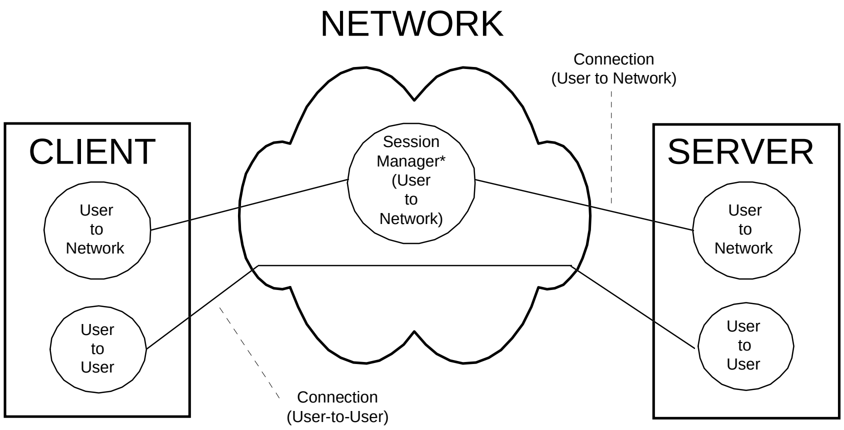
- *User-to-Network Configuration*
- *User-to-Network Messages*
- *User-to-User Playback*
- *User-to-User Record and Edit Assembly*
- *User-to-User File Access*
- *User-to-User Database Access*

1.1 Scope

This document provides the specifications for the controls of MPEG-2 bitstreams in both standalone and distributed environments with the following characteristics:

- Multi-server
  - DSM-CC clients may request service from multiple servers. The environment also contains servers communicating with other servers.
- Multi-session
  - A DSM-CC client has the ability to have multiple simultaneous calls in progress.
- Multi-client
  - A single piece of material may be accessed concurrently or sequentially by multiple clients.
- Connectivity
  - Broadcast
  - Point-to-point
  - Multicast
  - Multi-point to multi-point
- Multiprotocol
  - A DSM-CC client may request service of multiple servers, where each communication path may cross multiple diverse network protocols. These underlying network protocols must be transparent to the DSM-CC Extension.

1.2 DSM-CC Functional Reference Model



*May provide session, connection, and configuration management and control

Figure 1-1. DSM-CC Functional Reference Model

A functional reference model describing the signaling, control and User-User capabilities of the interactive network is shown in Figure 1-1. The overall set of functions is grouped by functional entities. The figure illustrates the functions in a platform independant manner and the depiction of any function does not neccessarily constrain its implementation in hardware or software.

Four functional entity types are defined:

- Session
- Connection
- Configuration
- User-User

The functional entity types Session, Connection and Configuration reside in the client, server and network. The functional entity type User-User resides in the client and the server only. The functional entities can interact with each other to provide the services requested, this is illustrated in the interconnections of the functional entity “bubbles” in the figure. For simplicity and readability the Session, Connection, and Configuration at each of the Client Server and Network are shown as one bubble rather than three.

1.2.1 Mapping of Functional Reference Model to DSM-CC

The interactions of functional entities are modeled as the exchange of DSM-CC protocol messages and their associated parameters.

The interactive relationships of the Session, Connection and Configuration functional entities at the client with their peers in the network are represented as the User-Network protocol. Likewise the mapping of the Session, Connection and Configuration functional entities at the server with their peers in the network are represented as the User-Network protocol.

The interactive relationship between the functional entities at the client and the server is represented as the User-User protocol.

1.3 DSM-CC Protocol Model

The protocol models below show the protocol stack of DSM-CC and its relationship with other protocols.

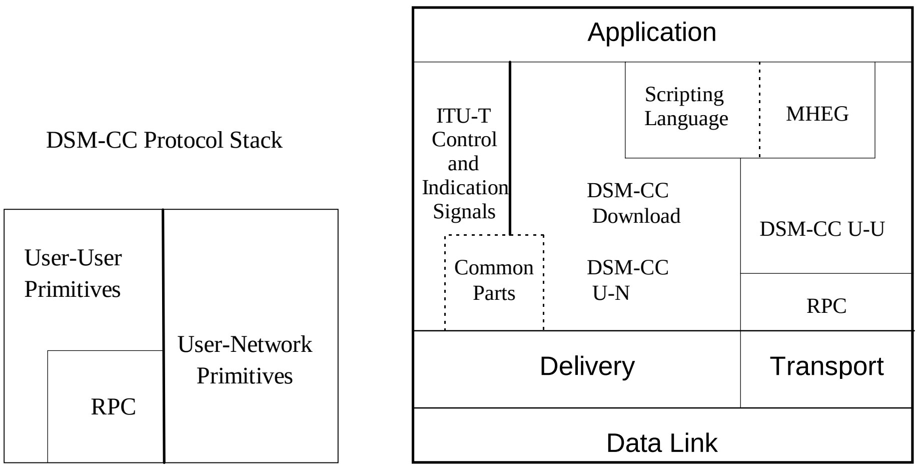


Figure 1-2. DSM-CC Protocol Model

DSM-CC provides access for general applications, MHEG applications, and scripting language application to primitives for establishing or deleting a network connection using User-Network (U-N) Primitive and communication between a client and a server accross a network using User-User (U-U) Primitive. U-U operations may use a Remote Procedure Call (RPC) protocol. Both U-U and U-N operations may employ a message passing scheme which involves a sequence of bit-pattern exchanges.

DSM-CC may be carried as a stream within an MPEG 1 System Stream, an MPEG 2 Transport Stream, or an MPEG 2 Program Stream. Alternatively, DSM-CC may be carried over other delivery mechanisms, such as TCP or UDP.

[The integration of DSM-CC with ITU-T Control and Indication signalling is currently under study. The uncertain relationship is illustrated by the dotted line in the figure. Possible common parts of DSM-CC and ITU-T C&I signalling, e.g., capability exchange, is to be defined. It is also possible that the ITU-T C&I signalling may share the same transport mechanism with DSM-CC.]

1.3.1 U-N and U-U Download Message Delivery Requirements

The delivery mechanism for U-N messages and U-U download messages is user defined. The table below summarizes the minimum requirements for a message delivery layer (MDL) for DSMCC Clients and Servers for these two types of messages.

Table 1-1

DeliveryFunction	Requirement
Integrity of Data	Error detection must be provided. Corrupted messages should be discarded.
Reliability of Delivery	The delivery of the message need not be guaranteed.
Flow Control	The delivery mechanism need not regulate the rate of transmission of messages.
Fragmentation and Reassembly	The delivery mechanism is responsible for any required fragmentation and reassembly of messages, up to some maximum given by the “maximum transmissible unit” (MTU) for the delivery mechanism, which may be as low as 4k bytes.
Delivery Order of Messages	Delivery mechanism need not be responsible for in order delivery of messages.

Service Access Point	The service access point is an end-system. Message delivery is to the end-system, not entities within the end system.
Addressing	None. There must be some form of connection to the SCCMC established by mechanisms outside the scope of DSMCC.
Concurrent transactions	There must be support for more than one simultaneous outstanding message transaction on an individual Client or Server system.

Examples of suitable MDLs are UDP/IP with a well known IP address and port for the SCCMC, and AAL5/ATM over a permanent virtual circuit (PVC) connected to the SCCMC. Both of these have capabilities that exceed the minimum requirements: UDP/IP supports finer grain service access points and AAL5 over a single VC guarantees in order delivery. However, DSM-CC U-N will not utilize any capabilities in excess of those specified above. As a further example, TCP/IP could be used as a DSM-CC U-N message delivery mechanism, but no advantage would be taken of its reliable delivery guarantee. The MTU may not be very large: MPEG2 TS, for example, has an MTU of less than 4k bytes.

These capabilities can be expressed with the following *semantic interfaces* (SI) that describe the facilities assumed by the network. Although they look like APIs, they are not intended as such; rather, they only identify the functionality required of the message delivery layer and make explicit all the information that must be provided to it by the DSM-CC U-N layer.

The SI for sending a DSMCC message (either a request or a response) is as follows:

```
STATUS SendToSCCMC(
      DSMCCMessageHeader * msg);
```

This SI says that DSMCC assumes the ability to send a message to “the network” (whose representative is the SCCMC) without knowing anything more than is in a DSMCCMessageHeader (which is defined in section 2.) In particular, DSMCC need not know any form of address of the SCCMC.

Two kinds of messages can be recieved: indications and confirmations. At a receiver, an indication is the first message of a message sequence transaction, and a confirmation is the concluding message.

```
STATUS RcvIndicationFromSCCMC(
      DSMCCMessageHeader * hdr,
      int maxMsglen);
```

where

`DSMCCMessageHeader`	is received DSMCC message
`maxMsgLen`	is the maximum length of the incoming message

This SI says that for the DSMCC to receive an indication, it needs to give the message delivery layer nothing more than the maximum length of the message that it can receive; that information, plus whatever is in defined to be a DSMCCMessageHeader, must be sufficient for the MDL to get the received message to the correct recipient.

```
STATUS RcvConfirmFromSCCMC(
      short XID,
      DSMCCMessageHeader * hdr,
      int maxMsglen);
```

where

`XID`	is the transaction ID expected on the reply.
`DSMCCMessageHeader`	is received DSMCC message
`maxMsgLen`	is the maximum length of the incoming message

This SI says that the DSMCC needs to specify the transaction ID of the confirmation message it wants to receive, and the maximum message size it can handle, and it will receive messages with that transaction ID. The transaction ID is necessary because there can be more than one simultaneous outstanding transaction, and there must be a way to route the confirmations to the sender of the original request.

1.3.2 Communications Requirement for Other U-U Functions

Other that U-U Download, the other U-U functions are specified in terms of user-defined RPC. The RPC mechanism must provide the following communications capabilities.

Table 1-1

DeliveryFunction	Requirement
Integrity of Data	Data integrity is guaranteed.
Reliability of Delivery	The delivery of the message is guaranteed.
Flow Control	The RPC mechanism regulates the rate of transmission of messages.
Fragmentation and Reassembly	The delivery mechanism is responsible for any required fragmentation and reassembly of messages, for arbitrarily large messages.
Delivery Order of Messages	The RPCmechanism is responsible for in order delivery of messages.
Service Access Point	The service access point is process or thread.
Addressing	RPC specific.
Concurrent transactions	There must be support for more than one simultaneous outstanding RPC on an individual Client or Server system.

1.4 References

[This list is incomplete and some of the detailed information of the references is still missing.]

1. RFC 1057 Remote Procedure Call (Internet)
2. RFC 1014 External Data Representation Standard (Internet)
3. DCE Distributed Computing Environment
4. NDR Network Data Representation, used by DCE
5. ISO RPC 11578 DIS
6. ITU-T Recommendation X.25
7. ITU-T Recommendation X.500
8. ITU-T Recommendation X.509
9. ITU-T Recommendation Q.931
10. ITU-T Recommendation Q.2931
11. ISO/IEC 14750-1 Working Draft: CORBA IDL as an Interface Definition Language for ODP Systems
12. ASN.1/BER
13. ITU-T Recommendation E164
14. ISO/IEC 11172 (MPEG-1)
15. ITU-T Rec. H.222.0|ISO/IEC 13818-1
16. TCP/IP (RFC #??)
17. SQL92

[According to 13818-1, a heading level 1 entitled "Technical Elements" starts here and encompasses the entire Normative text.]

1.5 Definitions

This section contains those definitions that are not yet defined in other MPEG (ISO/IEC 11172 and 13818) standards.

Client A consumer of a service from one or more Servers.

Connection A transport link that provides the capability to transfer information between two or more end points.

Network A collection of communicating elements that provides connections and may provide session control and/or connection control to User(s).

Server A provider of a service to one or more Clients.

Service A logical entity in the system that provides function(s) and interface(s) in support of one or more applications. The distinction of a service from other objects is that end-user access to it is controlled by a Service Gateway.

Session An association between two or more Users providing the capability to group together the resources needed for an instance of a service.

User An end system that is connected to a network that can transmit information to or receive information from other such end systems by means of the Network. A User may function as a Client, Server, or both.

Application Software that executes in a client environment

Application Download The procedure by which a server sends the requested application to a client.

Application Download Messaging Path The transport connection by which a client requests and receives a BLOB requested by the application download initiator. This messaging path may be point-to-point or point-to-multipoint.

Application Download Server The functional part of the server which is responsible for executing the server-side application download procedure.

Binary Large OBject (BLOB) Generic term used for a module that is transferred from one user to another (usually from a server to a client). This module may be an application, or it could be a different client specific module used to make updates or additions to resident software on the client, either by the network or by the server.

BLOB loader The software module resident on the client that is responsible for executing the client-side application download procedure.

Client: Describes the user that is receiving the BLOB, and therefore presenting the application to the ultimate user.

Application Download Initiator (ADI) An application on the client which communicates with the network over the user-network path and the application download server over the application download messaging path. The ADI communicates with the BLOB loader to perform the application download procedure.

Network-User Module Update Path The transport connection by which a network can use the application download procedure to make module updates to resident client software. It is required that the downloaded modules are able to be dynamically added and subsequently executed at run time. By the same token, these downloaded modules will be able to permanently replace existing resident modules on the client dynamically. *[to be further discussed]*

Ultimate User This term is used for the human element that will interact with the application in some way. The model is for the server to download an application onto a client. The client then begins execution of the application which presents the interface to and handles responses from the ultimate user.

User-Network messaging path The transport connection by which a user and network exchange messages. The protocol used to create this path is not specified. It is intended that the client and network be able to use this download procedure to load new resident modules or update older revisions.

User-User messaging path The transport connection by which a client and server exchange messages. There are two sub-types of user-user messaging paths, the application download messaging path and the application peer-to-peer messaging path. The latter messaging path refers to the connection between an existing application and its associated server.

1.6 Symbols and Abbreviations

[This section currently contains only those symbols and abbreviations that are not yet defined in other MPEG Standards. More related definitions will be added later.]

1.6.1 Symbols

[Tbd.]

1.6.2 Acronyms

[This acronym list is still incomplete.]

ABI	Application Boot Initiator
ABS	Application Boot Server
API	Application Programming Interface
ASN.1/BER	Abstract Syntax Notation 1/Basic Encoding Rules
BLOB	Binary Large OBject
BPDU	*[tbd] (see Annex C RFC 1483)*
CFS	Continuous Feed Server
CPDU	Common Protocol Data Unit
DAP	Directory Access Protocol (X.500)
DCE	Distributed Computing Environment
DIB	Directory Information Base (X.500)
DSA	Directory System Agent (X.500)
DSM	Digital Storage Media
DSM-CC	Digital Storage Media - Command and Control
DSP	Directory System Protocol (X.500)
DUA	Directory User Agent (X.500)
FCS	Frame Check Sum
GPDU	General Protocol Data Unit
IDL	Interface Definition Language
IP	Internet Protocol
MHEG	Multimedia/Hypermedia Experts Group
MPEG	Moving Picture Experts Group
MSL	Multimedia Scripting Language
NDR	Network Data Representation (DCE)
NPT	Normal Play Time
OMG	Object Management Group
ONC	Open Networked Computing
OPE	Other Protocol Element (MHEG)
PA	Physical Address
PDU	Protocol Data Unit
PES	Packetized Elementary Stream (13818-1)
PID	Packet Identifier (13818-1)
PIN	Personal Identification Number
PLL	Phase Locked Loop
PMT	Program Map Table
PS	Program Stream
PSI	Program Specific Information
RPC	Remote Procedure Call
SDL	Service Description Language
SE	SubElement
SNAP	SubNetwork Attachment Point
SQL	Structured Query Language
TCP	Transport Control Protocol

TS	Transport Stream
UDP	User Datagram Protocol
U-N	User-to-Network
U-U	User-to-User
XDR	External Data Representation

1.7 Methods of Specification

1.7.1 Remote Procedure Call

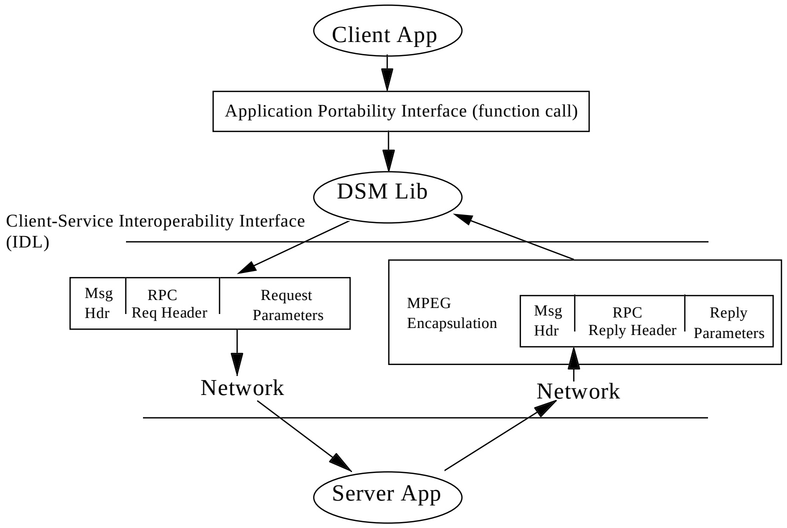


Figure 1-3. Example Data Flow of an RPC Execution

Digital Storage Media User-to-User (U-U) functionality exploits a Remote Procedure Call (RPC) protocol. An RPC allows implementation of a client-server model in which applications on a client are written to call functions that are similar to those that might be used if all actions were to be executed locally. The RPC includes a compiler which converts the functions to be called into (1) a piece of client code, which accepts arguments, prepares a packet containing the arguments, sends them to the server, accepts a packet from the server containing the server's response, and (2) a piece of server code which accepts packets from a client, performs the appropriate actions, and prepares and sends a packet containing the response to the client. For those U-U API primitives that use the RPC, the RPC protocol defines the actual bits that are exchanged as primitives are executed. Other U-U and U-N API primitives are message based.

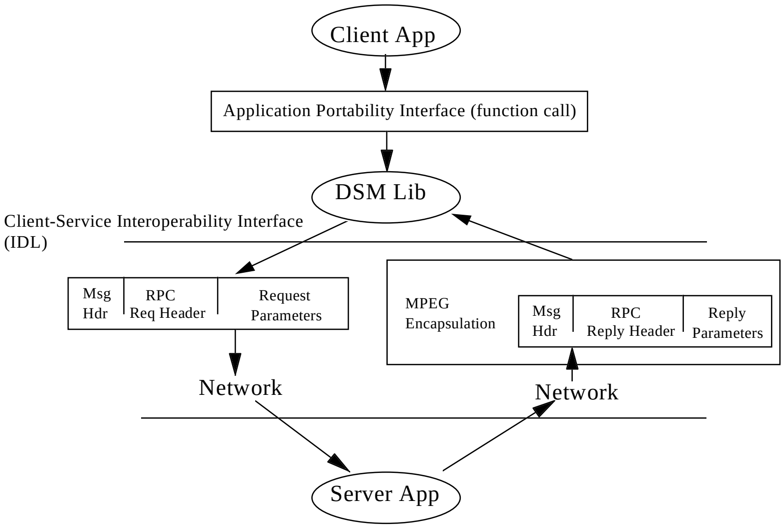


Figure 1-3 shows an example of the data flow of an RPC execution.

1.7.1.1 Independence of RPC

DSM-CC can be implemented using any RPC which can implement primitives that are legal within the Interface Definition Language (IDL). The RPC will include a data representation choice which defines how data structures are mapped to bits (e.g., Network Data Representation (NDR) or Abstract Syntax Notation 1 / Basic Encoding Rules (ASN.1/BER)).

Different implementations of RPC may generate different bit patterns on a communication link for the same primitive. Communication between a client using one RPC and a server using a different RPC would require a translator (executing on either the server or client side) to convert the RPC packet contents from one protocol to the other.

1.7.1.2 Local Equivalent Functions

For DSM-CC implementations in which the client and server functions are known to be entirely local (i.e., do not require message exchange over a network), those U-U and U-N primitives that use an RPC may be compiled by an alternative IDL compiler which produces a single equivalent local function call definition. This allows many applications to be simply ported between networked applications and stand-alone applications (e.g., CD-player). Alternatively, if separate server and client processes are executing locally, the RPC protocol may be used without modification.

1.7.2 Interface Definition Language

The U-U API primitives that use the RPC are defined in terms of an Interface Definition Language (IDL). The IDL provides a grammar for defining the function call like API specification for each primitive. Primitives written in the IDL are compiled by an IDL compiler to produce client and server stubs (executable code that implements packet formation, dispatch, receipt, and interpretation) and a header file used during compilation of the client and server applications.

1.7.3 Message Passing

[To be provided.]

1.7.4 Life Cycles

[To be provided.]

2. DSM-CC Message Header

All MPEG-2 DSM-CC messages begin with the DSM-CC MessageHeader with the exception of DSM-CC User-to-User primitives which use the RPC mechanism. This header contains information about the type of message being passed as well as any adaptation data which is needed by the transport mechanism including conditional access information needed to decode the data. Table 2 defines the format of a DSM-CC message header.

Table 2 MPEG-2 DSM-CC Message Header Format

Syntax	Num. of Bytes
dsmccMessageHeader() {	
protocolDiscriminator	**1**
dsmccType	**1**
transactionId	**8**
messageId	**2**
adaptationLength	**1**
messageLength	**2**
for(i=0;i<adaptationLength;i++) {	
dsmccAdaptationHeaderBytes	
}	
}	

The **protocolDiscriminator** field is used to indicate that the message is a MPEG-2 DSM-CC message. The value of this field shall be *[tbd]*.

[The protocolDiscriminator value is still under investigation. The intent was to align this field with the same field as that used by ITU-T Q.2931 general message format headers. This value must be assigned by ISO? ITU?]

The **dsmccType** field is used to indicate the type of MPEG2 DSM-CC message. Table 3 defines the possible dsmccTypes.

Table 3 MPEG-2 DSM-CC dsmccType values

dsmccType	Description
0x00	ISO/IEC 13818-6 Reserved
0x01	Identifies the message as an MPEG-2 DSM-CC User-to-Network configuration message.
0x02	Identifies the message as an MPEG-2 DSM-CC User-to-Network primitive message.
0x03	Identifies the message as an MPEG-2 DSM-CC User-to-User configuration message.
0x04	Identifies the message as an MPEG-2 DSM-CC User-to-User primitive message.
0x05-0x7f	ISO/IEC 13818-6 Reserved.
0x80-0xff	User Defined message type.

The **transactionId** field is used for session integrity and error processing. If a message is being originated, this field shall be set to a value which is unique within the network and shall remain unique for a period of time such that there is no chance that command sequences cannot collide.

The transactionId is constructed of the oringinators deviceId concatenated with a 2 byte transaction number.

[There is currently an issue between the size of the transactionId used in the dsmccMessageHeader() and the transactionId used by Download which is 2 bytes. This needs to be resolved.]

The **messageId** field indicates the type of message which is being passed. The values of the messageId are defined within the scope of the messageType.

[There was agreement to move to a 1 byte messageId field in both the dsmccMessageHeader() and the Download header. However, the way the messageIds are defined, 2 bytes are required. The messageId definition must be resolved before we can shrink the size of this field.]

The **adaptationLength** field indicates the total length of the adaptation headers which follow the message header. Each adaptation header contains a type and length field which are used to identify the individual adaptation headers.

The **messageLength** field is used to indicates the total length of the message which follows the general message header. This length includes any adaptation headers indicated in the adaptationLength and the message payload indicated by the messageId field.

The **DSMCCAdaptationHeader** is defined in Section 2.1.

2.1 DSM-CC Adaptation Header Format

The format of the DSM-CC Adaptation Header is defined in Table 4.

Table 4 Format of the DSM-CC Adaptation Header

Syntax	Num. of Bytes
dsmccAdaptationHeader() {	
adaptationType	**1**
adaptationLength	**1**
for(i=0;i<adaptationLength;i++) {	
adaptationDataByte	**1**
}	
}	

The **adaptationType** field is used to indicate the type of adaptation header. Table 5 defines the possible values of the adaptationType field.

Table 5 DSM-CC adaptationTypes

Adaptation Type	Description
0x00	ISO/IEC 13818-6 Reserved.
0x01	Indicates that this is a conditional access header.
0x02-0x7f	ISO/IEC 13818-6 Reserved.
0x80-0xff	User Defined adaptation header.

The **adaptationLength** and **adaptationDataByte** fields contain the adaptation data. The length and content of the adaptation data depends on the adaptationType.

2.1.1 DSM-CC Conditional Access Adaptation Header Format

Table 6 indicates the format of the conditional access adaptation header.

Table 6 Format of the DSM-CC Conditional Access Adaptation Header

Syntax	Num. of Bytes
dsmccConditionalAccessHeader() {	
conditionalAccessType	**2**
conditionalAccessLength	**1**
for(i=0;i<conditionalAccessLength;i++) {	
conditionalAccessDataByte	**1**
}	
}	

The **conditionalAccessType** field specifies the type of conditional access mechanism being used. The allowable values for this field are defined in *[systems document reference]*.

The **conditionalAccessLength** and **conditionalAccessDataByte** fields contain the data required for the particular type of conditional access mechanism being used. The length and content of the conditional access data depends on the conditionalAccessType.

3. User-to-Network Configuration Messages

This section refers to a broadcast capability which is used to send these messages. This then becomes a requirement on the network.

A list of tuples (which include serverIds and possibly other data) may be a postcondition of executing UN Config messages. This issue should be addressed in the LifeCycle Section [TBD]. It has been agreed that UNConfigMessages should support a network which does not use upstream signaling.

The following issues were added at Lausanne:

Either a User or the Network may assign the sessionId. There needs to be a method in U-N configuration to inform the User which method is being used. Only one method may be used in a Network.

Currently, only the Network may assign the resourceId, however, DSM-CC is investigating a method by which the User could assign this Id. If adopted, there needs to be a method in U-N configuration to inform the User which method is being used. Only one method may be used in a Network.

In some Network implementations, the User-to Network Configuration messages are not necessary. As in all parts of DSM-CC, the Network provider may choose not to implement U-N Configuration. However, if this section is implemented, all of the functions in this section must be implemented.

In addition to the User-to-Network configuration messages which allow the User devices to operate on the Network, the Users may use the User-to-User primitives may also execute a configuration among themselves.

The User-to-Network Configurations use the General Message Format defined in section **Error! Reference source not found.**. The dsmccType field shall be set to 0x01 to indicate that the message is a U-N Configuration message.

The User-to-Network (U-N) Configuration messages are used to allow a User device to gain access to a Network and configure the User device with the parameters which are required for the User Device to operate on the Network. Table 7 defines the messageIds for use in the U-N Configuration protocol:

Table 7 DSM-CC U-N Configuration messageIds

messageId	**Message Name**	**Description**
0x0000	Reserved	ISO/IEC 13818-6 Reserved.
0x0001	UNConfigRequest	Sent from the User to the Network to request the initial Network configuration.
0x0002	UNConfigConfirm	Sent from the Network to the User in response to the UNConfigRequest.
0x0003	UNConfigIndication	Sent from the Network to the User to reconfigure the User device.
0x0004	UNConfigResponse	Sent from the User to the Network in response to a UNConfigIndication message
0x0005	UNPageRequest	Sent from the User to the Network to request a page of data from the network.
0x0006	UNPageConfirm	Sent from the Network to the User in response to a UNPageRequest message.
0x0007	UNPageIndication	Sent from the Network to the User to update a page of data on the User device.
0x0008	UNPageResponse	Sent from the User to the Network in response to a UNPageIndication message
0x0009 - 0x7fff	Reserved	ISO/IEC 13818-6 Reserved.
0x8000 - 0xffff	User Defined	User Defined U-N Configuration message.

The U-N Configuration messages also provide the Network with a means to download pages of data to a User or group of users. This data may be specific to a particular device type and revision. The User may also request a data page from the Network at any time after initial configuration. Table 8 defines the page numbers available in DSM-CC.

Table 8 MPEG-2 DSM-CC User-to-Network Configuration page values

pageNumber	**Description**
0x0000	This page is reserved by the Network for the purpose of configuring Network specific options on a User device. The field types and actual values in this page are determined by the particular Network.
0x0001	This page is reserved by ISO/IEC 13818-6 for the purpose of configuring the client-specific DSM-CC parameters. See Table [*Page1*]
0x0001-0x7fff	ISO/IEC 13818-6 Reserved.
0x8000-0xffff	User defined pages.

The following table defines the content of Page 1:

Table 9 DSM-CC Page 1 definition

Syntax	**Num. of Bytes**
tCSesCnf	**4**
tCRelCnf	**4**
tCDownloadInfoReq	**4**

The fields in the table above are defined in *Table[timers]*.

Each Client device which uses the DSM-CC U-N protocol must have a unique Physical Address (PA) associated with it. The Physical Address is a 6 byte value assigned to the Client device at the time of manufacture. This address must be unique to all devices which access a Network. In order to assure uniqueness, this address should be assigned from a block of physical MAC addresses which are assigned to the manufacturer by the IEEE.

The U-N Configuration messages are assumed to be part of a larger protocol suite which may include U-N Primitives, U-U Primitives and U-U Configuration. The U-N Configuration messages use the DSM-CC message header defined on Page **Error! Bookmark not defined.** of this document. For U-N Configuration messages, the dsmccType shall be 0x01 which identifies the message as an MPEG-2 DSM-CC User-to-Network configuration message.

3.1 UNConfigRequest message definition

This message is sent from a User to the Network at initialization to request that the network assign the User a network address and any network specific information. Table 10 defines the syntax of the UNConfigRequest message.

Table 10 DSM-CC UNConfigRequest message

Syntax	Num. of Bytes
UNConfigRequest(){	
deviceId	**6**
deviceType	**2**
deviceMajorRev	**2**
deviceMinorRev	**2**
}	

The **deviceId** field is defined in Table [User-to-Network Message Field Data Types]. It is a globally unique number which defines a User or Network device. The Network uses this address to configure the User device.

The **deviceType** field shall be set to indicate the type of User device requesting the configuration.

The **deviceMajorRev** field shall be set to indicate the major revision level of the User device.

The **deviceMinorRev** field shall be set to indicate the minor revision level of the User device.

3.2 UNConfigConfirm message definition

This message is sent from the Network to the User in response to a UNConfigRequest message. Table 11 defines the syntax of the UNConfigConfirm message.

Table 11 DSM-CC UNConfigConfirm message

Syntax	Num. of Bytes
UNConfigConfirm(){	
reason	**2**
deviceId	**6**
clientId	**20**
networkId	**20**
pageCount	**2**
for(i=0;i<pageCount;i++) {	
pageNumber	**2**
pageLength	**2**
for(i=0;i<pageLength;i++) {	
pageDataByte	**1**
}	
}	
}	

The **response** field shall be set by the Network to indicate the status of the config request. Codes for this field are defined in Table [*User-to-Network Reason Codes*].

The **deviceId** field is defined in Table [*User-to-Network Message Field Data Types*]. It is a globally unique number which defines a User or Network device.

The **clientId** field is defined in Table [*User-to-Network Message Field Data Types*]. It is a globally unique OSI NSAP address which identifies a Client.

The **networkId** is defined in Table [*User-to-Network Message Field Data Types*.]. It is a globally unique OSI NSAP address which is used to send subsequent User-to-Network messages.

The **pageCount** field shall be set to indicate the number of pages of data which are included in the message.

The **pageNumber** field shall be set to indicate the number of the page which follows. The content of this page is determined by the type of User which is indicated in the deviceId, deviceType, deviceMajorRev and deviceMinorRev fields.

The **pageLength** field shall be set to indicate the number of bytes of page data for this page.

The **pageDataByte** field shall be set to contain the data which makes up the page. There shall be exactly pageLength of pageDataBytes included in the message.

Note that the **pageCount** and **pageLength** fields could allow a large amount of data to be downloaded in this message. However, this message is delivered using the same mechanism as all User-to-Network messages which may have size constraints. If a large amount of data must be downloaded to the client, the Download mechanism defined in [Download Section] should be used.

3.3 UNConfigIndication message definition

This message is sent from the Network to the User to reconfigure a User device. Table 12 defines the syntax of the UNConfigIndication message.

Table 12 DSM-CC UNConfigIndication message

Syntax	Num. of Bytes
UNConfigIndication(){	
response	**2**
deviceId	**6**
clientId	**20**
networkId	**20**
pageCount	**2**
for(i=0;i<pageCount;i++) {	
pageNumber	**2**
pageLength	**2**
for(i=0;i<pageLength;i++) {	
pageDataByte	**1**
}	
}	
}	

The **response** field shall be set by the Network to indicate the status of the config request. Codes for this field are defined in Table [*User-to-Network Reason Codes*].

The **deviceId** field is defined in Table [*User-to-Network Message Field Data Types*]. It is a globally unique number which defines a User or Network device.

The **clientId** field is defined in Table [User-to-Network Message Field Data Types]. It is a globally unique OSI NSAP address which identifies a Client

The **networkId** field is defined in Table [User-to-Network Message Field Data Types]. It is a globally unique OSI NSAP address used for subsequent User-to-Network messages.

The **pageCount** field shall be set to indicate the number of pages of data which are included in the message.

The **pageNumber** field shall be set to indicate the number of the page which follows. The content of this page is determined by the type of User which is indicated in the deviceId, deviceType, deviceMajorRev and deviceMinorRev fields.

The **pageLength** field shall be set to indicate the number of bytes of page data for this page.

The **pageDataByte** field shall be set to contain the data which makes up the page. There shall be exactly pageLength of pageDataBytes included in the message.

3.4 UNConfigResponse message definition

This message is sent from the User to the Network in response to a UNConfigIndication message. Table 13 defines the syntax of the UNConfigResponse message.

Table 13 DSM-CC UNConfigResponse message

Syntax	Num. of Bytes
UNConfigResponse(){	
response	**2**
}	

The **response** field shall be set to indicate the Users response to the UNConfigIndication message. Codes for this field are defined in Table [*User-to-Network Reason Codes*].

3.5 UNPageRequest message definition

This message is sent from a User to the Network at initialization to request that the network send a specific page or pages of configuration data. Table 14 defines the syntax of the UNPageRequest message.

Table 14 DSM-CC UNPageRequest message

Syntax	Num. of Bytes
UNPageRequest(){	
deviceId	**6**
deviceType	**2**
deviceMajorRev	**2**
deviceMinorRev	**2**
pageCount	**2**
for(i=0;i<pageCount;i++) {	
pageNumber	**2**
}	
}	

The **deviceId** field is defined in Table [User-to-Network Message Field Data Types]. It is a globally unique number which defines a User or Network device.

The **deviceType** field shall be set to indicate the type of User device requesting the configuration pages.

The **deviceMajorRev** field shall be set to indicate the major revision level of the User device.

The **deviceMinorRev** field shall be set to indicate the minor revision level of the User device.

The **pageCount** and **pageNumber** fields shall be set to indicate the number of pages of configuration data being requested and the page number of each page being requested.

3.6 UNPageConfirm message definition

This message is sent from the Network to the User in response to a UNPageRequest message. Table 15 defines the syntax of the UNPageConfirm message.

Table 15 DSM-CC UNPageConfirm message

Syntax	Num. of Bytes
UNPageConfirm(){	
response	**2**
deviceId	**6**
pageCount	**2**
for(i=0;i<pageCount;i++) {	
pageNumber	**2**
pageLength	**2**
for(i=0;i<pageLength;i++) {	
pageDataByte	**1**
}	
}	
}	

The **response** field shall be set by the Network to indicate the status of the config request. Codes for this field are defined in Table [*User-to-Network Reason Codes*].

The **deviceId** field is defined in Table [User-to-Network Message Field Data Types]. It is a globally unique number which defines a User or Network device.

The **pageCount** field shall be set to indicate the number of pages of data which are included in the message.

The **pageNumber** field shall be set to indicate the number of the page which follows. The content of this page is determined by the type of User which is indicated in the deviceId, deviceType, deviceMajorRev and deviceMinorRev fields.

The **pageLength** field shall be set to indicate the number of bytes of page data for this page.

The **pageDataByte** field shall be set to contain the data which makes up the page. There shall be exactly pageLength of pageDataBytes included in the message.

3.7 UNPageIndication message definition

This message is sent from the Network to the User to update a page or pages of configuration data on a User device. Table 16 defines the syntax of the UNPageIndication message.

Table 16 DSM-CC UNPageIndication message

Syntax	Num. of Bytes
UNPageIndication(){	
response	**2**
deviceType	**6**
deviceMajorRev	**2**
deviceMinorRev	**2**
pageCount	**2**
for(i=0;i<pageCount;i++) {	
pageNumber	**2**
pageLength	**2**
for(i=0;i<pageLength;i++) {	
pageDataByte	**1**
}	
}	
}	

The **response** field shall be set by the Network to indicate the status of the config request. Codes for this field are defined in Table [*User-to-Network Reason Codes*].

The **deviceId** field shall be set to indicate the User device the configuration pages are for.

The **deviceType** field shall be set to indicate the type of User device the configuration pages are for.

The **deviceMajorRev** field shall be set to indicate the major revision level of the User device.

The **deviceMinorRev** field shall be set to indicate the minor revision level of the User device.

The **pageCount** field shall be set to indicate the number of pages of data which are included in the message.

The **pageNumber** field shall be set to indicate the number of the page which follows. The content of this page is determined by the type of User which is indicated in the deviceId, deviceType, deviceMajorRev and deviceMinorRev fields.

The **pageLength** field shall be set to indicate the number of bytes of page data for this page.

The **pageDataByte** field shall be set to contain the data which makes up the page. There shall be exactly pageLength of pageDataBytes included in the message.

3.8 UNPageResponse message definition

This message is sent from the User to the Network in response to a UNPageIndication message. Table 17 defines the syntax of the UNPageResponse message.

Table 17 DSM-CC UNPageResponse message

Syntax	Num. of Bytes
UNPageResponse(){	
response	**2**
}	

The **response** field shall be set to indicate the Users response to the UNPageIndication message. Codes for this field are defined in Table [*User-to-Network Reason Codes*].

3.9 User Initiated UNConfigRequest message Sequence

When a User device initializes, it sends a UNConfigRequest message to the Network. At the time of initialization, the User device does not know the address of the Network device so it must broadcast the request over the network using the broadcast mechanism provided by the Network.

When the Network device which processes U-N Configuration messages receives a configuration request from a Client device, it first validates the Client device to ensure that the device is authorized to operate on the Network. This validation may be done locally at the Network device or the Network device may use external directory services to obtain authorization and configuration information about the User device. After the Network device validates the User device, it assigns the device a network address that other devices on the network will use to communicate with the User device. This address may be assigned by the Network device or by the external directory service.

Figure 4 illustrates the sequence of events that occur for a Client device to request its address and configuration information from the Network using the User-to-Network bootstrap protocol. The Network optionally uses the external directory service to obtain address, configuration and authentication information about the User device.

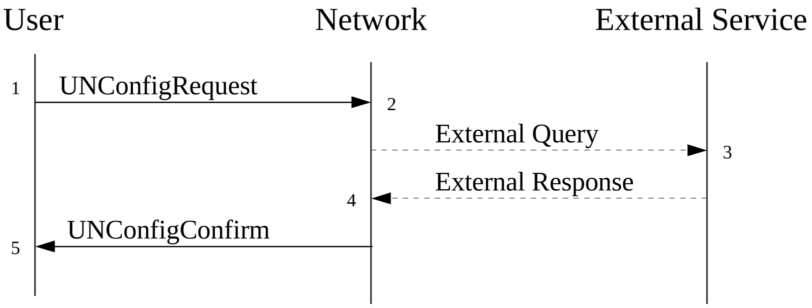


Figure 4 Sequence of events for Client initiated UNConfigRequest

Step 1 (User)

At configuration-time, the User device sends a UNConfRequest message over the network using the broadcast mechanism of the network. The messageId contains the code for a UNConfigRequest message and the deviceId field contains the factory assigned address of the Client device. The message also contains the deviceType, deviceMajorRev, and deviceMinorRev values.

Step 2 (Network)

The Network device that is responsible for configuring the User devices receives the message and determines if the User device is authorized to operate on the network. If the User device is not authorized, the Network device sends a UNConfigConfirm message addressed to the deviceId of the User device but containing a clientId and networkId of all zeros which indicates to the User device that it cannot connect to the network.

Step 3 (optional external directory service)

The Network may request configuration information for the device from an external directory service which is outside the scope of this specification.

Step 4 (Network)

If the User device is authorized to operate on the network, the Network device assigns the User device a network address and sends a UNConfigConfirm message addressed to the deviceId of the User device which contains the assigned address of the User device and the networkId of the Network device that will handle subsequent communications between the User device and the Network.

Step 5 (User)

When the User device receives the UNConfigConfirm message, it first checks the clientId field to determine if it has been assigned a network address. If this field is set to all 0's, the client shall not continue communicating on the Network. The User may however retry the UNConfigRequest message.

3.10 Network Initiated UNConfigIndication message Sequence

The Network may initiate a U-N Configuration by issuing a UNConfigIndication message which may be addressed to either the specific deviceId or broadcast to a group of Users using the broadcast mechanism of the Network. Only a specifically addressed message may be used to change the clientId of the Client device. A broadcast message may be used to change the networkId but, if a broadcast message contains a networkId of all zeros, the networkId at the User is not changed. These messages may be used to change the address of a User or to signal the User that the Network address has changed.

Figure 5 illustrates the sequence of events that occur during a Network initiated U-N configuration sequence. The Network optionally uses an external directory service to obtain address, configuration and authentication information about the User devices.

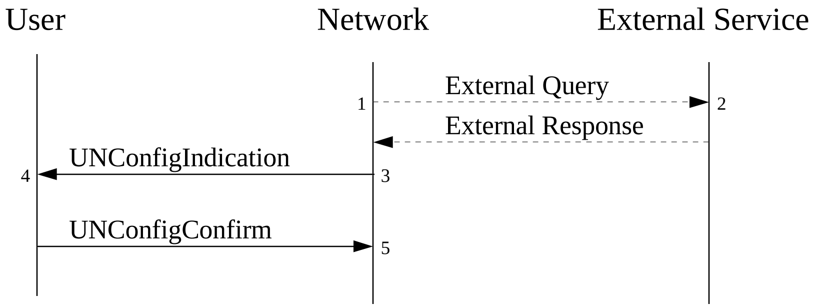


Figure 5 Sequence of events for Network initiated UNConfigRequest

Step 1 (optional external directory service)

The Network may request configuration information for the device from an external directory service which is outside the scope of this specification.

Step 2 (optional external directory service)

The external directory service sends a response to the Network which contains configuration information about the User device.

Step 3 (Network)

The Network issues a UNConfigIndication message to a User or group of Users. This message contains new configuration parameters which were obtained locally at the Network or optionally via an external directory service.

Step 4 (User)

The User updates its configuration parameters using the data received in the UNConfigIndication message. The User then sends a UNConfigResponse message to the Network indicating that it received the message.

Step 5 (Network)

When the Network device receives the UNConfigResponse message, the sequence is terminated.

3.11 User Initiated UNPageRequest message Sequence

The User device may request a specific page of configuration data anytime after it has successfully booted from the network. To do this it sends a UN Configuration Request message over the network using the address of the Network device.

The messageId contains the code for a Network Page Request message. The deviceId field contains the factory assigned address of the Client device. The deviceType, deviceMajorRev, and deviceMinorRev values are also used. The page number field is set to the user_data page that is being requested. If the page number field is all ones, all pages which apply to the deviceType, deviceMajorRev, and deviceMinorRev are sent sequentially to the Client device.

After a User device has been configured, it may request one or more pages of configuration data for its particular device type. Figure 6 shows the sequence of events that occur for a Client device to request a specific page of data.

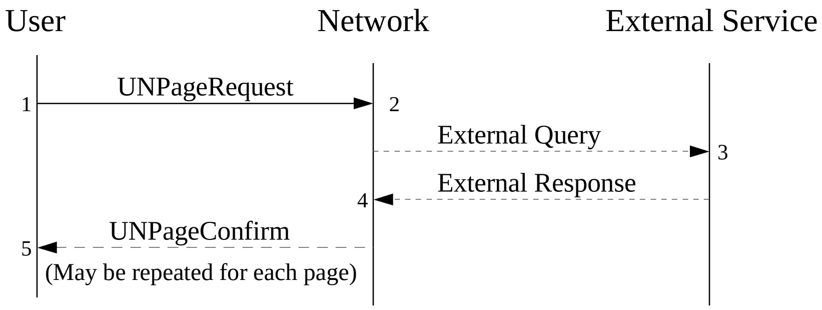


Figure 6. Sequence of events for Client initiated UNPageRequest

3.12 Network Initiated UNPageIndication message Sequence

The Network may send one or more pages of configuration data to a configured User device or it may broadcast the pages to a group of User devices. Figure 7 shows the sequence of events that occur when the Network sends unsolicited pages of data to User devices.

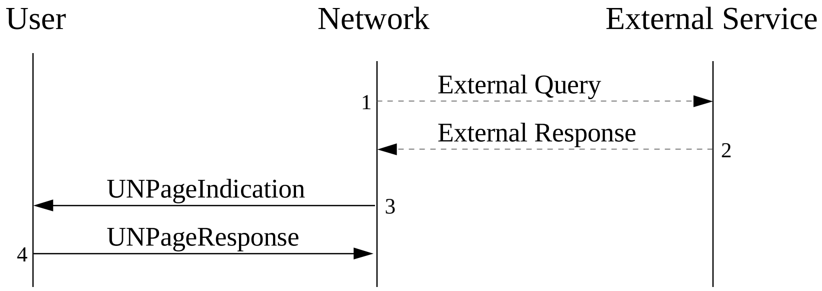


Figure 7. Sequence of events for Network initiated page indication

4. User-to-Network Messages

4.1 Overview and the General Message Format

[Although in principle a mechanism exists to deliver userData from U-N Client and Server SessionSetup messages to the appropriate User entity, the exact nature of this mechanism is not currently defined. Also, there may be different users of the userData (e.g. U-U entities, Download) and this use must be coordinated.]

[The current Network model and resource scenarios assume that the Server knows the resources required for a session and will make all of the Resource Add/Delete Requests. Further investigation will determine the need for Resource Add/Delete Request Scenarios from the Client or how a mixed environment of resource assignment/ownership would work. These capabilities are subjects for further study.]

[Note: Command sequences for ClientClearIndication and ServerClearIndication are tbd.]

The User-to-Network (U-N) messages are assumed to be part of a larger protocol stack. The U-N messages are designed to be carried on top of various protocols (e.g., UDP/IP, TCP/IP, or none). Constraints on specific lower level protocols are given in section *[Transport]*.

This section describes the format of all User-to-Network Messages including Network-to-Client and Network-to-Server messages. Subsequent sections describe how these messages are used in the operation of Network. User-to-Network messages are used to establish a session (and connections, as resources of the session) between a Client and a Server. Network assigned resources are described by resource descriptors, each of which consist of resource data elements.

The syntax has been designed to be extensible. Additional messages, resource descriptors used within those messages, and resource data elements which make up those resource descriptors may all be defined.

In general, **Request** messages are generated when the Server or Client initiates a message sequence. The Network responds to a Request message with a **Confirm** message. Messages which are sent asynchronously to the Server or Client from the Network are **Indication** messages. The Client and Server respond to an Indication message with a **Response** message.

Most of the control messages in this document use a request/response mechanism. When a Network, Client or Server issues a request message, the destination device issues a definite response to the request.

All messages between the Network and Users have a common message format. Table 18 defines the User-to-Network message format.

Table 18 General Format of DSM-CC User-Network Message

Syntax	Num. of Bytes
userNetworkMessagePacket() {	
DSMCCMessageHeader()	
MessagePayload()	
}	

The **userNetworkMessagePacket** is the general message format header defined in section (*[DSM-CC Message Header]*).

The **MessagePayload** differs depending on the function of the particular message; however, all **MessagePayload** fields are constructed from resource descriptors (see section *[Resource Descriptors]*) and data fields (see section *[User-to-Network Message Field Data Types]*).

4.2 Session Messages

This section defines the User-to-Network messages. Each message is identified by a specific messageId which is encoded to indicate the class and direction of the message. Table 19 defines the encoding of the messageId fields used in User-to-Network messages:

Table 19 Encoding of DSM-CC User-to-Network messageid

Bit Value	Description
14-15	messageDiscriminator.
2-13	messageScenario
0-1	messageType

The **messageDiscriminator** field is used to indicate if the message flow is between the Network and the Client or between the Network and the Server. Table 20 defines the possible values for the messageDiscriminator field.

Table 20 messageDiscriminator field values

messageDiscriminator	Message Flow
0x00	ISO/IEC 13818-6 Reserved.
0x01	Client and Network
0x02	Server and Network
0x03	ISO/IEC 138181-6 Reserved.

The **messageScenario** field is used to indicate the message sequence that the message is in. Table 21 defines the possible values for the messageScenario field.

[The following table may be removed -- is the information redundant with the messageId field in the U-N Messages table?]

Table 21 messageScenario field values

messageScenario	Description
0x000	ISO/IEC 13818-6 Reserved.
0x001	SessionSetUp
0x002	SessionRelease
0x003	AddResource
0x004	DeleteResource
0x005	PassThruMessage
0x006	StatusMessage
0x007	Clear
0x008	Proceeding
0x009	Connect
0x00a-7ff	ISO/IEC 138181-6 Reserved.
0x800-0xfff	User Defined Message Scenario

The **messageType** field is used to indicate the directionality of the message. Table 22 defines the possible values for the messageScenario field.

Table 22 messageType field values

messageType	Description
0x0	Request Message. This indicates that the message is being sent from the user to the Network to begin a scenario.
0x1	Confirm Message. This indicates that the message is being sent from the Network to the User in response to a Request message.
0x2	Indication Message. This indicates that the message is being sent from the Network to the User.
0x3	Response Message. This indicates that the message is being sent from the User to the Network in response to an Indication Message.

Table 23 defines the messageIds which are used in the DSM-CC User-to-Network messages.

Table 23 MPEG-2 DSM-CC U-N Messages

Command	messageId	messageId	Command
ISO/IEC 13818-6 reserved.	0x0000-0x400f	0x8000-0x800f	ISO/IEC 13818-6 reserved.
ClientSessionSetUpRequest	0x4010	0x8010	ServerSessionSetUpRequest
ClientSessionSetUpConfirm	0x4011	0x8011	ServerSessionSetUpConfirm
ClientSessionSetUpIndication	0x4012	0x8012	ServerSessionSetUpIndication
ClientSessionSetUpResponse	0x4013	0x8013	ServerSessionSetUpResponse
ISO/IEC 13818-6 reserved	0x4014-0x401f	0x8014-0x801f	ISO/IEC 13818-6 reserved
ClientReleaseRequest	0x4020	0x8020	ServerReleaseRequest
ClientReleaseConfirm	0x4021	0x8021	ServerReleaseConfirm
ClientReleaseIndication	0x4022	0x8022	ServerReleaseIndication
ClientReleaseResponse	0x4023	0x8023	ServerReleaseResponse
ISO/IEC 13818-6 reserved	0x4024-0x402f	0x8024-0x802f	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4030	0x8030	ServerAddResourceRequest
ISO/IEC 13818-6 reserved	0x4031	0x8031	ServerAddResourceConfirm
ClientAddResourceIndication	0x4032	0x8032	ISO/IEC 13818-6 reserved
ClientAddResourceResponse	0x4033	0x8033	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4034-0x403f	0x8034-0x803f	ISO/IEC 13818-6 reserved

Command	messageId	messageId	Command
ISO/IEC 13818-6 reserved	0x4040	0x8040	ServerDeleteResourceRequest
ISO/IEC 13818-6 reserved	0x4041	0x8041	ServerDeleteResourceConfirm
ClientDeleteResourceIndication	0x4042	0x8042	ISO/IEC 13818-6 reserved
ClientDeleteResourceResponse	0x4043	0x8043	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4044-0x404f	0x8044-0x804f	ISO/IEC 13818-6 reserved
ClientPassThruRequest	0x4050	0x8050	ServerPassThruRequest
ISO/IEC 13818-6 reserved	0x4051	0x8051	ISO/IEC 13818-6 reserved
ClientPassThruIndication	0x4052	0x8052	ServerPassThruIndication
ISO/IEC 13818-6 reserved	0x4053	0x8053	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4054-0x405f	0x8054-0x805f	ISO/IEC 13818-6 reserved
ClientStatusRequest	0x4060	0x8060	ServerStatusRequest
ClientStatusConfirm	0x4061	0x8061	ServerStatusConfirm
ClientStatusIndication	0x4062	0x8062	ServerStatusIndication
ClientStatusResponse	0x4063	0x8063	ServerStatusResponse
ISO/IEC 13818-6 reserved	0x4064-0x406f	0x8064-0x806f	ISO/IEC 13818-6 reserved
ClientClearRequest	0x4070	0x8070	ServerClearRequest
ISO/IEC 13818-6 reserved	0x4071	0x8071	ISO/IEC 13818-6 reserved
ClientClearIndication	0x4072	0x8072	ServerClearIndication
ISO/IEC 13818-6 reserved	0x4073	0x8073	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4074-0x407f	0x8074-0x807f	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4080	0x8080	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4081	0x8081	ISO/IEC 13818-6 reserved
ClientProceedingIndication	0x4082	0x8082	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4083	0x8083	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4084-0x408f	0x8084-0x808f	ISO/IEC 13818-6 reserved
ClientConnectRequest	0x4090	0x8090	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4091	0x8091	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4092	0x8092	ServerConnectIndication
ISO/IEC 13818-6 reserved	0x4093	0x8093	ISO/IEC 13818-6 reserved

Command	messageId	messageId	Command
ISO/IEC 13818-6 reserved	0x4094- 0x5fff	0x8094 - 0x9fff	ISO/IEC 13818-6 reserved
User defineable message ids	0x6000- 0x7fff	0xa000 - 0xffff	User defineable message ids

[Note: Command sequences for ClientClearIndication and ServerClearIndication are tbd.]

4.2.1 ClientSessionSetUpRequest message definition

This message is sent from a Client to the Network to request that a session be established with the requested serverId. The Network responds with a ClientSessionSetUpConfirm message. Before sending the ClientSessionSetUpConfirm message, the Network may send 0 or more ClientSessionProceedingIndication messages. Table 24 defines the syntax of the ClientSessionSetUpRequest message.

Table 24 DSM-CC U-N ClientSessionSetUpRequest message

Syntax	Num. of Bytes
ClientSessionSetUpRequest(){	
sessionId	**10**
clientId	**20**
serverId	**20**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

If the Network configuration indicates that the User (the originator of the command sequence) is responsible for generating the sessionId, the **sessionId** field shall be generated by the Client and be unique within the domain of the Network. If the Network configuration indicates that the Network is responsible for generating the sessionId, the **sessionId** field shall be set to 0 and the Network shall assign the sessionId in the ClientSessionSetUpConfirm message. The Network shall use the identical sessionId in all messages sent to the Client which refer to this session and the Client shall use the identical sessionId in all messages sent which refer to this session.

The **clientId** field shall be set by the Client and shall contain a value which uniquely identifies the the Client within the domain of the Network.

The **serverId** field shall be set by the Client and shall contain a value which uniquely identifies the Server with which the Client is establishing a session.

The **userDataCount** field specifies the number of userDataBytes that follow. The Client shall set the userDataCount to the number of userDataBytes that are included in the message. The userDataCount field shall be a value between 0 and 0x0400.

The **userDataByte** field is used to transport data transparently from the Client to the Server. When the Network sends the ServerSessionSetUpIndication message to the Server, the userDataByte field in the message shall be identical to the user data in the ClientSessionSetUpRequest message.

4.2.2 ClientSessionSetUpConfirm message definition

This message is sent from the Network to a Client in response to a ClientSessionRequest message. Table 25 defines the syntax of the ClientSessionSetUpConfirm message.

Table 25 DSM-CC U-N ClientSessionSetUpConfirm message

Syntax	Num. of Bytes
ClientSessionSetUpConfirm(){	
sessionId	**10**
response	**2**
resourceCount	**2**
for(i=0;i<resourceCount;i++) {	
resourceDescriptor()	
}	
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

If the Network configuration indicates that the User is responsible for assigning the sessionId, the Network shall set the **sessionId** field to the exact value of the sessionId which was received in the ClientSessionSetUpRequest message. If the Network configuration indicates that the Network is responsible for assigning the sessionId, the **sessionId** field shall be set to a unique value which identifies the session in the Network if the response field indicates that the session set-up request succeeded.

The **response** field shall be set by the Network to indicate the status of the session request. If this field is set to rspOK, this is an indication to the Client that the requested service has been established.

resourceCount and **resourceDescriptor** fields define the downstream and upstream resources which are assigned to the Client for this session. The resourceDescriptor fields shall be assigned by the Network. The number and type of resource descriptors that are passed depend on the User application and the type of service being requested. For all Client resources the requestType shall be non-negotiable.

The **userDataCount** field specifies the number of userDataBytes that follow. The Network shall set the userDataCount to the number of userDataBytes that are included in the message. The userDataCount field shall be a value between 0 and 0x0400.

The **userDataByte** field is used to transport data transparently from the Client to the Server. When the Network sends the ServerSessionSetUpIndication message to the Server, the userDataByte field in the message shall be identical to the user data in the ClientSessionSetUpRequest message.

4.2.3 ClientSessionSetUpIndication message definition

This message is sent from the Network to a Client to establish a session which was requested by the Server. Table 9 defines the syntax of the ClientSessionSetUpIndication message.

Table 26 DSM-CC U-N ClientSessionSetUpIndication message

Syntax	Num. of Bytes
ClientSessionSetUpIndication(){	
sessionId	**10**
clientId	**20**
serverId	**20**
resourceCount	**2**
for(i=0;i<resourceCount;i++) {	
resourceDescriptor()	
}	
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

If the Network configuration indicates that the User is responsible for assigning the sessionId, the Network shall set the **sessionId** field to the exact value of the sessionId which was received in the ServerSessionSetUpRequest message. If the Network configuration indicates that the Network is responsible for assigning the sessionId, the **sessionId** field shall be set to a unique value which identifies the session in the Network. The Client shall use this sessionId to identify this session in future messages.

The **clientId** field shall be set by the Network to the value of the clientId field which was received in the ServerSessionSetUpRequest message when the session was initially requested.

The **serverId** field shall be set by the Network to the value of the serverId field which was received in the ServerSessionSetUpRequest message when the session was initially requested. The Client shall use this field to identify the server which requested the session.

resourceCount and **resourceDescriptor** fields define the downstream and upstream resources which are assigned to the Client for this session. The resourceDescriptor fields shall be assigned by the Network. The number and type of resource descriptors that are passed depend on the User application and the type of service being requested.

userDataCount and **userDataByte** fields shall be set by the Network to be identical to the values of the same fields received in the ServerSessionSetUpRequest message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.4 ClientSessionSetUpResponse message definition

This message is sent from a Client to the Network in response to a ClientSessionSetUpIndication message. Table 27 defines the syntax of the ClientSessionSetUpResponse message.

Table 27 DSM-CC U-N ClientSessionSetUpResponse message

Syntax	Num. of Bytes
ClientSessionSetUpResponse(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set to the value of the sessionId field received in the ClientSessionSetUpIndication message.

The **response** field shall be set by the Client to a value which indicates the Client's response to the ClientSessionSetUpIndication message.

userDataCount and **userDataByte** fields shall be set by the Client to a value which shall be passed by the Network to the requesting Server in the ServerSessionSetUpConfirm message.

4.2.5 ClientReleaseRequest message definition

This message is sent from a Client to the Network to request that a session be torn-down. The Network responds with a ClientReleaseConfirm message. Before sending the ClientReleaseConfirm message, the Network shall also tear-down the session between the Network and the Server. Table 28 defines the syntax of the ClientReleaseRequest message.

Table 28 DSM-CC U-N ClientReleaseRequest message

Syntax	Num. of Bytes
ClientReleaseRequest(){	
sessionId	**10**
reason	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Client to the sessionId of the session that the Client is requesting to be torn-down.

The **reason** field shall be set by the Client to indicate the reason that the session is being requested to be torn-down.

The **userDataCount** and **userDataByte** fields are used to transport data transparently from the Client to the Server with the tear-down request. The Client shall set the userDataCount to the number of userDataBytes that are included in the message. When the Network sends the ServerReleaseIndication message to the Server, the userDataCount and userDataByte fields in the message shall be identical to those in the ClientReleaseRequest message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.6 ClientReleaseConfirm message definition

This message is sent from the Network to a Client in response to a ClientReleaseRequest message. Table 29 defines the syntax of the ClientReleaseConfirm message.

Table 29 DSM-CC U-N ClientReleaseConfirm message

Syntax	Num. of Bytes
ClientReleaseConfirm(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the value of the sessionId which was received in the ClientReleaseRequest message.

The **response** field shall be set by the Network to indicate the status of the session tear-down request. If this field is set to rspOK, this is an indication to the Client that the requested session has been released.

userDataCount and **userDataByte** fields shall be set by the Network to be identical to the values of the fields received in the ServerReleaseResponse message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataBytes field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.7 ClientReleaseIndication message definition

This message is sent from the Network to a Client initiate a session tear-down. Table 30 defines the syntax of the ClientReleaseIndication message.

Table 30 DSM-CC U-N ClientReleaseIndication message

Syntax	Num. of Bytes
ClientReleaseIndication(){	
sessionId	**10**
reason	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the value of the sessionId which is being requested to be torn-down.

The **reason** field shall be set by the Network to indicate the reason that the session is being torn-down. If the tear-down was initiated by the Server, this field shall be identical to the reason field which was received in the ServerReleaseRequest message.

userDataCount and **userDataByte** fields shall be set by the Network to be identical to the values of the same fields received in the ServerReleaseRequest message if the release was initiated by the server. If the release request was initiated by the Network the userDataCount field shall be set to 0 and no userDataBytes shall be sent. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.8 ClientReleaseResponse message definition

This message is sent from a Client to the Network in response to a ClientReleaseIndication message to indicate the Clients response to the request. Table 31 defines the syntax of the ClientReleaseResponse message.

Table 31 DSM-CC U-N ClientReleaseResponse message

Syntax	Num. of Bytes
ClientReleaseResponse(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set to the value of the sessionId field received in the ClientReleaseIndication message.

The **response** field shall be set by the Client to a value which indicates the Clients response to the ClientReleaseIndication message.

userDataCount and **userDataByte** fields shall be set by the Client to values which shall be sent to the Server in the ServerReleaseConfirm message.

4.2.9 ClientAddResourceIndication message definition

This message is sent from the Network to a Client to indicate that new resources have been added to the session as requested by the Server. Table 32 defines the syntax of the ClientAddResourceIndication message.

Table 32 DSM-CC U-N ClientAddResourceIndication message

Syntax	Num. of Bytes
ClientAddResourceIndication(){	
sessionId	**10**
resourceCount	**2**
for(i=0;i<resourceCount;i++) {	
resourceDescriptor()	
}	
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the value of the sessionId to which the resources are being added.

resourceCount and **resourceDescriptor** fields define any new resources which have been added to the Client side of the session. The resourceDescriptor fields shall be assigned by the Network. The number and type of resource descriptors that are passed depend on the User application and the type of service being requested.

userDataCount and **userDataByte** fields shall be set by the Network to be identical to the values of the same fields received in the ServerAddResourceRequest message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.10 ClientAddResourceResponse message definition

This message is sent from a Client to the Network in response to a ClientAddResourceIndication message to indicate the Clients response to the request. Table 33 defines the syntax of the ClientAddResourceResponse message.

Table 33 DSM-CC U-N ClientAddResourceResponse message

Syntax	Num. of Bytes
ClientAddResourceResponse(){	
sessionId	**10**
response	**2**
resourceCount	**2**
for(i=0;i<resourceCount;i++) {	
resourceDescriptor()	
}	
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set to the value of the sessionId field received in the ClientAddResourceIndication message.

The **response** field shall be set by the Client to a value which indicates the Client’s response to the ClientAddResourceIndication message.

resourceCount and **resourceDescriptor** fields contain the new resource descriptors which were added by the Network.

userDataCount and **userDataByte** fields shall be set by the Client to a value which shall be passed by the Network to the requesting Server in the ServerAddResourceConfirm message.

4.2.11 ClientDeleteResourceIndication message definition

This message is sent from the Network to a Client to indicate that resources have been deleted from the session as requested by the Server. Table 31 defines the syntax of the ClientDeleteResourceIndication message.

Table 34 DSM-CC U-N ClientDeleteResourceIndication message

Syntax	Num. of Bytes
ClientDeleteResourceIndication(){	
sessionId	**10**
reason	**2**
resourceCount	**2**
for(i=0;i<resourceCount;i++) {	
resourceNum	**2**
}	
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the value of the sessionId from which the resources are being deleted.

The **reason** field shall be set by the Network to be identical to the reason field received in the ServerDeleteResourceRequest message.

resourceCount and **resourceNum** fields define the resources which are being deleted from the Client side of the session. The sessionId in combination with the resourceNum comprise the resourceId.

userDataCount and **userDataByte** fields shall be set by the Network to be identical to the values of the same fields received in the ServerDeleteResourceRequest message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content, but the length of this field shall be the length specified by the userDataCount field.

4.2.12 ClientDeleteResourceResponse message definition

This message is sent from a Client to the Network in response to a ClientDeleteResourceIndication message to indicate the Client’s response to the request. Table 35 defines the syntax of the ClientDeleteResourceResponse message.

Table 35 DSM-CC U-N ClientDeleteResourceResponse message

Syntax	Num. of Bytes
ClientDeleteResourceResponse(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set to the value of the sessionId field received in the ClientDeleteResourceIndication message.

The **response** field shall be set by the Client to a value which indicates the Client’s response to the ClientDeleteResourceIndication message.

userDataCount and **userDataByte** fields shall be set by the Client to a value which shall be passed by the Network to the requesting Server in the ServerDeleteResourceConfirm message.

4.2.13 ClientPassThruRequest message definition

This message is sent from a Client to the Network to request that the network deliver a message to the requested Server. Table 36 defines the syntax of the ClientPassThruRequest message.

Table 36 DSM-CC U-N ClientPassThruRequest message

Syntax	Num. of Bytes
ClientPassThruRequest(){	
clientId	**20**
serverId	**20**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **clientId** field shall be set by the client to indicate the identifier of the Client which is sending the message.

The **serverId** field shall be set by the client to indicate the identifier of the Server which the message is being sent to.

The **userDataCount** and **userDataByte** fields are used to transport the message to the indicated serverId.

4.2.14 ClientPassThruIndication message definition

This message is sent from the Network to a Client to deliver a message from the indicated Server. Table 37 defines the syntax of the ClientPassThruIndication message.

Table 37 DSM-CC U-N ClientPassThruIndication message

Syntax	Num. of Bytes
ClientPassThruIndication(){	
clientId	**20**
serverId	**20**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **clientId** field shall be set by the Network to indicate the identifier of the Client to which the message was sent.

The **serverId** field shall be set by the Network to indicate the identifier of the Server which sent the message.

The **userDataCount** and **userDataByte** fields are used to transport the message from the indicated serverId.

4.2.15 ClientStatusRequest message definition

This message is sent from a Client to the Network to request a status message. Table 38 defines the syntax of the ClientStatusRequest message.

Table 38 DSM-CC U-N ClientStatusRequest message

Syntax	Num. of Bytes
ClientStatusRequest(){	
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **statusType** field shall be set by the client to indicate the type of status being requested.

The **statusCount** and **statusByte** fields are used to transport any data which is required to indicate additional information about the status being requested.

4.2.16 ClientStatusConfirm message definition

This message is sent from the Network to a Client in response to a ClientStatusRequest message. Table 39 defines the syntax of the ClientStatusConfirm message.

Table 39 DSM-CC U-N ClientStatusConfirm message

Syntax	Num. of Bytes
ClientStatusConfirm(){	
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **statusType** field shall be set by the Network to indicate the type of status being returned.

statusCount and **statusByte** fields shall be set by the Network to contain the status information indicated by the statusType field.

4.2.17 ClientStatusIndication message definition

This message is sent from the Network to a Client to request a status message from the Client. Table 40 defines the syntax of the ClientStatusIndication message.

Table 40 DSM-CC U-N ClientStatusIndication message

Syntax	Num. of Bytes
ClientStatusIndication(){	
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **statusType** field shall be set by the Network to indicate the type of status being requested.

The **statusCount** and **statusByte** fields are used to transport any additional data which is required to indicate additional information about the status being requested..

4.2.18 ClientStatusResponse message definition

This message is sent from a Client to the Network in response to a ClientStatusIndication message to indicate the Clients response to the request. Table 41 defines the syntax of the ClientStatusResponse message.

Table 41 DSM-CC U-N ClientStatusResponse message

Syntax	Num. of Bytes
ClientStatusResponse(){	
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **statusType** field shall be set by the Network to indicate the type of status being returned.

statusCount and **statusByte** fields shall be set by the Network to contain the status information indicated by the statusType field.

4.2.19 ClientSessionClearRequest message definition

This message is sent from a Client to the Network to unconditionally terminate a session at the Network. The Network shall also send a ServerSessionClearIndication to the Server upon receiving this message. Table 42 defines the syntax of the ClientSessionClearRequest message.

Table 42 DSM-CC U-N ClientSessionClearRequest message

Syntax	Num. of Bytes
ClientSessionClearRequest(){	
sessionId	**10**
reason	**2**
}	

The **sessionId** field shall be set by the client to indicate the session which is being cleared.

The **reason** field is used to indicate the reason that the session is being cleared.

4.2.20 ClientSessionClearIndication message definition

This message is sent from the Network to a Client to unconditionally terminate a session on the Client. Table 43 defines the syntax of the ClientSessionClearIndication message.

Table 43 DSM-CC U-N ClientSessionClearIndication message

Syntax	Num. of Bytes
ClientSessionClearIndication(){	
sessionId	**10**
reason	**2**
}	

The **sessionId** field shall be set by the Network to indicate the session which is being cleared.

The **reason** shall be set by the Network to indicate the reason that the session is being cleared.

4.2.21 ClientSessionProceedingIndication message definition

This message is sent from the Network to a Client in response to a ClientSessionSetUpRequest message to inform the client that the request is being processed. The Network may send this message 0 or more times before sending the ClientSessionSetUpConfirm message. The Client shall reset timer **Error! Reference source not found.** to its initial value upon receipt of this message. Table 44 defines the syntax of the ClientSessionProceedingIndication message.

Table 44 DSM-CC U-N ClientSessionProceedingIndication message

Syntax	Num. of Bytes
ClientSessionProceedingIndication(){	
reason	**2**
}	

The **reason** field shall be set by the Network to indicate the reason that the ClientSessionProceeding message is being sent.

4.2.22 ClientConnectRequest message definition

This is an optional message which is sent from a Client to the Network to signal the network that the Client has connected to a session and is ready to proceed with User-to-User messages. Table 45 defines the syntax of the ClientConnectRequest message.

Table 45 DSM-CC U-N ClientConnectRequest message

Syntax	Num. of Bytes
ClientConnectRequest(){	
sessionId	**10**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Client to indicate the session which has been connected.

The **userDataCount** and **userDataByte** fields are used to transport data transparently from the Client to the Server with the connect indication. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.23 ServerSessionSetUpRequest message definition

This message is sent from a Server to the Network to request that a session be established with the requested clientId. The Network responds with a ServerSessionSetUpConfirm message. Table 46 defines the syntax of the ServerSessionSetUpRequest message.

Table 46 DSM-CC U-N ServerSessionSetUpRequest message

Syntax	Num. of Bytes
ServerSessionSetUpRequest(){	
sessionId	**10**
serverId	**20**
clientId	**20**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Server and be unique within the domain of the Network if the Network configuration indicates that the User is responsible for generating the sessionId. The Network shall use the identical sessionId in all messages sent to the Server which refer to this session and the Server shall use the identical sessionId in all messages sent which refer to this session. If the Network configuration indicates that the Network is responsible for generating the sessionId, this field shall be set to 0 and the Network shall assign the sessionId in the ServerSessionSetUpConfirm message.

The **serverId** field shall be set by the Server and uniquely identify the Server within the domain of the Network.

The **clientId** field shall be set by the Server and uniquely identify the client with which the Server is requesting a session be established.

The **userDataCount** and **userDataByte** fields are used to transport data transparently from the Server to the Client with the set-up request. The Server shall set the userDataCount to the number of userDataBytes that are included in the message. When the Network sends the ClientSessionSetUpIndication message to the Client, the userDataCount and userDataByte fields in the message shall be identical to those in the ServerSessionSetUpRequest message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.24 ServerSessionSetUpConfirm message definition

This message is sent from the Network to a Server in response to a ServerSessionSetUpRequest message. Table 47 defines the syntax of the ServerSessionSetUpConfirm message.

Table 47 DSM-CC U-N ServerSessionSetUpConfirm message

Syntax	Num. of Bytes
ServerSessionSetUpConfirm(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network. If the Network configuration indicates that the User is responsible for assigning the sessionId, this field shall be set to the exact value of the sessionId which was received in the ServerSessionSetUpRequest message. If the Network configuration indicates that the Network is responsible for assigning the sessionId, this field shall be set to a unique value which identifies the session in the Network if the response field indicates that the session set-up request succeeded.

The **response** field shall be set by the Network to indicate the status of the session request. If this field is set to rspOK, this is an indication to the Server that the requested service has been established.

userDataCount and **userDataByte** fields shall be set by the Network to be identical to the values of the fields received in the ClientResourceResponse message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataBytes field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.25 ServerSessionSetUpIndication message definition

This message is sent from the Network to a Server to establish a session which was requested by a Client. Table 48 defines the syntax of the ServerSessionSetUpIndication message.

Table 48 DSM-CC U-N ServerSessionSetUpIndication message

Syntax	Num. of Bytes
ServerSessionSetUpIndication(){	
sessionId	**10**
clientId	**20**
serverId	**20**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

If the Network configuration indicates that the User is responsible for assigning the sessionId, the Network shall set the **sessionId** field to the exact value of the sessionId which was received in the ClientSessionSetUpRequest message. If the Network configuration indicates that the Network is responsible for assigning the sessionId, the **sessionId** field shall be set to a unique value which identifies the session in the Network. The Server shall use this sessionId to identify this session in future messages.

The **clientId** field shall be set by the Network to the value of the clientId field which was received in the ClientSessionSetUpRequest message when the session was initially requested. The Server shall use this field to identify the client which has requested the session.

The **serverId** field shall be set by the Network to the value of the serverId field which was received in the ClientSessionSetUpRequest message when the session was initially requested.

userDataCount and **userDataByte** fields shall be set by the Network to be identical to the values of the same fields received in the ClientSessionSetUpRequest message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.26 ServerSessionSetUpResponse message definition

This message is sent from a Server to the Network in response to a ServerSessionSetUpIndication message. Table 49 defines the syntax of the ServerSessionSetUpResponse message.

Table 49 DSM-CC U-N ServerSessionSetUpResponse message

Syntax	Num. of Bytes
ServerSessionSetUpResponse(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set to the value of the sessionId field received in the ServerSessionSetUpIndication message.

The **response** field shall be set by the Server to a value which indicates the Server’s response to the ServerSessionSetUpIndication message.

userDataCount and **userDataByte** fields shall be set by the Server to a value which shall be passed by the Network to the requesting Client in the ClientSessionSetUpConfirm message.

4.2.27 ServerConnectIndication message definition

This is an optional message which is sent from a Network to the Server to signal the network that the Client has connected to a session and is ready to proceed with User-to-User messages. The Network sends this message upon receipt of the ClientConnectIndication message. Table 50 defines the syntax of the ServerConnectIndication message.

Table 50 DSM-CC U-N ServerConnectIndication message

Syntax	Num. of Bytes
ServerConnectIndication(){	
sessionId	**10**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the sessionId which was received in the ClientConnectRequest message.

The **userDataCount** and **userDataByte** fields are used to transport data transparently from the Client to the Server with the connect indication. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.28 ServerReleaseRequest message definition

This message is sent from a Server to the Network to request that a session be torn-down. The Network responds with a ServerReleaseConfirm message. Before sending the ServerReleaseConfirm message, the Network shall also tear-down the session between the Network and the Client. Table 51 defines the syntax of the ServerReleaseRequest message.

Table 51 DSM-CC U-N ServerReleaseRequest message

Syntax	Num. of Bytes
ServerReleaseRequest(){	
sessionId	**10**
reason	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Server to the sessionId of the session that the Server is requesting to be torn-down.

The **reason** field shall be set by the Server to indicate the reason that the session is being requested to be torn-down.

The **userDataCount** field specifies the number of userDataBytes that follow. The Server shall set the userDataCount to the number of userDataBytes that are included in the message. The userDataCount field shall be a value between 0 and 0x0400.

The **userDataByte** field is used to transport data transparently from the Server to the Client. When the Network sends the ClientSessionSetUpIndication message to the Client, the userDataByte field in the message shall be identical to the user data in the ServerSessionSetUpRequest message.

4.2.29 ServerReleaseConfirm message definition

This message is sent from the Network to a Server in response to a ServerReleaseRequest message. Table 52 defines the syntax of the ServerReleaseConfirm message.

Table 52 DSM-CC U-N ServerReleaseConfirm message

Syntax	Num. of Bytes
ServerReleaseConfirm(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the value of the sessionId which was received in the ServerReleaseRequest message.

The **response** field shall be set by the Network to indicate the status of the session tear-down request. If this field is set to rspOK, this is an indication to the Server that the requested session has been released.

The **userDataCount** field specifies the number of userDataBytes that follow. The Network shall set the userDataCount to the number of userDataBytes that are included in the message. The userDataCount field shall be a value between 0 and 0x0400.

The **userDataByte** field is used to transport data transparently from the Server to the Client. When the Network sends the ClientSessionSetUpIndication message to the Client, the userDataByte field in the message shall be identical to the user data in the ServerSessionSetUpRequest message.

4.2.30 ServerReleaseIndication message definition

This message is sent from the Network to a Server to initiate a session tear-down which was requested by the Client. Table 53 defines the syntax of the ServerReleaseIndication message.

Table 53 DSM-CC U-N ServerReleaseIndication message

Syntax	Num. of Bytes
ServerReleaseIndication(){	
sessionId	**10**
reason	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the value of the sessionId which is being requested to be torn-down.

The **reason** field shall be set by the Network to indicate the reason that the session is being torn-down. If the tear-down was initiated by the Client, this field shall be identical to the reason field which was received in the ClientReleaseRequest message.

userDataCount and **userDataByte** fields shall be set by the Network to be identical to the values of the same fields received in the ClientReleaseRequest message. The userDataCount field shall be a value between 0x0 and 0x0400. The value of the userDataByte field is not restricted to content but, the length of this field shall be the length specified by the userDataCount field.

4.2.31 ServerReleaseResponse message definition

This message is sent from a Server to the Network in response to a ServerReleaseIndication message. Table 54 defines the syntax of the ServerReleaseResponse message.

Table 54 DSM-CC U-N ServerReleaseResponse message

Syntax	Num. of Bytes
ServerReleaseResponse(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set to the value of the sessionId field received in the ServerReleaseIndication message.

The **response** field shall be set by the Server to a value which indicates the Server’s response to the ServerReleaseIndication message.

userDataCount and **userDataByte** fields shall be set by the Server to values which shall be sent to the Client in the ClientReleaseConfirm message.

4.2.32 ServerAddResourceRequest message definition

This message is sent from a Server to the Network to request that resources be added to a session. The Network responds with a ServerAddResourceConfirm message. Table 55 defines the syntax of the ServerAddResourceRequest message.

Table 55 DSM-CC U-N ServerAddResourceRequest message

Syntax	Num. of Bytes
ServerAddResourceRequest(){	
sessionId	**10**
resourceCount	**2**
for(i=0;i<resourceCount;i++) {	
resourceDescriptor()	
}	
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Server to the sessionId of the session to which the resources are being added.

The **resourceCount** and **resourceDescriptor** fields indicate the resources that the server is requesting that the Network add to the session.

userDataCount and **userDataByte** fields shall be set by the Server to values which shall be sent to the Client in the ClientAddResourcesIndication message.

4.2.33 ServerAddResourceConfirm message definition

This message is sent from the Network to a Server in response to a ServerAddResourceRequest message. Table 56 defines the syntax of the ServerAddResourceConfirm message.

Table 56 DSM-CC U-N ServerAddResourceConfirm message

Syntax	Num. of Bytes
ServerAddResourceConfirm(){	
sessionId	**10**
response	**2**
resourceCount	**2**
for(i=0;i<resourceCount;i++) {	
resourceDescriptor()	
}	
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the value of the sessionId which was received in the ServerAddResourceRequest message.

The **response** field shall be set by the Network to indicate the result of the add resource request.

resourceCount and **resourceDescriptor** fields shall be set by the Network to values which were assigned to the requested resources.

userDataCount and **userDataByte** fields shall be set by the Network to the values which were received in the ClientAddResourceResponse message.

4.2.34 ServerDeleteResourceRequest message definition

This message is sent from a Server to the Network to request that resources be deleted from a session. The Network responds with a ServerDeleteResourceConfirm message. Table 57 defines the syntax of the ServerDeleteResourceRequest message.

Table 57 DSM-CC U-N ServerDeleteResourceRequest message

Syntax	Num. of Bytes
ServerDeleteResourceRequest(){	
sessionId	**10**
reason	**2**
resourceCount	**2**
for(i=0;i<resourceCount;i++) {	
resourceNum	**2**
}	
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Server to the sessionId of the session that the resources are being deleted from.

The **reason** field shall be set to indicate the reason that the resources are being deleted.

The **resourceCount** and **resourceNum** fields indicate the resources that the server is requesting that the Network delete from the session. The sessionId field in conjunction with the resourceNum field comprise the resourceId.

The **userDataCount** and **userDataByte** fields are used to transport data transparently from the Server to the Client in the ClientDeleteResourceIndication message. The Server shall set the userDataCount to the number of userDataBytes that are included in the message. The userDataCount field shall be a value between 0x0 and 0x0400.

4.2.35 ServerDeleteResourceConfirm message definition

This message is sent from the Network to a Server in response to a ServerDeleteResourceRequest message. Table 58 defines the syntax of the ServerDeleteResourceConfirm message.

Table 58 DSM-CC U-N ServerDeleteResourceConfirm message

Syntax	Num. of Bytes
ServerDeleteResourceConfirm(){	
sessionId	**10**
response	**2**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **sessionId** field shall be set by the Network to the value of the sessionId which was received in the ServerDeleteResourceRequest message.

The **response** field shall be set by the Network to indicate the result of the ServerDeleteResourceRequest message.

The **userDataCount** and **userDataByte** fields shall be set by the Network to the values received in the ClientDeleteResourceResponse message.

4.2.36 ServerPassThruRequest message definition

This message is sent from a Server to the Network to request that the network deliver a message to the requested User. Table 59 defines the syntax of the ServerPassThruRequest message.

Table 59 DSM-CC U-N ServerPassThruRequest message

Syntax	Num. of Bytes
ServerPassThruRequest(){	
clientId	**20**
serverId	**20**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **clientId** field shall be set by the server to indicate the identifier of the User which the message is being sent to.

The **serverId** field shall be set by the server to indicate the identifier of the User which is sending the message.

The **userDataCount** and **userDataByte** fields are used to transport the message to the indicated clientId. The userDataCount field shall be a value between 0x0 and 0x0400.

4.2.37 ServerPassThruIndication message definition

This message is sent from the Network to a Server to deliver a message from the indicated User. Table 60 defines the syntax of the ServerPassThruIndication message.

Table 60 DSM-CC U-N ServerPassThruIndication message

Syntax	Num. of Bytes
ServerPassThruIndication(){	
clientId	**20**
serverId	**20**
userDataCount	**2**
for(i=0;i<userDataCount;i++) {	
userDataByte	**1**
}	
}	

The **clientId** field shall be set by the Network to indicate the identifier of the User which sent the message.

The **serverId** field shall be set by the Network to indicate the identifier of the User to which the message was sent.

The **userDataCount** and **userDataByte** fields are used to transport the message from the indicated serverId. The userDataCount field shall be a value between 0x0 and 0x0400.

4.2.38 ServerStatusRequest message definition

This message is sent from a Server to the Network to request a status message. Table 61 defines the syntax of the ServerStatusRequest message.

[The association of the request parameter with data returned needs to be defined explicitly.]

Table 61 DSM-CC U-N ServerStatusRequest message

Syntax	Num. of Bytes
ServerStatusRequest(){	
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **statusType** field shall be set by the server to indicate the type of status being requested.

The **statusCount** and **statusByte** fields are used to transport any additional data which is required to indicate additional information about the status being requested.

4.2.39 ServerStatusConfirm message definition

This message is sent from the Network to a Server in response to a ServerStatusRequest message. Table 62 defines the syntax of the ServerStatusConfirm message.

Table 62 DSM-CC U-N ServerStatusConfirm message

Syntax	Num. of Bytes
ServerStatusConfirm(){	
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **statusType** field shall be set by the Network to indicate the type of status being returned.

statusCount and **statusByte** fields shall be set by the Network to contain the status information indicated by the statusType field.

4.2.40 ServerStatusIndication message definition

This message is sent from the Network to a Server to request a status message from the Server. Table 63 defines the syntax of the ServerStatusIndication message.

Table 63 DSM-CC U-N ServerStatusIndication message

Syntax	Num. of Bytes
ServerStatusIndication(){	
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **statusType** field shall be set by the Network to indicate the type of status being requested.

The **statusCount** and **statusByte** fields are used to transport any additional data which is required to indicate additional information about the status being requested.

4.2.41 ServerStatusResponse message definition

This message is sent from a Server to the Network in response to a ServerStatusIndication. Table 64 defines the syntax of the ServerStatusResponse message.

Table 64 DSM-CC U-N ServerStatusResponse message

Syntax	Num. of Bytes
ServerStatusResponse(){	
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **statusType** field shall be set by the Network to indicate the type of status being returned.

statusCount and **statusByte** fields shall be set by the Network to contain the status information indicated by the statusType field.

4.2.42 ServerSessionForwardRequest message definition

[To be provided]

4.2.43 ServerSessionTransferRequest message definition

[To be provided]

4.2.44 MPEG-2 DSM-CC statusTypes

The **statusType** field is used to indicate the type of status being requested or returned. Table 65 defines the possible statusTypes.

Table 65 MPEG-2 DSM-CC statusType values

statusType	Description	User Status Request / Indication	User Status Response / Confirm
0x0000	ISO/IEC 13818-6 Reserved		
0x0001	ISO/IEC 13818-6 Reserved		

0x0002	Identify Session Status. This status indicates the current status of a session.	sessionId	sessionId
			response
			serverId
			resourceCount
			for(resourceCount)
			{
			resourceDescriptor()
			}

0x0003	Identify Configuration. This status indicates the current configuration.	*[TBD]*	*[TBD]*
0x0004-0x7fff	ISO/IEC 13818-6 Reserved.		
0x8000-0xffff	User Defined statusType.		

4.3 User-to-Network Message Field Data Types

This table may be omitted after the data types have been moved into the message descriptions

Table 66 defines the data fields used in the User-to-Network messages.

Table 66 User-to-Network Message Field Data Types

Field Name	**Length (Bytes)**	**Range**	**Description**
cfSessionId	10	composite field	This field is identical to the sessionId field. It is used to connect a session to a continuous feed session. It contains the sessionId of the continuous feed session which is to be connected to the session.
clientId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Client. The clientId must be a specific address or be able to be resolved to a specific address by the Network. *[reference ISO OSI NSAP Specification]*.
deviceId	6	0x000000000000 - 0x3fffffffffff	A globally unique number which defines a User or Network device. For networks which use the ATM BHLI field, the second most significant bit of the deviceId is set to 1 to indicate that the last three octets contain the deviceNum. For example, in the IEEE 802 MAC if the second most significant bit is set to 1 in the OUI (Organization Unique Identifier), the last three octets are set by the OU (Organizational Unit).
NetworkId *[use is still tbd]*	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies the Network. The serverId must be a specific address or be able to be resolved to a specific address by the Network. *[reference ISO OSI NSAP Specification]*.
originatorId	*[TBD]*	*[TBD]*	*[TBD]*
reason	2	0x0000 - 0xffff	This field indicates the reason for a failure or action.
resourceCount	1	0x00 - 0xff	This field contains the number of resource descriptors which follow the resourceCount field in the message.
resourceDescriptor	variable	descriptor dependent	An array of resource descriptors of size resourceCount. See section *[resource descriptors]*.

Field Name	**Length (Bytes)**	**Range**	**Description**
serverId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Server. The serverId must be a specific address or be able to be resolved to a specific address by the Network. *[reference ISO OSI NSAP Specification]*.
sessionId	10	composite	The sessionId field shall consist of a unique 6 byte deviceId and a 4 byte session number. The sessionId may be assigned by either the User who initiates a session request or by the Network. This is determined by the U-N configuration protocol or by some other network configuration method.
sessionNum	4	0x00000000 - 0xffffffff	A number which uniquely identifies a session on the device which assigns the sessionId.
statusCount	2	0x0000 - 0xffff	This field indicates the number of status records that are being returned in a status response.
statusType	1	enumerated	This field indicates the type of status that is being requested or returned in a status transaction.
TransactionId *[Part of Message Header now, so may be removed from this table]*	*[TBD pending resolution of originatorId]*	*[TBD pending resolution of originatorId]*	This field is used to correlate request messages with confirm messages and indication messages with response messages. The transactionId is composed of the originatorId and the transactionNum.
transactionNum	2	0x0000 - 0xffff	The transactionNum combined with the originatorId forms the transactionId. The transactionNum is unique to the device which generates the message.
userDataByte	userDataCount	binary data	The userDataByte field contains data which is passed through the Network to the destination User device.
userDataCount	2	varies depending on message length and type of message	The userDataCount field contains the number of bytes of data which are included in a message.
userId	*[tbd]*	*[tbd]*	This field uniquely identifies an individual User device. *[Use of this parameter?]*

4.3.1 OSI NSAP Address Format

This section applies to the clientId, serverId, and networkId. These fields will always be a fixed 20-byte OSI NSAP. A generic OSI NSAP address consists of two domains:

1. Initial Domain Part (IDP) which consists of two sub-parts:

    a. 1-byte Authority and Format Identifier (AFI)

    b. variable-length Initial Domain Identifier (IDI) which depends on the value of the AFI.

2. Domain Specific Part (DSP) which depends on the value of the IDI.

The NSAP address type described here is already defined in ISO/IEC 8348. While any AFI is acceptable for DSM-CC (i.e., inter-networking is beyond the scope of DSM-CC), an example is the ATM End System Address as defined in ATM Forum's UNI Specification, Version 3.1. The characteristics of this format are as follows:

IDP				
	IDI	DSP		
AFI 1-byte	E.164 8-byte	HO-DSP 4-byte	ESI 6-byte	SEL 1-byte

where the

Total length:	20 bytes
AFI:	45 (ISO/IEC 8348 registered)
IDI:	8-byte BCD-encoded E.164 address
DSP:	Contains the Internet Protocol (IP) address in the 4-byte High Order-DSP (HO-DSP), the MAC address in the 6-byte End System Identifier (ESI), and a subscriber's identifier in the 1-byte Selector (SEL).

For Clients, the E.164 address in the IDI identifies an ATM-to-the-curb drop. The MAC address identifies the set-top terminal that is serviced by the drop. The SEL byte allows the set-top terminal to support up to 256 logical subscribers from one hardware platform, such as in a dormitory environment where one terminal may be shared by more than one roommate.

For servers, the E.164 address identifies the ATM address of the server. The ESI identifies a service that runs in that server.

IP addresses can be embedded within the above format to be used by the interactive multi-media applications between the Client and Server (User-to-User communication).[(*It is assumed that Ipv6 will be acommadated into the OSI NSAP. This work is outside the scope of DSM-CC)]*

4.4 Resource Descriptors

Session Set-Up and Resource Allocation message classes use resource descriptors to request and assign the various resources to a session. Table 67 defines the general format of a resource descriptor:

Table 67 DSM-CC User-Network Resource Descriptor

Syntax	Num. of Bytes
userNetworkResourceDescriptor() {	
resourceRequestId	**2**
requestType	**2**
resourceType	**2**
resourceNum	**2**
resourceLength	**2**
for(i=0;i<resourceLength;i++) {	
resourceDataByte	**1**
}	
}	

The **resourceRequestId** field is set by the User to correlate the resource specified in the Request message with the result given in the Confirm message. The Network shall return this resourceRequestId the Confirm message.

[If the User can set the ResourceNum, then a separate resourceRequestId to track individual resources is not needed -- this is under study]

The **requestType** field defines how the Server and Network will negotiate a resource. This field shall be defined as 0xff in ClientIndication and ClientResponse message types. The requestType field may have the following values:

Table 68 DSM-CC U-N Resource Descriptor requestTypes

requestType	In Message Type	Value	Description
Mandatory & Non-Negotiable	Request	0x00	Indicates that the Network must either satisfy the requested value exactly or the entire Resource Request command sequence fails.
Mandatory, but Negotiable	Request	0x01	Indicates that the Network must either satisfy the negotiable range/list of values or the entire resource request sequence fails. If the range/list has value *tbd*, then the sequence only fails if no resources are available.
Non-Mandatory, but Non-Negotiable	Request	0x02	Indicates that the Network must either satisfy the requested value exactly or the resource assignment fails (does not affect that state of the Resource Request command sequence).
Non-Mandatory, & Negotiable	Request	0x03	Indicates that the Network may either satisfy the negotiable range/list of values value exactly or the resource assignment fails (does not affect that state of the Resource Request command sequence). If the range/list has value *tbd*, then any resource value may be assigned (“don’t care” condition).
Recommended	Confirm	0x10	Indicates that an alternative value within the specified range/list was assigned in response to a Negotiable resource.
Assigned	Confirm	0x30	Indicates that the exact resource value was

requestType	In Message Type	Value	Description
			assigned
Failed	Confirm	0x31	Indicates that the Network was unable to assign a resource to satisfy the requestType.
Unprocessed	Confirm	0x32	Indicates that the Network did not process the request because a Mandatory resource failed prior to the processing of this descriptor.
Invalid	Confirm	0x33	Indicates that the resource requested is not valid.
reserved		0x0a - 0xfe	ISO/IEC 13818-6 reserved.
reserved	Indication, Response	0xff	ISO/IEC 13818-6 reserved.

For each requested resource descriptor that the User sends to the SM to request a new resource, the User shall indicate its resquestType as:

1. Mandatory&Non-Negotiable
2. Mandatory&Negotiable
3. Non-Mandatory&Non-Negotiable
4. Non-Mandatory&Negotiable

A resource's attribute is conveyed through the requestType parameter field in the requested resource descriptor. When multiple resources are included in the AddResourceRequest, the requested resources should be grouped and processed in the above listed order.

A User specifies a resource as Mandatory&Non-Negotiable when it will not consider an alternative offered by the SM in case of failure. Thus, the SM will not propose one. The failure of a Mandatory&Non-Negotiable resource will cause the failure of this resource and the remaining resource requests within the same AddResourceRequest message shall not be processed.

A User specifies a resource as Mandatory&Negotiable when it will consider an alternative offered by the SM in case of failure. Thus, if possible, the SM will propose one within the range given in the resource descriptor. For any Mandatory&Negotiable resource that the SM fails to allocate, the SM will if possible provide an alternative based on the capability of the network and any policy set up by the network provider. The failure of a Mandatory&Negotiable resource (i.e., the SM cannot offer a resource within the requested range) will cause the failure of this resource and the remaining resource requests within the same AddResourceRequest message shall not be processed.

A User specifies a resource as Non-Mandatory&Non-Negotiable when it will not consider an alternative offered by the SM in case of failure. Thus, the SM will not propose one. The failure of a Non-Mandatory&Non-Negotiable resource does not stop the SM from processing other resource requests within the same AddResourceRequest message.

A User specifies a resource as Non-Mandatory&Negotiable when it is willing to consider an alternative offered by the SM in case of failure. Thus, if possible, the SM will propose one within the range given in the resource descriptor. For any Non-Mandatory&Negotiable resource that the SM fails to allocate, the SM will do its best to provide an alternative based on the capability of the network and any policy set up by the network provider. The failure of a Non-Mandatory&Negotiable resource (i.e., the SM cannot provide a resource within the requested range) does not stop the SM from processing other resource requests within the same AddResoursceRequest message.

Within each AddResourceRequest message sent to the SM, the User can have a mixture of each of the resourceTypes. The SM will always process Mandatory&Non-Negotiable resource requests first and Mandatory&Negotiable resource requests second since failure in one such resource means the service cannot be provided and the remaining resource requested within the same AddResourceRequest will not be processed.

A User can always re-negotiate the Mandatory&Non-Negotiable or Mandatory&Negotiable resource that fails by initiating a new Resource Addition command sequence with new values.

The **resourceType** field defines the specific resource being requested. The resourceType field may have the following values:

Table 69 DSM-CC U-N Resource Descriptor resourceTypes

resourceType	Value	Description
Reserved	0x0000	ISO/IEC 13818-6 reserved.
ServerAtm	0x0001	Sent from the Server to the Network to request the downstream ATM resources required to deliver a session and from the Network to the Server to assign the downstream ATM resources.
ClientAtm	0x0002	Sent from the Client to the Network to request the downstream ATM resources required to deliver a session and from the Network to the Client to assign the downstream ATM resources.
MpegPsiDownstream	0x0003	Sent from the Server to the Network to request the MPEG resources required to deliver a session and from the Network to the Server to assign the MPEG resources. This resource is used if it is necessary that the ISO/IEC 13818-1 program specific information (PSI) be inserted by the Network.
MpegDownstream	0x0004	Sent from the Server to the Network to request the MPEG resources required to deliver a session and from the Network to the Server to assign the MPEG resources. This resource is used when a portion of the PSI information (e.g., program_map_sections) is inserted by the Server.

resourceType	Value	Description
ServerUpstream	0x0005	This is sent between the Server and the Network to request/assign the upstream resources required to deliver session data.
ServerDownstream	0x0006	Sent from the Server to the Network to request the downstream bandwidth necessary for the Server to deliver session data.
AtmConnection	0x0007	Sent from either the Client or Server to the Network, requesting an ATM connection .
Reserved	0x0008 - 0x00ff	ISO/IEC 13818-6 reserved.
ClientDownstream	0x0100	Sent from the Network to the Client to assign the downstream resources necessary for the Client to receive session data.
ClientCtrlDownstream	0x0101	Sent from the Network to the Client to assign the resources necessary for the Client to receive a control stream.
ClientUpstream	0x0102	Sent from the Network to the Client to assign the upstream resources necessary for the Client to deliver session data to the server.
Reserved	0x0103 - 0x7fff	ISO/IEC 13818-6 reserved.
UserDefined	0x8000 - 0xffff	Resource descriptors in this range are user definable.

The **resourceNum** field is an identifier assigned by the Network *[User assignable resourceNum is tbd]* to an assigned resource. The resourceNum values shall be unique within a session. The resourceNum field is used along with sessionId to derive the resourceId, a unique reference to an assigned resource within the Network. resourceId is a composite of sessionId and resourceNum and is not explicit syntax. resourceNum is retained by the Network and Users as long as the resource (and hence the session) to which it is referring is active, and is used when the User modifies or deletes a resource from a session. The resourceNum field(s) is/are also returned when the Network performs a session audit on a User. When a User is requesting a resource from the Network, it shall set the resourceNum to 0x0000. Values 0x0001 through 0xffff are available to the Network for assignment.

The **resourceLength** field defines the total length of the resource descriptor. The resourceLength depends on the particular resourceType being defined and the actual data in the resource descriptor.

The content of the **resourceDataByte** field depends on the particular resourceType being defined.

4.4.1 Resource Descriptor Definitions

This section defines the individual resource descriptors. These descriptors may be associated under a session to form the total resources required by the session. It is possible to instantiate more than one of the same resourceType in a session.

4.4.1.1 ServerContinuousFeedSession resource descriptor definition

The **ServerContinuousFeedSession** resource descriptor is requested by the Server to connect a Client session to a continuous feed session. Table 70 defines the format of the ServerContinuousFeedSession descriptor.

Table 70 ServerContinuousFeedSession resource descriptor

Syntax	Num. of Bytes
ServerContinuousFeedSession() {	
cfSessionId	10
}	

The **cfSessionId** field is used to indicate the sessionId of the continuous feed session to which the Client session will be associated. The Network shall be responsible for mapping the continuous feed session resources to the Client session.

4.4.1.2 ServerDownstream resource descriptor definition

The **ServerDownstream** resource descriptor is requested by the Server to allocate a portion of the downstream transport stream for a session. Multiple ServerDownstream descriptors may be requested for a session. Table 71 defines the format of the ServerDownstream descriptor.

Table 71 ServerDownstream resource descriptor

Syntax	Num. of Bytes
ServerDownstream() {	
downstreamBandwidth	4
coSessionId	10
}	

The **downstreamBandwidth** field indicates the data rate in bits per second required to deliver the session over the downstream portion of the network. The entire range of values from 0x00000000 to 0xffffffff for downstreamBandwidth are valid.

The **coSessionId** field is used to request that the Network allocate resources for the session on the same transport stream as an existing session. If this field is set to 0, the Network shall be free to assign the session to any transport stream which satisfies the requirements of the session.

This field is generally used to locate multiple sessions on the same transport stream to prevent the Client from having to switch between transport streams to receive multiple sessions.

4.4.1.3 ServerAtm resource descriptor definition

The **ServerAtm** resource descriptor is requested by the Server to instruct the Network that an ATM PVC connection should be established from the Network to the Server. This resource descriptor is sent when requesting a PVC ATM connection resource.

Table 72 defines the format of the ServerAtm resource descriptor.

Table 72 ServerAtm resource descriptor

Syntax	Num. of Bytes
ServerAtm () {	
serverAtmAddress	16
atmVci	2
atmVpi	2
}	

The **serverAtmAddress** field indicates the ATM address of the Server. The Network may use this address to establish a connection between the Network and the Server. This field may be sent to the appropriate Network device to be used in the ATM connection establishment procedure.

The **atmVci** and **atmVpi** parameters may be supplied by the Network after an ATM connection has been established. These fields are used in the case where the Network sets up the connection to the server without using ATM signaling to the Server. This type of connection is most common in third-party and proxy signaling methods. The Server shall always set these fields to 0 when requesting a new resource.

4.4.1.4 ClientAtm resource descriptor definition

The ClientAtm resource descriptor is requested by the Client to instruct the Network that an ATM PVC connection should be established from the Network to the Client. This resource descriptor is sent when requesting a PVC ATM connection resource.

Table 73 defines the format of the ClientAtm resource descriptor.

Table 73 ClientAtm resource

Syntax	Num. of Bytes
ClientAtm() {	
clientAtmAddress	16
atmVci	2
atmVpi	2
}	

The **clientAtmAddress** field indicates the ATM address of the Client. The Network may use this address to establish a connection between the Network and the Client. This field may be sent to the appropriate Network device to be used in the ATM connection establishment procedure.

The **atmVci** and **atmVpi** parameters may be supplied by the Network after an ATM connection has been established. These fields are used in the case where the Network sets up the connection to the server without using ATM signaling to the Client. This type of connection is most common in third-party and proxy signaling methods. The Server shall always set these fields to 0 when requesting a new resource.

4.4.1.5 AtmConnection resource descriptor definition

The **AtmConnection** resource descriptor is used by the Server or Client to request the Network for an ATM connection between the Server and Client. The Network may pass this request to the ATM device which sets up the connection using standard Q.2931 or UNI signaling or the Network may set up the connection on behalf of the devices. Table x defines the format of the AtmConnection resource descriptor.

This resource descriptor is sent when requesting an ATM connection resource.

Table 74 AtmConnection resource descriptor

Syntax	Num. of Bytes
AtmConnection () {	
atmVersion	2
serverAtmAddress	20
clientAtmAddress	20
atmTrafficParameter	For ATM Forum 3.1, See ATM Forum UNI 3.1 specification, section 5.4.5.6
broadbandBearerCapability	For ATM Forum 3.1, See ATM Forum UNI 3.1 specification, section 5.4.5.7
qualityOfService~~3.1~~	For ATM Forum 3.1, See ATM Forum UNI 3.1 specification, section 5.4.5.18
serverVpi	2
serverVci	2
clientVpi	2
clientVci	2
}	

The **atmVersion** defines the ATM version being used.

The **serverAtmAddress** field indicates the ATM address of UNI interface specified by requestor of the AddResource. The recipient of the information may use this address to establish a connection between itself and the Client or Server. This connection may be established using any of several means of ATM connection establishment including, but not limited to, Q.2931 or UNI associated, non-associated, proxy, or third-party signaling.

The **clientAtmAddress** must be the supplied address of the UNI inerface of the other side of the serverAtmAddress

Table 75 DSM-CC U-N Resource Descriptor AtmSvcConnection atmVersion

atmVersion	Value	Description
reserved	0x0000	13818-6 reserved
ATM Forum 3.0	0x0001	ATM Forum 3.0 being used
ATM Forum 3.1	0x0002	ATM Forum 3.1 being used
ATM Forum 4.0	0x0003	ATM Forum 4.0 being used
reserved	0x0004-0x0010	ISO/IEC 13818-6 reserved.
ITU-T Q.2931 Capability Set 1	0x0011	ITU-T Q.2931 Capability Set 1 being used
reserved	0x0012-0xffff	ISO/IEC 13818-6 reserved.

atmTrafficParameter-- For ATM 3.1, See ATM Forum UNI 3.1 specification, section 5.4.5.6.

broadbandBearerCapability-- For ATM 3.1, See ATM Forum UNI 3.1 specification, section 5.4.5.7.

qualityOfService-- For ATM 3.1, See ATM Forum UNI 3.1 specification, section 5.4.5.18.

4.4.1.6 MpegPsiDownstream resource descriptor definition

The MpegPsiDownstream resource descriptor is requested by the Server to instruct the Network that it is responsible for inserting the PSI information for the session into the transport stream. Table 76 defines the format of the MpegPsiDownstream resource descriptor.

[As transport descriptors (other than ATM, for instance) are defined, this descriptor may need to be reviewed/updated.]

Table 76 MpegPsiDownstream resource descriptor

Syntax	Num. of Bytes
MpegPsiDownstream(){	
mpegCaPidReq	2
mpegCaPid	2
mpegPmtLen	2
for(i=0;i<mpegPmtLen;i++) {	
mpegPmtByte	1
}	
}	

The **mpegCaPidReq** field is a binary field which indicates if the Server is requesting the Network to assign a PID to the Server which may be used for the insertion of Conditional Access messages at the Server. If this field is set to 1, the Network shall assign a Conditional Access PID to the session and include the PID in the Conditional Access Table which is sent over the transport stream(s) which support this session. If this field is set to 0, the Network shall not assign a Conditional Access PID. All other values for this field are reserved.

The **mpegCaPid** field is assigned by the Network if the Server requests a Conditional Access PID to be assigned to the session. The Network shall include the PID in the Conditional Access Table which is sent over the transport stream(s) which support this session. If the Server sets this field to 0 and the mpegCaPidReq field is set to 1, the Network shall assign a PID to this field. If the Server sets this field to the value of a Conditional Access PID which is already assigned to the Server, the Network shall validate the PID already belongs to the Server and is assigned as a Conditional Access PID and assure that the PID is included in the Conditional Access Table which is sent over the transport stream(s) which support this session. The Server may use a single Conditional Access PID to support multiple sessions. If the mpegCaPidReq field is set to 0, the Server shall set the mpegCaPid to 0. The Network shall not assign a value to this field.

The **mpegPmtLen** field indicates the number of bytes which are included in the Program Map Table which follows. The Server shall set this field when requesting a resource from the Network.

The **mpegPmtByte** field includes the Program Map Table for the session. The Network may use the pmt for purposes of re-mapping and re-timing the elementary streams in the session.

The format of the PMT is defined in the *[MPEG-2 systems specification]*.

4.4.1.7 MpegDownstream resource descriptor definition

The MpegDownstream resource descriptor is requested by the Server to inform the Network that the Server is responsible for inserting the PSI information for the session into the transport stream. [As transport descriptors (other than ATM, for instance) are defined, this descriptor may need to be reviewed/updated.]

Table 77 defines the format of the MpegDownstream resource descriptor.

[As transport descriptors (other than ATM, for instance) are defined, this descriptor may need to be reviewed/updated.]

Table 77 MpegDownstream resource descriptor

Syntax	Num. of Bytes
MpegDownstream (){	
mpegProgramNum	2
mpegPmtPid	2
mpegPidCount	2
for(i=0;i<mpegPidCount;i++) {	
mpegPid	2
}	
}	

The **mpegProgramNum** field is used by the Server to request an MPEG Program Number to be assigned by the Network. The Server shall set this field to 0 when requesting a new resource. The Network shall assign an MPEG Program Number to this field which is unique over all transport streams over which the session is transported.

The **mpegPmtPid** field is used by the Server to request an MPEG PID to be assigned by the Network. The Server uses this PID to insert the Program Map Table into the transport stream. The Server shall set this field to 0 when requesting a new resource. The Network shall assign an MPEG PID to this field which is unique over all transport streams over which the session is transported. The Network shall also add this field to the Program Access Table which is sent over the transport stream(s) which support this session.

The **mpegPidCount** field is used by the Server to request a number of MPEG PIDs to be assigned by the Network. The Server uses these PIDs for the elementary streams which are used to deliver the session. The Server shall set this field to the required number of PIDs when requesting a new resource. The Network shall assign MPEG PIDs for each PID requested.

The **mpegPid** field includes the MPEG PIDs which are assigned by the Network. There shall be mpegPidCount MPEG PID fields included in the descriptor. The Server shall set these fields to 0 when requesting a new resource. The Network shall assign MPEG PIDs to these fields which are unique over all transport streams over which the session is transported. The Network shall also route these PIDs over the transport stream(s) which support this session.

4.4.1.8 ServerUpstream resource descriptor definition

The ServerUpstream resource descriptor is requested by the Server to allocate a portion of the upstream transport stream for a session. Multiple ServerUpstream resource descriptors may be requested for a session. Table 78 defines the format of the ServerUpstream resource descriptor.

Table 78 ServerUpstream resource descriptor

Syntax	Num. of Bytes
ServerUpstream (){	
upstreamBandwidth	4
}	

The **upstreamBandwidth** field indicates the data rate in bits per second required to deliver the session over the upstream portion of the network. The entire range of values from 0x00000000 to 0xffffffff for downstreamBandwidth are valid.

4.4.1.9 ClientDownstream resource descriptor definition

The ClientDownstream resource descriptor is assigned by the Network and sent to the Client to instruct the Client about how/where to receive the downstream transport information The Server may also request this resource if it has a need for this information (for informational purposes only). Table 79 defines the format of the ClientDownstream resource descriptor.

Table 79 ClientDownstream resource descriptor

Syntax	Num. of Bytes
ClientDownstream (){	
mpegProgramNum	4
downstreamTransportId	4
}	

The **mpegProgramNum** field indicates the MPEG Program Number that has been assigned to the session. The Client should use this program number to look-up the Program Map Table in the Program Association Table (see ISO/IEC 13818-1 for details). The Network assigns this resource as a result of creating a ServerDownstream resource descriptor and an associated MPEG downstream resource. If the Server includes this resource, this field shall be set to 0 and the Network shall assign the field when an MPEG program number is assigned to the session.

The **downstreamTransportId** field indicates the transport stream that the Client should monitor to receive the PAT, PMT, and all associated elementary streams for the session. The Network assigns this resource as a result of creating a ServerDownstream resource descriptor. If the Server includes this resource, this field shall be set to 0 and the Network shall assign the value of this field at the time when the session is associated with a transport stream.

4.4.1.10 ClientCtrlDownstream resource descriptor definition

The ClientCtrlDownstream resource descriptor is assigned by the Network to the Client to instruct the Client how to receive session data. This descriptor is used in the case that the Client device is unable to process PSI information or to expedite acquisition of a session. The Server may also request this resource if it has a need for this information however, this resource is informational only on the Server side. Table 80 defines the format of the ClientCtrlDownstream resource descriptor.

Table 80 ClientCtrlDownstream resource descriptor

Syntax	Num. of Bytes
ClientCtrlDownstream (){	
mpegProgramNum	4
downstreamTransportId	4
mpegPidCount	2
for(i=0;i<mpegPidCount;i++) {	
mpegPid	2
}	
}	

The **mpegProgramNum** field indicates the MPEG Program Number that has been assigned to the session. The Client should use this program number to look-up the Program Map Table in the Program Association Table (see ISO/IEC 13818-1 for details). The Network assigns this resource as a result of creating a ServerDownstream resource descriptor and an associated MPEG downstream resource. If the Server includes this resource, this field shall be set to 0 and the Network shall assign the field when an MPEG program number is assigned to the session.

The **downstreamTransportId** field indicates the transport stream that the Client should monitor (in the use of this descriptor, the the PAT and PMT present may be bypassed; the PIDs are specified by **mpegPid**). The Network assigns this resource as a result of creating a ServerDownstream resource descriptor. If the Server includes this resource, this field shall be set to 0 and the Network shall assign the field when the session is assigned to a transport stream.

The **mpegPidCount** and **mpegPid** fields indicate the number of elementary streams and the PIDs assigned to the elementary streams. The Network shall assign the mpegPidCount field and the mpegPid fields.

4.4.1.11 ClientUpstream resource descriptor definition

The ClientUpstream resource descriptor is assigned by the Network to the Client to instruct the Client how to send session data to the Server. Table 81 defines the format of the ClientUpstream resource descriptor.

Table 81 ClientUpstream resource descriptor

Syntax	Num. of Bytes
ClientUpstream(){	
upstreamBandwidth	4
upstreamTransportId	4
}	

The **upstreamBandwidth** field indicates the data rate in bits per second which have been allocated to the session for delivery of application data to the Server. The entire range of values from 0x00000000 to 0xffffffff for upstreamBandwidth are valid. The Server shall set this field to the desired bandwidth to instruct the Network to assign this bandwidth to the session. The Network shall set this field to the actual bandwidth assigned to the session.

The **upstreamTransportId** field indicates the transport stream that the Client should use to send the upstream application data to the Server. The Server shall set this field to 0 when requesting a ClientUpstream resource descriptor. The Network shall assign this field.

[How are values allocated when a non-MPEG TS is used for the upstream?]

4.4.1.12 Resource Negotiation

4.4.1.12.1 Request Phases

The session layer messages that the User exchanges with the Session Manager (SM) in order to request resources can be divided into two phrases:

AddResourceRequest phase
AddResourceConfirm phase

4.4.1.12.2 AddResourceRequest Phase

The AddResourceRequest phase consists of one message, AddResourceRequest message, sent from the User to the SM. In this message, the User lists the resources that it is requesting by specifying the corresponding requested resource descriptors.

When the SM receives the AddResourceRequest message, it shall process the request in the following order:

1. Mandatory&Non-Negotiable:
   - If the SM can satisfy the Mandatory&Non-Negotiable resource, the scenario shall proceed with the processing of the Mandatory&Non-Negotiable resources.
   - If the SM fails at least one Mandatory&Negotiable resource, the remaining resources shall not be processed and the scenario shall proceed to the AddResoureConfirm phase.
   - If the SM has processed all the Mandatory&Negotiable requested resources, the scenario shall proceed with the processing of the Mandatory&Negotiable resources.
2. Mandatory&Negotiable:
   - If the SM can satisfy the Mandatory&Negotiable resource, the scenario shall continue with the processing of the Mandatory&Negotiable resources.

- If the SM fails at least one Mandatory&Negotiable resource for which there is an alternative within the range given by the requestor, the alternative should be returned and the scenario shall continue with the processing of the Mandatory&Negotiable.
- If the SM fails at least one Mandatory&Negotiable resource for which there is not an alternative within the range given by the requestor, the remaining resources shall not be processed and the scenario shall continue to the AddResourceConfirm phase.
- If the SM has processed all the Mandatory&Negotiable resources, the scenario shall proceed with the processing of the of the Non-Mandatory&Non-Negotiable resources.

3. Non-Mandatory&Non-Negotiable
   - If the SM can satisfy the Non-Mandatory&Non-Negotiable resource, the scenario shall continue with the processing of the Non-Mandatory&Non-Negotiable resources.
   - If the SM fails one Non-Mandatory&Non-Negotiable resource, the remaining resources shall be processed and the scenario shall continue with the processing of the Non-Mandatory&Non-Negotiable resources.
   - If the SM has processed all the Mandatory&Negotiable resources, the scenario shall proceed with the processing of the Non-Mandatory&Negotiable resources.
4. Non-Mandatory&Negotiable
   - If the SM can satisfy the Non-Mandatory&Negotiable resource the scenario shall continue with the processing of the Non-Mandatory&Negotiable resources.
   - If the SM fails one NonMandatory&Negotiable resource for which there is an alternative within the range given by the requestor, the alternative should be returned and the scenario shall continue with the processing of the NonMandatory&Negotiable.
   - If the SM fails one Mandatory&Negotiable resource for which there is not an alternative within the range given by the requestor, the remaining resources shall be processed and the scenario shall with the processing of the Non-Mandatory&Negotiable.
   - If the SM has processed all the Non-Mandatory&Negotiable resources, the scenario shall proceed with the processing of the of the Non-Mandatory&Negotiable resources.

During the AddResourceRequest phase, all resources that the SM can satisfy, shall be allocated by the SM and returned via the AddResourceConfirm message with their corresponding assigned resource descriptors. The resourceId field contains a valid non-null identifier assigned by the SM, and the requestType field shall be labeled Assigned.

For any Negotiated resource that the SM fails to allocate, the SM will do a best effort to provide an alternative based on the capability of the network and any policy set up by the network provider. The resource descriptor of an alternative will be encoded the same as the original requested resource descriptor with the necessary data fields modified to reflect values supported by the SM. For example, a User may request a negotiable transport resource with the bandwidth field containing 10 (Mbps); if the maximum bandwidth allowed in the network is 6 (Mbps), the SM will return the original requested resource descriptor with the bandwidth field changed to 6 (Mbps).

The resourceId field in the recommended resource descriptor contains a valid non-null identifier assigned by the SM and the requestType field shall be labeled Recommended.

If a Negotiable resource fails and the SM does not have a alternative, the resource descriptor returned will be an assigned resource descriptor with a null resourceId and a Failed requestType.

4.4.1.12.3 AddResourceConfirm Phase

The purpose of the AddResourceConfirm phase is to (1) inform the far-end User (e.g., Client if the Server has been requesting resources) of any successively assigned resources and (2) inform the requesting User of the final result of its resource request.

The AddResourceConfirm phase is labeled negative if the SM has failed to allocate (1) all the Mandaatory resource requested or (2) if there are no Mandatory resource requests, at least one Non-Mandatory resource. In this case, the far-end User shall not be informed (i.e., the SM shall not send an AddResourceIndication message to the far-end User).

The AddResourceComfirm phase is labeled positive if the SM has successfully allocated (1) all the Mandatory resource requests or (2) if there are no Mandatory resource requests, at least one Non-Mandatory resource. In this case, depending on the resources, the SM may send an AddResourceIndication message to the far-end User to inform it of any resources successively assigned.

The far-end User shall acknowledge the AddResourceIndication message with an AddResourceResponse message. If the far-end User does not accept any of the indicated resources, it will indicate this in the AddResourceResponse with a response code set to XX.

Upon receipt of the AddResourceResponse message from the far-end User, the SM shall send a concluding AddResourceConfirm message to the requesting User, containing the resource descriptors of all resources that the User has requested.

For the concluding AddResourceConfirm message sent to the requesting User, the requestType field of each resource descriptor shall be encoded as follows:

1. Assigned, if the resource has been successfully allocated by the SM and informed to the far-end User.

2. Failed, if the resource has failed to be allocated by the SM.

3. Unprocessed, if the resource request has not been processed by the SM due to a failure of a Mandatory&Non-Negotiable or Mandatory&Negotiable resource.

4. Invalid, if the requested resource descriptor contains invalid values in one of the parameters, either in the Header section or Data section.

It is also the responsibility of the requesting User to handle Failed resources based on its knowledge of the running interactive application. The requesting User options are (1) to ignore this resource or (2) to initiate a Session Tear-Down scenario to close the session.

4.5 Client Initiated Command Sequences

The following Client initiated command sequences are defined in this section:

- Session set-up command sequence.
- Connection to a continuous feed session command sequence.
- Tear-down of a session command sequence.
- Tear-down of a connection to a continuous feed session command sequence.

There are two types of sessions possible between the Client and a Server. The first type is a session where the Server delivers a service to a Client using network resources dedicated to that service. If an interactive service is desired, each Client session is allocated an upstream resource. A session may be requested by either the Client or the Server.

The second type of session is a Continuous Feed Session (CFS) which is set up by the Server. A CFS is not connected to a particular Client. Any number of Clients on the network may connect to a CFS after it has been set up. Theoretically, all Client sessions which connect to the CFS are allocated the same downstream resources thereby sharing a single channel and MPEG program between all Clients; however, actual network implementations may require some segmentation. If interactive connection to a CFS is desired, each Client will be given dedicated upstream resources allowing each Client to communicate with the Server.

The Server may notify the Clients that a Continuous Feed Session has been set-up via the broadcast or pass-thru messages described in section *[Broadcast and Pass-Thru Messages]* or User-to-User messages described in section *[User-to-User Interface]*. The Client connects to a CFS using the User-to-Network messages for session set-up. It passes the information about the CFS in userDataByte portion of the set-up request.

4.5.1 Client Session Set-Up Command Sequence

Figure 8 illustrates the procedure for session establishment initiated by the Client.

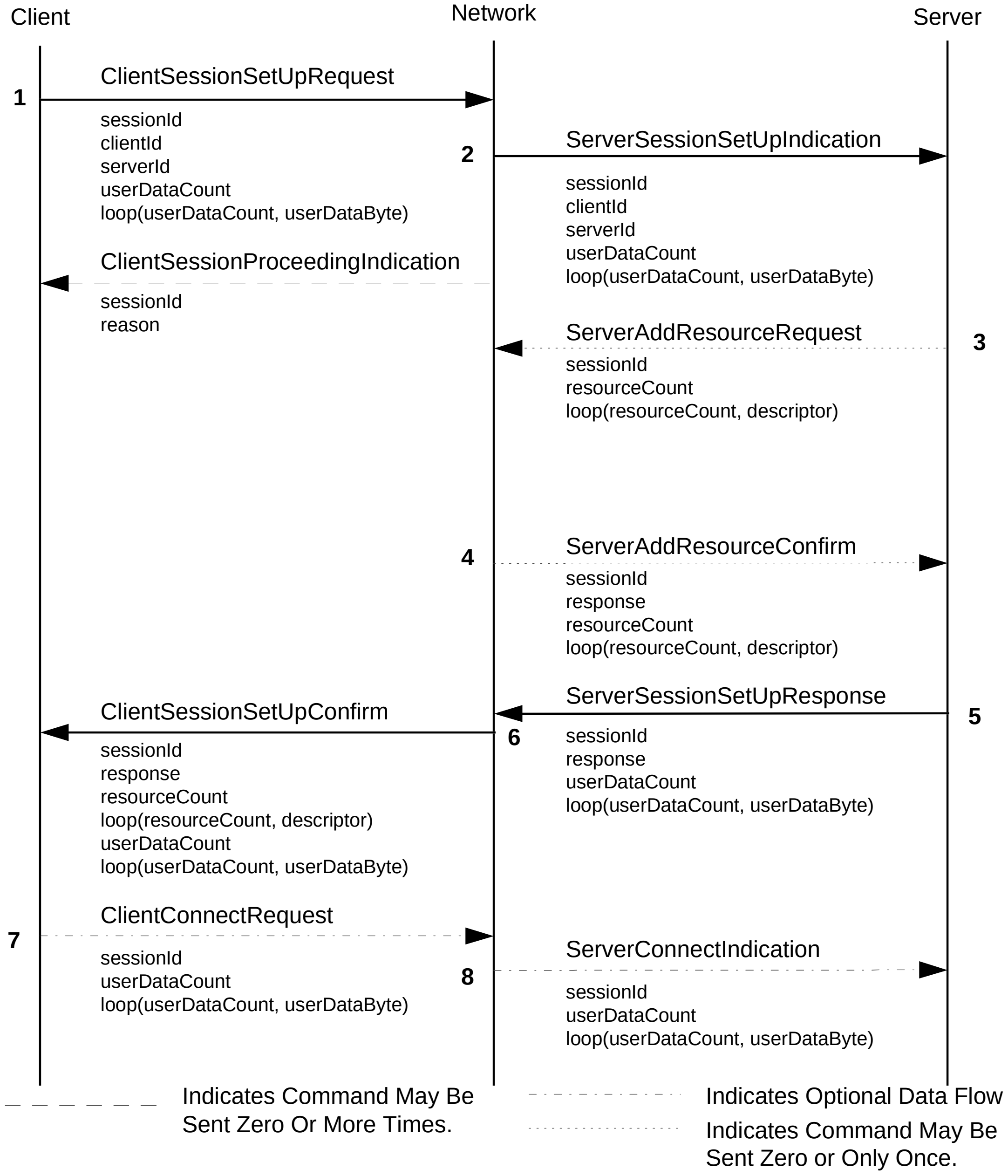

Client
Network
Server
1
ClientSessionSetUpRequest
sessionId
clientId
serverId
userDataCount
loop(userDataCount, userDataByte)
2
ServerSessionSetUpIndication
sessionId
clientId
serverId
userDataCount
loop(userDataCount, userDataByte)
ClientSessionProceedingIndication
sessionId
reason
ServerAddResourceRequest
3
sessionId
resourceCount
loop(resourceCount, descriptor)
4
ServerAddResourceConfirm
sessionId
response
resourceCount
loop(resourceCount, descriptor)
ServerSessionSetUpResponse
5
ClientSessionSetUpConfirm
6
sessionId
response
userDataCount
loop(userDataCount, userDataByte)
sessionId
response
resourceCount
loop(resourceCount, descriptor)
userDataCount
loop(userDataCount, userDataByte)
ClientConnectRequest
7
sessionId
userDataCount
loop(userDataCount, userDataByte)
ServerConnectIndication
8
sessionId
userDataCount
loop(userDataCount, userDataByte)
Indicates Command May Be Sent Zero Or More Times.
Indicates Optional Data Flow
Indicates Command May Be Sent Zero or Only Once.

Figure 8 Sequence of Events for Client Session Set-up

4.5.1.1 Client Initiates Session Set-Up

Step 1 (Client):

To begin establishing a new session, the Client shall send ClientSessionSetUpRequest to the Network and start timer **Error! Reference source not found.**. The value of the transactionId shall be selected by the Client and shall be used to correlate replies from the Network if there are multiple outstanding ClientSessionSetUpRequest messages. If the Client is assigning the sessionId, the value of the sessionId field shall be selected by the Client and shall contain the client's deviceId plus a sessionNum which is unique to the Client. If the Network is assigning the sessionId, then the Client shall set the value of the sessionId to 0. The value of the serverId field shall identify the Server with which the Client is requesting to establish a session. The value of the userDataCount field shall be equal to the number of userDataBytes present in the remainder of the message.

If timer **Error! Reference source not found.** expires before the ClientSessionSetUpConfirm message is received, flow shall shift to the "Network does not respond to ClientSessionSetUpRequest" scenario.

Step 2 (Network):

On receipt of ClientSessionSetUpRequest, the Network shall verify that the clientId and serverId fields represent entities known to the Network. If the values of these fields are valid and the Network believes that the network can support a new session, the Network shall send the ServerSessionSetUpIndication to the Server identified in the serverId field and shall start timers **Error! Reference source not found.** and **Error! Reference source not found.**. If the sessionId was set to 0 by the Client, the value of the sessionId field shall be selected by the Network. The sessionId shall be used to identify the new session throughout its duration. The value of the clientId field identifies the Client that requested the session and shall be identical to the value received from the Client. The values of the userDataCount and userDataByte fields shall be identical to the values received in the ClientSessionSetUpRequest (i.e., the Network passes this information transparently through to the Server).

If the value of the serverId field is invalid or if the network cannot support a new session, flow shall shift to the "Network rejects ClientSessionSetUpRequest" scenario.

If timer **Error! Reference source not found.** expires before the either ServerSessionSetUpResponse message is received, the Network shall send ClientSessionProceedingIndication to the Client. The reason field shall be set to rsn. Timer **Error! Reference source not found.** shall then be restarted.

If timer **Error! Reference source not found.** expires before the ServerSessionSetUpResponse message is received, flow shall shift to the "Server does not respond to ServerSessionSetUpIndication" scenario.

Step 3 (Server):

On receipt of ServerSessionSetUpIndication, the Server may validate the clientId field to ensure that it represents a Client known to the Server. If the value of the clientId field is acceptable and the Server can support a new session, it shall begin resource negotiation for the session by sending ServerAddResourceRequest to the Network. The value of the sessionId field shall be identical to the value received from the Network in ServerSessionSetUpIndication. The value of the response field shall be set to rspOK. The Server shall set the value of the resourceCount to the number of resource types required to initially establish the session. For each type of resource requested, the Server shall include a resourceDescriptor field in ServerAddResourceRequest. Within each resource descriptor:

- The value of the resourceRequestId field shall be selected by the Server and can be used to correlate replies from the Network.
- The value of the requestType field shall be encoded to indicate whether the specific values requested for the resource are MANDATORY NEGOTIABLE, MANDATORY NON-NEGOTIABLE, NON-MANDATORY NEGOTIABLE, or NON-MANDATORY NON-NEGOTIABLE. If the resource is tagged MANDATORY NEGOTIABLE, the Server will accept alternative values with in its suggested range proposed by the Network. If the resource is tagged MANDATORY NON-NEGOTIABLE, the Server will not accept any other value for the resource, and session establishment shall fail if the Network cannot assign the requested value. If the

resource is tagged NON-MANDATORY NEGOTIABLE, the Server will accept any alternative including not getting the resource at all. If the resource is tagged NON-MANDATORY NON-NEGOTIABLE, the Server will accept only the requested resource or no resource at all.

- The value of the resourceId field shall be encoded as all 0's.
- The value of the resourceType field shall be encoded to indicate the specific type of resource being requested for the session.
- The value of the resourceLength field shall indicate the number of bytes remaining in the resourceDescriptor.
- The values of the resourceDataByte field shall be encoded as appropriate for the specific type of resource requested.

If the value of the clientId field is not acceptable or if the Server cannot support a new session, flow shall shift to the "Server rejects ServerSessionIndication" scenario.

Step 4 (Network):

On receipt of ServerAddResourceRequest, the Network shall iterate through the list of requested resources to determine if the network can fulfill each individual resource request. If the Network can assign the requested values for all resources that the Server tagged MANDATORY NON-NEGOTIABLE, the Network shall then attempt to assign values to those resources that the Server tagged MANDATORY NEGOTIABLE, NON-MANDATORY NEGOTIABLE, and NON-MANDATORY NON-NEGOTIABLE and shall construct a ServerAddResourceConfirmation message. The value of the resourceCount field shall be identical to the value received from the Server in ServerAddResourceRequest. For each resource requested, the Network shall include a resourceDescriptor field in ServerAddResourceConfirmation. Within the resourceDescriptor:

- The value of the resourceRequestId field shall be identical to the value received from the Server.
- The value of the responseType field shall be encoded as one of the following values:
  1. rspResourceOK to indicate that the Network is able to assign the exact resource values requested by the Server.
  2. rspResourceNegotiate to indicate that the Network is not able to assign the exact resource values requested by the Server but has assigned the values included in the remainder of the descriptor. The Network shall not use this response if the Server indicated that the resource values requested was NON-NEGOTIABLE.
  3. rspResourceFailed to indicate that the Network is not able to assign the exact resource values requested by the Server, and the Server indicated that the resource values requested was NON-NEGOTIABLE. rspResourceFailed shall also indicate that the Network cannot assign the resource at all regardless of whether the Server tagged the request as NEGOTIABLE or NON-NEGOTIABLE.
- If the responseType field is encoded as either rspResourceOK or rspResourceNegotiate the value of the resourceId field shall be assigned by the Network to uniquely identify the assigned resources for their duration within the session. If the responseType field is encoded as rspResourceFailed, the value of the resourceId field shall be encoded as all 0's.
- The value of the resourceType field shall indicate a specific type of resource and shall be identical to the value received from the Server.
- The value of the resourceLength field shall indicate the number of bytes remaining in the descriptor.
- The values of the resourceDataByte field shall be encoded as appropriate for the specific type of resource being assigned. If the requestType field is encoded as rspResourceOK or rspResourceFailed, the values shall match those received from the Server. If the response field is encoded as rspResourceNegotiate, one or more values shall differ from those received from the Server.

The value of the response field in ServerAddResourceConfirm message shall be encoded as one of the following values:

- rspOK if, for all of the requested resources, the requestType field is encoded as ASSIGNED or RECOMMENDED.
- rspResourceFailed if for any of the requested resources, the responseType field was encoded as rspResourceFailed.

If the Network cannot assign the requested values for any resource that the Server has tagged as MANDATORY NON-NEGOTIABLE, flow shall shift to the ”Network Unable to Assign MANDATORY NON-NEGOTIABLE Resource” scenario.

Step 5 (Server):

On receipt of ServerAddResourceConfirm with the response field encoded as rspOK, the Server shall iterate through the resource list and provision itself to use the assigned resources for the session. It shall send ServerSessionSetUpResponse to the Network with the response field encoded as rspResourceCompleted. The value of the userDataCount field shall indicate the number of userDataBytes in the remainder of the message.

The Server may consider the session established at this point.

Step 6 (Network):

On receipt of ServerSessionSetUpResponse with the response field encoded as rspResourceCompleted, the Network shall terminate timers **Error! Reference source not found.** and **Error! Reference source not found.** and then send ClientSessionSetUpConfirm to the Client. The value of the transactionId field shall be identical to the value received from the Client in ClientSessionSetUpRequest. The value of the sessionId field shall be identical to the value received in ServerResourceResponse. The value of the response field shall be rspOK. The value of the resourceCount field shall indicate the total number of resources initially assigned to the Client side of the session. For each type of resource assigned, the Network shall include a resourceDescriptor field. Within each resourceDescriptor:

- The value of the resourceId field shall be assigned by the Network to uniquely identify the assigned resources for their duration within the session.
- The value of the resourceType field shall be encoded to indicate the specific type of resource being assigned for the session.
- The value of the resourceLength field shall indicate the number of bytes remaining in the resourceDescriptor.
- The values of the resourceDataByte field shall be encoded as appropriate for the specific type of resource being assigned.

The values of the userDataCount and userDataByte fields sent in ClientSessionSetupConfirm shall be identical to those received in the ServerResourceResponse message.

Step 7 (Client):

On receipt of ClientSessionProceedingIndication with a valid transactionId, the Client shall reset timer **Error! Reference source not found.**.

On receipt of ClientSessionSetUpConfirm, the Client shall determine if it is capable of using the resources assigned for the session by the Network. If the Client can use all of the assigned resources, the Client shall determine if it has userDataBytes to be delivered to the Server. If the Client does not have userDataBytes to be delivered to the Server, the session shall be considered to be active and shall start timer **Error! Reference source not found.**.

If the Client cannot use one or more of the assigned resources, flow shall shift to the “Client Unable to Use Resources” scenario.

If the Client has userDataBytes to be delivered to the Server, flow shall shift to the “Client Has Final userDataBytes” scenario.

4.5.1.2 Network does not respond to ClientSessionSetUpRequest

If timer **Error! Reference source not found.** expires before the ClientSessionSetUpConfirm message is received, the Client shall consider the session set-up sequence to be terminated and the session request failed. If after the Client has terminated the Session Request, the ClientSessionSetUpConfirm message is received (or anytime a ClientSessionSetUpConfirm is received with an unknown transactionId) the Client shall send a ClientReleaseRequest message to the Network with the sessionId field set to the value of the sessionId in the ClientSessionSetUpConfirm message and the reason code field set to rsnClUnkRequestID.

4.5.1.3 Network Rejects Client Session Request

Step 2 (Network):

On receipt of ClientSessionSetUpRequest, if the value of the clientId or serverId field is invalid or if the network cannot support a new session, the Network shall send ClientSessionSetUpConfirm to the Client. The value of the transactionId field shall be identical to the value received in ClientSessionSetUpRequest, and the value of the response field shall indicate why session establishment is rejected. The value of the userDataCount field shall be 0, and no userDataByte shall be sent. The following response codes shall apply:

- rspNeNoCalls - Indicates that the Network is unable to accept new sessions.
- rspNeInvalidClient - Indicates that the Network rejected the request due to an invalid clientId.
- rspNeInvalidServer - Indicates that the Network rejected the request due to an invalid serverId.

Step 3 (Client):

On receipt of ClientSessionSetupConfirm with a valid sessionRequestId, and a response code which indicates that the session was rejected, the Client shall terminate session establishment.

4.5.1.4 Server Rejects Server Session Indication

Step 3 (Server):

If the value of the clientId field is invalid or if the Server cannot support a new session, the Server shall send ServerSessionSetUpResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, the value of the reason field shall indicate why the session establish request is rejected, and the value of the resourceCount field shall be set to 0 and there shall be no resource descriptors. The following response codes shall apply:

- rspSeNo- Indicates that the Server is unable to accept new sessions.
- rspSeInvalidClient- Indicates that the Server rejected the request due to an invalid clientId.
- rspSeNoService - Indicates that the Server could not be provide the requested service.

Step 4 (Network):

On receipt of ServerSessionSetUpResponse with a valid sessionId and a response field which indicates that the Server rejected the session request, the Network shall terminate session establishment with the Server and send ClientSessionSetUpConfirm to the Client. The value of the transactionId field shall be identical to the value in ClientSessionSetUpRequest received from the Client, and the value of the reason field shall indicate why the session establish request is rejected. The values of the userDataCount field shall be 0 and no userDataByte shall be sent.

Step 5 (Client):

On receipt of ClientSessionSetupConfirm with a valid transactionId and a response field which indicates that the session request was rejected, the Client shall terminate session establishment.

4.5.1.5 Network Unable to Assign MANDATORY and NON-NEGOTIABLE Resource

Step 4 (Network):

If the Network cannot assign the requested values for any resource that the Server has tagged as MANDATORY and NON-NEGOTIABLE or cannot assign any values for one or more resources regardless of how the Server tagged the resource(s), the Network shall send ServerAddResourceConfirm to the Server. The value of the reason field shall be set to rsnNeNoResource to indicate that the Network could not assign the requested values for the resource indicated by the resourceRequestId field.

Step 5 (Server):

On receipt of ServerAddResourceConfirm with the reason code set to rsnNeNoResource, the Server shall terminate session establishment and send ServerResourceResponse to the Network. The value of the reason field shall be set to rsnSeNoResource to indicate that there are no resources available. The value of the userDataCount shall indicate the number of userDataByte in the remainder of the message.

Step 6 (Network):

On receipt of ServerSessionSetUpResponse, the Network shall terminate session establishment with the Server and send ClientSessionSetUpConfirm to the Client. The value of the transactionId field shall be identical to the value received from the Client in ClientSessionRequest. The value of the reason field shall be set to rsnSeNoResource to indicate that the session could not be established. The values of the userDataCount and userDataByte fields shall be identical to those received in ServerSessionSetUpResponse.

Step 7 (Client):

On receipt of ClientSessionSetUpConfirm with the response code which indicates that the session was rejected, the Client shall terminate session establishment.

4.5.1.6 Server Terminates Resource Negotiation

Step 5 (Server):

On receipt of ServerAddResourceConfirm with the response field encoded as rspResourceCompleted or rspResourceContinue, if the Server decides to terminate resource negotiation because one or more resource values assigned by the Network are unacceptable, the Server shall send ServerResourceResponse to the Network. The value of the sessionId field shall be identical to the value received in ServerResourceIndication, and the reason field shall indicate why the Server has terminated resource negotiation. The value of the userDataCount field shall indicate the number of userDataByte in the remainder of the message. The Server shall consider the session as terminated at this point.

Step 6 (Network):

On receipt of ServerSessionSetUpResponse with the response field set to indicate that the resources were rejected, the Network shall terminate session establishment with the Server and send ClientSessionSetupConfirm to the Client. The value of the transactionId field shall be identical to the value received from the Client in ClientSessionSetUpRequest. The value of the reason field shall indicate why the requested session could not be established. The values of the userDataCount and userDataByte fields shall be identical to those received in ServerSessionSetUpResponse. At this point, the Network shall consider the session as terminated.

Step 7 (Client):

On receipt of ClientSessionSetUpConfirm with the response field set to indicate that the session was rejected, the Client shall terminate session establishment.

4.5.1.7 Client Unable to Use Resources

Step 7 (Client):

On receipt of ClientSessionSetUpConfirm, if the Client cannot use one or more of the assigned resources, it shall send ClientReleaseRequest to the Network. The value of the sessionId field shall be identical to the value received in ClientSessionSetupConfirm, and the reason field shall be set to rsnClNoResource to indicate that the Client cannot use one or more of the assigned resources.

Step 8 (Network):

On receipt of ClientReleaseRequest, the Network shall release all Client interface resources assigned to the session and shall send ClientReleaseConfirm and ServerReleaseIndication. The value of the sessionId field shall be identical to the value received from the Client in ClientReleaseRequest, and the value of the reason field shall indicate that the Client was unable to provision itself to use the resources assigned to the session. The value of the userDataCount field shall be 0.

Step 9 (Server):

On receipt of ServerReleaseIndication, the Server shall first release all resources assigned to the session and then send ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall be 0. At this point, the session is terminated from the Server’s perspective.

Step 10 (Network):

On receipt of ServerReleaseResponse, the Network shall release all Server interface resources assigned to the session, and shall consider the session to be terminated.

Step 11 (Client):

On Receipt of the ClientReleaseConfirm, the Client shall consider the session to be released.

4.5.1.8 Client Has Final userDataBytes

Step 7 (Client):

If the Client has userDataBytes to be delivered to the Server, it shall send ClientConnectRequest to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall indicate the number of userDataByte present.

Step 8 (Network):

On receipt of ClientConnectRequest with a valid sessionId, the Network shall send ServerConnectIndication to the Server. The values of the sessionId, userDataCount, and userDataByte fields shall be identical to the corresponding values received from the Client. After sending the message. There is no change of state for the session at the Network.

Step 9 (Server):

On receipt of ServerConnectIndication, the Server shall consider the session to be established end-to-end through the network.

4.5.2 Client Connection to Continuous Feed Session Command Sequence

A continuous feed session is where a Server is delivering continuous session information to the Network. One or more Clients may connect to this session and receive this shared data. In addition, each Client session which is connected to the continuous feed session may also be allocated additional Client-specific resources.

The Client connects to a continuous feed session using the same method as the client initiated session set up command sequence. The userDataBytes that are passed to the Server may be used by the applications running on the Server to determine if the Client is requesting a connection to a continuous feed session. If the Server determines that the Client is requesting a connection to a continuous feed session, it connects the Client session to the continuous feed session by adding a continuous feed session resource descriptor to the client session.

Due to the nature of a shared session, only the continuous feed resource descriptor and Client-specific resource descriptors may be included in the ServerAddResourceRequest message.

The ClientSessionRequest sequence is a confirmed service. The Network connects the session to the CFS before confirming the request.

Figure 9 describes the sequence of events that occur when a Client connects to a Continuous Feed Session.

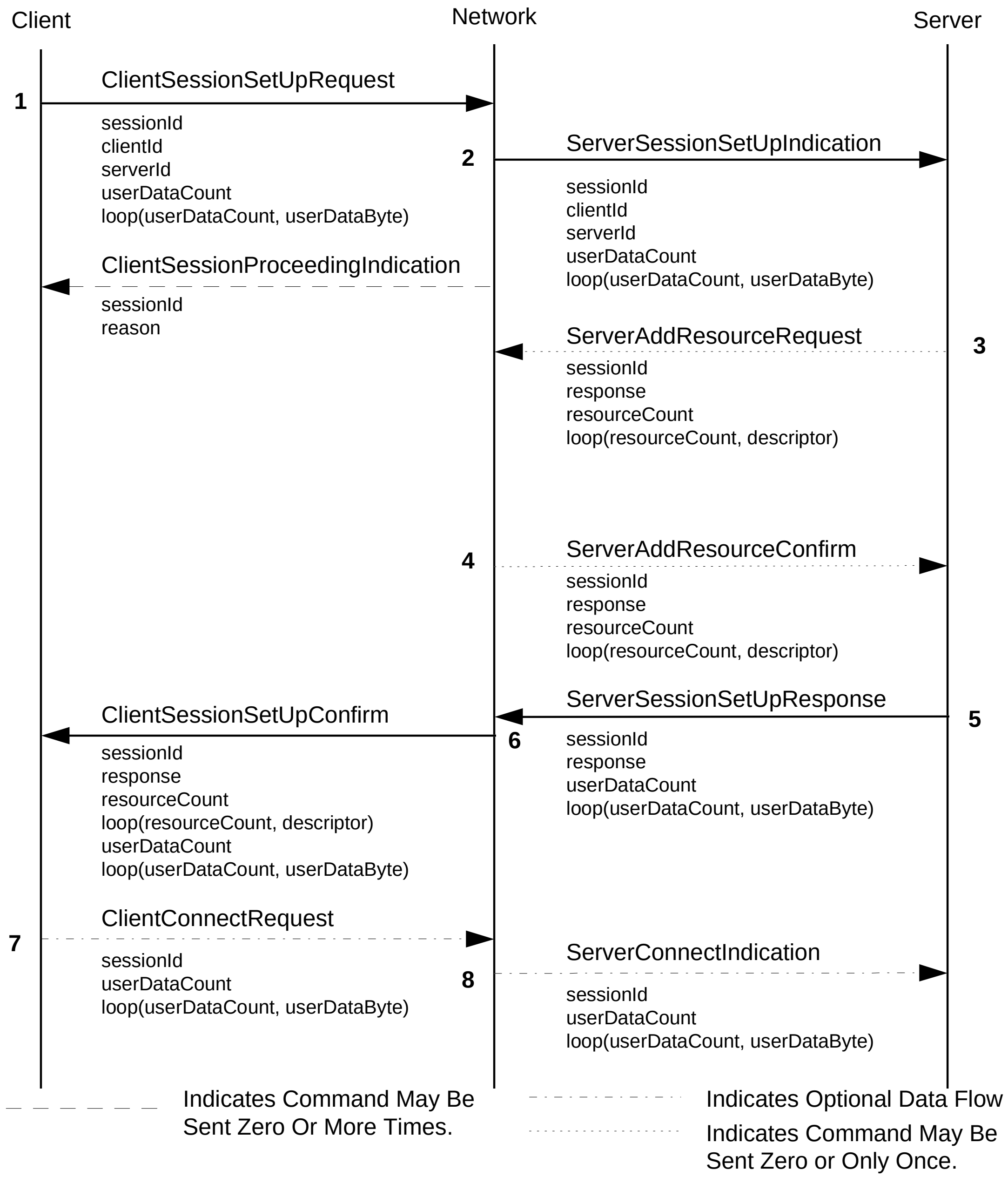

Client
Network
Server
1
ClientSessionSetUpRequest
sessionId
clientId
serverId
userDataCount
loop(userDataCount, userDataByte)
2
ServerSessionSetUpIndication
sessionId
clientId
serverId
userDataCount
loop(userDataCount, userDataByte)
ClientSessionProceedingIndication
sessionId
reason
ServerAddResourceRequest
3
sessionId
response
resourceCount
loop(resourceCount, descriptor)
4
ServerAddResourceConfirm
sessionId
response
resourceCount
loop(resourceCount, descriptor)
ServerSessionSetUpResponse
5
ClientSessionSetUpConfirm
6
sessionId
response
userDataCount
loop(userDataCount, userDataByte)
sessionId
response
resourceCount
loop(resourceCount, descriptor)
userDataCount
loop(userDataCount, userDataByte)
ClientConnectRequest
7
sessionId
userDataCount
loop(userDataCount, userDataByte)
ServerConnectIndication
8
sessionId
userDataCount
loop(userDataCount, userDataByte)
Indicates Command May Be Sent Zero Or More Times.
Indicates Optional Data Flow
Indicates Command May Be Sent Zero or Only Once.

Figure 9 Sequence of events for Client connection to a Continuous Feed Session

4.5.2.1 Client initiates connection to a Continuous Feed Session

Step 1 (Client):

To begin the connection to a continuous feed session, the Client shall send ClientSessionRequest to the Network and start timer **Error! Reference source not found.**. The value of the sessionRequestId field shall be selected by the Client and shall be used to correlate replies from the Network if there are multiple outstanding ClientSessionRequest messages. The value of the clientId field is the MAC address of the Client. The value of the serverId field shall identify the Server that the Client is requesting to establish a session with. The value of the userDataCount field shall be equal to the number of userDataBytes present in the remainder of the message. The userDataBytes field may contain the data required to inform the Server that this is a request to connect to a continuous feed session.

If timer **Error! Reference source not found.** expires before the ServerSessionResponse message is received, flow shall shift to the “Network does not respond to ClientSessionRequest” scenario.

Step 2 (Network):

On receipt of ClientSessionRequest, the Network shall verify that the clientId and serverId fields represent entities known to the Network. If the values of these fields are valid and the Network believes that the network can support a new session, the Network shall send the ServerSessionIndication to the Server identified in the serverId field and shall start timers **Error! Reference source not found.** and **Error! Reference source not found.**. The value of the sessionId field shall be selected by the Network and shall be used to identify the new session throughout its duration. The value of the clientId field identifies the Client that requested the session and shall be identical to the value received from the Client. The values of the userDataCount and userDataByte fields shall be identical to the values received in the ClientSessionRequest (i.e., the Network passes this information transparently through to the Server).

If the value of the serverId field is invalid or if the network cannot support a new session, flow shall shift to the “Network rejects ClientSessionRequest” scenario.

If timer **Error! Reference source not found.** expires before the ServerSessionResponse message is received, the Network shall send ClientSessionProceeding to the Client. The value of the sessionRequestId field shall be identical to the value received in ClientSessionRequest and the reason field shall be set to rsn. Timer **Error! Reference source not found.** shall then be restarted.

If timer **Error! Reference source not found.** expires before the ServerSessionResponse message is received, flow shall shift to the “Server does not respond to ServerSessionIndication” scenario.

Step 3 (Server):

On receipt of ServerSessionIndication, the Server may validate the clientId field to ensure that it represents a Client known to the Server. The Server determines that the request is a connection to a continuous feed session based on the data contained in the userDataBytes field. If the value of the clientId field is valid and the Server can support a connection to the requested continuous feed session, it may begin resource negotiation for additional resources for the session the session by sending ServerContinuousFeedSessionResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network in ServerSessionIndication. The value of the cfSessionId field shall be set to the sessionId that the session is to be connected. The value of the response field shall be set to rspOK. The Server shall set the value of the resourceCount to the number of additional resource types required for the session. For each type of resource requested, the Server shall include a resourceDescriptor field in ServerSessionResponse. Within each resource descriptor:

- The value of the resourceRequestId field shall be selected by the Server and can be used to correlate replies from the Network.
- The value of the request field shall be encoded to indicate whether the specific values requested for the resource are NEGOTIABLE or NON-NEGOTIABLE. If the resource is tagged NEGOTIABLE, the Server may accept alternative values proposed by the Network. If the resource is tagged NON-NEGOTIABLE, the Server will not accept any other value for the resource, and session establishment shall fail if the Network cannot assign the requested value.

- The value of the resourceId field shall be encoded as all 0's.
- The value of the resourceType field shall be encoded to indicate the specific type of resource being requested for the session.
- The value of the resourceLength field shall indicate the number of bytes remaining in the resourceDescriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource requested.

If the value of the clientId field is invalid or if the Server cannot support a connection to the requested continuous feed session, flow shall shift to the "Server rejects ServerSessionIndication" scenario.

Step 4 (Network):

On receipt of ServerContinuousFeedSessionResponse, the Network shall iterate through the list of requested resources to determine if the network can fulfill each individual resource request. If the Network can assign the requested values for all resources that the Server tagged NON-NEGOTIABLE, the Network shall then attempt to assign values to those resources that the Server tagged NEGOTIABLE and shall construct a ServerResourceIndication message. The value of the resourceCount field shall be identical to the value received from the Server in ServerSessionResponse. For each resource requested, the Network shall include a resourceDescriptor field in ServerResourceIndication. Within the resourceDescriptor:

- The value of the resourceRequestId field shall be identical to the value received from the Server.
- The value of the response field shall be encoded as one of the following values:
  1. rspResourceOK to indicate that the Network is able to assign the exact resource values requested by the Server.
  2. rspResourceNegotiate to indicate that the Network is not able to assign the exact resource values requested by the Server but has assigned the values included in the remainder of the descriptor. The Network shall not use this response if the Server indicated that the resource values requested was NON-NEGOTIABLE.
  3. rspResourceFailed to indicate that the Network is not able to assign the exact resource values requested by the Server, and the Server indicated that the resource values requested was NON-NEGOTIABLE. rspResourceFailed shall also indicate that the Network cannot assign the resource at all regardless of whether the Server tagged the request as NEGOTIABLE or NON-NEGOTIABLE.
- If the response field is encoded as rspResourceOK, the value of the resourceId field shall be assigned by the Network to uniquely identify the assigned resources for their duration within the session. If the response field is encoded as either rspResourceNegotiate or rspResourceFailed, the value of the resourceId field shall be encoded as all 0's.
- The value of the resourceType field shall indicate a specific type of resource and shall be identical to the value received from the Server.
- The value of the resourceLength field shall indicate the number of bytes remaining in the descriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource being assigned. If the response field is encoded as rspResourceOK or rspResourceFailed, the values shall match those received from the Server. If the response field is encoded as rspResourceNegotiate, one or more values shall differ from those received from the Server.

The value of the response field in ServerResourceIndication message shall be encoded as one of the following values:

- rspOK if, for all of the requested resources, the response field is encoded as ASSIGN.
- rspResourceContinue if, for all of the requested resources indicated as NON-NEGOTIABLE by the Server, the response field is encoded as ASSIGN, and for one or more of the requested resources indicated as NEGOTIABLE by the Server, the response field is encoded as NEGOTIATE.

- rspResourceFailed if for any of the requested resources, the response field was encoded as rspResourceFailed.

If the Network cannot assign the requested values for any resource that the Server has tagged as NON-NEGOTIABLE, flow shall shift to the ”Network Unable to Assign NON-NEGOTIABLE Resource” scenario.

Step 5 (Server):

On receipt of ServerResourceIndication with the response field encoded as rspOK, the Server shall iterate through the resource list and provision itself to use the assigned additional resources for the session. It shall send ServerResourceResponse to the Network with the response field encoded as rspResourceCompleted. The value of the userDataCount field shall indicate the number of userDataByte in the remainder of the message.

On receipt of ServerResourceIndication with the response field encoded as rspResourceContinue, the Server shall iterate through the resource list to determine which resources the Network was able to assign as requested by the Server and which resources the Network has assigned alternate values for. If the Server accepts the alternate resource values assigned by the Network, the Server shall provision itself to use the assigned additional resources for the session and shall send ServerResourceResponse to the Network with the response field encoded as rspResourceCompleted. The value of the userDataCount field shall indicate the number of userDataByte in the remainder of the message.

If the alternate resource values assigned by the Network are unacceptable to the Server, the Server shall terminate the session establish procedure, the flow shall shift to the “Server Terminates Resource Negotiation” scenario.

Step 6 (Network):

On receipt of ServerResourceResponse with the response field encoded as rspResourceCompleted, the Network shall terminate timers **Error! Reference source not found.** and **Error! Reference source not found.** and then send ClientSessionSetupConfirm to the Client. The value of the sessionRequestId field shall be identical to the value received from the Client in ClientSessionRequest. The value of the sessionId field shall be identical to the value received in ServerResourceResponse. The value of the response field shall be rspOK. The value of the resourceCount field shall indicate the total number of resources from the continuous feed session and additional resources assigned to the Client side of the session. For each type of resource assigned, the Network shall include a resourceDescriptor field. Within each resourceDescriptor:

- The value of the resourceId field shall be assigned by the Network to uniquely identify the assigned resources for their duration within the session.
- The value of the resourceType field shall be encoded to indicate the specific type of resource being assigned for the session.
- The value of the resourceLength field shall indicate the number of bytes remaining in the resourceDescriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource being assigned.

The values of the userDataCount and userDataByte fields sent in ClientSessionSetupConfirm shall be identical to those received in the ServerResourceResponse message.

Step 7 (Client):

On receipt of ClientSessionProceedingIndication with a valid sessionRequestId, the Client shall reset timer **Error! Reference source not found.**.

On receipt of ClientSessionSetupConfirm, the Client shall determine if it is capable of using the resources assigned for the session by the Network. If the Client can use all of the assigned resources, the Client shall determine if it will be using the optional ClientConnectRequest message. If it will not, the session shall be considered to be active and shall start timer **Error! Reference source not found.**.

If the Client cannot use one or more of the assigned resources, flow shall shift to the “Client Unable to Use Resources” scenario.

If the Client is using the optional ClientConnectRequest message, flow shall shift to the “Client sends ClientConnectRequest” scenario

4.5.2.2 Network does not respond to ClientSessionRequest

If timer **Error! Reference source not found.** expires before the ClientSessionResponse message is received, The Client shall consider the session set-up sequence to be terminated and the session request failed. If after the Client has terminated the Session Request, the ClientSessionResponse message is received (or anytime a ClientSessionResponse is received with an unknown sessionRequestId) the Client shall send a ClientReleaseRequest message to the Network with the sessionId field set to the value of the sessionId in the ClientSessionResponse message and the reason code field set to rsnClUnkRequestID.

4.5.2.3 Network Rejects Client Session Request

Step 2 (Network):

On receipt of ClientSessionRequest, if the value of the clientId or serverId field is invalid or if the network cannot support a new session, the Network shall send ClientSessionSetupConfirm to the Client. The value of the sessionRequestId field shall be identical to the value received in ClientSessionRequest, and the value of the response field shall indicate why session establishment is rejected. The value of the userDataCount field shall be 0, and no userDataByte shall be sent. The following response codes shall apply:

- rspNeNoCalls - Indicates that the Network is unable to accept new sessions.
- rspNeInvalidClient - Indicates that the Network rejected the request due to an invalid clientId.
- rspNeInvalidServer - Indicates that the Network rejected the request due to an invalid serverId.

Step 3 (Client):

On receipt of ClientSessionSetupConfirm with a valid sessionRequestId, and a response code which indicates that the session was rejected, the Client shall terminate session establishment.

4.5.2.4 Server Rejects Server Session Indication

Step 3 (Server):

If the value of the clientId field is invalid or if the Server cannot support a connection to a continuous feed session, the Server shall send ServerSessionResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, the value of the reason field shall indicate why the session establish request is rejected, and the value of the resourceCount field shall be set to 0 and there shall be no additional resource descriptors. The following response codes shall apply:

- rspSeNo- Indicates that the Server is unable to accept new sessions.
- rspSeInvalidClient- Indicates that the Server rejected the request due to an invalid clientId.
- rspSeNoCFS- Indicates that the Server could not connect the session to the requested CFS.

Step 4 (Network):

On receipt of ServerContinuousFeedSessionResponse with a valid sessionId and a response field which indicates that the Server rejected the session request, the Network shall terminate session establishment with the Server and send ClientSessionSetupConfirm to the Client. The value of the sessionRequestId field shall be identical to the value in ClientSessionRequest received from the Client, and the value of the reason field shall indicate why the session establish request is rejected. The values of the userDataCount field shall be 0 and no userDataByte shall be sent.

Step 5 (Client):

On receipt of ClientSessionSetupConfirm with a valid sessionRequestId and a response field which indicates that the session request was rejected, the Client shall terminate session establishment.

4.5.2.5 Network Unable to Assign NON-NEGOTIABLE Resource

Step 4 (Network):

If the Network cannot assign the requested values for any resource that the Server has tagged as NON-NEGOTIABLE or cannot assign any values for one or more resources regardless of how the Server tagged the resource(s), the Network shall send ServerResourceIndication to the Server. The value of the reason field shall be set to rsnNeNoResource to indicate that the Network could not assign the requested values for the resource indicated by the requestId field.

Step 5 (Server):

On receipt of ServerResourceIndication with the reason code set to rsnNeNoResource, the Server shall terminate session establishment and send ServerResourceResponse to the Network. The value of the reason field shall be set to rsnSeNoResource to indicate that there are no resources available. The value of the userDataCount shall indicate the number of userDataByte in the remainder of the message.

Step 6 (Network):

On receipt of ServerResourceResponse, the Network shall terminate session establishment with the Server and send ClientSessionSetupConfirm to the Client. The value of the sessionRequestId field shall be identical to the value received from the Client in ClientSessionRequest. The value of the reason field shall be set to rsnSeNoResource to indicate that the session could not be established. The values of the userDataCount and userDataByte fields shall be identical to those received in ServerResourceResponse.

Step 7 (Client):

On receipt of ClientSessionSetupConfirm with the response code which indicates that the session was rejected, the Client shall terminate session establishment.

4.5.2.6 Server Terminates Resource Negotiation

Step 5 (Server):

On receipt of ServerResourceIndication with the response field encoded as rspResourceCompleted or rspResourceContinue, if the Server decides to terminate resource negotiation because one or more resource values assigned by the Network are unacceptable, the Server shall send ServerResourceResponse to the Network. The value of the sessionId field shall be identical to the value received in ServerResourceIndication, and the reason field shall indicate why the Server has terminated resource negotiation. The value of the userDataCount field shall indicate the number of userDataByte in the remainder of the message. The Server shall consider the session as terminated at this point.

Step 6 (Network):

On receipt of ServerResourceResponse with the response field set to indicate that the resources were rejected, the Network shall terminate session establishment with the Server and send ClientSessionSetupConfirm to the Client. The value of the sessionRequestId field shall be identical to the value received from the Client in ClientSessionRequest. The value of the reason field shall indicate why the requested session could not be established. The values of the userDataCount and userDataByte fields shall be identical to those received in ServerResourceResponse. At this point, the Network shall consider the session as terminated.

Step 7 (Client):

On receipt of ClientSessionSetupConfirm with the response field set to indicate that the session was rejected, the Client shall terminate session establishment.

4.5.2.7 Client Unable to Use Resources

Step 7 (Client):

On receipt of ClientSessionSetupConfirm, if the Client cannot use one or more of the assigned resources, it shall send ClientReleaseRequest to the Network. The value of the sessionId field shall be identical to the value received in ClientSessionSetupConfirm, and the reason field shall be set to rsnClNoResource to indicate that the Client cannot use one or more of the assigned resources.

Step 8 (Network):

On receipt of ClientReleaseRequest, the Network shall release all Client interface resources assigned to the session and shall send ClientReleaseConfirm and ServerReleaseIndication. The value of the sessionId field shall be identical to the value received from the Client in ClientReleaseRequest, and the value of the reason field shall indicate that the Client was unable to provision itself to use the resources assigned to the session. The value of the userDataCount field shall be 0.

Step 9 (Server):

On receipt of ServerReleaseIndication, the Server shall first release all resources assigned to the session and then send ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall be 0. At this point, the session is terminated from the Server's perspective.

Step 10 (Network):

On receipt of ServerReleaseResponse, the Network shall release all Server interface resources assigned to the session, and shall consider the session to be terminated.

Step 11 (Client):

On Receipt of the ClientReleaseConfirm, the Client shall consider the session to be released.

4.5.2.8 Client Sends ClientConnectRequest message

Step 7 (Client):

The Client sends the ClientConnectRequest to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall indicate the number of userDataByte present.

Step 8 (Network):

On receipt of ClientConnectRequest with a valid sessionId, the Network shall send ServerConnectIndication to the Server. The values of the sessionId, userDataCount, and userDataByte fields shall be identical to the corresponding values received from the Client. After sending the message. There is no change of state for the session at the Network.

Step 9 (Server):

On receipt of ServerConnectIndication, the Server shall consider the session to be established end-to-end through the network.

4.5.2.9 Server does not respond to ServerSessionIndication

[TBD]

4.5.3 Client Session Tear-Down Command Sequence

Figure 10 illustrates the normal procedure for session release initiated by the Client.

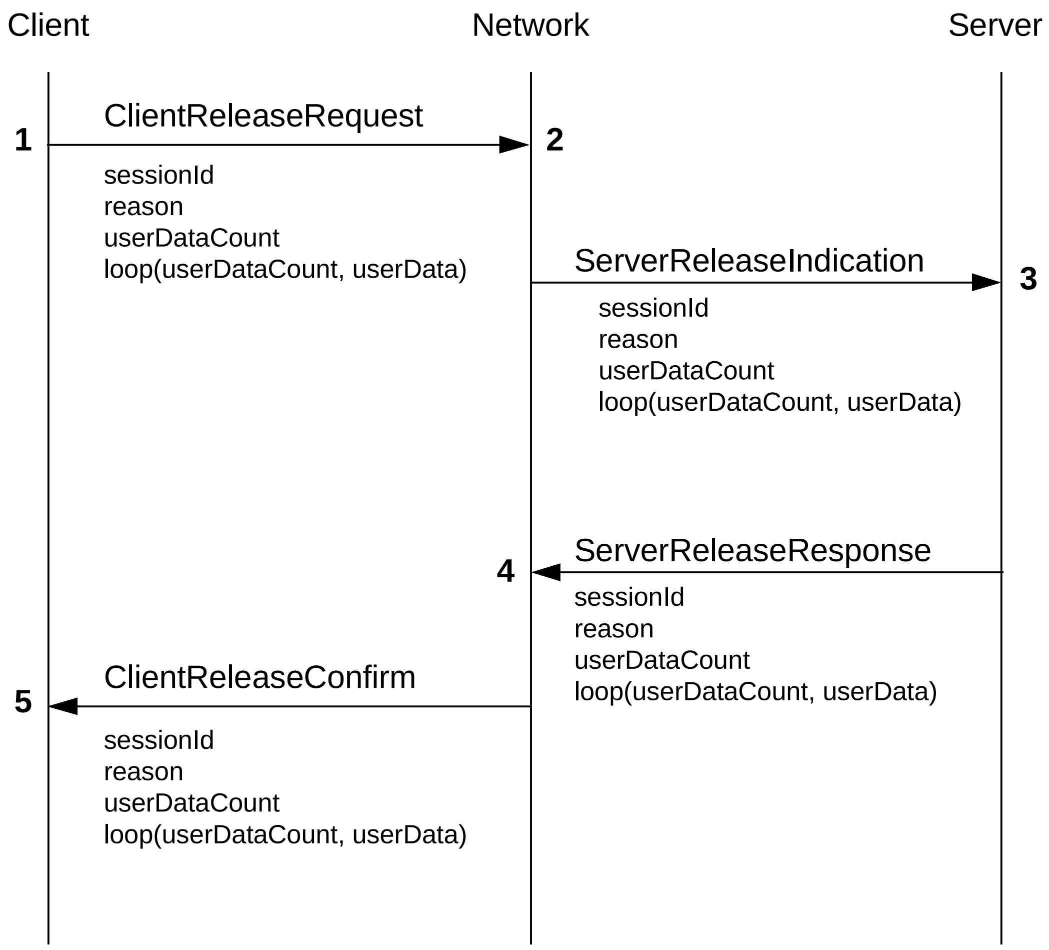


Figure 10 Sequence of events for Client initiated session tear-down

4.5.3.1 Client Initiates Release Request

Step 1 (Client):

To start the procedure for releasing an existing session, the Client shall send ClientReleaseRequest to the Network. The value of the sessionId field shall correspond to an existing session, and the value of the reason field shall indicate why the Client is releasing the session. The value of the userDataCount field shall indicate the number of userDataByte present in the message. Upon sending the ClientReleaseRequest message, the Client shall not use any of the resources assigned to the session.

Step 2 (Network):

On receipt of ClientReleaseRequest, the Network shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid and owned by the Client, the Network shall send ServerReleaseIndication to the Server. The value of the sessionId field shall be identical to the sessionId received from the Client, and the value of the reason field shall indicate that the session is being released at the request of the Client. The values of the userDataCount and userDataByte fields shall be identical to the values received from the Client.

If the Network determines that the sessionId received in ClientReleaseRequest is invalid or that the sessionId does not belong to the Client, flow shall shift to the “Network Rejects Client Release Request” scenario.

Step 3 (Server):

On receipt of ServerReleaseIndication, the Server shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid, the Server shall first release all resources assigned to the session and then send ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall indicate the number of userDataByte in the message. At this point, the Server shall consider the session to be terminated.

If the Server determines that the sessionId is invalid, flow shall shift to the “Server Rejects Server Release Indication” scenario.

Step 4 (Network):

On receipt of ServerReleaseResponse, the Network shall release all resources assigned to the session and send ClientReleaseConfirm to the Client. The values of the sessionId, userDataCount, and userDataByte fields shall be identical to the values received in ServerReleaseResponse.

Step 5 (Client):

On receipt of ClientReleaseConfirm, the Client shall release all resources assigned to the session. At this point, the Client shall consider the session to be terminated.

4.5.3.2 Network Rejects Client Release Request

Step 2 (Network):

If the Network determines that the value of the sessionId field received in ClientReleaseRequest is invalid, the Network shall send ClientReleaseConfirm to the Client. The value of the sessionId field shall be identical to the value received in ClientReleaseRequest, and the value of the reason parameter shall be set to indicate that the sessionId is invalid. After sending ClientReleaseRequest, the Network may take other actions such as initiating an audit with the Client however no change in session state occurs at this time. The following reason codes may be used in the ClientReleaseConfirm message.

- rspNeNoSession- Indicates that a request was made for a non-existent sessionId.
- rsnNeNotOwner - Indicates that the requested sessionId was not owned by the user.

Step 3 (Client):

On receipt of ClientReleaseConfirm, the Client shall terminate the release procedure. The Client may take other actions such as initiating an audit with the Network or initiating internal diagnostics to determine the state of the session.

4.5.3.3 Server Rejects Server Release Indication

Step 3 (Server):

If the Server determines that the value of the sessionId field received in ServerReleaseIndication is invalid, the Server shall send ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received in ServerReleaseIndication, and the value of the reason field shall be set to rspSeNoSession to indicate that the sessionId is invalid. After sending ServerReleaseResponse, the Server may take other actions such as initiating an audit with the Network.

Step 4 (Network):

On receipt of ServerReleaseResponse, the Network shall assume that the sessionId is invalid and release all resources assigned to the session. The Network shall send ClientReleaseConfirm to the Client. The value of the sessionId field shall be identical to the value received in ServerReleaseResponse because the Network previously validated it on receipt of ClientReleaseRequest. The value of the reason field shall be set to rspSeProcError to indicate that a procedure error has occurred with the Server. The value of the userDataCount field shall be 0. After sending ClientReleaseConfirm, the Network may initiate an audit with the Server.

Step 5 (Client):

On receipt of ClientReleaseConfirm, the Client shall release all resources assigned to the session. At this point, the Client shall consider the session to be terminated.

4.5.4 Client Continuous Feed Session Tear-Down Command Sequence

When the Client terminates a session which is connected to a continuous feed session, it issues the ClientReleaseRequest message which indicates the sessionId that is being terminated. The message also contains userDataBytes which is passed to the Server. There is no Network requirement as to the content of the userDataBytes.

The ClientReleaseRequest sequence is a confirmed service. The Network disconnects the service locally at the Network and issues the ClientReleaseConfirm message.

Figure 11 describes the sequence of events that occur during a Client initiated tear-down of a connection to a continuous feed session.

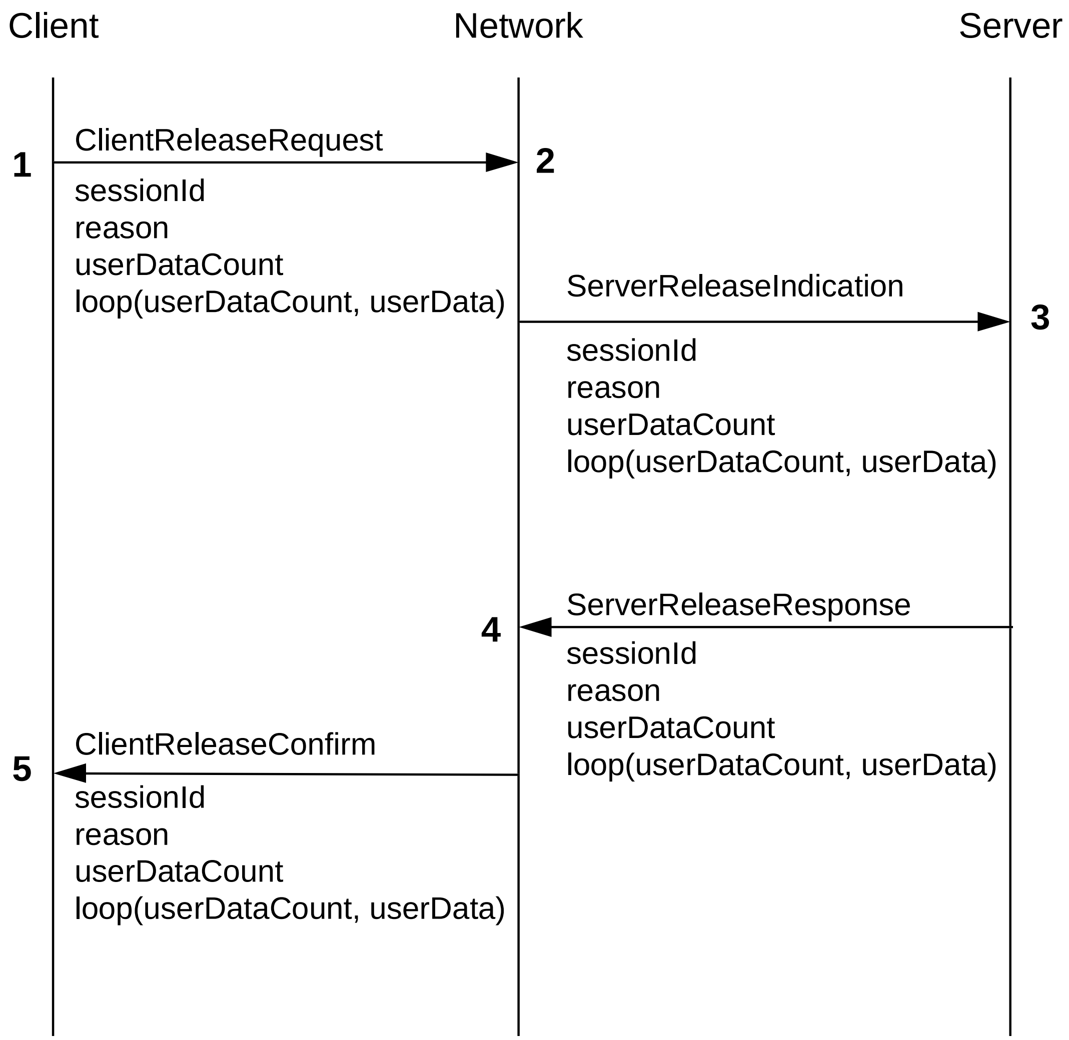


Figure 11 Sequence of events for Client initiated Continuous Feed Session tear-down

4.5.4.1 Client Initiates Release Request

Step 1 (Client):

To start the procedure for releasing an existing session, the Client shall send ClientReleaseRequest to the Network. The value of the sessionId field shall correspond to an existing session, and the value of the reason field shall indicate why the Client is releasing the session. The value of the userDataCount field shall indicate the number of userDataByte present in the message. Upon sending the ClientReleaseRequest message, the Client shall not use any of the resources assigned to the session.

Step 2 (Network):

On receipt of ClientReleaseRequest, the Network shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid and owned by the Client, the Network shall send ServerReleaseIndication to the Server. The value of the sessionId field shall be identical to the sessionId received from the Client, and the value of the reason field shall indicate that the session is being released at the request of the Client. The values of the userDataCount and userDataByte fields shall be identical to the values received from the Client.

If the Network determines that the sessionId received in ClientReleaseRequest is invalid or that the sessionId does not belong to the Client, flow shall shift to the “Network Rejects Client Release Request” scenario.

Step 3 (Server):

On receipt of ServerReleaseIndication, the Server shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid, the Server shall first release all resources assigned to the session and then send ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall indicate the number of userDataByte in the message. At this point, the Server shall consider the session to be terminated.

If the Server determines that the sessionId is invalid, flow shall shift to the “Server Rejects Server Release Indication” scenario.

Step 4 (Network):

On receipt of ServerReleaseResponse, the Network shall release all resources assigned to the session and send ClientReleaseConfirm to the Client. The values of the sessionId, userDataCount, and userDataByte fields shall be identical to the values received in ServerReleaseResponse.

Step 5 (Client):

On receipt of ClientReleaseConfirm, the Client shall release all resources assigned to the session. At this point, the Client shall consider the session to be terminated.

4.5.4.2 Network Rejects Client Release Request

Step 2 (Network):

If the Network determines that the value of the sessionId field received in ClientReleaseRequest is invalid, the Network shall send *[message TBD]* to the Client. The value of the sessionId field shall be identical to the value received in ClientReleaseRequest, and the value of the reason parameter shall be set to indicate that the sessionId is invalid. After sending *[message TBD]*, the Network may take other actions such as initiating an audit with the Client however no change in session state occurs at this time. The following reason codes may be used in the *[message TBD]* message.

- rspNeNoSession- Indicates that a request was made for a non-existent sessionId.
- rsnNeNotOwner - Indicates that the requested sessionId was not owned by the user.

Step 3 (Client):

On receipt of *[message TBD]*, the Client shall terminate the release procedure. The Client may take other actions such as initiating an audit with the Network or initiating internal diagnostics to determine the state of the session.

4.5.4.3 Server Rejects Client Release Indication

Step 3 (Client):

If the Server determines that the value of the sessionId field received in ServerReleaseIndication is invalid, the Server shall send *[message TBD]* to the Network. The value of the sessionId field shall be identical to the value received in ServerReleaseIndication, and the value of the reason field shall be set to rspSeNoSession to indicate that the sessionId is invalid. After sending *[message TBD]*, the Server may take other actions such as initiating an audit with the Network.

Step 4 (Network):

On receipt of *[message TBD]*, the Network shall assume that the sessionId is invalid and release all resources assigned to the session. The Network shall send ClientReleaseConfirm to the Client. The value of the sessionId field shall be identical to the value received in *[message TBD]* because the Network previously validated it on receipt of ClientReleaseRequest. The value of the reason field shall be set to rspSeProcError to indicate that a procedure error has occurred with the Server. The value of the userDataCount field shall be 0. After sending ClientReleaseConfirm, the Network may initiate an audit with the Server.

Step 5 (Client):

On receipt of ClientReleaseConfirm, the Client shall release all resources assigned to the session. At this point, the Client shall consider the session to be terminated.

4.6 Server Initiated Command Sequences

The following Server initiated command sequences are defined in this section:

- Server Initiated Session set-up command sequence.
- Server Initiated Continuous feed session set-up command sequence.
- Server Initiated Session resource re-provision command sequence.
- Server Initiated Add resources to a session command sequence.
- Server Initiated Session tear-down command sequence.
- Server Initiated Continuous feed session tear-down command sequence.

4.6.1 Server Session Set-Up Command Sequence

Figure 12 illustrates the normal procedure for session establishment initiated by the Server.

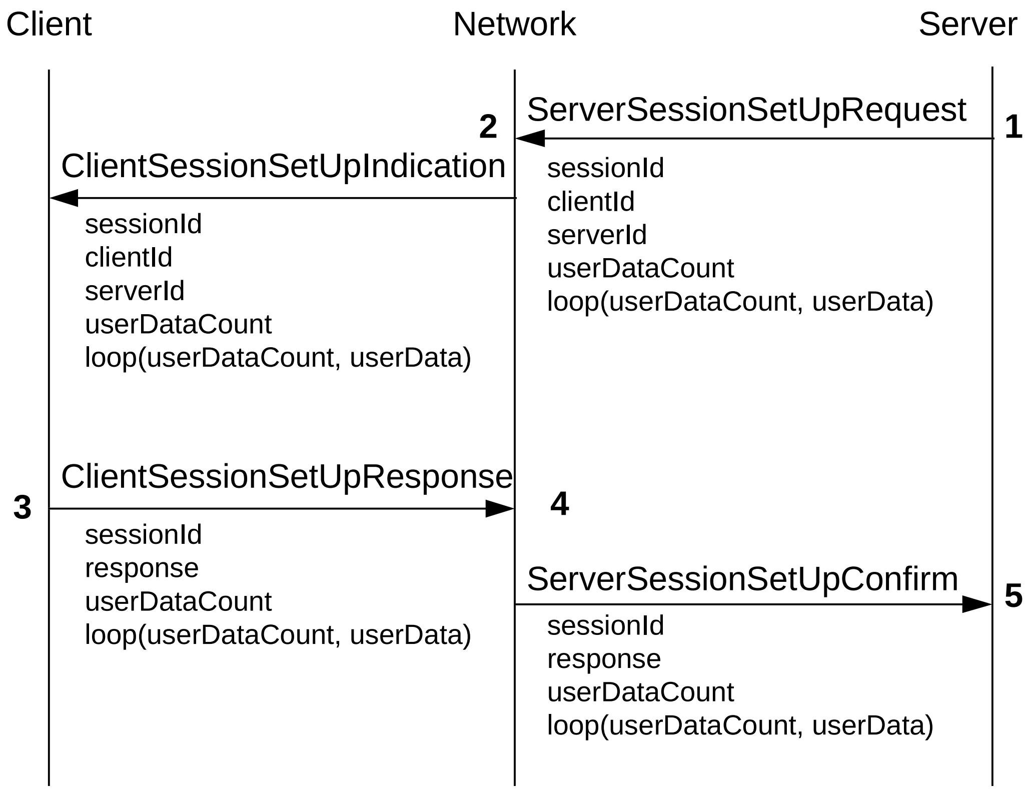

Client
Network
Server
1
ServerSessionSetUpRequest
2
sessionId
clientId
serverId
userDataCount
loop(userDataCount, userData)
ClientSessionSetUpIndication
sessionId
clientId
serverId
userDataCount
loop(userDataCount, userData)
ClientSessionSetUpResponse
3
4
sessionId
response
userDataCount
loop(userDataCount, userData)
ServerSessionSetUpConfirm
5
sessionId
response
userDataCount
loop(userDataCount, userData)

Figure 12 Sequence of Events for Server Session Set-up

4.6.1.1 Server Initiates Session Set-Up

Step 1 (Server):

To begin establishing a new session, the Server shall send ServerSessionRequest to the Network. The value of the requestId field shall be selected by the Server and shall be used to correlate replies from the Network if there are multiple outstanding ServerSessionRequest or ServerContinuousFeedSessionRequest messages. The value of the serverId field shall identify the Server, and the value of the clientId field shall identify the Client that the Server is requesting to establish a session with. The Server shall set the value of the resourceCount field to the number of resource types required to initially establish the session. For each type of resource requested, the Server shall include a resourceDescriptor field in ServerSessionRequest. Within each resource descriptor:

- The value of the requestId field shall be selected by the Server and can be used to correlate replies from the Network.
- The value of the request field shall be encoded to indicate whether the specific values requested for the resource are NEGOTIABLE or NON-NEGOTIABLE. If the resource is tagged NEGOTIABLE, the Server may accept alternative values proposed by the Network. If the resource is tagged NON-NEGOTIABLE, the Server will not accept any other value for the resource, and session establishment shall fail if the Network cannot assign the requested value.
- The value of the resourceId field shall be encoded as all 0’s.
- The value of the resourceType field shall be encoded to indicate the specific type of resource being requested for the session.

- The value of the resourceLength field shall indicate the number of bytes remaining in the resourceDescriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource requested.

Step 2 (Network):

On receipt of ServerSessionRequest, if the Network is able to accept a request for a new session the Network shall verify that the clientId and serverId fields represent entities known to the Network. If the Network can accept a new session request and if the values of these fields are valid, the Network shall iterate through the list of requested resources to determine if the network can fulfill each individual resource request. If the Network can assign the requested values for all resources that the Server tagged NON-NEGOTIABLE, the Network shall then attempt to assign values to those resources that the Server tagged NEGOTIABLE and shall construct a ServerSessionConfirm message. The value of the requestId field shall be identical to the requestId in the ServerSessionRequest message. The resourceCount field shall be identical to the value received from the Server in ServerSessionRequest. For each resource requested, the Network shall include a resourceDescriptor field in ServerSessionConfirm. Within the resourceDescriptor:

- The value of the requestId field shall be identical to the value received from the Server.
- The value of the response field shall be encoded as one of the following values:
  1. rspResourceOK to indicate that the Network is able to assign the exact resource values requested by the Server.
  2. rspResourceNegotiate to indicate that the Network is not able to assign the exact resource values requested by the Server but has assigned the values included in the remainder of the descriptor. The Network shall not use this response if the Server indicated that the resource values requested was NON-NEGOTIABLE.
  3. rspResourceFailed to indicate that the Network is not able to assign the exact resource values requested by the Server, and the Server indicated that the resource values requested was NON-NEGOTIABLE. rspResourceFailed shall also indicate that the Network cannot assign the resource at all regardless of whether the Server tagged the request as NEGOTIABLE or NON-NEGOTIABLE.
- If the response field is encoded as rspResourceOK, the value of the resourceId field shall be assigned by the Network to uniquely identify the assigned resources for their duration within the session. If the response field is encoded as either rspResourceNegotiate or rspResourceFailed, the value of the resourceId field shall be encoded as all 0’s.
- The value of the resourceType field shall indicate a specific type of resource and shall be identical to the value received from the Server.
- The value of the resourceLength field shall indicate the number of bytes remaining in the descriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource being assigned. If the response field is encoded as rspResourceOK or rspResourceFailed, the values shall match those received from the Server. If the response field is encoded as rspResourceNegotiate, one or more values shall differ from those received from the Server.

The value of the response field in ServerSessionConfirm message shall be encoded as one of the following values:

- rspOK if, for all of the requested resources, the response field is encoded as ASSIGN.
- rspResourceContinue if, for all of the requested resources indicated as NON-NEGOTIABLE by the Server, the response field is encoded as ASSIGN, and for one or more of the requested resources indicated as NEGOTIABLE by the Server, the response field is encoded as NEGOTIATE.
- rspResourceFailed if the response field in any of the requested resource descriptors is encoded with rspResourceFailed.

If the Network cannot support a session set-up, the clientId or serverId fields are invalid, or the Network cannot assign the requested values for any resource that the Server has tagged as either NEGOTIABLE or NON-NEGOTIABLE flow shall shift to the ”Network rejects ServerSessionRequest” scenario.

Step 3 (Client):

On receipt of ClientSessionSetUpConfirm with a valid requestId field and the response field encoded as rspOK, the Client shall iterate through the resource list and provision itself to use the assigned resources for the session and send the ClientSessionSetUpResponse message.

Step 4 (Network):

On receipt of the ClientSessionSetUpResponse message. The Network shall send the ServerSessionSetUpConfirm message.

Step 7 (Server)

On receipt of the ServerSessionUpConfirm message. The Server shall consider the session to be active.

4.6.1.2 Network rejects ServerSessionSetUpRequest

Step 2 (Network):

On receipt of ServerSessionSetUpRequest, the Network may reject the request for the following reasons:

- The Network is unable to accept a request for a new session.
- The serverId field is invalid.
- The clientId field is invalid.
- The Network is unable to assign requested resources.

In these cases, the Network shall send ServerSessionSetUpConfirm to the Server. The value of the sessionRequestId shall be identical to the value received in ServerSessionSetUpRequest, and the value of the response field shall indicate why the Network is rejecting session establishment. The Network shall terminate the session set-up procedure at this point. The following response codes apply:

- rspNeNoCalls - Indicates that the Network is not accepting new session requests.
- rspNeInvalidServer - Indicates that the serverId field is invalid.
- rspNeInvalidClient - Indicates that the clientId field is invalid.
- rspResourceFailed - Indicates that the Network could not satisfy one or more requested resources.

Step 3 (Server):

On receipt of ServerSessionSetUpConfirm with the response field set to indicate that the request was rejected, the Server shall terminate the session establishment procedure.

4.6.1.3 Server terminates resource negotiation

Step 3 (Server):

On receipt of ServerSessionSetUpConfirm, the Server determines that it is unable to use any of the assigned resources, the Server may terminate the session by sending the ServerReleaseRequest message. The sessionId field shall be identical to the sessionId in the ServerSessionConfirm message. The reason field shall be set to rsnSeNoResource to indicate that the server could not use the assigned resources.

Step 4 (Network)

On receipt of the ServerReleaseRequest message, the Network shall send the ServerReleaseConfirm message to the Server. At this time, the Network shall consider the session to be torn down and may release all resources assigned to the session.

On receipt of the ServerReleaseConfirm message, the Server shall consider the session to be torn down and may release all resources allocated for the session.

4.6.1.4 Client rejects ClientSessionSetUpIndication

Step 5 (Client):

On receipt of Client SessionSetUpIndication, if the Client cannot accept the session set-up request or if it cannot use one or more of the assigned resources, it shall send ClientSessionSetUpResponse to the Network. The value of the sessionId field shall be identical to the value received in ClientSessionSetUpIndication, and the response field shall indicate the reason that the session request was rejected. The value of the userDataCount field shall indicate the number of userDataBytes in the remainder of the message. After sending the message, the Client shall consider the session to be released. The following response codes apply:

rsnClNoCalls - Indicates that the Client was not accepting session requests.

rsnClNoResource - Indicates that the Client was unable to use one or more resources.

Step 6 (Network):

On receipt of ClientSessionSetUpResponse with the response code set to indicate that the session set-up was rejected, the Network shall release all resources assigned to the session and shall send ServerConnectConfirm. The value of the sessionId field shall be identical to the value received from the Client in ClientSessionSetUpResponse, and the value of the reason field shall be identical to the reason field in the ClientSessionSetUpResponse message. The value of the userDataCount and userDataByte fields shall be identical to the fields in the ClientSessionSetUpResponse message.

4.6.2 Server Continuous Feed Session Set-Up Command Sequence

The Server may set-up a session which is not connected to a particular Client. This type of session is a Continuous Feed Session (CFS). Any number of Clients may connect to a single CFS and share the downstream resources of that CFS. Each Client session which is connected to a CFS may have a separate upstream bandwidth allocation.

A CFS is assigned a SessionId by the Server that sets up the CFS. After a CFS is set up, any number of Clients can connect to the session. When a Client is connected to a CFS, that Client assigns an individual sessionId to its connection with the CFS which allows each Client connection to be tracked individually.

Figure 13 describes the sequence of events that occur during a Server Continuous Feed Session set-up.

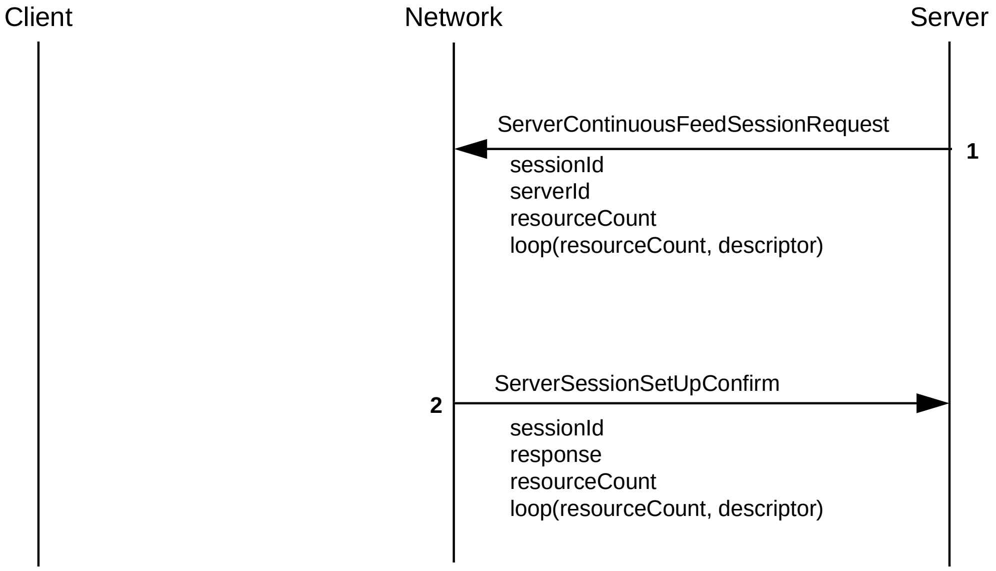

Client
Network
Server
ServerContinuousFeedSessionRequest
1
sessionId
serverId
resourceCount
loop(resourceCount, descriptor)
ServerSessionSetUpConfirm
2
sessionId
response
resourceCount
loop(resourceCount, descriptor)

Figure 13 Sequence of events for Server initiated Continuous Feed Session Set-Up

4.6.2.1 Server Initiates Continuous Feed Session Set-Up

Step 1 (Server):

To begin establishing a new continuous feed session, the Server shall send ServerContinuousFeedSessionRequest to the Network. The value of the transactionId field shall be selected by the Server and shall be used to correlate replies from the Network, if there are multiple outstanding ServerSessionRequest or ServerContinuousFeedSessionRequest messages. The value of the serverId field shall identify the Server. The Server shall set the value of the resourceCount field to the number of resource types required to initially establish the session. For each type of resource requested, the Server shall include a resourceDescriptor field in ServerContinuousFeedSessionRequest. Within each resource descriptor:

- The value of the requestId field shall be selected by the Server and can be used to correlate replies from the Network.
- The value of the request field shall be encoded to indicate whether the specific values requested for the resource are NEGOTIABLE or NON-NEGOTIABLE. If the resource is tagged NEGOTIABLE, the Server may accept alternative values proposed by the Network. If the resource is tagged NON-NEGOTIABLE, the Server will not accept any other value for the resource, and session establishment shall fail if the Network cannot assign the requested value.
- The value of the resourceId field shall be encoded as all 0’s.
- The value of the resourceType field shall be encoded to indicate the specific type of resource being requested for the session.
- The value of the resourceLength field shall indicate the number of bytes remaining in the resourceDescriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource requested.

Step 2 (Network):

On receipt of ServerContinuousFeedSessionRequest, if the Network is able to accept a request for a new session the Network shall verify that the serverId field represents an entity known to the Network. If the Network can accept a new session request and if the values of these fields are valid, the Network shall iterate through the list of requested resources to determine if the network can fulfill each individual resource request. If the Network can assign the requested values for all resources that the Server tagged NON-NEGOTIABLE, the Network shall then attempt to assign values to those resources that the Server tagged NEGOTIABLE and shall construct a ServerSessionConfirm message. The value of the sessionRequestId field shall be identical to the sessionRequestId in the ServerContinuousFeedSessionRequest message. The resourceCount field shall be identical to the value received from the Server in ServerContinuousFeedSessionRequest. For each resource requested, the Network shall include a resourceDescriptor field in ServerSessionConfirm. Within the resourceDescriptor:

- The value of the resourceRequestId field shall be identical to the value received from the Server.
- The value of the response field shall be encoded as one of the following values:
  1. rspResourceOK to indicate that the Network is able to assign the exact resource values requested by the Server.
  2. rspResourceNegotiate to indicate that the Network is not able to assign the exact resource values requested by the Server but has assigned the values included in the remainder of the descriptor. The Network shall not use this response if the Server indicated that the resource values requested was NON-NEGOTIABLE.
  3. rspResourceFailed to indicate that the Network is not able to assign the exact resource values requested by the Server, and the Server indicated that the resource values requested was NON-NEGOTIABLE. rspResourceFailed shall also indicate that the Network cannot assign the resource at all regardless of whether the Server tagged the request as NEGOTIABLE or NON-NEGOTIABLE.
- If the response field is encoded as rspResourceOK, the value of the resourceId field shall be assigned by the Network to uniquely identify the assigned resources for their duration within the session. If the response field is encoded as either rspResourceNegotiate or rspResourceFailed, the value of the resourceId field shall be encoded as all 0’s.
- The value of the resourceType field shall indicate a specific type of resource and shall be identical to the value received from the Server.
- The value of the resourceLength field shall indicate the number of bytes remaining in the descriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource being assigned. If the response field is encoded as rspResourceOK or rspResourceFailed, the values shall match those received from the Server. If the response field is encoded as rspResourceNegotiate, one or more values shall differ from those received from the Server.

The value of the response field in ServerSessionConfirm message shall be encoded as one of the following values:

- rspOK if, for all of the requested resources, the response field is encoded as ASSIGN.
- rspResourceContinue if, for all of the requested resources indicated as NON-NEGOTIABLE by the Server, the response field is encoded as ASSIGN, and for one or more of the requested resources indicated as NEGOTIABLE by the Server, the response field is encoded as NEGOTIATE.
- rspResourceFailed if for any of the requested resources, the response field was encoded as rspResourceFailed.

At this point, the Network shall consider the Continuous Feed Session to be active.

If the Network cannot support a continuous feed session set-up or the serverId field is invalid, or the Network cannot assign the requested values for any resource that the Server has tagged as either NEGOTIABLE or NON-NEGOTIABLE flow shall shift to the ”Network rejects ServerContinuousFeedSessionRequest” scenario.

Step 3 (Server):

On receipt of ServerSessionConfirm with a valid sessionRequestId field and the response field encoded as rspOK, the Server shall iterate through the resource list and provision itself to use the assigned resources for the session. At this point the Server shall consider the Continuous Feed Session to be active.

On receipt of ServerSessionConfirm with the response field encoded as rspResourceContinue, the Server shall iterate through the resource list to determine which resources the Network was able to assign as requested by the Server and which resources the Network has assigned alternate values for. If the Server accepts the alternate resource values assigned by the Network, the Server shall provision itself to use the assigned resources. At this point, the Server shall consider the Continuous Feed Session to be active.

If the alternate resource values assigned by the Network are unacceptable to the Server, the Server shall terminate the session establish procedure, the flow shall shift to the “Server terminates resource negotiation” scenario.

4.6.2.2 Network rejects ServerContinuousFeedSessionSetUpRequest

Step 2 (Network):

On receipt of ServerContinuousFeedSessionSetUpRequest, the Network may reject the request for the following reasons:

- The Network is unable to accept a request for a new continuous feed session.
- The serverId field is invalid.
- The Network is unable to assign requested resources.

In these cases, the Network shall send ServerSessionConfirm to the Server. The value of the sessionRequestId shall be identical to the value received in ServerSessionRequest, and the value of the response field shall indicate why the Network is rejecting session establishment. The Network shall terminate the session set-up procedure at this point. The following response codes apply:

- rspNeNoCalls - Indicates that the Network is not accepting new session requests.
- rspNeInvalidServer - Indicates that the serverId field is invalid.
- rspResourceFailed - Indicates that the Network could not satisfy one or more requested resources.

Step 3 (Server):

On receipt of ServerSessionConfirm with the response field set to indicate that the request was rejected, the Server shall terminate the continuous feed session establishment procedure.

4.6.2.3 Server terminates resource negotiation

Step 3 (Server):

On receipt of ServerSessionConfirm, if the Server determines that it is unable to use any of the assigned resources, the Server may terminate the session by sending the ServerReleaseRequest message. The sessionId field shall be identical to the sessionId in the ServerSessionConfirm message. The reason field shall be set to rsnSeNoResource to indicate that the server could not use the assigned resources.

Step 4 (Network)

On receipt of the ServerReleaseRequest message, the Network shall send the ServerReleaseConfirm message to the Server. At this time, the Network shall consider the session to be torn down and may release all resources assigned to the session.

On receipt of the ServerReleaseConfirm message, the Server shall consider the session to be torn down and may release all resources allocated for the session.

4.6.3 Server Add Resource Command Sequence

After a session has been established, the Server may add additional resources to the session.

Figure 14 illustrates the normal procedure for adding new resources to an existing session.

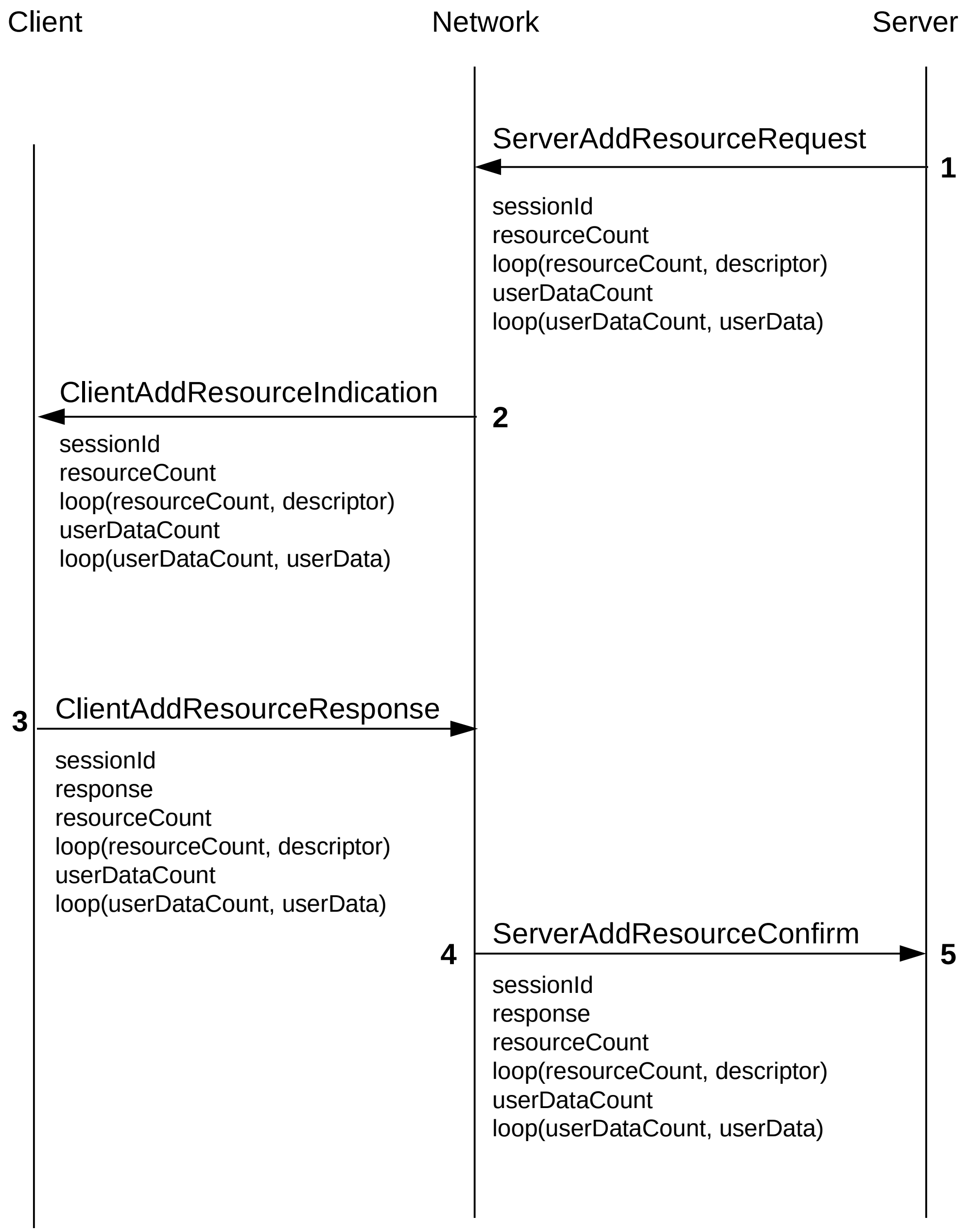

Client
Network
Server
ServerAddResourceRequest
1
sessionId
resourceCount
loop(resourceCount, descriptor)
userDataCount
loop(userDataCount, userData)
ClientAddResourceIndication
2
sessionId
resourceCount
loop(resourceCount, descriptor)
userDataCount
loop(userDataCount, userData)
3
ClientAddResourceResponse
sessionId
response
resourceCount
loop(resourceCount, descriptor)
userDataCount
loop(userDataCount, userData)
ServerAddResourceConfirm
4
5
sessionId
response
resourceCount
loop(resourceCount, descriptor)
userDataCount
loop(userDataCount, userData)

Figure 14 Sequence of events for Adding Resources to a Session

4.6.3.1 Server Initiates Add Resource Request

Step 1(Server):

To start the procedure for adding resources to an existing session, the Server shall send ServerAddResourceRequest to the Network. The value of the sessionId field shall indicate the session to which the resources are to be added. The Server shall select the value of the transactionId to be unique. The value of the resourceCount field shall indicate the number of resources to be added to the session. For each resource to be added, the Server shall include a resourceDescriptor field encoded as follows:

- The value of the resourceRequestId field shall be selected by the Server and can be used to correlate replies from the Network.
- The value of the request field shall be encoded to indicate whether the specific values requested for the resource are NEGOTIABLE or NON-NEGOTIABLE. If the resource is tagged NEGOTIABLE, the Server may accept alternative values proposed by the Network. If the resource is tagged NON-NEGOTIABLE, the Server will not accept any other value for the resource, and session establishment shall fail if the Network cannot assign the requested value.
- The value of the resourceId field shall be encoded as all 0’s.
- The value of the resourceType field shall be encoded to indicate the specific type of resource being requested for the session.
- The value of the resourceLength field shall indicate the number of bytes remaining in the resourceDescriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource requested.

Step 2 (Network):

On receipt of ServerAddResourceRequest, the Network shall iterate through the list of requested resources to determine if the network can fulfill each individual resource request. If the Network can assign the requested values for all resources that the Server tagged NON-NEGOTIABLE, the Network shall proceed to assign values to those resources that the Server tagged NEGOTIABLE and to construct a ServerAddResourceConfirm message. The value of the resourceCount field shall be identical to the value received from the Server in ServerAddResourceRequest. For each resource requested, the Network shall include a resourceDescriptor field in ServerAddResourceConfirm. Within the resourceDescriptor:

- The value of the resourceRequestId field shall be identical to the value received from the Server.
- The value of the response field shall be encoded as one of the following values:
  1. rspResourceOK to indicate that the Network is able to assign the exact resource values requested by the Server.
  2. rspResourceNegotiate to indicate that the Network is not able to assign the exact resource values requested by the Server but has assigned the values included in the remainder of the descriptor. The Network shall not use this response if the Server indicated that the resource values requested was NON-NEGOTIABLE.
  3. rspResourceFailed to indicate that the Network is not able to assign the exact resource values requested by the Server, and the Server indicated that the resource values requested was NON-NEGOTIABLE. rspResourceFailed shall also indicate that the Network cannot assign the resource at all regardless of whether the Server tagged the request as NEGOTIABLE or NON-NEGOTIABLE.
- If the response field is encoded as rspResourceOK, the value of the resourceId field shall be assigned by the Network to uniquely identify the assigned resources for their duration within the session. If the response field is encoded as either rspResourceNegotiate or rspResourceFailed, the value of the resourceId field shall be encoded as all 0’s.
- The value of the resourceType field shall indicate a specific type of resource and shall be identical to the value received from the Server.
- The value of the resourceLength field shall indicate the number of bytes remaining in the descriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource being assigned. If the response field is encoded as rspResourceOK or rspResourceFailed, the

values shall match those received from the Server. If the response field is encoded as rspResourceNegotiate, one or more values shall differ from those received from the Server.

The value of the response field in ServerAddResourceConfirm message shall be encoded as one of the following values:

- rspOK if, for all of the requested resources, the response field is encoded as ASSIGN.
- rspResourceContinue if, for all of the requested resources indicated as NON-NEGOTIABLE by the Server, the response field is encoded as ASSIGN, and for one or more of the requested resources indicated as NEGOTIABLE by the Server, the response field is encoded as NEGOTIATE.
- rspResourceFailed if for any of the requested resources, the response field was encoded as rspResourceFailed.

If the Network cannot assign the requested values for any resource that the Server has tagged as NON-NEGOTIABLE, flow shall shift to the “Network Unable to Assign NON-NEGOTIABLE Resource” scenario.

Step 3 (Server):

On receipt of ServerAddResourceConfirm with the response field encoded as rspOK, the Server shall iterate through the resource list and provision itself to use the assigned resources for the session. It shall send ServerAddResourceRequest to the Network with the response field encoded as rspResourceCompleted. The value of the userDataCount field shall indicate the number of userDataBytes in the remainder of the message.

On receipt of ServerAddResourceConfirm with the response field encoded as rspResourceContinue, the Server shall iterate through the resource list to determine which resources the Network was able to assign as requested by the Server and which resources the Network has assigned alternate values for. If the Server accepts the alternate resource values assigned by the Network, the Server shall provision itself to use the assigned resources for the session and shall send ServerAddResourceRequest to the Network with the response field encoded as rspResourceCompleted. The Server shall select the value of the transactionId to be unique within the context of the session. The value of the userDataCount field shall indicate the number of userDataByte in the remainder of the message.

If the alternate resource values assigned by the Network are unacceptable to the Server, the Server shall terminate the add resource procedure, the flow shall shift to the “Server terminates resource negotiation” scenario.

Step 4 (Network):

On receipt of ServerAddResourceRequest with the reason field encoded as rspResourceCompleted, the Network shall send ClientAddResourceIndication to the Client. The value of the sessionId field shall be identical to the value received in ServerAddResourceRequest. The value of the resourceCount field shall indicate the total number of resources being added to the Client side of the session. If there are no additional resources to be added to the Client side of the session, the Network shall send the ClientAddResourceIndication message with the resourceCount field set to 0. For each type of resource added, the Network shall include a resourceDescriptor field. Within each resourceDescriptor:

- The value of the resourceId field shall be assigned by the Network to uniquely identify the assigned resources for their duration within the session.
- The value of the resourceType field shall be encoded to indicate the specific type of resource being assigned for the session.
- The value of the resourceLength field shall indicate the number of bytes remaining in the resourceDescriptor.
- The values of the resource field shall be encoded as appropriate for the specific type of resource being assigned.

The value of the transactionId field shall be identical to the value of the transactionId field received in the ServerAddResourceRequest message. The values of the userDataCount and userDataByte fields sent in

ClientAddResourceIndication shall be identical to those received in the ServerAddResourceRequest message.

Step 5 (Client):

On receipt of ClientAddResource Indication, the Client shall determine if it is capable of using the additional resources assigned for the session by the Network. If the Client can use all of the additional resources, it shall send ClientAddResourceResponse to the Network with the response field set to rspOK. The value of the transactionId field shall be identical to the value received from the Network. The value of the userDataCount field shall indicate the number of userDataBytes present in the message. At this point, the Client shall consider the additional resources as committed to the session.

If the Client cannot use one or more of the additional resources, flow shall shift to the “Client unable to use additional resources” scenario.

Step 6 (Network):

On receipt of ClientAddResourceResponse message with the response field set to indicate that the Client accepted the additional resources, the Network shall send ServerAddResourceConfirm to the Server. The values of the transactionId, userDataCount, and userDataByte fields shall be identical to the values received from the Client. After sending the message, the Network shall consider the additional resources to be committed to the session.

Step 7 (Server):

On receipt of ServerAddResourceConfirm, the Server shall consider the additional resources to be committed to the session.

4.6.3.2 Network Unable to Assign NON-NEGOTIABLE Resource

Step 2 (Network):

If the Network cannot assign the requested values for any resource that the Server has tagged as NON-NEGOTIABLE or cannot assign any values for one or more resources regardless of how the Server tagged the resource(s), the Network shall send ServerAddResourceConfirm to the Server. The value of the reason field shall be set to rsnNeNoResource to indicate that the Network could not assign the requested values for the resource indicated by the requestId field. The network shall terminate the resource addition at this point. The session state shall be maintained with the resources that existed before the message was received.

Step 5 (Server):

On receipt of ServerAddResourceConfirm with the reason code set to rsnNeNoResource, the Server shall terminate resource addition. The session state shall be maintained with the resources that existed before the procedure was started.

4.6.3.3 Server Terminates Resource Negotiation

Step 3 (Server):

On receipt of ServerAddResourceConfirm with the response field encoded as rspResourceCompleted or rspResourceContinue, if the Server decides to terminate resource negotiation because one or more resource values assigned by the Network are unacceptable, the Server shall send ServerProvisionResourceRequest to the Network. The value of the sessionId field shall be identical to the value received in ServerAddResourceConfirm, and the reason field shall indicate why the Server has terminated resource negotiation. The value of the userDataCount field shall be 0 and no userDataByte field shall be included.

Step 4 (Network):

On receipt of ServerProvisionResourceRequest with the reason field set to rsnSeRejResource to indicate that the resources were rejected, the Network shall terminate the resource addition procedure and send ServerProvisionResourceConfirm to the Server. The values of the userDataCount and userDataByte fields shall be identical to those received in ServerProvisionResourceRequest. At this point, the Network shall consider the resource addition procedure as terminated. The session state shall be maintained with the resources that existed before the resource addition procedure was started. Steps 5 and 6 do not occur.

Step 7 (Server):

On receipt of message with the response field set to rsnSeRejResource to indicate that the resource addition procedure was rejected, the Server shall consider the resource addition procedure as terminated. The session state shall be maintained with the resources that existed before the resource addition procedure was started.

4.6.3.4 Client Unable to Use Additional Resources

Step 7 (Client):

On receipt of ClientAddResourceIndication, if the Client cannot use one or more of the additional resources, it shall send ClientAddResourceConfirm to the Network. The value of the sessionId field shall be identical to the value received in ClientAddResourceIndication, and the reason field shall be set to rsnClNoResource to indicate that the Client cannot use one or more of the additional resources. At this point, the Client shall consider the resource addition procedure to be terminated. The session state shall be maintained with the resources that existed before the resource addition procedure was started.

Step 8 (Network):

On receipt of ClientAddResourceResponse message with the response set to indicate that the Client rejected the additional resources, the Network shall send ServerProvisionResourceConfirm with the response field set to indicate that the add resource request was rejected. At this point, the Network shall consider the resource addition procedure to be terminated. The session state shall be maintained with the resources that existed before the resource addition procedure was started.

Step 9 (Server):

On receipt of ServerProvisionResourceConfirm with the response field set to indicated that the add resource request was rejected, the Server shall consider the resource addition procedure to be terminated. The session state shall be maintained with the resources that existed before the resource addition procedure was started.

4.6.3.5 Network Unable to Assign NON-NEGOTIABLE Resource

Step 2 (Network):

If the Network cannot assign the requested values for any resource that the Server has tagged as NON-NEGOTIABLE or cannot assign any values for one or more resources regardless of how the Server tagged the resource(s), the Network shall send ServerAddResourceConfirm to the Server. The value of the reason field shall be set to rsnNeNoResource to indicate that the Network could not modify the requested values for the resource indicated by the requestId field. The network shall terminate the resource addition at this point. The session state shall be maintained with the resources that existed before the message was received.

Step 5 (Server):

On receipt of ServerAddResourceConfirm with the reason code set to rsnNeNoResource, the Server shall terminate resource modification. The session state shall be maintained with the resources that existed before the resource modification procedure was started.

4.6.4 Server Session Delete Resource Command Sequence

Figure 15 illustrates the normal procedure for deleting resources from an existing session.

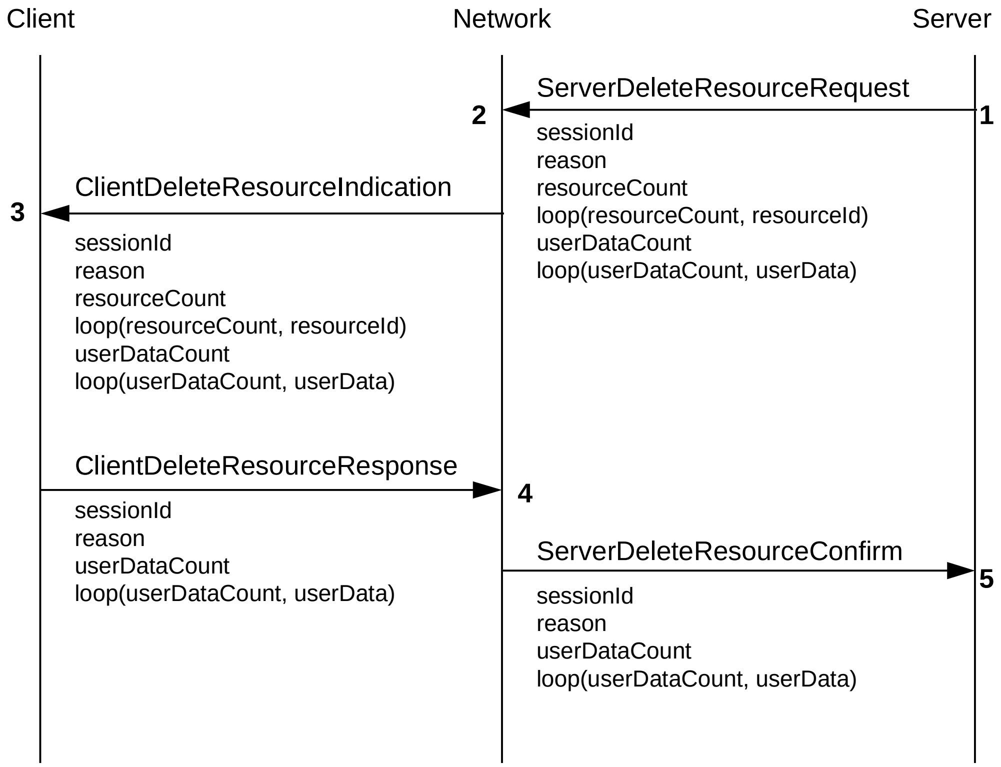

Client
Network
Server
ServerDeleteResourceRequest
2
1
sessionId
reason
resourceCount
loop(resourceCount, resourceId)
userDataCount
loop(userDataCount, userData)
ClientDeleteResourceIndication
3
sessionId
reason
resourceCount
loop(resourceCount, resourceId)
userDataCount
loop(userDataCount, userData)
ClientDeleteResourceResponse
4
sessionId
reason
userDataCount
loop(userDataCount, userData)
ServerDeleteResourceConfirm
5
sessionId
reason
userDataCount
loop(userDataCount, userData)

Figure 15 Sequence of events for Deleting Resources from a Session

Step 1 (Server):

To begin the procedure for deleting resources from a session, the Server shall stop using the resources that it intends to delete and send ServerDeleteResourceRequest to the Network. The sessionId shall identify the session from which the resources are to be deleted. The value of the reason field shall indicate why the Server is deleting the resources from the session. The value of the resourceCount field shall indicate the number of resourceId fields present in the remainder of the message. The userDataCount and userDataByte fields shall be set to values which will be passed to the Client in the ClientDeleteResourceIndication message.

Step 2 (Network):

On receipt of ServerDeleteResourceRequest, the Network shall verify that the session exists and is associated with the Server. The Network shall also verify that all of the resourceIds are valid for the session. If these conditions are met, the Network shall send ClientDeleteResourceIndication to the Client. The value of the sessionId, userDataCount, and userDataByte fields shall be identical to the values received from the Server. The value of the reason field shall indicate that the Server has requested that the resources be deleted from the session. The value of the resourceCount field shall indicate the number of resourceId fields present in the remainder of the message. If the resource deletion procedure does not require any resources to be deleted from the Client, the resourceCount field shall be set to 0 and no resourceIds shall be included in the message.

If the sessionId is invalid or not associated with the Server or if any of the resourceIds are invalid or not connected to the session, flow shall shift to the “Network Rejects Delete Resource” scenario.

Step 3 (Client):

On receipt of ClientDeleteResourceIndication, the Client shall verify that the session exists. The Client shall also verify that all of the resourceIds are valid for the session. The Client shall send ClientDeleteResourceResponse to the Network. The value of the sessionId and transactionId fields shall be identical to the values received from the Network. The userDataCount and userDataByte fields shall be set to values which will be passed to the Server in the ServerDeleteResourceConfirm message. At this point, the Client shall consider the resource deletion procedure completed and shall not use the deleted resources.

If the sessionId is invalid or one or more of the indicated resourceIds is invalid, flow shall shift to the "Client Rejects Delete Resource" scenario.

Step 4 (Network):

On receipt of ClientDeleteResourceResponse, the Network shall send ServerDeleteResourceConfirm to the Server. The value of the sessionId, userDataCount, and userDataByte fields shall be identical to the values received from the Client. At this point the Network shall consider the resource deletion completed and may release the deleted resources.

Step 5 (Server):

On receipt of ServerDeleteResourceConfirm, the Server shall consider the resource deletion procedure completed.

4.6.4.1 Network Rejects Delete Resource

Step 2 (Network):

If the requested resources are not valid for the session or server, the Network shall send ServerDeleteResourceConfirm to the Server. The value of the reason field shall be set to rsnNeNoResource to indicate that the Network could not delete the requested resources indicated by the resourceId fields. The network shall terminate the resource deletion at this point. The session state shall be maintained with the resources that existed before the ServerDeleteResourceRequest was received.

Step 4 (Server):

On receipt of ServerDeleteResourceConfirm with the reason code set to rsnNeNoResource, the Server shall terminate resource deletion procedure. The session state shall be maintained with the resources that existed before the resource deletion procedure was started.

4.6.4.2 Client Rejects Delete Resource

Step 3 (Client):

On receipt of ClientDeleteResourceIndication, if the one or more of the resources is invalid, it shall send ClientDeleteResourceConfirm to the Network. The value of the sessionId field shall be identical to the value received in ClientDeleteResourceIndication, and the reason field shall be set to rsnClNoResource to indicate that the Client cannot delete one or more of the indicated resourceIds. At this point, the Client shall consider the resource deletion procedure to be terminated. The session state shall be maintained with the resources that existed before the resource deletion procedure was started.

Step 4 (Network):

On receipt of ClientDeleteResourceResponse message with the response set to indicate that the Client rejected the delete resource command, the Network shall send ServerDeleteResourceConfirm with the response field set to indicate that the delete resource request was rejected. At this point, the Network shall consider the resource deletion procedure to be terminated. The session state shall be maintained with the resources that existed before the resource deletion procedure was started.

Step 5 (Server):

On receipt of ServerDeleteResourceConfirm with the response field set to indicated that the delete resource request was rejected, the Server shall consider the resource deletion procedure to be terminated. The session state shall be maintained with the resources that existed before the resource deletion procedure was started.

4.6.5 Server Session Tear-Down Command Sequence

Figure 16 illustrates the normal procedure for session release initiated by the Server.

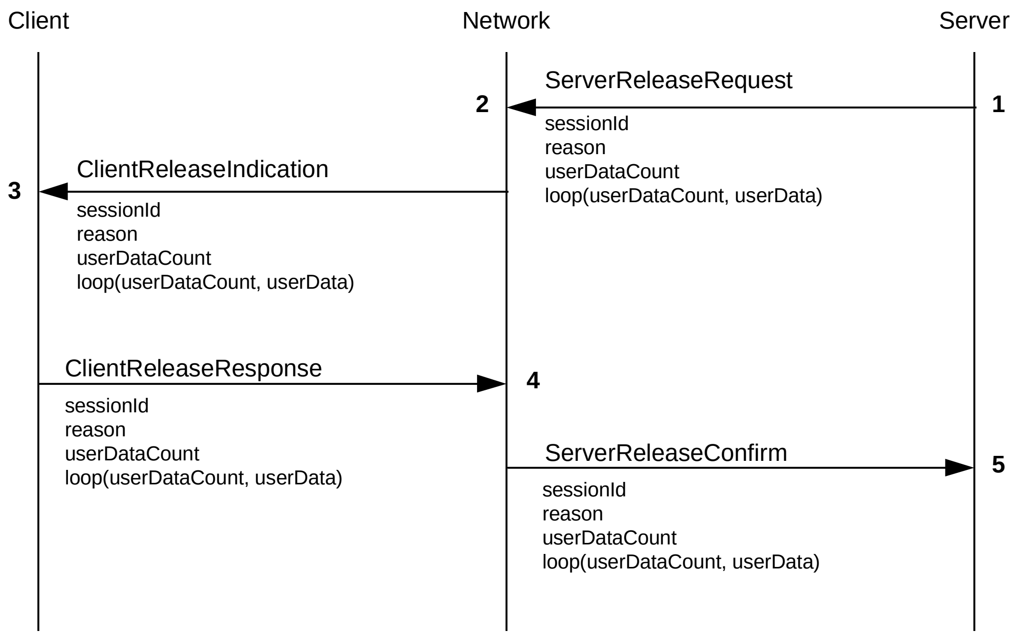


Figure 16 Sequence of events for Server initiated session tear-down

4.6.5.1 Server Initiates Release Request

Step 1 (Server):

To start the procedure for releasing an existing session, a Server shall send ServerReleaseRequest to the Network. The value of the sessionId field shall correspond to an existing session, and the value of the reason field shall indicate why the Server is releasing the session. The value of the userDataCount field shall indicate the number of userDataByte present in the message.

Step 2 (Network):

On receipt of ServerReleaseRequest, the Network shall verify that the value of the sessionId field corresponds to an existing session and the session belongs to the Server. If the sessionId is valid and owned by the Server, the Network shall send ClientReleaseIndication to the Client. The value of the sessionId field shall be identical to the sessionId received from the Server, and the value of the reason field shall indicate that the session is being released at the request of the Server. The values of the userDataCount and userDataByte fields shall be identical to the values received from the Server.

If the Network determines that the sessionId received in ServerReleaseRequest is invalid or that the Server does not own the session, flow shall shift to the “Network Rejects ServerReleaseRequest” scenario.

Step 3 (Client):

On receipt of ClientReleaseIndication, the Client shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid, the Client shall first release all resources assigned to the session and then send ClientReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall indicate the number of userDataByte in the message. At this point, the Client shall consider the session to be terminated and may release all session resources.

If the Client determines that the sessionId is invalid, flow shall shift to the “Client Rejects ClientReleaseIndication” scenario.

Step 4 (Network):

On receipt of ClientReleaseResponse, the Network shall release all Client interface resources assigned to the session and send ServerReleaseConfirm to the Server. The values of the sessionId, userDataCount, and userDataByte fields shall be identical to the values received in ClientReleaseResponse. At this point, the Network shall consider the session to be terminated and may release all session resources.

Step 5 (Server):

On receipt of ServerReleaseConfirm, the Server shall release all resources assigned to the session. At this point, the Server shall consider the session to be terminated and may release all session resources.

4.6.5.2 Network Rejects Server Release Request

Step 2 (Network):

If the Network determines that the value of the sessionId field received in ServerReleaseRequest is invalid or that the Server is not the owner of the session, the Network shall send ServerReleaseConfirm to the Server. The value of the sessionId field shall be identical to the value received in ServerReleaseRequest, and the value of the reason parameter shall indicate that the sessionId is invalid or not owned by the Server. At this point the session release procedure is terminated. After sending ServerReleaseConfirm, the Network may take other actions such as initiating an audit with the Server.

Step 3 (Server):

On receipt of ServerReleaseConfirm, the Server shall terminate the session release procedure. The Server may take other actions such as initiating an audit with the Network or initiating internal diagnostics.

4.6.5.3 Client Rejects Client Release Indication

Step 3 (Client):

If the Client determines that the value of the sessionId field received in ClientReleaseIndication is invalid, the Client shall send ServerReleaseConfirm to the Network. The value of the sessionId field shall be identical to the value received in ClientReleaseResponse, and the value of the reason field shall indicate that the sessionId is invalid. After sending ServerReleaseConfirm, the Client may take other actions such as initiating an audit with the Network.

Step 4 (Network):

On receipt of ClientReleaseResponse which indicates that the session is invalid, the Network shall release resources assigned to the session. The Network shall send ServerReleaseConfirm to the Server. The value of the sessionId field shall be identical to the value received in ClientReleaseResponse since the Network previously validated it on receipt of ServerReleaseRequest The value of the reason field shall indicate that a procedure error has occurred with the Client. The value of the userDataCount field shall be 0. At this point, the Network may consider the release procedure as complete and may release all resources associated with the session. After sending ServerReleaseConfirm, the Network may initiate an audit with the Client.

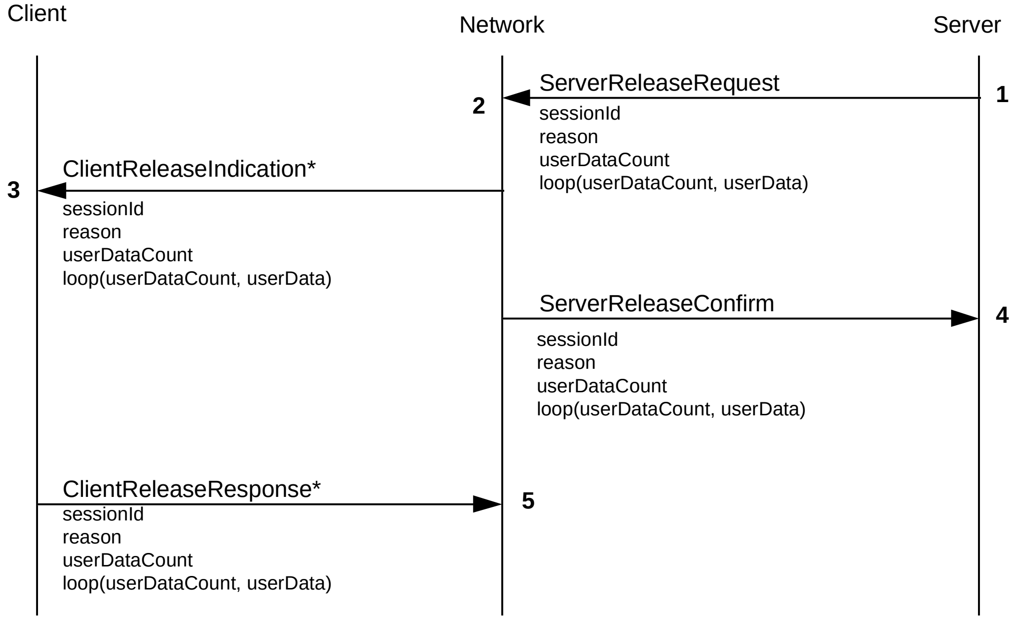
Step 5 (Server):

On receipt of ServerReleaseConfirm, the Server shall consider the release procedure as completed and may release all resources assigned to the session.

4.6.6 Server Continuous Feed Session Tear-Down Command Sequence

After a continuous feed session has been established, it may be terminated using the Server Release sequence. A Session Tear-Down command sequence for a continuous feed session may be initiated only by the Server.

Figure 17 describes the sequence of events that occur during a Server initiated tear-down of a continuous feed session. When the continuous feed session is torn down, any sessions which are connected to the continuous feed session are first torn down by the Network.



* - Indicates that the Network repeats this procedure for each session connected to the CFS.

Figure 17 Sequence of events for Server initiated Continuous Feed Session Tear-Down

4.6.6.1 Server Initiates Continuous Feed Session Release Request

Step 1 (Server):

To start the procedure for releasing a Continuous Feed Session, a Server shall send ServerReleaseRequest to the Network. The value of the sessionId field shall correspond to an existing continuous feed session, and the value of the reason field shall indicate why the Server is releasing the session. The value of the userDataCount field shall indicate the number of userDataByte present in the message.

Step 2 (Network):

On receipt of ServerReleaseRequest, the Network shall verify that the value of the sessionId field corresponds to an existing session and the session belongs to the Server. If the sessionId is valid and owned by the Server, the Network shall send ClientReleaseIndication to each Client which is connected to the continuous feed session. The value of the sessionId field shall be the sessionId of the session which is connected to the continuous feed session, and the value of the reason field shall indicate that the session is being released because the continuous feed session to which the session is connected has been released by the Server. The values of the userDataCount and userDataByte fields shall be identical to the values received from the Server. The Network does not wait for the ClientReleaseResponse to be received from each Client before confirming the Server request. After sending the ClientReleaseIndication messages, the Network shall release all resources assigned to the continuous feed session and send ServerReleaseConfirm to the Server. The value of the sessionId shall be the sessionId of the continuous feed session, the userDataCount shall be set to 0, and no userDataBytes shall be included. At this point, the Network shall consider the continuous feed session to be terminated and may release all resources for the session.

If the Network determines that the sessionId received in ServerReleaseRequest is invalid or that the Server does not own the session, flow shall shift to the “Network Rejects ServerReleaseRequest” scenario.

Step 3 (Client):

On receipt of ClientReleaseIndication, the Client shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid, the Client shall first release all resources assigned to the session and then send ClientReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall indicate the number of userDataByte in the message. At this point, the Client shall consider the session to be terminated and may release all session resources.

If the Client determines that the sessionId is invalid, flow shall shift to the “Client Rejects ClientReleaseIndication” scenario.

Step 4 (Server):

On receipt of ServerReleaseConfirm, the Server shall release all resources assigned to the session. At this point, the Server shall consider the continuous feed session to be terminated and may release all session resources. The Server should also consider all sessions which were connected to the continuous feed session to be terminated and may release all resources allocated for the sessions.

Step 5 (Network):

On receipt of each ClientReleaseResponse, the Network shall release all Client interface resources assigned to the session. At this point, the Network shall consider the Client session to be terminated and may release any client session resources.

4.6.6.2 Network Rejects Server Release Request

Step 2 (Network):

If the Network determines that the value of the sessionId field received in ServerReleaseRequest is invalid or that the Server is not the owner of the session, the Network shall send *[message TBD]* to the Server. The value of the sessionId field shall be identical to the value received in ServerReleaseRequest, and the value of the reason parameter shall indicate that the sessionId is invalid or not owned by the Server. At this point the session release procedure is terminated. After sending *[message TBD]*, the Network may take other actions such as initiating an audit with the Server.

Step 3 (Server):

On receipt of *[message TBD]*, the Server shall terminate the session release procedure. The Server may take other actions such as initiating an audit with the Network or initiating internal diagnostics.

4.6.6.3 Client Rejects Client Release Indication

Step 3 (Client):

If the Client determines that the value of the sessionId field received in ClientReleaseIndication is invalid, the Client shall send ClientReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received in ClientReleaseIndication, and the value of the reason field shall indicate that the sessionId is invalid. After sending ClientReleaseResponse, the Client may take other actions such as initiating an audit with the Network.

Step 4 (Network):

On receipt of ClientReleaseResponse which indicates that the session is invalid, the Network shall release Client resources assigned to the session. At this point, the Network may consider the release procedure as complete and may release all Client resources associated with the session. The Network may initiate an audit with the Client.

4.6.7 Server Session Forward Command Sequence

[to be provided]

4.6.8 Server Session Transfer Command Sequence

[To be provided]

4.7 Network Initiated Command Sequences

The following Network initiated command sequences are defined in this section:

- Session tear-down command sequence.
- Continuous feed session tear-down command sequence.
- Client status request command sequence.
- Server status request command sequence.

4.7.1 Network Initiated Session Tear-Down Command Sequence

Figure 18 illustrates the normal procedure for session release initiated by the Network.

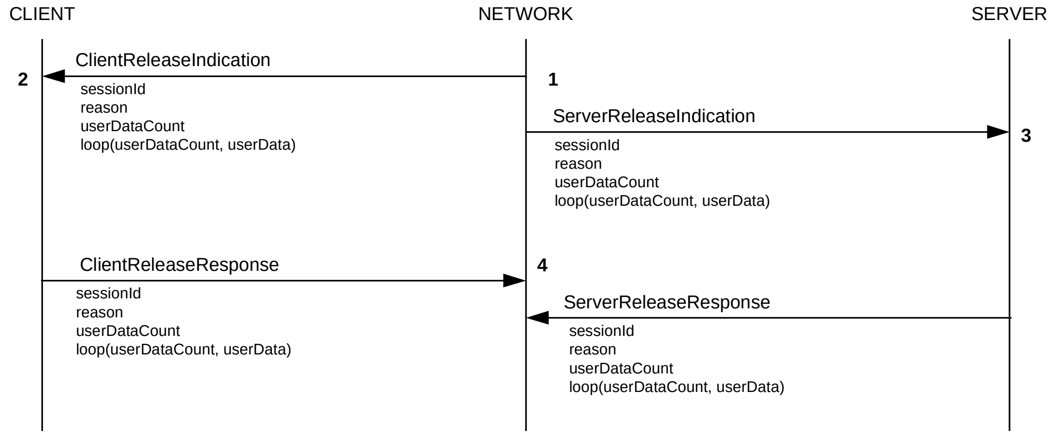


Figure 18 Sequence of events for Network initiated session tear-down

4.7.1.1 Network Initiates Session Tear-Down

Step 1 (Network):

To start the procedure for releasing an existing session, the Network shall send ClientReleaseIndication to the Client and send ServerReleaseIndication to the Server. The value of the sessionId field shall correspond to the existing session that is to be released, and the value of the reason field shall indicate why the Network is releasing the session. The value of the userDataCount field shall be 0 and no userDataBytes shall be included.

Step 2 (Client):

On receipt of ClientReleaseIndication, the Client shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid, the Client shall first release all resources assigned to the session and then send ClientReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall be 0 and no userDataBytes shall be included. At this point, the Client shall consider the session to be terminated and may release any resources allocated to the session.

If the Client determines that the sessionId is invalid, flow shall shift to the “Client Rejects ClientReleaseIndication” scenario.

Step 3 (Server):

On receipt of ServerReleaseIndication, the Server shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid, the Server shall first release all resources assigned to the session and then send ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall be 0 and no userDataBytes shall be included. At this point, the Server shall consider the session to be terminated and may release any resources allocated to the session.

If the Server determines that the sessionId is invalid, flow shall shift to the “Server Rejects Server Release Indication” scenario.

Step 3 (Network):

On receipt of ClientReleaseResponse, the Network shall release all Client interface resources assigned to the session.

On receipt of ServerReleaseResponse, the Network shall releases all Server interface resources assigned to the session.

After both the ClientReleaseResponse and ServerReleaseResponse have been received, the Network shall consider the session to be terminated.

4.7.1.2 Client Rejects Client Release Indication

Step 2 (Client):

If the Client determines that the value of the sessionId field received in ClientReleaseIndication is invalid, the Client shall send ClientReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received in ClientReleaseIndication, and the value of the reason field shall indicate that the sessionId is invalid. After sending ClientReleaseResponse, the Client may take other actions such as initiating an audit with the Network.

Step 3 (Network):

On receipt of ClientReleaseResponse, the Network releases all Client interface resources assigned to the session and terminate the release procedure with the Client. The Network may take other actions such as initiating an audit with the Client.

4.7.1.3 Server Rejects Server Release Indication

Step 4 (Server):

If the Server determines that the value of the sessionId field received in ServerReleaseIndication is invalid, the Server shall send ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received in ServerReleaseIndication, and the value of the reason field shall indicate that the sessionId is invalid. After sending ServerReleaseResponse, the Server may initiate an audit with the Network.

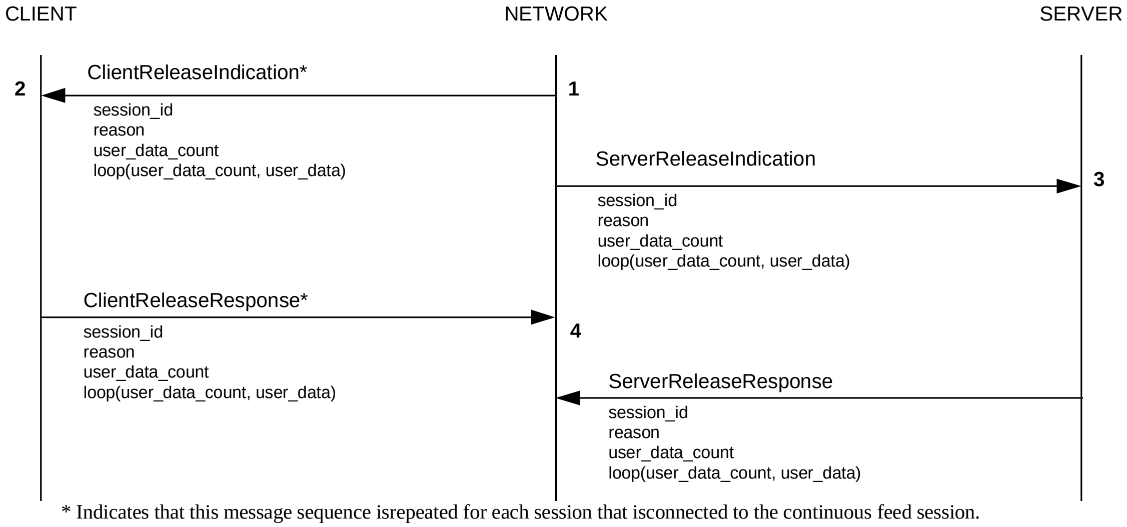
Step 5 (Network):

On receipt of ServerReleaseResponse, the Network shall release all Server interface resources assigned to the session and shall terminate the release procedure with the Server. The Network may take other actions such as initiating an audit with the Server.

4.7.2 Network Initiated Continuous Feed Session Tear-Down Command Sequence

The Network may initiate a continuous feed session tear-down sequence using the Server Disconnect Indication message and the Client Disconnect Indication message.

Figure 19 describes the sequence of events that occur during a Network initiated continuous feed session tear-down sequence.



* Indicates that this message sequence isrepeated for each session that isconnected to the continuous feed session.

Figure 19 Sequence of events for Network initiated continuous feed session tear-down.

4.7.2.1 Network Initiates Continuous Feed Session Tear-Down

Step 1 (Network):

To start the procedure for releasing a continuous feed session, the Network shall send ClientReleaseIndication to each Client which is connected to the continuous feed session and send ServerReleaseIndication to the Server. The value of the sessionId field shall correspond to the existing session that is to be released in the case of the Client and the continuous feed sessionId in the case of the Server, and the value of the reason field shall indicate why the Network is releasing the session. The value of the userDataCount field shall be 0 and no userDataBytes shall be included.

Step 2 (Client):

On receipt of ClientReleaseIndication, the Client shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid, the Client shall first release all resources assigned to the session and then send ClientReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall be 0 and no userDataBytes shall be included. At this point, the Client shall consider the session to be terminated and may release any resources allocated to the session.

If the Client determines that the sessionId is invalid, flow shall shift to the “Client Rejects ClientReleaseIndication” scenario.

Step 3 (Server):

On receipt of ServerReleaseIndication, the Server shall verify that the value of the sessionId field corresponds to an existing session. If the sessionId is valid, the Server shall first release all resources assigned to the continuous feed session. The Server shall also release all resources for any sessions which are connected to the continuous feed session. The Server then sends ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the value of the userDataCount field shall be 0 and no userDataBytes shall be included. At this point, the Server shall consider the session to be terminated and any sessions connected to the continuous feed session to be terminated.

If the Server determines that the sessionId is invalid, flow shall shift to the “Server Rejects ServerReleaseIndication” scenario.

Step 3 (Network):

On receipt of ClientReleaseResponse, the Network shall release all Client interface resources assigned to the session The Network shall consider the connected session to be terminated..

On receipt of ServerReleaseResponse, the Network shall releases all Server interface resources assigned to the session. The Network shall consider the continuous feed session to be terminated.

4.7.2.2 Client Rejects Client Release Indication

Step 2 (Client):

If the Client determines that the value of the sessionId field received in ClientReleaseIndication is invalid, the Client shall send ClientReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received in ClientReleaseIndication, and the value of the reason field shall indicate that the sessionId is invalid. After sending ClientReleaseResponse, the Client may take other actions such as initiating an audit with the Network.

Step 3 (Network):

On receipt of ClientReleaseResponse, the Network releases ant additional resources assigned to the session and consider the connected session to be terminated. The Network may take other actions such as initiating an audit with the Client.

4.7.2.3 Server Rejects Server Release Indication

Step 4 (Server):

If the Server determines that the value of the sessionId field received in ServerReleaseIndication is invalid, the Server shall send ServerReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received in ServerReleaseIndication, and the value of the reason field shall indicate that the sessionId is invalid. After sending ClientReleaseResponse, the Server may initiate an audit with the Network.

Step 5 (Network):

On receipt of ClientReleaseResponse, the Network shall release all resources assigned to the continuous feed session and shall consider the session to be terminated. The Network may take other actions such as initiating an audit with the Server.

4.7.3 Network Initiated session list Client Status Command Sequence

Figure 20 illustrates the procedure used by the Network for determining the list of sessions in which the Client believes it is participating.

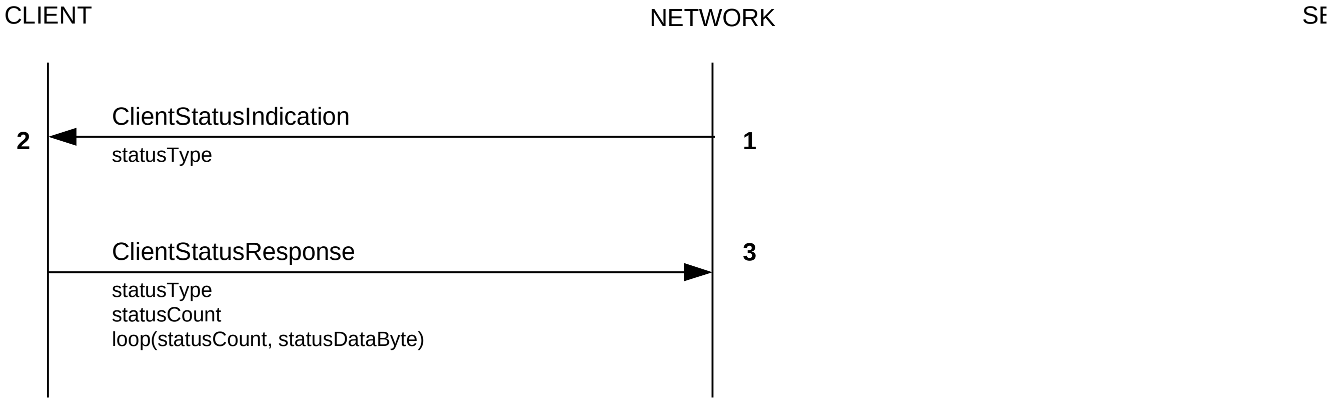


Figure 20 Sequence of events for Network Requests Session List Status from Client

4.7.3.1 Network Initiates session list Client Status command sequence

Step 1 (Network):

The network initiates a session audit by issuing a ClientStatusIndication message. The sessionType indicates that a session list is being requested.

Step 2 (Client):

The Client receives the ClientStatusIndication message and sends a ClientStatusResponse to the Network. The sessionCount field in the status contains the number of sessions which are active on the Client. A list of sessionIds follow the sessionCount field. There is one sessionId for each active session on the Client.

Step 3 (Network)

The Network receives the ClientStatusResponse message which terminates the sequence. The network may use the list of sessions to perform a session audit on the Client.

4.7.4 Network Initiated session list Server Status Command Sequence

Figure 21 illustrates the procedure used by the Network for determining the list of sessions in which the Server believes it is participating.

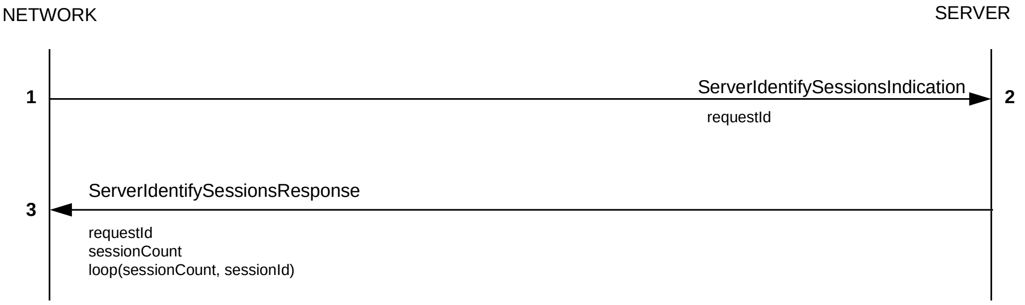

NETWORK
SERVER
1
ServerIdentifySessionsIndication
2
requestId
ServerIdentifySessionsResponse
3
requestId
sessionCount
loop(sessionCount, sessionId)

Figure 21 Sequence of events for Network Requests Session List status from Server

4.7.4.1 Network Initiates session list Server Status command sequence

Step 1 (Network):

The network initiates a session audit by issuing a ServerStatusIndication message. The requestId contains a unique identifier at the Network which will be used to correlate the response.

Step 2 (Server):

The Server receives the ServerStatusIndication message and sends a ServerStatusResponse to the Network. The requestId contains the requestId that was received in the ServerStatusIndication message. The sessionCount field contains the number of sessions which are active on the Server. A list of sessionIds follow the sessionCount field. There is one sessionId for each active session on the Server.

Step 3 (Network)

The Network receives the ServerStatusResponse message which terminates the sequence. The network may use the list of sessions to perform a session audit on the Server.

4.7.5 Network Initiated Audit of Client Session Command Sequence

Figure 22 illustrates the procedure used by the Network for determining the list of resources that the Client believes are assigned to a session in which the Client is participating.

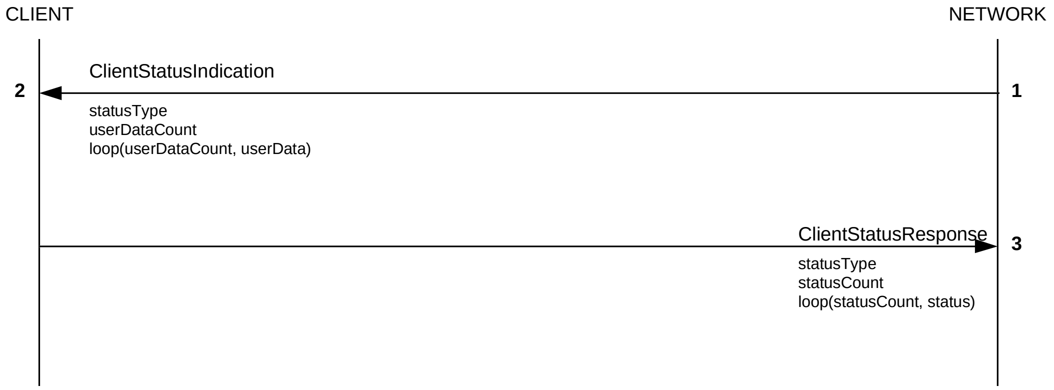


Figure 22 Sequence of events for a session audit Client Status by the Network

4.7.5.1 Network Initiates session audit Client Status command sequence

Step 1 (Network):

To start the procedure for determining the list of resources that the Client believes are assigned to a session, the Network shall send ClientStatusIndication to the Client.

Step 2 (Client):

On receipt of ClientStatusIndication with a valid sessionId, the Client shall determine the resources assigned to the requested sessions and shall send ClientStatusResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. The value of the response field shall indicate that the sessionId is valid and that the Client is reporting the resources that it believes are assigned. The value of the resourceCount field shall indicate the number of resource Descriptors in the remainder of the message.

On receipt of ClientStatusIndication with an invalid sessionId, the Client shall send ClientStatusResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the response field shall indicate that the sessionId is invalid. The value of the resourceCount field shall be 0.

Step 3 (Network):

On receipt of ClientStatusResponse, the Network shall consider the procedure to be completed and may use the returned session status to perform an audit on the session.

4.7.6 Network Initiated session audit Server Status Command Sequence

Figure 23 illustrates the procedure used by the Network for determining the list of resources that the Server believes are assigned to a session in which the Server is participating.

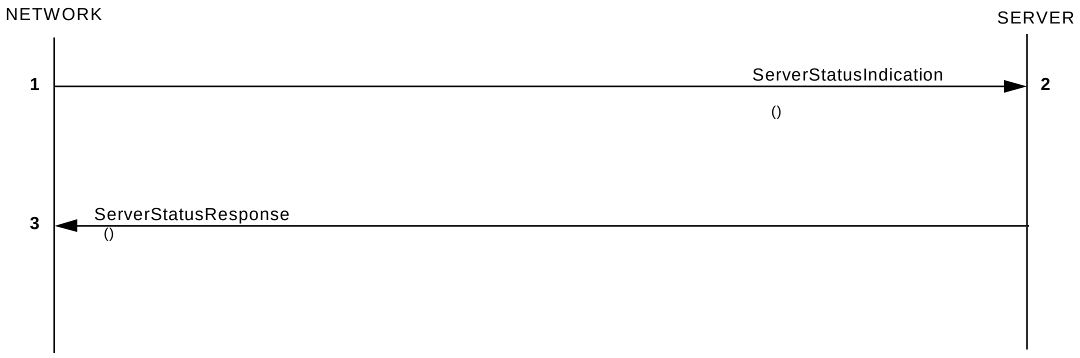


Figure 23 Sequence of events for a session audit Server Status initiated from the Network

4.7.6.1 Network Initiates session audit Server Status command sequence

Step 1 (Network):

To start the procedure for determining the list of resources that the Server believes are assigned to a session, the Network shall send ServerStatusIndication to the Server. The value of the sessionId field shall indicate a session in which the Server is participating.

Step 2 (Server):

On receipt of ServerStatusIndication with a valid sessionId, the Server shall determine the resources assigned to the session and shall send ServerStatusResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. The value of the response field shall indicate that the sessionId is valid and that the Server is reporting the resources that it believes are assigned. The value of the clientId shall indicate the Client that the Server believes is participating in the session. If the session is a continuous feed session, the clientId field shall be set to 0. The value of the resourceCount field shall indicate the number of resource Descriptors in the remainder of the message.

On receipt of ServerStatusIndication with an invalid sessionId, the Server shall send ServerSessionStatusResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network, and the response field shall indicate that the sessionId is invalid. The value of the resourceCount field shall be 0.

Step 3 (Network):

On receipt of ServerStatusResponse, the Network shall consider the procedure to be completed and may use the returned session status to perform an audit on the session.

4.8 Pass-Thru and Broadcast Messages

The Client and the Server may communicate between themselves using the Pass-Thru commands. These commands pass a message payload through the Network. The Pass-Thru message may be sent in either direction. The format of the payload of these messages is defined by the User.

Additionally, a Server may broadcast a Pass-Thru message to a subset of the Client population using the broadcast addressing mode.

4.8.1 ServerPassThru Message sent to a Client

Figure 24 describes a Pass-Thru message which is sent to a Client from a Server.

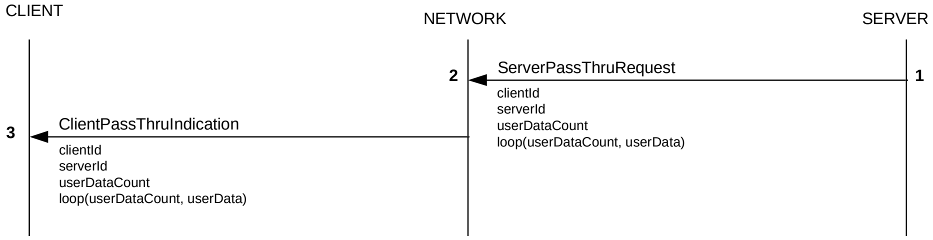


Figure 24 Sequence of events for Server initiated Pass-Thru Message

4.8.1.1 The Server sends a ServerPassThruRequest message to the Network

Step 1 (Server)

The Server creates a ServerPassThruRequest message which contains the addressMode and destAddress of the intended Client.

Step 2 (Network)

The Network validates the address mode and destination address. If the address is known to the Network it creates a ClientPassThruIndication message and delivers it to the indicated Client.

Step 3 (Client)

The Client receives the ClientPassThruIndication messageIndication message and processes it. The payload of the message is defined by User. There is no specific response to a ClientPassThru message indication. The Client may reply to a Pass-Thru message by sending a new ClientPassThruRequest message.

4.8.2 ClientPassThru Message sent to a Server

Figure 25 describes a Pass-Thru message which is sent to a Client from a Server.

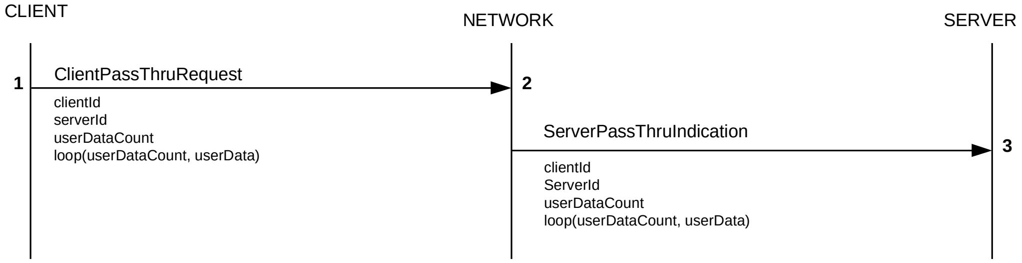


Figure 25 Sequence of events for Client initiated Pass-Thru Message

4.8.2.1 The Client sends a ClientPassThruRequest message to the Network

Step 1 (Client)

The Client creates a ClientPassThruRequest message which contains the addressMode and destAddress of the intended Server.

Step 2 (Network)

The Network validates the address mode and destination address. If the address is known to the Network it creates a ServerPassThruIndication message and delivers it to the indicated Server.

Step 3 (Server)

The Server receives the ServerPassThruIndication message and processes it. The payload of the message is defined by User. There is no specific response to a Pass-Thru message indication. The Server may reply to a message by sending a new ServerPassThruRequest message.

4.9 Error Handling

This section to be added

4.10 Timers

The DSM-CC protocol uses timers to help it recover from the unreliable nature of the underlying network. Upon sending a message, each entity will set a timer whenever the DSM-CC protocol state machine is expecting a return event. If the timer expires prior to receiving the expected event, the state machine will assume the event has been lost and initiate recovery. The only exception to this rule is the Network Session Proceeding timer (tNSesPcd) which is used by the Network to inform the Client that the command sequence is proceeding and it should reset its inactivity timer.

Timer values are 4 byte fields representing microseconds. The values that the timers are set to are sent to the Client or Server as part of their U-N Configuration. Provisioning of timers for the Network is a local matter and is outside the scope of DSM-CC. Refer to *[U-N Config section]* for a description of the configuration timer fields.

The following are the defined timer values:

Timer Value Name	Description

tCSesCnf	Client is waiting for a ClientSessionSetUPConfirm message
tCRelCnf	Client is waiting for a ClientReleaseConfirm message.
tNAddResReq	Network is waiting for a ServerAddResourceRequest message.
tNSesPcd	Network should send a ClientSessionProceedingIndication message to Client.
tNSSesRsp	Network is waiting for a ServerSessionSetUpResponse message.
tNCSesRsp	Network is waiting for a ClientSessionSetUpResponse message.
tNDelResRsp	Network is waiting for a ClientDeleteResourceResponse message.
tNSRelRsp	Network is waiting for a ServerSessionReleaseResponse message.
tNetRel	Network is waiting for either a Client or Server SessionRelesaseResponse.
tNSvrRelRsp	Network is waiting for a ServerReleaseResponse message during network initiated session teardown.
tNCliRelRsp	Network is waiting for a ClientReleseResponse message during network initiated session teardown.
tNCIdSes	Network is waiting for a ClientIdentifySessions message.
tNSIdSes	Network is waiting for a SessionIdentifySessions message.
tNCStat	Network is waiting for a ClientStatusConfirm message.
tNSStat	Network is waiting for a ServerStatusConfirm message.
tNAddResRsp	Network is waiting for a ClientAddResourceResponse message.
tSAddResCnf	Server is waiting for a ServerAddResourceConfirm message.
tSSesCnf	Server is waiting for a ServerSessionConfirm message.
tSCFSesCnf	Server is waiting for A ServerSessionConfirm from a continuous feed request.
tSAddResCnf	Server is waiting for a ServerAddResourceConfirm message.
tSDelResCnf	Server is waiting for a ServerDeleteResourceConfirm message.
tSSRelCnf	Server is waiting for a ServerReleaseConfirm message.
tSCFRelCnf	Server is waiting for a ServerReleaseConfirm for a Continuous Feed Session.

4.11 Reason Codes

The following reason codes are defined for use by the U-N messages.

Table 82 User-to-Network reason codes

Reason	Value	Description
rsnOK	0x0000	This indicates that the command sequence is proceeding normally
rsnClSessProcced	0x0001	Indicates that the Network is waiting on a response from the server.
rsnClUnkRequestID	0x0002	Indicates that the Client received a message which contained an unknown sessionRequestId.
rsnClNoResource	0x0003	Indicates that the Client rejected a session set-up because it was unable to use the assigned resources.
rsnClNoCalls	0x0004	Indicates that the Client rejected a session set-up because it was not accepting calls at that time.
rsnNeNoResource	0x0103	Indicates that the network is unable to assign one or more resources to a session.
rsnNeNotOwner	0x0104	Indicates that the network is unable to process a request from a User because the User is not the owner of the session.

rsnSeNoResource	0x0201	Indicates that the server is unable to complete a session set-up because the required resources are not available.
rsnSeRejResource	0x0202	Indicates that the server rejected the assigned resources.

4.12 Response Codes

The following response codes are defined for use by the U-N messages.

Table 83 User-to-Network response codes

Response	Value	Description
rspOK	0x0000	This indicates that the requested command completed with no errors.
rspNeNoCalls	0x0001	Indicates that the Network is unable to accept new sessions.
rspNeInvalidClient	0x0002	Indicates that the Network rejected the request due to an invalid clientId.
rspNeInvalidServer	0x0003	Indicates that the Network rejected the request due to an invalid serverId.
rspNeNoSession	0x0004	Indicates that the Network rejected the request because the requested sessionId did not exist.
rspSeNoCalls	0x0101	Indicates that the Server is unable to accept new sessions.
rspSeInvalidClient	0x0102	Indicates that the Server rejected the request due to an invalid clientId.
rspSeNoService	0x0103	Indicates that the Server rejected the request because the requested service could not be provided.
rspSeNoCFS	0x0104	Indicates that the Server rejected the request because the requested Continuous Feed Session could not be found.
rspSeNoSession	0x0105	Indicates that the Server rejected the request because the requested sessionId did not exist.
rspSeProcError	0x0106	Indicates that the Server generated a procedure error as the result of a request.
rspResourceContinue	0x1001	This indicates that a resource request completed with no errors but, an assigned resource was assigned an alternate value by the Network.
rspResourceFailed	0x1002	This indicates that a resource request failed because the Network was unable to assign the requested resources.
rspResourceOK	0x1003	Indicates that the requested command completed with no errors.
rspResourceNegotiate	0x1004	Indicates that the Network was able to complete a request but has assigned alternate values to a negotiable field.
rspResourceCompleted	0x1005	Indicates that the Server has accepted

		the resources assigned by the Network.

4.13 User-Network Messages State Tables

[These state table shave been updated to match the current User-Network Command Sequence scenarios described in a previous section. In addition, the tables include an inactivity recovery timer strategy. Please note, however, that it is understood that there are still some deficiencies in error detection and recovery. This is a subject of further work.]

[The state tables currently assume that a single transactionId is used throughput an entire Command Sequence rather than on a per request/response and indication/confirm pairing basis as agreed to in Boston. This will be updated in the future.]

The purpose of the state tables is to provide a more precise description of the U-N messages protocols. They describe the state of a DSM-CC Session: the events that occur in the protocols, the actions taken, timer management, and the resultant state. It is not the intent of these tables to rigorously define each condition met and action performed, but instead provide a framework for understanding the control and operation of the DSM-CC U-N protocol and help insure that the protocol is complete. Specific actions such as resource allocation are described only in general terms by the tables and referenced to the appropriate sections of this Recommendation for a full description.

There are three interacting state tables, one for each Client, Server and Network portion of a session. Each session runs an instance of the state tables with their own state variables and timers. These instances are independent of other sessions which may be running concurrently with them.

It is assumed that each Client, Network and Server entity has an entity-wide state machine which controls the operation of that entity and defines when the entity is available for the creation and operation of sessions, and when it is unavailable for such activities. For instance, an entity that is in the middle of a U-N config may not be in the proper state to accept new sessions until the configuration step is complete. It is outside the scope of this section to describe these entity-wide state machines and assumes that the Client, Network, or Server entity are in a state which allows for session creation and operation.

[Table numbering “X-n” needs to be updated, and cross referenced with corresponding table captions.]

The state tables are depicted in Tables X-6 (Client), X-12 (Network), and X-18 (Server). Each state table entry consists of a:

- Current State Field
- Incoming Event Field
- Extra Conditions Field
- Actions Performed Field
- Timer Management (Start Timers Field, Stop Timers Field)
- Next state Field

The states are represented in their abbreviated names as described in Tables X-2, X-8, and X-14. If more than one state is present in the Current State field, then the entry pertains to any of the states specified.

Incoming events are represented in their abbreviated name as described in Tables X-1, X-7, and X-13. Internal events, such as a request initiated by a entity user, or a timer expiring will be enclosed in square brackets, [ ]. If more than one event is present in the Incoming Event field, then the entry pertains to any of the Incoming events specified.

Conditions, if present, will be a sequence of predicates which may be joined by logical operators. In keeping with ISO 13818-1, symbols similar to the C programming language will be used:

&&	Logical AND.
||	Logical OR.
!	Logical NOT.

The conditions will be represented by abbreviated terms as described in Tables X-3, X-9, and X-15.

Unless otherwise specified, it is always assumed that the incoming event will be checked and validated as follows:

- If it is a message, that the message syntax is verified against the definitions in section 2.3.1.
- If the message has a transactionId, that it is valid and represents an active request.
- If the message has a sessionid, that it is valid and represents a known session.
- If the message has a response field, that its value is in accordance with the expected value in section 2.3.
- If the message has a reason field, that its value is in accordance with the expected value in section 2.3.

Actions are represented by abbreviated terms as described in Tables X-4, X-10, and X-16. There may be several actions performed per incoming event. Action entries may also include a reference to a section which better describes the specifics of actions to be performed. These references will be in parenthesis, ( ).

Timers are represented by their abbreviated names as described in Tables X-6, X-11, and X-17. If more than one entry is present in the Start Timers field, then all timers specified should be started. If more than one entry is present in the Stop Timers field, then any of the timers defined which are active should be stopped. A Reset of a timer value is represented by that timer being present in both the Stop Timers and Start Timers fields of an entry.

Table 84. Session Incoming Events

Event Name	Description
[initiate-ses]	Client-initiated request for a session.
[initiate-cf-ses]	Client-initiated request for attachment to a continuous feed session.
[initiate-rel]	Client-initiated request for session release.
[timer expire]	A timer expired.
[initiate-svr-stat]	Network initiates a ServerStatusRequest.
[initiate-cli-stat]	Network initiates a ClientStatusRequest.
[initiate-svr-id-req]	Network initiates a ServerIdentifySessionsRequest.
[initiate-cli-id-req]	Network initiates a ClientIdentifySessionsRequest.
[initiate-add-res]	Server-initiated request to add resources to an existing session.
[initiate-del-res]	Server-initiated request to delete resources from an existing session.
CliSesInd	Received a ClientSessionIndication from the Network.
CliSesProceeding	Received a ClientSessionProceeding from the Network.
CliSesCnf	Received a ClientSessionConfirm from the Network.
CliStatInd	Received a ClientStatusIndication from the Network.
CliIdSesInd	Received a ClientIdentifySessions Indication from the

	Network.
CliRelInd	Received a ClientReleaseIndication from the Network.
CliDelResInd	Received a ClientDeleteResourceIndication from the Network.
CliAddResInd	Received a ClientAddResoureIndication from the Network.
CliRelCnf	Received a ClientReleaseConfirm from the Network.
CliSesReq	Received a ClientSessionRequest message.
CliSesRsp	Received a ClientSessionResponse message.
CliRelReq	Received a ClientReleaseRequest message.
CliConnectReq	Received a ClientConnectRequest message.
CliStatRsp	Received a ClientStatusResponse message.
CliIdSesRsp	Received a ClientIdentifySessionsResponse message.
CliRelRsp	Received a ClientReleaseResponse message.
CliSesDelResRsp	Received a ClientDeleteResourceResponse message.
SvrSesReq	Received a ServerSessionRequest message.
SvrCFSesReq	Received a ServerContinuousFeedSessionRequest message.
SvrDelResReq	Received a ServerDeleteResourceRequest message.
SvrRelReq	Received a ServerReleaseRequest message.
SvrSesRsp	Received a ServerSessionResponse message.
SvrRelRsp	Received a ServerReleaseResponse message.
SvrAddResReq	Received a ServerAddResourceRequest message.
SvrStatRsp	Received a ServerStatusResponse message.
SvrIdSesRsp	Received a ServerIdentifySessionsResponse message.
SvrSesInd	Received a ServerSessionIndication message.
SvrStatInd	Received a ServerStatusIndication message.
SvrIdSesInd	Received a ServerIdentifySessionsIndication message.
SvrRelInd	Received a ServerReleaseIndication message.
SvrConnectInd	Received a ServerConnectIndication message.
SvrSesCnf	Received a ServerSessionConfirm message.
SvrDelResCnf	Received a ServerDeleteResourceConfirm message.
SvrRelCnf	Received a ServerReleaseConfirm message.
SvrAddResCnf	Received a ServerAddResourceConfirm message.

Table 85. Client Session States

State Name	**Description**
CSIdle	Client is Idle. Ready to receive/initiate sessions or provide status information to the Network.
WFCSCnf	Client has initiated a session request and is waiting for a reply.
CSActive	A client session has been created and is active.

WFCRelCnf	Client has initiated a release of the active session and is waiting for a reply.

Table 86. Network Session States

State Name	Description
NSIdle	Network Session is Idle. Ready to receive session requests or pass-through messages.
NSActive	The Network sessionId is active
WFSesRsp	The Network is waiting for a ServerSessionResponse during session establishment.
WFCSRelRsp	The Network is waiting for a ClientReleaseResponse and a ServerReleaseResponse during network-initiated session tear-down.
WFSRelRsp1	The Network is waiting for a ServerReleaseResponse. The Client will not need to be informed of the session release.
WFSRelRsp2	The Network is waiting for a ServerReleaseResponse. The Client will need to be informed of the session release.
WFCSRsp	The Network is waiting for a ClientSessionResponse during session establishment.
WFCAddResRsp	The Network is waiting for a ClientAddResourceResponse.
WFCRelRsp1	The Network is waiting for a ClientReleaseResponse. The Server does not need to be informed of the session release.
WFCRelRsp2	The Network is waiting for a ClientReleaseResponse. The Server still needs to be informed of the session release.
WFCRelRsp3	The Network is waiting for ClientReleaseResponses from all the Clients attached to a torn-down continuous feed session.
WFCDelResRsp	The Network is waiting for a ClientDeleteResourceResponse.
WFSStatRsp	The Network is waiting for a ServerStatusResponse.
WFCStatRsp	The Network is waiting for a ClientStatusResponse.
WFSIdSesRsp	The Network is waiting for a ServerIdentifySessionsResponse.
WFCIdSesRsp	The Network is waiting for a ClientIdentifySessionsResponse.

Table 87. Server Session States

State Name	Description
SSIdle	Server sessionId is idle.
SSActive	Server sessionId identifies an active session.
WFSSCnf	Server is waiting for a ServerSessionConfirm message during session establishment.
WFSDelResCnf	Server is waiting for a ServerDeleteResourceConfirm message.
WFSRelCnf	Server is waiting for a ServerReleaseConfirm during session release.
WFSAddResCnf	Server is waiting for a ServerAddResourceConfirm message

Table 88. State Table Conditions

Abbreviation	Description
resources	Resources in resource list are valid and can be allocated to the session.

acceptable	
user data	Client has user data to send
valid sessionId	The sessionId in the received message is known to the Client.
valid transactionId	The transactionId field in the received message is known to the Client
valid clientId	The clientId is known to the Server and has the correct privileges.
valid serverId	The serverId is the Server’s
support new session	The Server is capable of supporting a new session.
CF session	The session in question is a continuous feed session.
reject response field	Response field of the received message indicates that the corresponding request was rejected by the Network or the Server.
rsnXXXX	Set the reason field in the message being sent to the value rsnXXXX.
rspXXXX	Set the response field in the message being sent to the value rspXXXX.
valid resources	ResourceIds in the resource list are valid and can be provisioned according to the conventions in the protocol for the U-N primitive sequence being executed.
last CF client	there is only one Client attached to the continuous feed session in question.
Clients	The continuous feed session has at least one Client.

Table 89. Session Actions

Action Abbreviation	Description
CliSesReq	Send ClientSessionRequest message to the Network.
CliSesRsp	Send a ClientSessionResponse message to the Network.
CliConnectReq	Send a ClientConnectRequest message to the Network.
CliStatRsp	Send a ClientStatusResponse message to the Network.
CliIdSesRsp	Send a ClientIdentifySessionsResponse message to the Network.
CliRelReq	Send a ClientReleaseRequest message to the Network.
CliRelRsp	Send a ClientReleaseResponse message to the Network.
CliDelResRsp	Send a Client Session Delete Resources Response message to the Network.
CliAddResRsp	Send a ClientAddResourcesResponse message to the Network.
allocate transactionId	Select a unique transactionId.
free transactionId	Free the transactionId. transactionId values should be “frozen” for a period of time so that there are no collisions of packets with the same transactionIds.
continuous feed	Set up UserData field to indicate this is a continuous feed session
reject response	Set the value of the response field in the message being sent to a value which indicates that the corresponding request message was rejected.
provision	Provision Client to use the resources as requested in previous

resources	primitives.
free sessionId if enabled	Invalidate sessionId for subsequent messages for some period of time. This may be done by the client, network or server depending on which has been configured to do so.
free resources	Client will discontinue use of indicated resources.
rsnXXXX	Set the reason field in the message being sent to the value rsnXXXX.
rspXXXX	Set the response field in the message being sent to the value rspXXXX.
SvrSesReq	Send a ServerSessionRequest message.
SvrIdSesRsp	Send a ServerIdentifySessionsResponse message.
SvrStatRsp	Send a ServerStatusResponse message
SvrRelReq	Send a ServerReleaseRequest message.
SvrResRsp	Send a ServerResourceResponse message.
SvrRelRsp	Send a ServerReleaseResponse message.
SvrDelResReq	Send a ServerDeleteResourceRequest message.
SvrAddResReq	Send a ServerAddResourceRequest message.
allocate sessionId if enabled	Allocate a unique sessionId value for the created session. This may be done by the client, network or server depending on which is configured to do so.
provision Server res.	Provision resources for the Server part of the session.
provision Client res.	Provision resources for the Client part of the session
provision resources	Provision resources as requested in the current U-N primitive sequence.
free Server resources	Free resources used by the Server part of the session.
free Client resources	Free resources used by the Client part of the session.

4.13.1 Client-Related State Tables

Table 90. Client Initiated Session Set-Up Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSIdle	[initiate-ses]		allocate transactionId CliSesReq		tCSesCnf	WFCSCnf
	CliSesProceeding CliSesCnf	unknown transactionId	CliRelReq rsnClUnkTransID			CSIdle
WFCSCnf	CliSesProceeding			tCSesCnf	tCSesCnf	WFCSCnf
	[tCSesCnf]		free transactionId			CSIdle

	CliSesCnf	!resources acceptable	CliRelReq rsnClNoResource free transactionId	tCSesCnf		CSIdle
		reject response field	free transactionId	tCSesCnf		CSIdle
		resources acceptable && !user data	free transactionId	tCSesCnf		CSActive
		resources acceptable && user data	CliConnectReq free transactionId	tCSesCnf		CSActive
CSActive	CliSesProceeding					

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSIdle	CliSesReq	valid clientId && valid serverId && support new session	allocate sessionId if enabled SvrSesInd		tNAddResReq tNSesPcd	WFSAddResReq
		!valid clientId || !serverId || !support new session	CliSesCnf			NSIdle
WFSSesRsp	SvrSesRsp	rspResourceCompleted	provision resources CliSesCnf	tNSSesRsp		NSActive
		!rspResourceCompleted	CliSesCnf	tNSSesRsp		NSIdle
	[tNSSesRsp]		free sessionId if enabled CliSesCnf rspTimeExpire			NSIdle
WFSAddResReq	[tNSesPcd]		CliSesProceeding		tNSesPcd	WFSAddResReq
	[tNAddResReq]		free sessionId if enabled	tNSesPcd		NSIdle
	SvrSesRsp		CliSesCnf rspOK	tNSesPcd tNAddResReq		NSActive
		rspSeNoCalls	CliSesCnf rspSeNoCalls	tNSesPcd tNAddResReq		
	SvrAddResReq		SvrAddResCnf allocate Server res.	tNSesPcd	tNSSesRsp	WFSSesRsp
		!valid resources	SvrAddResCnf rsnNeNoResource	tNSesPcd	tNSSesRsp	WFSSesRsp
		reject response field	CliSesCnf free sessionId	tNSesPcd tNAddResReq		NSIdle
NSActive	CliConReq		SvrConInd			NSActive

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSIdle	SvrSesInd	valid clientId && valid serverId && support new session && resources to provision || CF Session	SvrAddResReq		tSAddResCnf	WFSAddResCnf2
		valid clientId && valid serverId && support new session && !resources to provision	SvrSesRsp rspOK			SSActive
		!valid clientId || !valid serverId || !support new session	SvrSesRsp rspSeNoCalls			SSIdle
WFSAddResCnf2	SvrAddResCnf		SvrSesRsp provision resources	tSAddResCnf		SSActive
		!valid resources	SvrSesRsp reject response field	tSAddResCnf		SSIdle
		rspNeNoResource	SvrSesRsp rspSeNoResource	tSAddResCnf		SSIdle
	[tSAddResCnf]			tSAddResCnf		SSIdle
SSActive						

Table 91. Client Connection to a Continuous Feed Session Command Sequence

Same as Client SessionSetup Command Sequence.

Table 92. Client Initiated Session Tear-Down Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSActive	[initiate-rel]		CliRelReq allocate transactionId free resources		tCRelCnf	WFCRelCnf
WFCRelCnf	CliRelCnf		free sessionId	tCRelCnf		CSIdle
	[tClRelCnf]		free sessionId			CSIdle
	CliRelInd	rsnUnknownSessionId	free sessionId Audit procedures	tCRelCnf		CSIdle
CSIdle						

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSActive	CliRelReq		SvrRelInd free Client resources		tNSRelRsp	WFSRelRsp1
WFSRelRsp1	SvrRelRsp		free sessionId if enabled free resources CliRelCnf	tNSRelRsp		NSIdle
		rspUnknownSessionId	free sessionId if enabled free Server resources CliRelCnf	tNSrelRsp		NSIdle
	[tNSRelRsp]		free sessionId if enabled free Server resources CliRelCnf			NSIdle
NSIdle	CliRelReq	invalid sessionId	CliRelInd rsnUnknownSessionId			NSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSActive	SvrRelInd		free resource SvrRelRsp			SSIdle
SSIdle	SvrRelInd	unknown sessionId	SvrRelRsp rspUnknownSessionId			SSIdle

Table 93. Client Continuous Feed Session Tear-Down Command Sequence

Same as Client Session Tear-Down Command Sequence.

4.13.2 Server- Related State Tables

Table 94. Server Initiated Session Set-Up Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSIdle	CliSesInd	!resources acceptable	CliSesRsp reject response			CSIdle
		valid clientId && valid sessionId && resources acceptable	CliSesRsp			CSActive

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State

NSIdle	SvrSesReq	valid serverId && valid clientId && support new session	allocate sessionId if enabled CliSesInd		tNCSesRsp	WFCSesRsp
		!valid serverId || !valid clientId || !support new session	SvrSesCnf reject response field			NSIdle
WFCSesRsp	CliSesRsp		SvrSesCnf	tNCSesRsp		NSActive
		reject response field	SvrSesCnf reject response field free sessionId if enabled	tNCSesRsp		NSIdle
	[tNSCnReq]		free sessionId if enabled CliRelInd SvrRelInd			NSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSIdle	[initiate-ses]		allocate transactionId allocate sessionId if enabled SvrSesReq		tSSesCnf	WFSSesCnf
WFSSesCnf	SvrSesCnf			tSSesCnf		SSActive
		!valid resources || reject response field	free sessionId if enabled free transactionId	tSSesCnf		SSIdle
	[tSSesCnf]		free sessionId if enabled free transactionId			SSIdle
SSActive						

Table 95. Server Continuous Feed Session Set-Up Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSIdle	SvrCFSesReq	valid serverId && valid clientId && valid resources&& support CF session	allocate sessionId if enabled SvrSesCnf			NSActive

		!valid serverId || !valid clientId || !valid resources || !support CF session	SvrSesCnf reject response field			NSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSIdle	[initiate-cf-ses]		allocate transactionId allocate sessionId if enabled SvrCFSesReq		tSCFSesCnf	WFCFSesCnf
WFCFSesCnf	SvrSesCnf			tSCFSesCnf		SSActive
		!valid resources || reject response field	free transactionId free sessionId if enabled	tSCFSesCnf		SSIdle
	[tSCFSesCnf]		free transactionId			SSIdle
SSActive						

Table 96. Server Initiated Add Resource Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSActive	CliAddResInd	resources acceptable	CliAddResRsp			CSActive
		!resources acceptable	CliAddResRsp rspCLNoResource			CSActive

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSActive	SvrAddResReq		CliAddResInd		tNAddResRsp	WFSPrvResReq
		!valid resources	SvrAddResCnf rspNeNoResource			NSActive
WFCAddResRsp	CliAddResRsp		SvrAddResCnf provision resources	tNAddResRsp		NSActive
		rspClNoResources	SvrAddResCnf free new resources	tNAddResRsp		NSActive
	[tNAddResRsp]		free transactionId !provision resources			NSActive
NSIdle						

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State

SSActive	[initiate-add-res]		allocated transactionId SvrAddResReq		tSAddResCnf	WFSAddResCnf2
WFSAddResCnf2	SvrAddResCnf		provision resources	tSAddResCnf		SSActive
		rspNeNoResource	!provision resources	tSAddResCnf		SSActive
	[tSAddResCnf]		!provision resources free transactionId			SSActive

Table 97. Server Session Delete Resources Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSActive	CliDelResInd	valid resources	CliDelResRsp free resources			CSActive
		!valid resources	CliDelResRsp rspClRejResource			CSActive

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSActive	SvrDelResReq		CliDelResInd		tNDelResRsp	WFCDelResRsp
		!valid resource	SvrDelResCnf rspNeNoResource			NSActive
WFCDelResRsp	CliDelResRsp		free transactionId SvrDelResCnf free resources	tNDelResRsp		NSActive
		rspClNoResource	free transactionId SvrDelResCnf rspClNoResource !free resources	tNDelResRsp		NSActive
	[tNDelResRsp]		free transactionId SvrDelResCnf rspTimeExpire !free resources			NSActive
NSIdle						

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSActive	[initiate-del-res]		allocate transactionId SvrDelResReq free resources		tSDelResCn	WFSDelResCnf

WFSDelResCnf	SvrDelResCnf		free transactionId	tSDelResCnf		SSActive
		rspNeNoResource || rspClNoResource	!free resources free transactionid	tSDelResCnf		SSActive
	[tSDelResCnf]		!free resources free transactionId			SSActive
SSIdle						

Table 98. Server Iniatiated Session Tear-Down Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSActive	CliRelInd		free resources CliRelRsp			CSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSActive	SvrRelReq		CliRelInd		tNCliRel	WFCRelRsp2
		!valid serverId	SvrProcErr			NDiagnostic
WFCRelRsp2	CliRelRsp CliProcErr		SvrRelCnf free all resources	tNCliRel		NSIdle
	[tNCliRel]		free all resource			NSIdle
NSIdle						

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSActive	[initiate-ses-rel]		allocate transactionId SvrRelReq		tSSRelCnf	WFSRelCnf
WFSRelCnf	SvrRelCnf		free resources free transactionId free sessionId if enabled	tSSRelCnf		SSIdle
	[tSSesRel]		free resources free transactionId free sessionId if enabled	tSSRelCnf		SSIdle
SSIdle						

Table 99. Server Continuous Feed Session Tear-Down Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSActive	CliRelInd		free resources CliRelRsp			CSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSActive	SvrRelReq		CliRelInd		tNCliRelRsp	WFCRelRsp3
		CF session	CliRelInd all Clients SvrRelCnf free resources		tNCFRelRsp	WFCRelRsp3
		CF session && !Clients	free resources SvrRelCnf			NSIdle
WFCRelRsp3	CliRelRsp	!last CF Client		tNCliRelRsp	tNCliRelRsp	WFCRelRsp3
		last CF Client	free sessionId	tNCliRelRsp		NSIdle
	[tNCliRel]		free sessionId			NSIdle
WFCSRelRspCF	[tNCFRel]		free sessionId free resources			NSIdle
	CliRelRsp	last Client	free Client resources	tNCFRelRsp	tNSvrRelRsp	WFSRelRsp1
		!last Client				WFCSRelRspCF
	SvrRelRsp		free Server resources	tNCFRelRsp	tNCliRelRsp	WFCRelRsp3
	CliProcErr	last Client	free Client resources	tNCFRelRsp	tNSvrRelRsp	WFSRelRsp1
		!last Client				WFCSRelRspCF
	SvrProcErr		free Server resources	tNCFRelRsp	tNCliRelRsp	WFCRelRsp3
NSIdle						

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSActive	[initiate-cf-ses-rel]		allocate transactionId SvrRelReq		tSCFRelCnf	WFSRelCnf

WFSRelCnf	SvrRelCnf		free resources free sessionId if enabled free transactionId	tSCFRelCnf		SSIdle
	[tSSesRel]		free resources free sessionId if enabled free transactionId	tSCFRelCnf		SSIdle
SSIdle						

4.13.3 Network-Related State Tables

Table 100. Network Initiated Session Tear-Down Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSActive	CliRelInd		free resources CliRelRsp			CSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSActive	[initiate-rel]		allocate transactionId CliRelInd SvrRelInd		tNetRel	WFCSRelRsp
WFCSRelRsp	CliRelRsp		free Client resources	tNetRel	tNSvrRelRsp	WFSRelRsp1
	SvrRelRsp		free Server resources	tNetRel	tNCliRelRsp	WFCRelRsp1
	[tNetRel]		free resources free sessionId if enabled free transactionId			NSIdle
WFSRelRsp1	SvrRelRsp		free sessionId if enabled free Server resources free transactionId	tNSvrRelRsp		NSIdle
	[tNSvrRelRsp]		free sessionId if enabled free Server resources free transactionId			NSIdle
WFCRelRsp1	CliRelRsp		free sessionId if enabled free Client resources free transactionId	tNCliRelRsp		NSIdle

	[tNCliRelRsp]		free sessionId if enabled free Client resources free transactionId			NSIdle
NSIdle						

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSActive	SvrRelInd		free resources SvrRelRsp			SSIdle
SSIdle						

Table 101. Network Initiated Continuous Feed Session Tear-Down Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSActive	CliRelInd		free resources CliRelRsp			CSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSActive	[inititate-rel]	CF session	allocate transactionId CliRelInd all Clients SvrRelInd		tNetRel	WFCSRelRspCF
		CF session && !Clients	allocated transactionId SvrRelInd		tNSvrRelRsp	WFSRelRsp1
WFCSRelRspCF	CliRelRsp	last Client	free Client's resources		tNSvrRelRsp	WFSRelRsp1
		!last Client	free Client's resources	tNetRel	tNetRel	WFCSRelRspCF
	SvrRelRsp		free Server resources	tNetRel	tNCliRelRsp	WFCRelRsp3
	[tNRelCF]		free sessionId if enabled free resources free transactionId			NSIdle
WFSRelRsp1	SvrRelRsp		free sessionId free Server resources free transactionId	tNSvrRelRsp		NSIdle
	[tNSvrRelRsp]		free sessionId free Server resources free transactionId			NSIdle
WFCRelRsp3	CliRelRsp	!last CF Client	free Client's resources	tNCliRelRsp	tNCliRelRsp	WFCRelRsp3

		last CF Client	free Client's resources free sessionId if enabled free transactionId	tNCliRelRsp		NSIdle
	[tNCliRel]		free resources free sessionId if enabled free transactionId			NSIdle
NSIdle						

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSActive	SvrRelInd		free resource SvrRelRsp			SSIdle
SSIdle						

Table 102. Network Initiated Client Session List Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSIdle	CliIdSesInd		CliIdSesRsp			CSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSIdle	SvrIdSesRsp CliIdSesRsp SvrStatRsp	!valid transactionId				NSIdle
	[initiate-cli-id-ses]		allocate transactionId CliIdSesInd		tNCIdSes	WFCIdSesRsp
WFCIdSesRsp	CliIdSesRsp		free transactionId	tNCIdSes		NSIdle
	[tNCIdSes]		free transactionId			NSIdle

Table 103. Network Initiated Server Session List Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSIdle	SvrIdSesRsp CliIdSesRsp	!valid transactionId				NSIdle
	[initiate-svr-id-ses]		allocate transactionId SvrIdSesInd		tNSIdSes	WFSIdSesRsp
WFSIdSesRsp	SvrIdSesRsp		free transactionId	tNSIdSes		NSIdle
	[tNSIdSes]		free transactionId			NSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSIdle	SvrIdSesInd		SvrIdSesRsp			SSIdle

Table 104. Network Initiated Audit of Client Session Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
CSIdle	CliStatInd		CliStatRsp			CSIdle

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSIdle	[initiate-cli-stat]		allocate transactionId CliStatInd		tNCStat	WFCStatRsp
WFCStatRsp	CliStatRsp		free transactionId	tNCStat		NSIdle
	[tNCStat]		free transactionId			NSIdle

Table 105. Network Initiated Audit of Server Session Command Sequence

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
NSIdle	[initiate-svr-stat]		allocate transactionId SvrStatInd		tNSStat	WFSStatRsp
WFSStatRsp	SvrStatRsp		free transactionId	tNSStat		NSIdle
	[tNSStat]		free transactionId			NSIdle
NDiagnostic						

Current State	Event	Conditions	Actions	Stop Timers	Start Timers	Next State
SSIdle	SvrSesInd	valid clientId && valid serverId && support new session	SvrSesRsp (2.3.5.1.1)		tSSesInd	WFSResInd
	SvrStatInd	!valid sessionId	SvrStatRsp			SSIdle
SDiagnostic	SvrStatInd		SvrStatRsp			SDiagnostic

5. User-to-User Interface

5.1 Introduction

This section of the ISO/IEC 13818-6 standard defines a framework and basic set of interfaces which support the life cycles of MPEG multimedia applications in a heterogeneous network environment. In order to enable the installation and usage of multimedia applications, a minimal set of basic access primitives are deemed necessary. These include interfaces for fundamental service types: directories, streams, files and data objects. The User-to-User primitives are concerned with ISO layer 7 application-specific interfaces only. Informative annexes will describe RPC and other ISO layer 6-1 implementations.

The primitives described herein are considered to define key interfaces for interoperability between clients and services, and key interfaces for portability of client applications. It is a goal of this framework and associated user-to-user primitives to enable information providers to load content into services, and to enable client applications to retrieve that content, in a way that is fully interoperable.

5.1.1 Scope

This section specifies the following:

- User-to-User System Environment
- DSM Library Common Definitions
- Application Runtime Procedures
- The Core Client-Service Interfaces
- The Extended Client-Service Interfaces
- The Application Portability Interfaces

5.1.2 Requirements

The DSM User-to-User primitives set is viewed as enabling a wide-range of multimedia applications to run using the MPEG delivery system in heterogeneous environments. As such, it is within the scope of the ISO/IEC 13818-6 charter. They are a minimum set of primitives for efficient operation over networks that may have a long latency between client and service, limited client storage and limited network request path bandwidth. The User-to-User primitives provide a consistent, unified interface for commonly-used multimedia types, and for the usage of a system whereby services may be registered, browsed, activated and accessed.

Application requirements drive the need for the DSM interfaces. Applications that expect to run in the DSM system environment include:

- Movies On Demand
- Movie Listing
- Tele-shopping
- Near Movies On Demand
- News on Demand
- Karaoke On Demand
- Games
- Tele-medicine
- Distance Learning,
- and others

Use of DSM by MHEG requires access to multimedia data objects, particularly remote access of streams, files and composite objects. Composite objects must be accessed in such a way that certain sub-objects will return a remote reference and other sub-objects will return data.

Interoperability in a heterogeneous environment requires that service and asset brokers be implemented. Key functions of these brokers are a) directories where objects can be browsed and requested, b) authentication of client and authorization of end-user, and finally c) a translation of the request into one of many possible heterogeneous service or asset instances.

It is a requirement that this system framework and interface support a settop client, where there is typically limited memory and no disk. While the interfaces are simplified to accomplish this, they may also be extended to support clients with more capabilities and resources. These primitives are defined in a way to allow the settop client to act as a stub to brokers in the service gateway and services. When it wishes access to an object, the stub asks for it by name (as in Open or Get), and the broker, which is not in the settop client, resolves which instance is to be used and returns the corresponding object reference or data to the client. The minimal client using these primitives will not be a name service, perform authentication or authorization, or choose which instance of service or asset to connect to. These functions will be performed by service brokers and asset brokers on the server side of the network.

The interfaces must allow for loading of services and content into the system as well as access by end-users.

The primitives must provide secure access based ownership, administrative, broker, writer and reader privileges.

Password and encryption access control must be implemented to prevent unauthorized updates or access.

The framework must allow for extensibility, where existing interfaces can be augmented and new interfaces added.

Functions will be specified in OMG IDL (ISO/IEC 14750) to permit use in multiple RPC and language environments, including, but not limited to UNO, ONC, DCE, C++ and C. Moreover, it is desirable that this specification be extensible so as to be operable in a CORBA system environment.

It is desirable that this specification be used in the C programming environment. As such, primitives will also be specified with normative C syntax, which is the result of C source generation from the corresponding IDL. Rules for translation from IDL to C will be specified or referenced.

The interfaces must address the network latency issue (client-server), by supporting synchronous deferred pipelining of requests. This will enable applications to prefetch information and prepare services in advance of the time needed to present results to the end-user. This will also assure that access is not blocked unnecessarily due to another outstanding request, by permitting multiple parallel requests to potentially multiple separate heterogeneous destinations.

It is required that the interfaces specified herein be consistent and simple, while not sacrificing the overriding need to address network latency. The interfaces may package other standard semantics and syntax (e.g., OMG IDL and SQL statement) in such a way as to provide a unified interface for MPEG digital storage operations. For example, a SQL statement may be packaged in an synchronous deferred, pipelined primitive.

5.1.3 Typographical Conventions

The type styles shown below are used in this section to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings, where no distinction is necessary, nor are the type styles used in text where their density would be distracting.

Helvetica	OMG IDL, CORBA language and syntax elements
Times New Roman bold	DSM specification is encapsulated in module DSM { ..};
`Courier`	C language elements

5.2 System Environment

There are multiple user entities of the distributed system encompassed by this standard. The user entities are seen as connecting to an MPEG network as defined in the User-to-Network section of ISO/IEC 13818-6, or connected by private network which is used for communication among server and service entities. These entities are listed under physical and logical categories as follows:

5.2.1 System Hardware User Entities

- Servers that store and deliver MPEG data and other information. Servers include a heterogeneous variety of processors and storage systems. The server has operating system software that supports the operation of services and applications that run on it and isolates these from underlying hardware devices and network protocols.

- Hardware clients that run applications and provide the underlying capability of decoding and displaying ISO/IEC 13818 streams. Hardware clients include a heterogeneous variety of settop boxes, PCs, etc. The hardware client has operating system software that supports the operation of applications that run on it and isolates these from underlying hardware devices and network protocols.

5.2.2 System Logical Entities

- End-user applications that control the viewing of digitally stored media and response to human user input.
- DSM library which mediates between the end-user application and the remote service interface, providing a simplified function call interface to the end-user application, and an RPC interface to the remote service.
- Services which contain application objects such as directories, streams and other data. A Service is a logical entity in the system that provides function(s) and interface(s) in support of one or more applications. The distinction of a service from other objects is that end-user access to it is controlled by a Service Gateway. Services may be located and distributed in any manner on heterogeneous server hardware platforms.
- A Service Gateway which provides an interface for browsing and discovering services, authentication and authorization of end-users, registration of services and end-users, and resolution of the connection between clients and services. The Service Gateway is a specialized form of service.
- Directory objects and associated interface, which may be configured into a service, providing name space and browsing of objects to applications.
- Stream objects and associated interface, which may be configured into a service, providing ISO/IEC 13818 continuous media streams to clients at their request.
- File objects and associated interface, which may be configured into a service, providing data storage and retrieval to applications.
- View objects and associated interface, which may be configured into a service, providing directory sort and filter operations and relational data access to applications.
- Any Custom object that is application-specific in function and interface, that invokes operations over and above what is specified by the DSM primitives. Custom objects are specified in IDL and configured into a service. The service with augmented interfaces is defined and registered with the Service Gateway.

The user entities are replaceable components that have standard interfaces between them. This standard defines the interfaces between these components, as illustrated in

Figure 26.

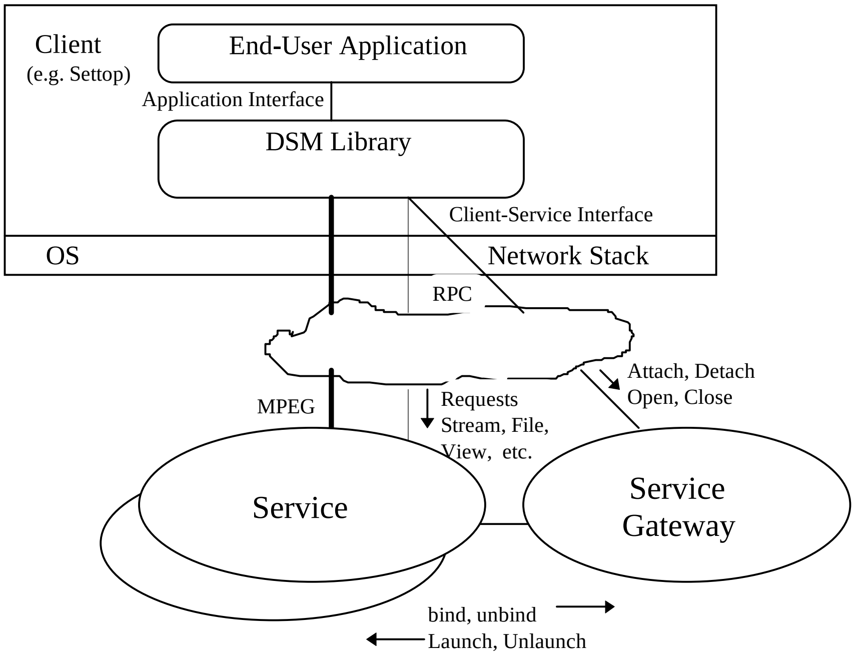


Figure 26: User-to-User System Environment

The component entities shown in Figure 1 are logical components; They may exist on a single machine that supports all the components, or they may be distributed across a set of special purpose machines. For example, requests for an individual media stream may be distributed across several machines in order to balance resource requirements even though a client sees a single stream reference.

The Service Gateway presents to the client a graph of service names and information, arranged as a hierarchy of directories and services. The top root of the Service Gateway has a default service, for clients requesting a session without specifying a service. There may be one or more applications indicated by the graph. There is one top service for each application, representing the name of the application. In addition to the entry service, an application may also utilize a variety of other services, e.g., stream services or order fulfillment services, which may or may not be shared with other applications. For shared services, the hierarchy will converge at those points.

For security reasons, only a Service Gateway may maintain a name graph with service objects. The manager of the service will register each service and its interfaces with the Service Gateway.

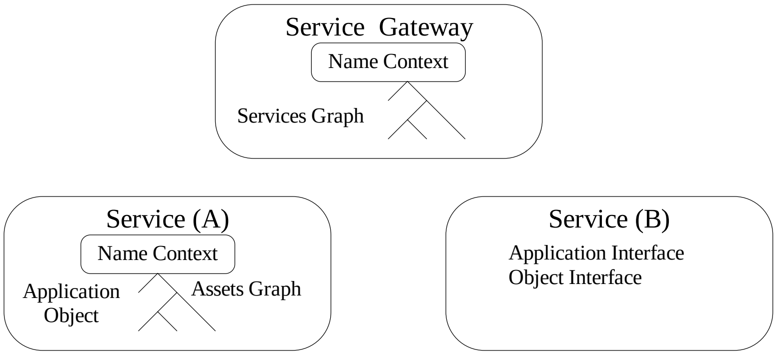


Figure 27: ServiceGateway and Service Name Spaces

A Service can define its interfaces by use of the Interfaces Define() operation. In this way it can declare whether it includes well-known interfaces such as DSM Directory, Stream, File and View, or whether it includes other interfaces which are known only to a limited set of applications. A service can therefore declare a Directory interface and as such will offer all of the Directory name context functionality. On the other hand, it could declare specific application interfaces without offering itself as a name context, i.e., a Service is not necessarily a Directory.

In DSM, the Name Context for objects is called a Directory. This is not to be confused with a traditional operating system Directory of files. It in no way implies a physical organization of objects, but rather presents to the client the appearance of scoped name spaces. DSM does not preclude an implementation that constructs Directories in the traditional manner, while at the same time supports implementations in which the Directory is a container of objects, and provides Name Server-like browsing functions by which a client can ‘discover’ objects. Within a service, there may be a directory hierarchy composed of a root Name Context at the top and a graph of sub directories, each serving to scope its own name space. The graphs may intersect in the implementation, enabling the sharing of objects.

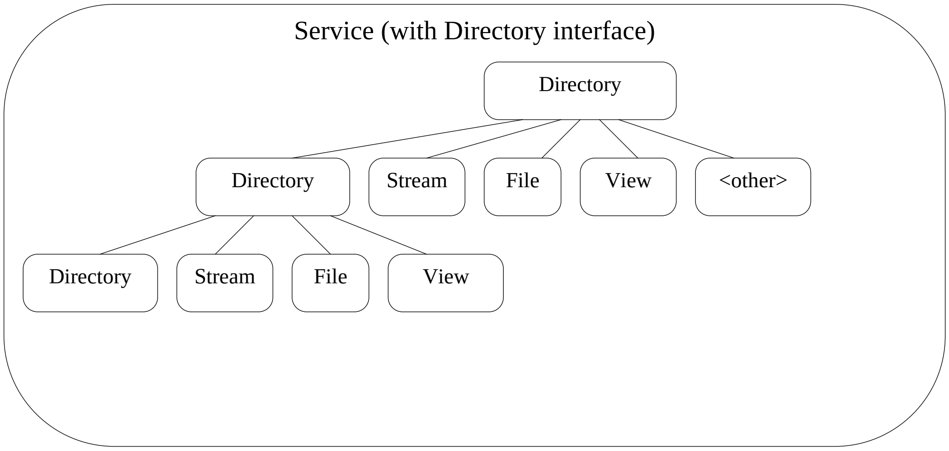


Figure 28: Possible Directory Locations of Service Components

5.2.3 Application and Service Interfaces

The Application Portablity Interface is the interface between an application and the greater client Operating System, which includes the DSM Library, the processor operating system and the communications transport stack. The goal of the DSM User-to-User interface is to provide applications a portable means of accessing the remote service (e.g. Directory, Stream, File and View). In some cases the interfaces supplied to the application by the client operating system are more abstract than the raw capability of the remote servers. In other words there is not necessarily a 1-1 mapping between the remote operations supported by the services/components and the application interfaces. This allows the client operating system to hide hardware and network dependencies and it allows the application to take advantage of information that is only known locally. Therefore, the standard defines two interfaces for service available to a client application; the interface seen by the application and the interface supported by the server, as shown in Figure 29.

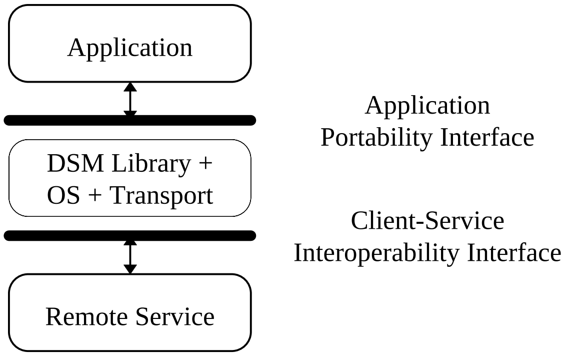


Figure 29: Application and Service Interfaces

This standard defines the following logical interfaces between the entities of the system:

Core Interfaces -

The Core interfaces represent the minimum requirement for a DSM-CC Service Complex.

- **Base** Interface.

⇒ This interface provides commonly used operations Close, Destroy and IsA. It is an abstract interface, meaning it is included (inherited) by other interfaces.

- **Access** Interface.

⇒ This interface provides commonly used attributes for size, history (version and date), lock status and permissions. It is also an abstract interface, included (inherited) by other interfaces.

- **Event** Interface.

⇒ This interface provides operations by which a client can subscribe/unsubscribe to asynchronous events, whereby an object can send events to the client over the MPEG stream.

- **Directory** Interface.

⇒ This interface provides a CORBA name service interface plus operations to access objects and object data through depth and breadth-first path traversal.

- **Stream** Interface.

⇒ This interface enables a client to interactively control MPEG continuous media streams.

- **File** Interface.

⇒ This interface enables a client to access data within an object which is a sequence of bytes.

- **ServiceGateway** Interface.

⇒ This interface provides a directory of services and enables a client to attach to a service environment and open access to services.

Extended Interfaces

The extended interfaces are optional:

- **View** Interface.

⇒ The interface enables a client to sort and filter objects by their attributes using standard SQL statements.

- **LifeCycle** Interface.

⇒ This interface is used by implementations for creation of objects, to insure unique object references in a DSM environment.

- **Interfaces** Interface.

⇒ The interface provides a method for defining and verifying new interfaces to an environment, to insure unique interface types in a DSM environment.

- **Security** Interface.

⇒ The interface provides a method to associate the passing of authentication parameters with open, resolve or get operations.

- **Service** Interface.

⇒ This interface enables activation and deactivation of user connections to a service, with the ServiceGateway acting as the primary service broker.

5.3 Application Runtime Procedures

5.3.1 User-to-Network Assumptions and Requirements

The section entitled “USER-TO-NETWORK OPERATIONS” of this standard provides a signaling message set for the establishment and teardown of multiple User-to-User connections represented as a Session. These connections are used by the User-to-User primitives for the request and delivery of MPEG-2 audio, video and private data.

5.3.1.1 Session Establishment

1. The User-to-User application portability function **DSM_ServiceGateway_Attach** is called to establish a Session. Its input parameters are marshaled together and placed in the **userData** field of **ClientSessionSetupRequest**. The input parameters in **DSM_ServiceGateway_Attach** are:

    - A Client reference which is a unique system-wide address/identification of the client node. This is placed in the **rClientRef** parameter.
    - A Client Configuration Information Element whereby the settop identifies its characteristics to the server. This is placed in the input **rClientProfile** parameter.
    - An End User identification of the consumer, unique within the context of the Service Gateway that the Client wishes to attach to. This is placed in the **aEndUser** parameter.
    - A User Context identifier, used for resumption of previously suspended application state. This is placed in the input **aSuspendContext** parameter.
    - A path specification identifying a path to the desired service. The first node in the path specification is a ServiceGateway Name to identify the desired ServiceGateway. The optional second node in the path specification will contain a Service Name to identify the desired initial service. The Service Name is used by the ServiceGateway to determine the initial network resources that should be requested for the session. These are placed in the **rPathSpec**

parameter. The Network will take the **userData** of **ClientSessionSetupRequest** and place it in the **userData** of **ServerSessionSetupIndication**, as part of the Session establishment sequence.

2. The server will use the Client Profile and Service Name to negotiate proper resources for the session using the appropriate User-to-Network messages.
3. Near the completion of the User-to-Network session establishment, The server will return **ServerSessionSetupResponse** to the network, with **userData** containing the marshaled output parameters of **DSM_ServiceGateway_Attach**. The output parameters of **DSM_ServiceGateway_Attach** are:
    - An exception indication. This is placed in the **ev** parameter.
    - A User Context identifier, used as an assignment of UserContext by the ServiceGateway. This is placed in the output **aResumeContext** parameter.
    - Resolved references for the ServiceGateway and optionally for the first service, as specified in **rPathSpec**. These are placed in the output **rPathRefs** parameter.
    - Local time of the ServiceGateway, placed in the **rDateTime** parameter.
4. The Network will take the **userData** of **ServerSessionSetupResponse** and place it in the **userData** of **ClientSessionSetupResponse**, thus passing the reply to **DSM_ServiceGateway_Attach** back to the client.
5. The format of the resolved reference mentioned above depends upon the RPC and network stack used between client and service. **rClientProfile** will provide this identity so that both client and service can determine the contents of the reference. The RPC is Universal Network Object (UNO), unless otherwise specified in **rClientProfile**. This will typically contain addressing information.
6. The format of datatypes will be Common Data Representation (CDR), unless otherwise specified in **rClientProfile**.
7. A Service Reference is always returned for the Service gateway. The Service Gateway is the service where Directory Open (specifying service name) is sent. It performs the role of object broker for the Server system. It is assumed that services use the Bind, Unbind, Launch and Unlaunch interfaces as defined by this standard.
8. Depending on the **rPathSpec** of the **ServiceGateway Attach**, References may be returned for

    a. Application Service. The Application service is the service where User-to-User messages are sent after the application boot has occurred.

    b. DSM object within the Application Service, e.g. a Stream for a Movie.

5.3.1.2 Session Teardown

1. The User-to-User application portability function **DSM_ServiceGateway_Detach** is called to teardown a Session. Its input parameters are marshaled together and placed in the **userData** field of **ClientReleaseRequest**. This **userData** is then forwarded by the network to the **ServiceGateway** in **ServerReleaseRequest**. The input parameters in **DSM_ServiceGateway_Detach** are:
    - A ServiceGateway reference which is a unique system-wide address/identification of the Service Gateway node. This is placed in the **object** parameter.
    - A suspend indication which indicates to the server that application state should be preserved for later resumption. This is placed in the **aSuspend** parameter.
2. The output parameters of **DSM_ServiceGateway_Detach** are marshaled together and placed by the ServiceGateway in the **userData field** of ServerReleaseConfirm. This **userData** is then forwarded by

the network to the Client in **ClientReleaseConfirm**. There is one output to **DSM_ServiceGateway_Detach:**

- An exception indication. This is placed in the **ev** parameter.

5.3.2 Initial Application State

In order for the application to access remote objects, it must make a call to open the application using **Service Gateway Attach** on the Application Interface. The corresponding remote interface to **ServiceGateway Attach** consists of User-to-Network **ClientSessionSetupRequest** and **ClientSessionSetupResponse**. The parameters to Attach are carried in the User-to-Network **userData** field. When the Client/EndUser has been authenticated and authorized by the Service Gateway, the connection between client and service is established. The Service Gateway is the broker for access to all services. A front end of the application must be downloaded from a Service to the client before it can begin execution. This Service (which contains a copy of the initial front end) may be either a Download Service, as described in the section entitled “Initial Application Download,” or a File Service, as described in this section. The Download Service has a specific protocol for communicating client configuration and downloading not only the application front end, but also the network stack it requires. In addition, it can perform cyclic downloads for application boots to many clients at a time. If a File Service is used, a well-known file name may be used for the initial front end.

An end-user may select an application by name, or may indicate that the application name is unknown. In the latter case, the application name will be chosen by the Service Gateway, e.g., it will choose a default application such as a catalog, within which the end-user can pick and choose an application. To start this process, the client will issue Directory Open (service name) to the DSM Library.

5.4 DSM-CC User-to-User System Specification

5.4.1 Interface Definition Language

The interfaces are defined in Object Management Group (OMG) Interface Definition Language (IDL),

which is standardized as ISO/IEC 14750. OMG IDL provides both language independence and protocol independence. It also supplies a strong typing system which can prevent mismatches when the various components are installed instead of detecting them by failure at runtime. Lastly, OMG IDL provides an interface inheritance mechanism which can allow component interface to be extended over time while maintaining backward compatibility.

In OMG IDL syntax, parameters are specified to be either input, output, or both input and output. In the underlying RPC-like implementation, input parameters will be placed in the RPC request, and output parameters will be returned in the RPC reply. Refer to ISO/IEC 14750 for further information about OMG IDL. The entire DSM interface is enclosed in IDL module named “DSM”. This prevents name collision with other OMG environments.

The application portability interface is compiled from the IDL. Using the synchronous deferred compile option, a DSM IDL compiler will generate a RequestHandle as the return value of the operation. The use of the RequestHandle allows an application process to overlap requests without blocking while waiting for replies from the remote service. This avoids forcing the client to wait for a possibly high latency response before continuing. For example, an open and several file requests can be started without waiting for a server response. The client need only to wait for a response at the point that the data is actually required.

5.4.2 DSM Primitives Interface Overview

DSM specifies interfaces as either abstract or instantiable. An abstract interface is never used to define a realizable object, it only provides interfaces that are useful for inclusion in an object’s overall interface. An instantiable interface maps directly to a real object, such as a Stream or a File. Using OMG IDL notation, a new interface can include other interfaces. For example, Stream includes the Base and Access interfaces in the following IDL:

```
module DSM {
    interface Stream: Base, Access {};
};
```

An Access Role (i.e., READER, WRITER, BROKER, OWNER or MANAGER) shall be associated with each operation and with Get and Put individually on each attribute.

The ServiceGateway complex must implement all Core interfaces completely in order to be DSM-CC compliant. Extended interfaces are optional. If the Server implements an extended interface, it must implement all operations and attributes of the interface. The client, however, need only implement those privilege groups that are required by the current application. A client can choose, for example, to implement only the READER group of an interface, meaning only those operations in the interface with READER Access Role.

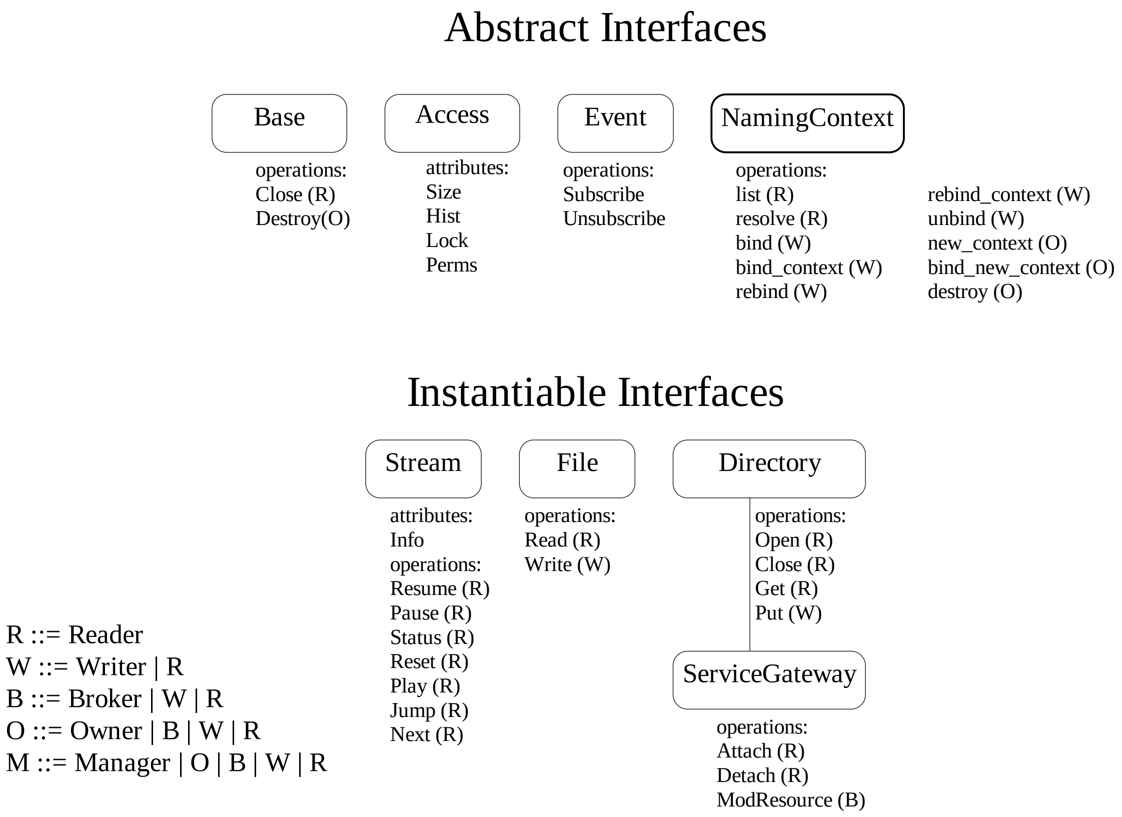


Figure 30: DSM Core Abstract and Instantiable Interfaces

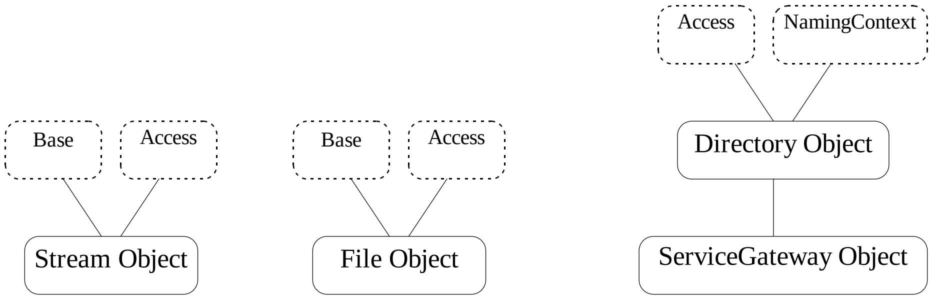


Figure 31: DSM Core Inheritance Hierarchy

5.4.3 DSM Common Types

The following are DSM basic type definitions commonly used throughout the DSM interface:

```
module DSM {
   // machine-independent basic types
  //
  typedef short s_short;              // 16 bit signed integer
  typedef long s_long;                // 32 bit signed integer
  typedef unsigned short u_short;// 16 bit unsigned integer
  typedef unsigned long u_long;   // 32 bit unsigned integer
  // u_longlong is used in File interface as aOffset, unsigned 64 bit integer
  // note: this is a placeholder until OMG formalizes the standard basic type
  // until then the array of 2 u_long is little-endian over the interface
  typedef u_long  u_longlong[2];
  typedef sequence<octet> ObjData;

  // entity identification
  //
  typedef string Profile;                          // user device capabilities info
  //  ObjRef  is a scoped handle,
  //  whereas ObjKey is complete unique system-wide identification,
  // ObjKey includes ObjRef plus RPC-specific address info
  // DSM Library maps between the two to provide ObjKey to the RPC header
  typedef u_long  ObjRef;
  typedef sequence<octet> ObjKey;
  typedef u_long UserContext;                  // context for an application run
  typedef u_longlong EndUser;                  // system-wide identification of end user

  // system management
  //
  typedef sequence<octet, 1024> EncryptData;
  typedef string Password;
  typedef char AccessRole;
  const char MANAGER = 'M';
  const char OWNER = 'O';
  const char BROKER = 'B';
  const char WRITER = 'W';
  const char READER = 'R';
  struct Version {u_long aMajor; u_long aMinor;};
  struct DateTime {                   // tm from ANSI C std.
          s_long tm_sec;              //  seconds, 0-59
          s_long tm_min;              // minutes, 0-59
          s_long tm_hour;             // hours, 0-23
          s_long tm_mday;             // day of the month, 1-31
          s_long tm_mon;              // months since Jan, 0-31
          s_long tm_year;             //years from 1900
          s_long tm_wday;             // days since Sunday, 0-6
          s_long tm_yday;             // days since Jan 31, 0-365
          s_long tm_isdst;};          // Daylight Savings Time indicator

};
```

5.4.4 Exceptions

An exception is an indication that an operation request was not performed successfully. An exception may be accompanied by additional, exception-specific information. Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exceptional condition has occurred during the performance of a request.

The standard OMG IDL exceptions are used by DSM. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard exceptions may not be listed in **raises** expressions. The OMG IDL reference standard has a complete description of how exceptions are handled, plus code examples of C mappings.

Each standard exception also includes a **completion_status** which takes one of the values { COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. These have the following meanings:

COMPLETED_YES	The object implementation has completed processing prior to the exception being raised.
COMPLETED_NO	The object implementation was never initiated prior to the exception being raised.
COMPLETED_MAYBE	The status of the implementation completion is indeterminate.

```
module CORBA {
 #define ex_body {u_long minor; completion_status completed;}
 enum completion_status {COMPLETED_YES, COMPLETED_NO,
                         COMPLETED_MAYBE};
 enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};
  exception UNKNOWN          ex_body;    //the unknown exception
  exception BAD_PARAM               ex_body;    //an invalid parameter was passed
  exception NO_MEMORY               ex_body;    //dynamic memory allocation failure
  exception IMP_LIMIT               ex_body;    //violated implementation limit
  exception COMM_FAILURE            ex_body;    //communication failure
  exception INV_OBJREF              ex_body;    //invalid object reference
  exception NO_PERMISSION           ex_body;    //no permission for attempted op
  exception INTERNAL         ex_body;    //ORB internal error
  exception MARSHALL         ex_body;    //error marshalling param/result
  exception INITIALIZE       ex_body;    //ORB initialization failure
  exception NO_IMPLEMENT     ex_body;    //operation implementation unavailable
  exception BAD_TYPECODE            ex_body;    //bad typecode
  exception BAD_OPERATION    ex_body;    //invalid operation
  exception NO_RESOURCES            ex_body;    //insufficient resources for request
  exception PERSIST_STORE           ex_body;    //persistent storage failure
  exception BAD_INV_ORDER           ex_body;    //routine invocations out of order
  exception TRANSIENT               ex_body;    //transient failure - reissue request
  exception FREE_MEM         ex_body;    //cannot free memory
  exception INV_IDENT        ex_body;    //invalid identifier syntax
  exception INV_FLAG                ex_body;    //invalid flag was specified
  exception INTF_REPOS       ex_body;    //error accessing interface repository
  exception BAD_CONTEXT             ex_body;    //error processing context object
  exception OBJ_ADAPTER             ex_body;    //failure detected by object adapter
  exception DATA_CONVERSION         ex_body;    //data conversion error
};
```

The following common user exceptions are defined by DSM:

```
module DSM {
    enum CompletionStatus {COMPLETED_YES, COMPLETED_NO,
        COMPLETED_MAYBE};
    #define ExceptUser {u_long minor; CompletionStatus aCompleted;}
    #define ExceptAuth {u_long minor; CompletionStatus aCompleted; EncryptData \
                                    aEncryptKey;}

    exception READ_LOCKED ExceptUser;          //READER operation denied, read locked
    exception WRITE_LOCKED ExceptUser;         //WRITER operation denied, write locked
    exception NO_AUTH ExceptAuth;              //not allowed to open without authentication
    exception OPEN_LIMIT ExceptUser; // too many resources are open
    exception MPEG_DELIVERY ExceptUser;        //error delivering MPEG stream
};
```

5.4.5 Access Control

Clients are granted access to object interfaces through the following:

- Privilege: by Access Role for each operation requiring OWNER, MANAGER, BROKER, WRITER or READER privileges to invoke.
- Identification: by client identification as being the authorized OWNER, or a member of an authorized READER, WRITER, BROKER or MANAGER group
- Authentication: Initial access to any object may be set up to require a password /PIN or encrypted key exchange. Access for all operations on an object can be setup to require secure transmissions of all messages to/from that object.

Authorization for access to services is performed by the Service Gateway.

Authorization for access to an application's objects is performed by the Service.

An AllSecure parameter can be set on an object which requires secure transmission for all messages to/from the object. It is expected that the lower layers of network protocol will perform this function via encryption or scrambling.

5.4.5.1 DSM IDL Definition for Access Control

The following rules enable the implementation of access control for DSM-CC:

Each primitive must have an associated invocation privilege OWNER, MANAGER, BROKER, WRITER or READER. This is defined by means of the Access Control Role (ACR) definition form:

```
const char <operation name>_ACR = <access role>;
```

for example, Stream Resume has the following access role definition:

```
module DSM {
    interface Stream {
            const char Resume_ACR = READER;
    };
};
```

Each attribute will have separate invocation privilege for both Get and Put operations against it. If the CORBA _set operation is used, the DSM Put invocation privilege applies. If the CORBA _get operation is used, the DSM Get invocation privilege applies. This is defined by means of the Access Control Role (ACR) definition form:

```
const char <attribute name>_Put_ACR = <access role>;
const char <attribute name>_Get_ACR = <access role>;
```

5.4.5.2 Setting Permissions

When an object is created the OWNER is associated with it. The OWNER may alter privileges by setting the permissions (Perms) attribute on an object to allow access by designated MANAGER, BROKER, WRITER or READER groups. Permissions are set by using **Directory Put** (where path specification is <object-name>, **Perms**).

A client may be in more than one group. An object can be accessed by the OWNER and by more than one group in each of MANAGER, BROKER, WRITER or READER AccessRoles. Groups identifiers are scoped within arbitrary object context boundaries, e.g., within the ServiceGateway, within the Service, etc. The client identifier will map to a set of groups in which that client is a member. Corresponding authentication databases will vary with the implementation, and are not specified in this standard..

The **Perms** attribute will identify groups that can access an object or invoke its operations. An object will limit access to the client who has the specified group or individual identification. Each method of that object will further restrict invocation by allowing access by role and group. For example, if a method requires WRITER privileges, the client must be in one of the WRITER groups specified in that object's **Perms** attribute. To **Open** or **resolve** a target object, the client must be in one of the READER groups specified in that object's **Perms** attribute.

The OWNER may set permissions to associate a password with an object, or to associate encrypt key data with an object. When a request to Open such an object is given, an exception is returned. If encryption is required, the exception contains an encrypt key challenge. In order to continue, the end-user must send an **Authenticate** containing the correct response to the encrypt key. If a password is required, the end-user must send an **Authenticate** containing the correct password. The **Authenticate** must immediately follow with the repeated Open, in order to be granted access.

The OWNER may set an **AllSecure** flag in an object's **Perms** attribute, indicating that the all messages to/from the object must be secure, e.g., encrypted or scrambled. From **Open** or **resolve** time through **Close**, the service will effect secure transmission through appropriate lower network layer messaging.

5.5 The Core Client-Service Interfaces

The Core DSM User interfaces are the minimum set that must be supported by a DSM-CC compliant Server Complex. They include Base, Access, Directory, Stream, File and ServiceGateway interfaces.

5.5.1 Base

The Base interface provides common operations for DSM objects. These operations are organized into a base class for convenience rather than a specific need. In particular, we do not expect the Base type to be used directly by applications.

A client (READER) may have obtained an object reference for an object, performed some operations against it, and now has no more use for that object. The **Close** operation indicates the client no longer needs transient resources associated with the object and will not make any further requests.

When a client (OWNER) wishes to delete the persistent data associated with an object, it may invoke the **Destroy** to destroy it, thus enabling the object's parent service to free all state and storage resources associated it.

In addition, all DSM object implementations must support a type query that asks whether the object supports the operations defined by a particular interface. The C Client API defines this operation as DSM_Base_IsA (see the C Client API for more details). In general, the implementation requirement depends on the underlying RPC communication. UNO's IIOP, for example, defines a special "_is_a" request with a type string parameter. A CORBA environment will normally handle this request automatically at the client or at the server skeleton without any effort from the object implementation. Non-CORBA environments, however, must be able to handle this kind of request.

5.5.1.1 Summary of Base Primitives

The following primitives are used by the interfaces of all objects.

Close	Close a reference to an object. (READER)
Destroy	Destroy an object instance. (OWNER)

```
module DSM
     interface Base {
            const AccessRole Close_ACR = READER;
            const AccessRole Destroy_ACR = OWNER;
     };
};
```

5.5.1.2 DSM Base Close

DSM Base Close	Close a reference to an object. (READER)

Client-Service Interface IDL Syntax

```
module DSM {
   interface Base {
            void  Close ();
    };
};
```

Semantics

Base Close is used by the client to indicate that access to the object is no longer required. This is primarily a resource issue and is not specifically required; however, the total number of references allowed is limited, and well behaved clients will close whenever reasonable. If OPEN_LIMIT has been received when attempting to Open an object reference, the client will need to use Close on one or more other active references in order to free resources, before retrying the Open.

Privileges Required:
READER

5.5.1.3 DSM Base Destroy

DSM Base Destroy	Destroy an object instance.

Client-Service Interface Syntax

```
module DSM {
   interface Base  {
           void Destroy  ();
   };
};
```

Semantics

Base Destroy is used by the client to delete an object instance. After this occurs, its reference will no longer be valid, and storage resources used for it will be freed.

Privileges Required:
OWNER

5.5.2 Access

The Access interface provides common description and access control attributes needed by most objects. These include size, version, date, lock and permissions attributes.

5.5.2.1 Access Definitions

```
module DSM {
     interface Access {
          // size
          attribute u_long Size;                  // object size in octets;
          const AccessRole Size_Get_ACR = READER;
          const AccessRole Size_Put_ACR = BROKER;

          // history
          struct Hist_T {
                    Version  aVersion;            // object version
                    DateTime  aDateTime;}; // time created or last updated
          attribute Hist_T Hist;                  // version and time of persistent object
          const AccessRole Hist_Get_ACR = READER;
          const AccessRole Hist_Put_ACR = BROKER;

          // lock status
          struct Lock_T {boolean ReadLock; boolean WriteLock;};
          attribute Lock_T Lock;
          const AccessRole Lock_Get_ACR = READER;
          const AccessRole Lock_Put_ACR = WRITER;

          // permissions
          struct Perms_T {
                    //the next 4 are binary masks of binary flags signifying
                    //PasswordReqd, EncryptReqd, and groups that can access the object
                    u_short ManagerPerm;
```

```
                u_short BrokerPerm;
                u_longlong WriterPerm;
                u_longlong ReaderPerm;
                u_longlong Owner;               //owner identifier
                string rPassword;               //PIN
                EncryptData rEncryptData;       //key
                // instruct lower layers to implement a secure connection for this object
                boolean AllSecure;};            // all methods parameters encrypted
        attribute Perms_T Perms;
        const AccessRole Perms_Get_ACR = OWNER;
        const AccessRole Perms_Put_ACR = OWNER;
    };
};
```

5.5.3 Directory

The **Directory** interface provides a general name space for binding names to services or data. A service gateway implements the directory interface, and as such provides the primary mechanism for accessing other services or applications.

Directory defines four kinds of operations:

- Binding a name to an object reference or data value
- Resolving a name to the bound object reference or data value
- Removing a name’s binding
- Listing the bindings

The **Directory** interface inherits the attributes defined by the **Access** interfaces to allow permission definitions for directories as well as individual services and data.

For the basic operations involving object references, **Directory** inherits from the NamingContext interface defined in the CORBA Object Services Naming module (CosNaming). Using the OMG NamingContext interface allows a CORBA environment to support DSM-CC easily, while not requiring a DSM-CC implementation to use a CORBA system.

Directory does *not* inherit from **Base** because both **Base** and NamingContext define destroy operations. The current IDL specification requires that operation names, including inherited operations, be unique and case-insensitive. Thus, Base::Destroy would collide with the NamingContext::destroy. Because it cannot inherit from **Base**, **Directory** defines its own **Close** operation.

For completeness, the NamingContext operations are presented below; however, the CORBA Naming specification is the correct definition of the operations that a directory must support. Any discrepancies between the CORBA Naming specification and the NamingContext operations described below represents an error in the copy of the operations presented here.

The CORBA Naming specification includes the definitions for names and bindings below. The DSM module defines equivalent types either by referencing the CosNaming definitions with a typedef or by having a complete definition of the type. A name component consists of two strings that must be unique within a specific context. A name is a sequence of components that can describe a path through a set of contexts.

5.5.3.1 Directory Definitions, Exceptions

```
module CosNaming{
   typedef string Istring;
   struct NameComponent {
```

```
            Istring id;
            Istring kind;
    };
    typedef sequence<NameComponent> Name;
    enum BindingType {nobject, ncontext };
    struct Binding {
            Name binding_name;
            BindingType binding_type;
    };
    typedef sequence <Binding> BindingList;

    interface BindingIterator;
};
```

```
module DSM {
    typedef CosNaming::Istring Istring;
    typedef CosNaming::NameComponent NameComponent;
    typedef CosNaming::Name Name;
    typedef CosNaming::BindingType BindingType;
    typedef CosNaming::Binding Binding;
};
```

NamingContext defines operations for naming object references but not general data, so **Directory** adds similar operations for binding names to data values (type “any” in IDL). In addition to naming data values, the directory interface extends the NamingContext interface with operations to bind or resolve a list of names in a single call. These operations allow a compact call to access a number of objects. The semantics of these operations are always identical to performing a sequence of the individual calls. To bind or resolve a single name, the sequence length can be set to 1.

The list of names specified in a single call is specified by a **PathType** and **PathSpec**. The type indicates the format of the spec, which may be a linear path of objects (DEPTH traversal) or child objects at a given level of hierarchy (BREADTH traversal). The spec is a sequence of **Step** structures, each of which contains a name component and a process flag. If the **process** flag in a **Step** structure is set then the step name is processed, otherwise the name is simply used to traverse further into the name space. If the operation using **PathSpec** returns object references, these will be contained in a separate path references output parameter. If the operation using **PathSpec** returns data values, these will be contained in a separate path values output parameter. The definition of these types is as follows:

```
module DSM {
   // two types of path traversal, depth and breadth match traditional methods
   const char DEPTH = 0;
   const char BREADTH = 1;
   typedef char PathType;          //DEPTH or BREADTH

   struct Step {
           NameComponent name;
           boolean process;
   };

   typedef sequence<Step> PathSpec;
   typedef sequence<ObjRef> PathRefs;
   typedef sequence<any> PathValues;
};
```

Directory operations return several types of exceptions, specified below. These exceptions are defined by the CosNaming module, and therefore must be either available as part of the DSM environment or defined explicitly by a DSM implementation.

Ed. note: There does not appear to be a way to define an alias for an exception in the same way that one creates a typedef, so a non-CORBA implementation will actually need to define a CosNaming module. If there is a way to alias exception types then these definition should indicate that.

```
module CosNaming {
    interface NamingContext {
         enum NotFoundReason { missing_node, not_context, not_object };

         exception NotFound {
                  NotFoundReason why;
                  Name rest_of_name;
         };

         exception CannotProceed {
                  NamingContext cxt;
                  Name rest_of_name;
         };

         exception InvalidName {};
         exception AlreadyBound {};
         exception NotEmpty {};
    };
};
```

```
module DSM {

  // these are identical to CosNaming, for use by DSM Directory primitives:

  enum NotFoundReason {
         missing_node, not_context, not_object
   };
   exception NotFound {
         NotFoundReason why;
         Name rest_of_name;
   };
   exception CannotProceed {
         CosNaming::NamingContext cxt;
         PathSpec rest_of_name;
   };
   exception InvalidName { };
   exception AlreadyBound { };
   exception NotEmpty { };
};
```

The **NotFound** and **CannotProceed** exceptions return the unresolved part of the requested name. For example, if a **bind** operation on the path (A,B,C) raises **NotFound** with the rest_of_name as (B,C) then the context named by A could not resolve the B name component. The meaning of the **NotFound**, **AlreadyBound**, and **InvalidName** exceptions is straightforward. The **CannotProceed** exception means that the resolving context did not have permission to do a resolve. In the example above, a **CannotProceed** exception means that the A context does not have permission to perform a resolve on the B context. In this case, the caller may wish to attempt to perform the resolve directly, as the caller might have permission even though the A context did not.

5.5.3.2 Summary of Directory Primitives

Inherited from **Access**:

attributes: **Size, Hist, Lock, Perms**

Inherited from **NamingContext**:

list	Return a list of all the bindings to object references in the context. (READER)
resolve	Return the object reference bound to a given name. (READER)
bind	Bind an object reference to a name. (WRITER)
bind_context	Bind a naming context to a name. (WRITER)
rebind	Bind an object reference to a name, overwriting any previous binding. (WRITER)
rebind_context	Bind a context to a name, overwriting any previous binding. (WRITER)
unbind	Remove a binding for a name. (WRITER)
new_context	Create a new naming context. (OWNER)
bind_new_context	Create a new naming context and bind it to the given name. (OWNER)
destroy	Destroy the naming context. (OWNER)

Defined in **Directory**:

DSM Directory Open	Resolve the objects associated with names in the given path (READER).
DSM Directory Close	Close a reference to a Directory. (READER)
DSM Directory Get	Return the values bound to names in a given path. (READER)
DSM Directory Put	Bind names in a given path to values, overwriting any previous bindings. (WRITER)

```
module DSM
    interface Directory : Access, CosNaming::NamingContext {
            const AccessRole list_ACR = READER;
            const AccessRole resolve_ACR = READER;
            const AccessRole bind_ACR = WRITER;
```

```
        const AccessRole bind_context_ACR = WRITER;
        const AccessRole rebind_ACR = WRITER;
        const AccessRole rebind_context_ACR = WRITER;
        const AccessRole unbind_ACR = WRITER;
        const AccessRole new_context_ACR = OWNER;
        const AccessRole bind_new_context_ACR = OWNER;
        const AccessRole destroy_ACR = OWNER;
        const AccessRole Open_ACR = READER;
        const AccessRole Close_ACR = READER;
        const AccessRole Get_ACR = READER;
        const AccessRole Put_ACR = WRITER;
    };
};
```

5.5.3.3 list

list	Return a list of all the bindings to object references in the context. (READER)

Client-Service IDL Syntax

```
module CosNaming {
    interface NamingContext {
        void list(
                in unsigned long count,
                out BindingList bindings, out BindingIterator itr
        );
    };
};
```

Semantics

The **list** operation returns a list of bindings in the NamingContext. The count parameter indicates how many bindings to return immediately; the remaining bindings can be retrieved from the returned iterator. The iterator interface simply has two operations defined as follows:

```
module CosNaming {
    interface BindingIterator {
        boolean next_one (out Binding b);
        boolean next_n (in unsigned long how_many,
                        out BindingList bl);
        void destroy ();
   };
};
```

```
module DSM {
   typedef CosNaming::BindingIterator BindingIterator;
};
```

The **next_one** and **next_n** operations return more bindings from the context, if there are any. Both operations return false if there were no additional bindings. The **destroy** operation discards any server-side storage associated with the iterator and makes the iterator no longer valid to access.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
unsigned long count	input	The maximum number of bindings to return.
BindingList bindings	output	A sequence containing up to count bindings.
BindingIterator itr	output	An iterator for retrieving additional bindings.

5.5.3.4 resolve

resolve	Bind a name to a data value. (READER)

Client-Service IDL Syntax

```
module CosNaming {
    interface NamingContext {
         Object resolve (in Name n)
              raises (NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **resolve** operation returns the object reference that is bound to the given name. If no name is bound, then the **NotFound** exception is raised.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
Object	output	The object reference that is bound to the name.

5.5.3.5 bind

bind	Bind an object reference to a name. (WRITER)

Client-Service IDL Syntax

```
module CosNaming {
    interface NamingContext {
         void bind (in Name n, in Object obj)
              raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
```

```
      };
};
```

Semantics

The **bind** operation associates an object reference with a name. This operation raises the **AlreadyBound** exception if the name is bound to another object or data value in this context. The name specifies one or more name components that indicate intermediate contexts through which to search. If any of the components is not bound, then the **NotFound** exception is raised. If an intermediate context is found that refuses permission to the outer context then a **CannotProceed** exception is raised.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
Object obj	input	The object reference that is bound to the name.

5.5.3.6 bind_context

bind_context	Bind a naming context to a name. (WRITER)

Client-Service IDL Syntax

```
module CosNaming {
      interface NamingContext {
             void bind_context (in Name n, in NamingContext nc)
                    raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
      };
};
```

Semantics

The **bind_context** operation associates a naming context with a name. This operation raises the AlreadyBound exception if the name is bound to another object or data value in this context.

This operation is distinct from the **bind** operation to allow the option of binding a context into a name space where it will not implicitly resolve components of a path. This approach also simplifies the resolution process, as a context knows exactly which contexts to search inside it rather than needing to narrow every bound object to see if it is a context.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
NamingContext nc	input	The naming context that is bound to the name.

5.5.3.7 rebind

rebind	Bind an object reference to a name, overwriting any previous binding. (WRITER)

Client-Service IDL Syntax

```
module CosNaming {
    interface NamingContext {
        void rebind (in Name n, in Object obj)
            raises (NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **rebind** operation associates an object reference with a name in a directory**.** Unlike the **bind** operation, this operation will replace the binding for a name if it was previously-bound.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
Object obj	input	The object reference that is bound to the name.

5.5.3.8 rebind_context

rebind_context	Bind a naming context to a name. (WRITER)

Client-Service IDL Syntax

```
module CosNaming {
    interface NamingContext {
        void rebind_context (in Name n, in NamingContext nc)
            raises (NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **rebind_context** operation associates a naming context with a name. Unlike the **bind_context** operation, this operation will replace the binding for a name if it was previously-bound.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
NamingContext nc	input	The naming context that is bound to the name.

5.5.3.9 unbind

unbind	Remove a binding for a name. (WRITER)

Client-Service IDL Syntax

```
module CosNaming {
    interface NamingContext {
        void unbind (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **unbind** operation removes the binding associated with the given name from the directory.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.

5.5.3.10 new_context

new_context	Create a new naming context. (OWNER)

Client-Service IDL Syntax

```
module CosNaming {
    interface NamingContext {
        NamingContext new_context();
    };
};
```

Semantics

The **new_context** operation returns a newly-created NamingContext.

Privileges Required:
OWNER
Parameters

type/variable	direction	description
NamingContext	output	The newly-created naming context.

5.5.3.11 bind_new_context

bind_new_context	Create a new naming context and bind it to the given name. (OWNER)

```
module CosNaming {
  interface NamingContext {
        NamingContext bind_new_context(in Name n)
                raises (AlreadyBound, NotFound, CannotProceed, InvalidName);
  };
};
```

Semantics

The **bind_new_context** operation creates a new context and associates it with the given name.

Privileges Required:
OWNER

Parameters

type/variable	direction	description
Name n	input	A name that describes a path through one or more directories, starting with this one.
NamingContext	output	The newly-created naming context.

5.5.3.12 destroy

destroy	Destroy the naming context. (OWNER)

Client-Service IDL Syntax

```
module CosNaming {
    interface NamingContext {
         void destroy ()
              raises (NotEmpty);
    };
};
```

Semantics

The **destroy** operation removes the persistent storage associated with the context.

Privileges Required:
OWNER

5.5.3.13 DSM Directory Open

Open	Find the objects associated with the names in the given path (READER).

Client-Service IDL Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
         void Open(
                   in PathType aPathType,
                   in PathSpec rPathSpec,
                   out PathRefs rPathRefs)
         raises(OPEN_LIMIT, NO_AUTH , NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **Directory Open** operation provides a path traversal with a resolve of object references from names at specified nodes in the path. The **aPathType** and **rPathSpec** parameters define the specific set of names and values that are resolved. The result is a PathSpec that corresponds to the input rPathSpec with the object references set. This operation looks up each path element sequentially, but not atomically (other directory operations may occur between the lookups of elements). If the a particular resolve fails, then the entire operation raises the appropriate exception.

The object reference(s) returned by rPathSpec are assigned by the client DSM Library, in order to enable deferred synchronous operation. It maps to the RPC destination address and other object qualifiers that are passed in the RPC messages over the network between the Client and the newly opened object.

Privileges Required:
READER

Parameters

type/variable	direction	description
PathType aPathType	input	A reference to a Directory object.
PathSpec rPathSpec	input	A sequence of steps, each representing a node in a directory hierarchy.
PathRefs rPathRefs	output	The object references resolved as a result of this operation.

5.5.3.14 DSM Directory Close

DSM Directory Close	Close a reference to a Directory. (READER)

Client-Service IDL Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        void  Close ();
     };
};
```

Semantics

Directory Close is used by the client to indicate that access to the directory is no longer required. This operation is sent to the Directory to be closed. Closing a Directory is not specifically required unless the directory is bound as a service to the ServiceGateway, in which case network resources may need to be freed as a result of the close.

Privileges Required:
READER

5.5.3.15 DSM Directory Get

Get	Return the values bound to the names in a given path. (READER)

Client-Service IDL Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        void Get(
                in PathType aPathType,
                in PathSpec rPathSpec,
                out PathValues rPathValues)
        raises(NO_AUTH, NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **Directory Get** operation provides a path traversal with a resolve of data values from names at specified nodes in the path. The **aPathType** and **rPathSpec** parameters define the specific set of names and values that are resolved. The result is a PathSpec that corresponds to the input rPathSpec with the object references set. This operation looks up each path element sequentially, but not atomically (other directory operations may occur between the lookups of elements). If the a particular resolve fails, then the entire operation raises the appropriate exception.

Privileges Required:
READER

Parameters

type/variable	direction	description
PathType aPathType	input	A reference to a Directory object.
PathSpec rPathSpec	input	A sequence of steps, each representing a node in a directory hierarchy.
PathValues rPathValues	output	The object or attribute values resolved as a result of this operation.

5.5.3.16 DSM Directory Put

Put	Bind a graph of names to the given values. (WRITER)

Client-Service IDL Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        void Put(
                in PathType aPathType,
                in PathSpec rPathSpec,
                in PathValues rPathValues)
        raises(NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **Directory Put** operation provides a path traversal with a binding of data values to names at specified nodes in the path. The **aPathType** and **rPathSpec** parameters define the specific set of names and values that are bound. This operation binds each path element sequentially, but not atomically (other directory operations may occur between the binding of elements). If the a particular resolve fails, then the entire operation raises the appropriate exception.

Privileges Required:
WRITER

Parameters

type/field	direction	description
PathType aPathType	input	The path type indicates these structure of rPathSpec. It may be a linear path (DEPTH traversal), or child nodes at one level of a hierarchy (BREADTH traversal).
PathSpec rPathSpec	input	A sequence of steps, each representing a node in a directory hierarchy.
PathValues rPathValues	input	The object or attribute values to be bound as a result of this operation.

5.5.4 Stream

Stream primitives are used to emulate VCR-like controls for manipulating MPEG continuous media streams. Streams differ from other datatypes in that, while in play mode, the rate of the stream delivery will be governed by an MPEG network flow control mechanism. Streams include data types such as video and audio, as defined by ISO/IEC 13818.

Stream Pause and **Stream Resume** behave much like their VCR counterparts. However, each primitive that initiates play mode includes a scale parameter which controls forward or reverse operation. Position is indicated in Normal Play Time (NPT), which is specified as seconds and sub-seconds. NPT indicates the stream absolute position relative to the beginning of the stream. **Stream Play** enables play from a start NPT position until a stop NPT position is reached. **Stream Jump** provides capability to jump when a stop NPT position is reached to any start NPT position in the stream.

A stream is first requested using **Directory Open**. It returns a reference for the stream, to be used with subsequent stream commands. Streams open in pause mode. More than one stream can be opened at a time. **Stream Next** is used to link a currently playing stream to another opened stream. When the current playing stream reaches its NPT stop time, the second stream will become active.

Successful execution of Stream commands require that the Service execute them in the exact sequence that the client has requested them. For example, **Directory Open** and **Stream Resume** can be sent in quick succession from the client if play mode is desired immediately after the completion of the Open. In this case, the operations could not be executed out of order, since at the service, the results of one will feed as an input to the second.

At any given time, the server will be in one of the following play modes for a given video delivery:

Open	The stream object is opened
Pause	The stream object does not transport the media stream
SearchXport	The stream object goes to start NPT and then transports the media stream
SearchXportPause	The stream object goes to start NPT, and then transports the media stream until stop NPT is reached.
PauseSearchXport	The stream object transports the stream until stop NPT, then searches to start NPT and transports the media stream.
Unknown	The stream is in an unknown state.

Stream Status is used to inquire as to the current NPT, Scale and Mode of an open stream.

5.5.4.1 Stream Definitions, Exceptions

```
module DSM {
   interface Stream : Base, Access {
       exception BAD_STOP ExceptUser;          //invalid StopNPT, can never be reached
       exception BAD_START ExceptUser;                 //stream does not contain this NPT
       exception BAD_SCALE ExceptUser;                 //invalid scale
       exception INV_NEXTREF ExceptUser;               //invalid next stream reference
       exception QUE_LIMIT ExceptUser;                 //stream que depth exceeded. max = 2
       struct NPT {s_long aSeconds; u_long aMicroSeconds;};     // Normal Play Time
       struct Scale {s_short aNumerator; u_short aDenominator;};          //+FF,-Rewind, Rate
       // stream modes
       const u_long PAUSE_M = 0;
       const u_long TRANSPORT_M = 1;
       const u_long TRANSPORT_PAUSE_M = 2;
       const u_long PAUSE_TRANSPORT_M = 3;
       const u_long END_OF_STREAM_M = 4;
       const u_long UNKNOWN_M = 5;
       typedef u_long Mode;
       struct Stat {
               NPT rPosition;
               Scale rScale;
               Mode aMode;};
       struct Info_T {
               string aDescription;
               NPT duration;
               boolean audio;
               boolean video;
               boolean data;};
       attribute Info_T  Info;
       const AccessRole Info_Get_ACR = READER;
       const AccessRole Info_Put_ACR = OWNER;
   };
};
```

Note: The Stream Info is not intended to be an attribute database, but rather to be a minimum set of stream identification and characteristics. Pertinent title information includes title and runtime length.

5.5.4.2 Normal Play Time Temporal Positioning

In order to support random positioning and a variety of play rates the Media stream primitives make use of a temporal addressing scheme called Normal Play Time (NPT). Intuitively NPT is the clock the viewer associates with a program. It is often digitally displayed on a VCR. NPT advances normally when in normal play mode (scale = 1/1), advances at a faster rate when in fast scan forward (high positive scale ratio), decrements when in scan reverse (high negative scale ratio) and is fixed in pause mode. NPT is roughly equivalent to SMPTE timecodes. NPT is defined as two values representing seconds and subseconds.

To understand DSM's NPT model one must separate the application's perspective of NPT from the underlying mechanism used to coordinate NPT between the client and server.

From the application's perspective NPT is a clock that is maintained in the client operating system. It is used to request position relative to a specific program (i.e. "where are we?") or to control the position of the stream (i.e. "jump to this position"). Consider the following example: As a reference time assume that real time starts at 0 and progresses in seconds. Note that RPC-latency is assumed to be 0. Suppose that the application makes the following calls:

1. At 0 seconds Stream Open is called. It is followed by Stream Resume requesting that the stream begin playing at normal play rate with NPT start time 30.
2. At 10 seconds Stream Pause is called. At this point NPT will be 40.
3. At 16 seconds Stream Resume is called requesting that the stream continue playing at NPT = 80 at ten times normal speed.
4. At 26 seconds Stream Close is called. At this point NPT will be 180.

The coordination of NPT between the server and client is independent of the API usage of NPT. There are two possible methods for maintaining NPT in the client. The first is method NPT descriptors as described in the section entitled “NORMAL PLAY TIME.” The other method is to explicitly query the server for NPT information. Due to latency considerations the second method may be less accurate.

5.5.4.3 Summary of Stream Primitives

Inherited from **Base**:

IsA, Close, Destroy

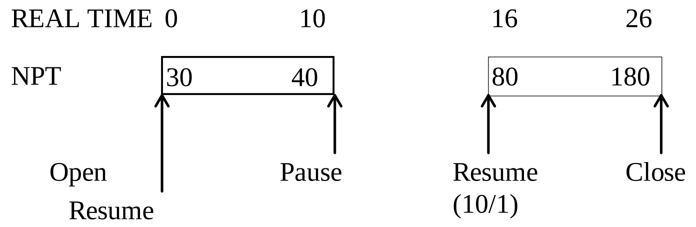
Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Defined in **Stream**:

attributes: **Info**

DSM Stream Pause	Stop sending stream when NPT position is reached. (READER)
DSM Pause Resume	Start sending stream at NPT positon within stream. (READER)
DSM Stream Status	Obtain status of a stream. (READER)
DSM Stream Reset	Reset the queue of pending operations on a stream (READER)
DSM Stream Jump	When stream reaches stop NPT, resume at start NPT. (READER)
DSM Stream Play	Play stream from start NPT until stop NPT. (READER)
DSM Stream Next	Establish a link to a successor stream when this stream completes. (READER)

```
module DSM
    interface Stream : Base, Access {
          const AccessRole Pause_ACR = READER;
          const AccessRole Resume_ACR = READER;
          const AccessRole Status_ACR = READER;
          const AccessRole Reset_ACR = READER;
          const AccessRole Jump_ACR = READER;
          const AccessRole Play_ACR = READER;
          const AccessRole Next_ACR = READER;
    };
};
```



The above drawing illustrates an example of Stream primitives usage. In this example, time is shown in seconds for ease of explanation. The latency between client and server is not shown (it could be considerable). In response to a viewer input, **Directory Open** followed by **Stream Resume** commands are sent to start playing the video at normal play rate (scale 1/1), with NPT start time 30 seconds into the stream. 10 seconds later the viewer presses the VCR pause button, causing a **Stream Pause** command to be given. At this time and during the pause the video NPT remains at a point 40 seconds from the start of the stream. 6 seconds later, the viewer initiates fast forward with NPT start time 80 seconds into the stream, causing a **Stream Resume** command to be sent with scale = 10/1. Finally, the viewer quits, causing **Base Close** to be sent.

5.5.4.4 Stream State Machine

The stream interface provides methods to control the advance of a media stream. The interface leverages the DSM Scale and NPT structures.

Note that the DSM Library provides the client stream Status with the Status() function. The Status value which the settop device provides allows the remote stream object to instrument the transport delay. The Status value which the remote stream object returns allows the settop device to instrument the transport delay. One application of the measurement is to configure transport buffers.

The time value which the client provides with the Resume(in NPT rStart) is the stream position at which to begin transport. The scale value describes both the direction (reverse is just a negative value) and the rate (normal Play is just a positive value of 1.0). The time value which the client provides with the Pause(in NPT rStop) is the stream position at which to suspend the transport. (There are transport mechanisms which require some message traffic to sustain the connection. It is implementation dependent, and not client visible, how the source of the media stream stimulates the connection for such a transport solution. The obvious technique is to transmit the data stream, with its Status fields.)

Reference to Stream Interface: Since the client can cascade methods, the sequence Resume(rStart) plus Pause(rStop) equates to Play(rStart, rStop). Also the sequence Pause(rStop) plus Resume(rStart) equates to Jump(rStop, rStart).

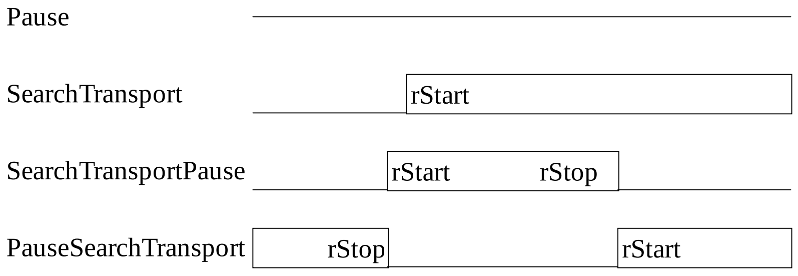
5.5.4.4.1 State Machine

The interface controls a state machine. The state machine, shown below with the some of the transitions, comprises the states of a) Open b) Pause c) Transport d) TransportPause and e) PauseTransport. The

Open() causes the state to transition to the Open state. The default values are a) rStart=0000.0000, b) rStop=EFFF.FFFF, and c) Scale=0001,0001.

If the state machine receives a Pause(rStop) after a Play(rStart, rStop), the Pause(rStop) replaces the Play(rStop). If the state machine receives a Jump(rStop, rStart) after a Play(rStart, rStop), the Jump(rStop) replaces the Play(rStop). This, in essence, cancels the Play() function. If the state machine receives Resume(rStart) after a Jump(rStop, rStart), the Resume(rStart) replaces the Jump(rStart). If the state machine receives Play(rStart, rStop) after a Jump(rStop, rStart), the Play(rStart) replaces the Jump(rStart).

The figure below shows the stream profile which corresponds to each state.



- Pause: There is one phase. The stream object does not transport the media stream.
- SearchTransport: There are two phases with one time value to describe the transition. The stream object searches to rStart, and the transports the media stream. Since there is no rStop, the stream object continues to advance the stream until it receives another function.
- SearchTransportPause: The common sequence which causes the state transition is either Play(rStart, rStop) or Resume(rStart) plus Pause(rStop).There are three phases with two time values to describe the transitions. The stream object searches to rStart, transports the stream until rStop, and enters the Pause state.
- PauseSearchTransport: The common sequence which causes the state transition is either Jump(rStop, rStart) or Pause(rStop) plus Resume(rStart). There are three phases with two time values to describe the transitions. The stream object transports the stream until rStop, searches to rStart, and transports the stream.

The table below shows how the state machine responds to the methods as a function of the previous state.

Previous State	Reset()	Resume()	Pause()	Play()	Jump()
O	O	ST	P	STP	PST
ST	O	ST	STP	STP	PST
TP	O	PST	TP	STP	PST
P	O	ST	P	STP	PST
PST	O	PST	PST	STP	PST

The semantics of Reset() are to return to the Open state. One situation which causes the transition to the Open state is the EndOfStream condition. If the client registers interest in the transition, the client can detect the EndOfStream through the transition. This example illustrates the situation where a client detects a transition which is not the direct result of methods which the client invokes. In the EndOfStream condition, it was the remote stream object, not the settop device, which invokes the Reset() method. Yet the settop device could detect the transition.

The invocation of certain methods cause state transitions. State transitions also occur because the

Previous State	rStart	rStop
O		
ST		
TP		P
P		
PST		ST

Note that the rStop causes transitions, but the rStart does not cause (client visible) transitions. The reason is that the transition from SearchTransport to a Transport (after the stream object searches to rStart) is not (with this state machine) client visible. The next section presents a state machine where the distinction between the SearchTransport state versus Transport state is visible. Thus the client can detect when the search phase is complete.

5.5.4.4.2 Complete State Machine

The state machine above does not expose one transition which was thought to be instantaneous. If the client invokes Resume(rStart), the stream object begins a search phase, advances to the rStart, and begins the transport phase. The transition is visible with the state machine of this section. There is both a SearchTransport state (during which the rStart is valid) and a Transport state (during which the rStart is stale). The table below describes the complete state machine.

	Reset()	Resume()	Pause()	Play()	Jump()
O	O	ST	P	STP	PST
ST	O	ST	STP	STP	PST
T	O	ST	TP	STP	PST
TP	O	PST	TP	STP	PST
P	O	ST	P	STP	PST
STP	O	ST	STP	STP	PST
PST	O	PST	TP	STP	PST

The table below shows the transitions which the rStart condition and the rStop condition cause:

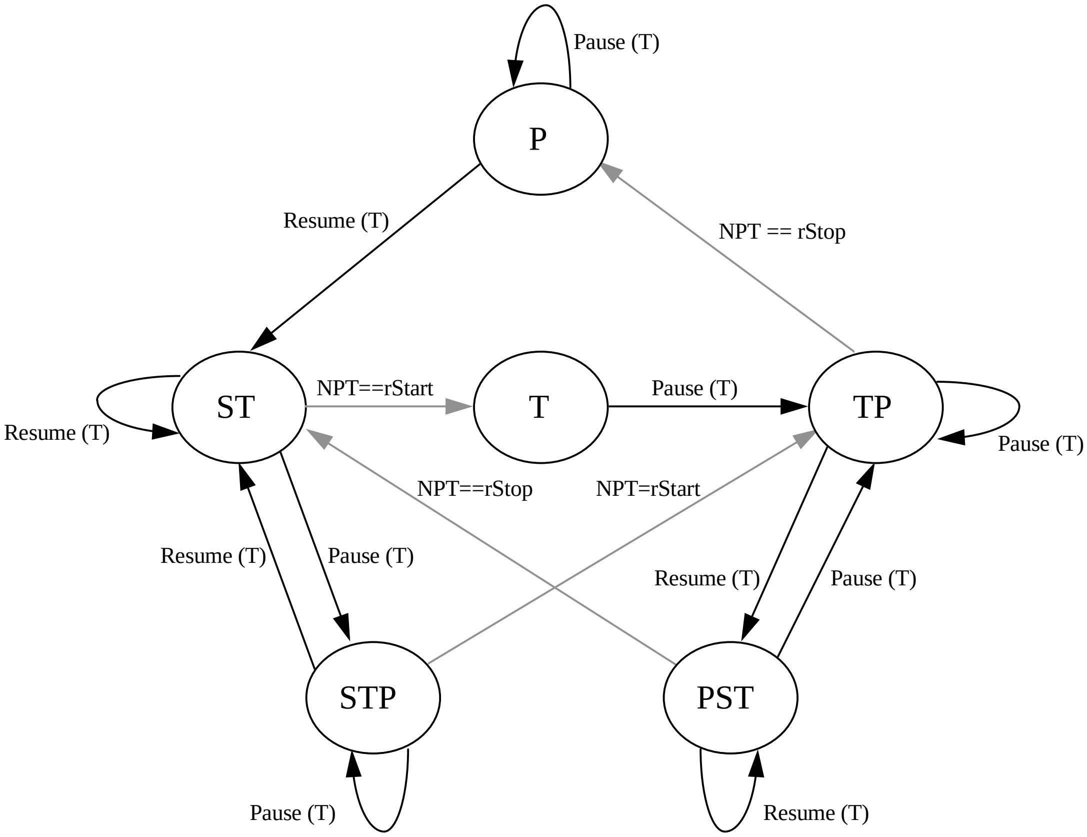
Previous State	rStart	rStop
O		
ST	T	
T		
TP		P
P		
STP	TP	P*
PST	T*	ST

If the state was SearchTransport, the machine transitions to Transport when the stream position reaches the rStart. If the state was SearchTransportPause, the machine transitions to TransportPause when the stream position reaches the rStart. If the state was TransportPause, the machine transitions to Pause when the stream position reaches the rStop. If the state was PauseSearchTransport, the machine transitions to SearchTransport when the stream position reaches the rStop.

The two states with the asterisk relate to exceptions. If the state is SearchTransportPause, the stream object expects to reach the rStart before the rStop. If the stream object encounters the rStop first, the machine transitions to the Pause state. If the state is PauseSearchTransport, the stream object expects to encounter the rStop before the rStart. If the stream object encounters the rStart first, the machine transitions to the Transport state.

5.5.4.4.3 Simple Stream Interface

The interface below realizes the identical state machine but with just the Reset(), Resume() and Pause() methods. The interface does not include Play(rStart, rStop) because the function is identical to the sequence Resume(rStart) plus Pause(rStop). The interface does not include Jump(rStop, rStart) because the function is identical to sequence Pause(rStop) plus Resume(rStart). Since the functions are redundant, the interface is just as expressive as the more complex interface. Since the client can cascade methods, there is no compromise in performance.



The state machine is shown below. Note that the table is identical (for columns which apply to both) to the state machine with Play(rStart, rStop) and Jump(rStop, rStart) functions.

Previous State	Reset()	Resume()	Pause()
O	O	ST	P
ST	O	ST	STP
T	O	ST	TP
TP	O	PST	TP
P	O	ST	P
STP	O	ST	STP
PST	O	PST	TP

The table below shows the transitions which the rStart condition and the rStop condition cause. Again the table is identical (for the columns which apply to both) to the state machine with Play(rStart, rStop) and Jump(rStop, rStart) functions.

Previous State	rStart	rStop

O		
ST	T	
T		
TP		P
P		
STP	TP	P*
PST	T*	ST

5.5.4.5 DSM Stream Pause

DSM Stream Pause	Stop sending stream when NPT position is reached.

Client-Service IDL Syntax

```
module DSM {
     interface Stream : Base, Access {
            void Pause (
                        in NPT rStop)
                        raises (MPEG_DELIVERY, BAD_STOP, QUE_LIMIT);
     };
};
```

Semantics

A client calls **Stream Pause** to cause the video server to stop sending the stream when it reaches **NPT rStop**.The actual presentation of video frames (freeze frame versus blanked or alternative display) is considered implementation-specific and is therefore not specified.

If the stream is in forward transport mode, either an **rStop** of negative infinity or an **rStop** less than the current **NPT** will indicate pause immediately if there are no other commands in the stream state machine queue. If the stream is in reverse transport mode, either an **rStop** of positive infinity or an **rStop** greater than the current **NPT** will indicate pause immediately if there are no other commands in the stream state machine queue.

Privileges Required
READER

Parameters

type/variable	direction	description
NPT rStop	input	NPT position at which the Pause will occur.

5.5.4.6 DSM Stream Resume

DSM Stream Resume	Start sending stream at NPT position.

Client-Service IDL Syntax

```
module DSM {
     interface Stream : Base, Access {
            void Resume (
                    in NPT rStart,
                    in Scale rScale)
                    raises (MPEG_DELIVERY, BAD_START, BAD_SCALE, QUE_LIMIT);
    };
};
```

Semantics

A client calls **Stream Resume** to cause the video server to resume sending the stream at **rStart**.

rScale is composed of a numerator and a denominator. An **rScale** of 1/1 indicates normal play at the normal forward viewing rate. It is recommended for efficiency that either the numerator or denominator have a value of 1. The ratio of numerator to denominator corresponds to the rate with respect to normal viewing rate. For example, a ratio of 2/1 indicates 2 times the normal viewing rate, and a ratio of 1/2 indicates one-half the normal viewing rate. The server will respond with best effort, that is at the closest rate to the requested rate that it can deliver. The **rScale** reply will indicate the actual rate delivered. A positive numerator indicates forward direction. A negative numerator indicates reverse direction. Either a numerator or denominator of 0 will result in a **BAD_SCALE** exception. If the Stream is in transport mode and a Stream Resume is sent which causes an exception, the stream will continue as if the operation causing the exception had not occurred. An rStart which equals or exceeds the Stream duration will result in a **BAD_START** exception.

Privileges Required
READER

Parameters

type/variable	direction	description
NPT rStart	input	NPT position at which to resume.
Scale rScale	input	The scale at which to resume. A numerator / denominator indicating rate and direction. A negative numerator indicates reverse direction, whereas a positive numerator indicates forward direction. 1/1 indicates normal play speed.

5.5.4.7 DSM Stream Status

DSM Stream Status	Obtain status of a stream.

Client-Service IDL Syntax

```
module DSM {
    interface Stream : Base, Access {
            void  Status (
                    in Stat rAppStatus,
                    out Stat rActStatus)
                    raises (MPEG_DELIVERY);
    };
};
```

Semantics

Stream Status is used to request status of a stream in progress. It returns the current **NPT** position, scale and mode of the stream. The application's estimation of current position may be specified in the call request. The reply will contain the actual position If mode is in an unknown state (i.e., there is an error) an exception will be returned.

Privileges Required
READER

Parameters

type/variable	direction	description
Stat rAppStatus	input	Expected stream status, NPT, Scale and Mode
Stat rActStatus	output	Actual stream status, NPT, Scale and Mode.

5.5.4.8 DSM Stream Reset

DSM Stream Reset	Reset the queue of pending operations on a stream (READER)

Client-Service IDL Syntax

```
module DSM {
    interface Stream : Base, Access{
          void Reset ();
    };
};
```

Semantics

Stream Reset is used to reset the Stream state machine. Since Stream operations may be queued in advance (the maximum queue depth is 2), it is possible that the application will queue some operations which are later determined to be incorrect. With Reset, the client can pull the pending operations from the queue without interrupting the current operation in progress for the Stream.

Privileges Required
READER

5.5.4.9 DSM Stream Jump

DSM Stream Jump	When stream reaches stop NPT, resume at start NPT.

Client-Service IDL Syntax

```
module DSM {
    interface Stream : Base, Access{
          void   Jump (
                    in NPT rStart,
                    in NPT rStop,
                    in Scale rScale)
                    raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE,
                                    QUE_LIMIT);
    };
};
```

Semantics

A client calls **Stream Jump** to cause the server to commence sending the stream at the **NPT** indicated by **rStart**, to occur when the stream reaches the **NPT** indicated by **rStop**. Stream commands may be overlapped to a queue depth of two. For example, **Stream Jump** may be followed by a **Stream Pause**, causing the Jump to change mode to Pause immediately after the Jump. Scale will indicate either normal play, scan forward or scan reverse by its value.

If a Play immediately follows a Jump, the play supersedes the **rStop** of the Jump.

Privileges Required
READER

Parameters

type/variable	direction	description
NPT rStop	input	NPT position at which the Jump will occur.
NPT rStart	input	NPT position to resume from as a result of the Jump.
Scale rScale	input	A numerator / denominator indicating rate and direction. A negative numerator indicates reverse direction, whereas a positive numerator indicates forward direction. 1/1 indicates normal play speed.

5.5.4.10 DSM Stream Play

DSM Stream Play	Play stream from start NPT until stop NPT.

Client-Service IDL Syntax

```
module DSM {
     interface Stream : Base, Access{
            void  Play (
                        in NPT rStart,
                        in NPT rStop,
                        in Scale rScale)
                        raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE,
                                        QUE_LIMIT);
     };
};
```

Semantics

A client calls **Stream Play** to cause the server to transmit the MPEG stream immediately at the stream **NPT** specified by **rStart**.

Play mode shall be set to normal speed, fast forward or reverse as indicated by **rScale**. If **rStart** is 0, playback will commence at the beginning of the stream. If a Jump immediately follows a Play, the Jump supersedes the **rStop** of the Play.

Privileges Required
READER

Parameters

type/variable	direction	description
NPT rStart	input	NPT position at which to resume Play.
NPT rStop	input	NPT position at which to stop Play and change mode to Pause.
Scale rScale	input	A numerator / denominator indicating rate and direction. A negative numerator indicates reverse direction, whereas a positive numerator indicates forward direction. 1/1 indicates normal play speed.

5.5.4.11 DSM Stream Next

DSM Stream Next	De-activate a video.

Client-Service IDL Syntax

```
module DSM {
    interface Stream : Base, Access {
          void  Next (
                    in Stream rNextStream)
                    raises (INV_NEXTREF, QUE_LIMIT);
    };
};
```

Semantics

The **Stream Next** will link the completion of a current stream to the start of a next stream. Both streams must be open prior to making this call.

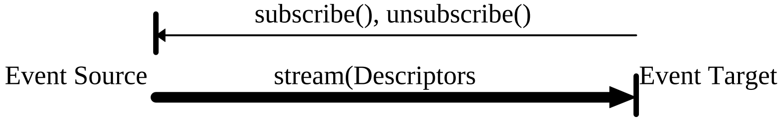
Privileges Required
READER

Parameters

Stream rNextStream	input	Reference of the successor Stream object, as returned by a Directory Open.

5.5.5 Event

The event interface in the Common Object Services specification of the Object Management Group is the foundation on which the basic event interface was built. The event interface, however, differs in two respects. First the interface packages the functions, which scatter across multiple interfaces in the Object Management Group design, into a single interface. Second, the audience for the interface is a client, such as the settop device, which receives the media stream. The stream object, to be specific, distributes the event data through the media stream.



The figure above shows the concept. The request to subscribe the events is an interface, cast as Interface Definition Language. The event distribution, however, is over the media stream as descriptor data found in a private data section.

The motivation for the stream event interface is the situation where the service and the client both understand the semantics of certain events which correlate to the media stream. The client invokes the Subscribe() function to express interest in the event. The client provides the event name, which is a simple string. The stream object returns a token which uniquely identifies the event.

The stream object at some point places the data found in the interface declaration into the media stream near the companion media data. Note that the data includes both the token and the time to which the event relates. The inclusion of the time value allows the client to schedule the reaction to the event to correlate with the presentation of the media stream.

5.5.5.1 Event Definitions, Exceptions

```
module DSM {
    interface Event {
        // In addition to the other descriptor fields, the stream object places the
        // StreamEvent in the private data section of the media stream:
        const u_short NULL_EVENT_TOKEN = 0;
        struct StreamEvent {
                u_short aEventToken;
                s_long aSeconds;
                u_long aSubSeconds;
                sequence<octet> OpaqueData;};
        exception INVALID_EVENT_NAME {string aEventName;};
        exception INVALID_EVENT_TOKEN {u_short EVENT_TOKEN;};
    };
};
```

The constant declaration captures the convention that if the token field of the descriptor data is the value zero, it is understood that the event data which follows is bogus. If the event trigger time is maximum negative value, the semantics are to immediately respond to the event.

While not a remote service interface, the section of the document which deals with the application perspective describes an method which returns the event data to the client application code. Since the event data returns through the media stream, the interface is outside the scope of the remote service interface specification.

5.5.5.2 Summary of Event Primitives

Defined in **Event**:

DSM Event Subscribe	Subscribe to receive an event over an MPEG stream. (READER)
DSM Event Unsubscribe	Indicate desire to no longer receive an event. (READER)

```
module DSM
     interface Event{
            const AccessRole Subscribe_ACR = READER;
            const AccessRole Unsubscribe_ACR = READER;
     };
};
```

5.5.5.3 DSM Event Subscribe

DSM Event Subscribe	Subscribe to receive an event over an MPEG stream. (READER)

Client-Service Interface Syntax

```
module DSM {
     interface Event {
            void Subscribe(
                        in string aEventName,
                        out u_short aEventToken)
                        raises(INVALID_EVENT_NAME);
     };
};
```

Semantics

The client invokes **Event Subscribe** to request that the specified event be sent when it occurs. The client provides the event name. (The mechanism through which the client discovers the event name space is outside the scope of this interface.) The service returns an event token to associate with the event name. The scope of the token is at least the media stream. The client, in other words, should not find duplicate tokens in the same stream. The exception relates to the situation where the client provides an event name which the service does not recognize.

Privileges Required
READER

Parameters

type/variable	direction	description
string aEventName	input	The symbolic name of the event.
u_short aEventToken	output	The token which the service assigns, and which the client should associate, with the event.

5.5.5.4 DSM Event Unsubscribe

DSM Event Unsubscribe	Indicate desire to no longer receive an event. (READER)

Client-Service Interface Syntax

```
module DSM {
    interface Event {
        void Unsubscribe(
                in u_short aEventToken)
                raises(INVALID_EVENT_TOKEN);
    };
};
```

Semantics

The client invokes **Event Unsubscribe** to instruct the service to not generate the event. The client provides the event token to describe the subscription to which the operation refers. The token with respect to the client becomes stale. The service can assign the token to other subscription requests. The exception relates to the situation where the client provides a bogus token, for example a token which was valid but is now stale.

Privileges Required
READER

Parameters

type/variable	direction	description
u_short aEventToken	input	The token which identifes the previous subscription.

5.5.6 File

This section describes the interface for two operations that read and write files. When combined with the other Core DSM interfaces such as OMG **NamingContext**, **Directory**, **Base**, and **Access**, a minimal file system interface is realized. When combined with Extended DSM interfaces such as **Lifecycle** and **View**, a more complete file system interface is realized. In the spirit of OMG, the IDL permits the server implementation to map to any of a variety of heterogeneous object and file systems,.

Multimedia access is largely read-only. Certain clients (e.g., settops) will implement only READER operations, in which case basic access to remote files is achieved by **File Read** and **Directory Get** primitives.

Write operations such as **File Write** and **Directory Put** primitives may be invoked only by authorized WRITERs.

The video network service presents a unique communication environment, with both asymmetric and symmetric data paths. The upstream or request data path can have significant restrictions in terms of bandwidth, reliability and latency. Within this environment there is a need to provide sequential data file service to the client. Further, there is a requirement that the file service provide high performance, even in the face of the communication restrictions, in particular, limited bandwidth request path and long round-trip latency. Finally, the client-side library must be compact in order to be used in limited-memory conditions such as are found in settop units.

5.5.6.1 File Definitions, Exceptions

```
module DSM {
    interface File : Base, Access {
        exception INV_OFFSET ExceptUser;            //size + offset exceeds file size
        exception INV_SIZE ExceptUser;              //size exceeds file size
    };
};
```

5.5.6.2 Summary of File Primitives

Inherited from **Base**:

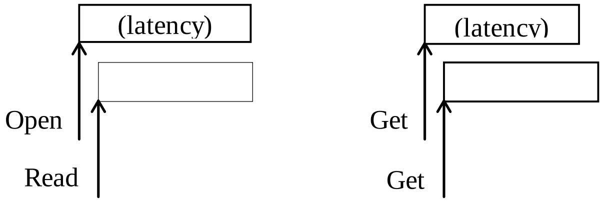
IsA, Close, Destroy

Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Defined in **File**:

DSM File Read	Random access read from a file. (READER)
DSM File Write	Random access write to a file. (WRITER)

```
module DSM
    interface File : Base, Access {
        const AccessRole Read_ACR = READER;
        const AccessRole Write_ACR = WRITER;
    };
};
```



The above drawing shows optimizations specifically made in response to video network requirements. The File interface is therefore well-suited to network environments where the round-trip latency is high and request bandwidth is low. **File Read** and **Directory Get** allow a large data return per request with a choice of retry option in the underlying RPC. **Directory Open** (of a file object) and **File Read** may be pipelined, where the **Directory Open** is immediately followed by a **File Read** as described in the application portability interface “Application Synchronous Deferred Operations.” All file reads may be overlapped as synchronous deferred requests. The RPC must assure that these requests are executed in order at the Server. The application may provide a reliable transfer hint to the underlying RPC to ignore errors for certain types of multimedia data, e.g., images, short audio. **Directory Get** will return the entire file.

5.5.6.3 DSM File Read

DSM File Read	Random access read from a file. (READER)

Client-Service IDL Syntax

```
module DSM {
    interface File : Base, Access {
        void Read (
                in u_longlong aOffset,
                in u_long aSize,
                in boolean aReliable,
                out ObjData rData)
                raises (INV_OFFSET, INV_SIZE);
    };
};
```

Semantics

File Read provides random access to opened files, using a File reference obtained from a previous **Directory Open**. Because offset and size are explicit parameters, seeks can be accomplished assuming the application maintains the current byte position in the file.

In the case where the network imposes a long round-trip latency, efficient operation of multimedia object access requires that the underlying RPC and network protocol stack support overlapped, synchronous deferred transactions. The application will need to prefetch files in an attempt to stay ahead of the anticipated user actions. The RPC must assure that the Server executes the operations for a client in the same order that the client has invoked them. The underlying RPC stack will retry the

If reliable delivery is set to FALSE for a **File Read** operation, the operation will not be retried in the event of timeout or error. This is useful in the case where media, e.g. short audio or image, is presented in fast-paced normal play application time, in which case it is more important for the presentation to move forward on schedule than to stall while an object is being refetched.

Privileges Required
READER

Parameters

type/variable	direction	description
u_longlong aOffset	input	64 bit value indicating starting byte position within the file.
u_long aSize	input	Number of bytes to read.
boolean aReliable	input	If aReliable = FALSE, client indicates that the RPC reply need not be reliable, e.g., for use with multimedia data for transient presentations.
ObjData rData	output	Pointer to data returned by the File Read.

5.5.6.4 DSM File Write

DSM File Write	Random access write to a file. (WRITER)

Client-Service IDL Syntax

```
module DSM {
    interface File : Base {
          void  Write (
                     in u_longlong aOffset,
                     in u_long aSize,
                     in ObjData rData)
                     raises (INV_OFFSET, INV_SIZE, WRITE_LOCKED);
    };
};
```

Semantics

File Write provides a mechanism to write data to a file starting at a designated offset. WRITER privileges are required. The **File Write** uses the File reference obtained from a previous **Directory Open**. Appends may be performed by using the size of the file as **aOffset**. **Size** is an exported attribute of the Access Interface and may be obtained through a **Directory Get** operation.

As a general rule, files which can be writable by settop clients should be kept in a separate service from services that contain files which are read-only by settop clients,. The data size is limited by client to server bandwidth constraints.

Privileges Required
WRITER

Parameters

type/variable	direction	description
u_longlong aOffset	input	64 bit value indicating starting byte position within the file.
u_long aSize	input	Number of bytes to write.
ObjData rData	input	Pointer to data to be written.

5.5.7 ServiceGateway

A network session is established as a result of invoking ServiceGateway Attach. At this time the client is connected with the ServiceGateway's top directory. Depending on the PathSpec in ServiceGateway Attach, the client may also have opened a first service, e.g. a Download Service. While in the Session, the client may connect to additional services by sending Directory Open to the ServiceGateway., or disconnect from services by sending the appropriate Close to the Service.

ServiceGateway inherits Directory bind and unbind operations. **bind**, **bind_context**, **rebind**, **rebind_context** and **unbind** require MANAGER privileges to be invoked on the ServiceGateway

5.5.7.1 ServiceGateway Definitions, Exceptions

```
module DSM {
     // these exceptions are shared with other interfaces
     exception UNK_USER ExceptUser;                 // unknown user
     exception BAD_PROFILE ExceptUser;              // bad client profile format or contents
     exception NO_SUSPEND  ExceptUser;              // unable to suspend state
     exception NO_RESUME ExceptUser;  // unable to resume a previous session
     struct NetResource {
            u_long Id;                              // resource identifier
            sequence<any> ResourceParams;};
    typedef sequence<NetResource> NetResources;
};
```

5.5.7.2 Summary of ServiceGateway Primitives

Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Inherited from **Directory and NamingContext**:

**Open, Close, Get, Put,
list, resolve, bind, bind_context, rebind,
rebind_context, unbind, new_context,
destroy**

Defined in **ServiceGateway**:

DSM ServiceGateway Attach	Attach to a ServiceGateway domain of services. (READER)
DSM ServiceGateway Detach	Detach from a ServiceGateway domain of services. (READER)
DSM ServiceGateway ModResource	Service request to add/delete network resources. (BROKER)

```
module DSM
    interface ServiceGateway : Directory {
            const AccessRole bind_ACR = MANAGER;
            const AccessRole bind_context_ACR = MANAGER;
            const AccessRole rebind_ACR = MANAGER;
            const AccessRole rebind_context_ACR = MANAGER;
            const AccessRole unbind_ACR = MANAGER;
            const AccessRole Attach_ACR = READER;
            const AccessRole Detach_ACR = READER;
            const AccessRole ModResource_ACR = BROKER;
    };
};
```

5.5.7.3 DSM ServiceGateway Attach

DSM ServiceGateway Attach	Attach to a ServiceGateway domain of services. (READER)

Client-Service IDL Syntax

```
module DSM {
    interface ServiceGateway : Directory {
            void  Attach (
                        in ObjRef rClientRef,
                        in Profile rClientProfile,
                        in EndUser aEndUser,
                        in UserContext aSuspendContext,
                        in PathSpec rPathSpec,
                        out UserContext aResumeContext,
                        out PathRefs rPathRefs,
                        out DateTime rDateTime)
                        raises (NO_AUTH, BAD_PROFILE, UNK_USER, NO_RESUME,
                                OPEN_LIMIT, NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

ServiceGateway Attach specifies the parameters to be included in the **userData** fields of the User-Network Session Establishment messages. It may be used as an RPC only is systems where there is no DSM-CC User-to-Network signaling, otherwise the portability interface input parameters are placed in **userData** field of the U-N **ClientSessionSetupRequest**, and the portability interface output parameters are placed in **userData** field of the U-N **ClientSessionSetupResponse**.

LINEAR path traversal is used for this operation. Therefore PathType is not specified. A client reference **rClientRef** is provided which maps to header information used to uniquely address the client. A client profile **rClientProfile** is provided to identify characteristics of the client, e.g. model and version number, etc. An identification of a previously suspended user context **aSuspendContext** enables the client to indicate that an application is to resume from previously suspended state. If this is set to 0, it indicates the application is starting up for the first time, and will therefore receive an **aResumeContext** reply identifying the new used context for this run of the application. A path specification **rPathSpec** names the path to ServiceGateway and possibly a first service to open. The reply **rPathRefs** will contain references for the Steps in rPathSpec that have process = TRUE.

Privileges Required:
READER

Parameters

type/variable	direction	description
ObjRef rClientRef	input	The client object reference which uniquely identifies the requesting node to the ServiceGateway. This maps to the ObjKey over the network.
Profile rClientProfile	input	A string in TermCap format which includes configuration information from the Client Configuration Information Element.
EndUser aEndUser	input	Contains the EndUser system-wide identifier
UserContext aSuspendContext	input	A user context identifier. A value of 0 indicates the client wishes to establish a new user context. A value > 0 indicates the client wishes to reconnect to a previously suspended user context.
PathSpec rPathSpec	input	A Sequence of Steps, representing a path to the ServiceGateway and possibly to a service.
UserContext aResumeContext	output	The user context assignment for this client to Service-Gateway session.
PathRefs rPathRefs	output	The object references opened as a result of this attach, e.g. ServiceGateway and a service. Each ObjRef maps to a corresponding ObjKey over the network.
DateTime rDateTime	output	ANSI-C standard holds date and time broken down into their elements. Local time of the ServiceGateway.

5.5.7.4 DSM ServiceGateway Detach

DSM ServiceGateway Detach	Detach from a ServiceGateway domain of services. (READER)

Client-Service IDL Syntax

```
module DSM {
    interface ServiceGateway : Directory {
          void  Detach (
                    in boolean aSuspend)
                    raises (NO_SUSPEND);
    };
};
```

Semantics

A client/end user may invoke **ServiceGateway Detach** to disconnect from a ServiceGateway. This will effectively remove the client from a Session. If **aSuspend** is true, the ServiceGateway will inform Services which are maintaining user context for this end user to save state for a possible resumption of those

Services. It is up to the application to determine how state will be saved and maintained between Sessions. The client may later invoke **ServiceGateway Attach** with a **UserContext** identifier equivalent to current **UserContext** (prior to the Detach), in order to Resume from the saved state.

Invoking **ServiceGateway Detach** will result in a User-to-Network Session Teardown sequence. The User-to-Network UserData consists of the arguments of this primitive.

Privileges Required:
READER

Parameters

type/variable	direction	description
boolean aSuspend	input	Direction to application as to whether to maintain state after the service connection is broken.

5.5.7.5 DSM ServiceGateway ModResource

DSM ServiceGateway ModResource	Service request to add/delete network resources. (BROKER)

Client-Service IDL Syntax

```
module DSM {
    interface ServiceGateway : Directory {
        void ModResource (
                in u_long aUserContext, // which maps to a Network Session
                // requested resources
                in NetResources rReqResources,
                // resources granted
                out NetResources rActResources);
    };
};
```

Semantics

If a Service receives a request from a Client that requires a change in Network resources between the Service and the Client, it can invoke **ServiceGateway ModResource** to initiate the change. rReqResources is used to identify a list of Resources and associated parameters. rActResources identifies the Resources and parameters that were granted.

Privileges Required:
READER

Parameters

type/variable	direction	description
ServiceGateway object	input	A reference to a DSM ServiceGateway object.

u_long aUserContext	input	Identification of a UserContext
NetResources rReqResources	input	Requested Resources.
NetResources rActResources	output	Resources Granted.

5.6 Extended Interfaces

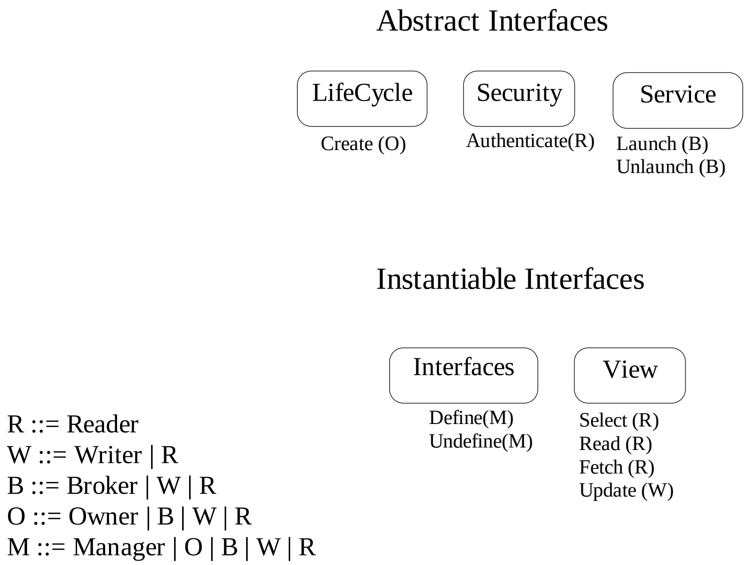
The DSM-CC Extended interfaces are optional. Each of these interfaces may be implemented at the discretion of the Service Provider. In a CORBA server environment, the equivalent of LifeCycle Create may be defined individually by each object, as is done in CosNaming. The DSM LifeCycle Create is offered as a means of having a common function that is inherited and recognized in DSM systems. In a non-CORBA system, LifeCycle Create assures uniqueness of object references.

Likewise, in a CORBA server environment, there is an Interface Repository where new interfaces are defined. In a CORBA server environment, the Interfaces object is not needed, whereas in a non-CORBA system, the Interfaces operations are used to verify interface definitions and to assure uniqueness of interface types.

The Security interface may not be required if passwords or encrypt keys are never exchanged.

The Service interface is not needed for small systems where ServiceGateway is the broker for all resolves. For scalable systems, the Service interface is used to enable the ServiceGateway to broker a connection to a service on a physically separate server, and to enable Services to act as brokers and make decisions based on client Profile and identification.

The View interface is not required if none of the system’s clients (providers and consumers) has need for sorting or filtering Directory information or accessing a database.



5.6.1 View

Multimedia client-server applications using MPEG for audio, video and file access also have a need for viewing information in the perspective of the end user, as opposed to how the information is stored at the server. The View primitives provide operations for sorting and filtering data such that directories and database information can be presented to the user in a more palatable form.

Using the View interface, the relational model can be applied to objects in directories. The View Type in this case is **NON_DB**, meaning the directory is not a database. The objects' exported attributes and their associated values are used in a **View Select** query to produce a sorted and filtered result set. The result set can then be browsed using **View Read**.

Alternatively, the View interface can be applied to an actual database at the server. The View Type for this case is either **SQL89, SQL92, or SQL3**. **View Select** and **View Read** are again used by the calling application to retrieve database attributes. Note that name for SQL3 may change as that standard nears completion.

For all View Types, the SQL language syntax is used as the basic query form.

View Fetch is provided which will return a window of results, as opposed to a single result row or attribute object. A **View Read** from the calling application can result in a **View Fetch** which will prefetch results in anticipation of the cursor location in the next **View Read**.

Using the application portability interface, **Directory Open** can be pipelined with **View Select** and **View Fetch** in deferred synchronous mode, resulting in the pipelined execution of the operations.

The overlapped execution and local results caching overcomes a potentially significant response time issue in long-latency networks.

5.6.1.1 Non-Database View

The View interface can be used as an extension of the Directory Interface to enable searching, sorting and filtering of Directory objects, using a minimal SQL set. A NON_DB View type indicates that the View is not a Database, but does support limited SQL queries against a container of objects, e.g., a Directory.

The result set from the view contains temporary attribute objects which are browsable by the client. For example, a client can sort objects by the **Access Size** attribute, using view.

The SQL set supported by a NON_DB ViewType is as follows:

SELECT, as defined in SQL92, with keywords (in order normally found):

ALL	is the default and specifies that all objects that satisfy the SELECT statement should be returned
FROM	indicates which object types to perform the query against
WHERE	specifies conditions
ASC	sort in ascending order
DESC	sort in descending order
GROUP BY	return summary information about groups of objects
HAVING	return summary information about groups of objects
ORDER BY	the order in which rows are returned
UNION	combine the results of two select statements
INTERSECT	combine the results of two select statements
MINUS	combine the results of two select statements

Conditions, i.e., [attribute operator value] combinations, as defined in SQL92 Some attributes are stored in structures. The query will specify attributes within structures in ANSI-C syntax, i.e., <attribute structure name>.<attribute name>. Operators in the query must compare a value to a basic type, e.g., an integer or string.

The following are strictly NOT allowed for NON_DB ViewType:

1. DISTINCT, since there are no duplicate objects within a name context
2. CONNECT BY, START WITH, since hierarchy is explicit through use of Directories
3. FOR UPDATE OF, since writes are not allowed through SQL on NON_DB View
4. NO WAIT, since locks may not be set with NON_DB View
5. plus any other non-SELECT statement

5.6.1.2 Database View

A View object may represent an actual database at the server. The View Types for a database are SQL89, SQL92 and SQL3. Each of these refers to a SQL standard. Based on the type, the syntax and semantics of that standard are applicable.

5.6.1.3 View Procedures

The following steps outline the query sequence:

1. The client application makes the **Directory Open** of a View object, followed immediately by a **View Select** with a SQL statement.

2. The DSM Library issues **Directory Open**, **View Select** and **View Fetch** RPCs in synchronous deferred mode, allowing them to be pipelined.

3. The RPC Server establishes the query, executes it, and creates a results area for all rows matched. The RPC Server fetches the initial set of result rows. In addition it marks which rows are to be returned.

4. The RPC reply sent to the client with a subset of the rows matched.

5. The rows returned from the **View Fetch** are stored in a local buffer at the client.

The following steps outline the browsing sequence:

1. The client obtains the initial set of rows or objects from **View Fetch** in its local buffer, as described above.

2. The client perform can obtain a row by invoking **View Read** with cursor value that points into the matching result set at the Server. The DSM Library will invoke the remote interface **View Fetch** as needed or to prefetch a window of rows in anticipation of further Reads.

3. Finally the client issues **View Close** is used to close the View and the query.

5.6.1.4 View Definitions, Exceptions

```
module DSM {
interface View {
        // ViewType identifies the Query set supported by the View
        // NON_DB indicates service is not a Database but performs minimal
        // searches, filters and sorts using SELECT as described in the DSM spec
        // SQL89 indicates the View is a SQL89-compliant database
        // SQL92 indicates the View is a SQL92-compliant database
        // SQL3 indicates the View is a SQL3-compliant database
        const char NON_DB = ‘N’;
        const char SQL89 = ‘1’;
        const char SQL92 = ‘2’;
        const char SQL3 = ‘3’;    // this is reserved for SQL3
        attribute char Type;
        const AccessRole Info_Get_ACR = READER;
        const AccessRole Info_Put_ACR = OWNER;
        typedef string SQLStatement;
        struct ResultDescribe {
                s_short aNumberOfColumns;       //number of attributes or columns
                s_short aPrecision;             //count of digits in numbers
                s_short aScale;                 //count of digits to right of decimal point
                boolean aNullOK; };             //NULL OK in numbers
        typedef sequence<any> BufDescribe;      //describes type, length of each attribute
        exception BUF_TOO_SMALL ExceptUser; //local buffer space requested is too small
        exception BUF_TOO_BIG ExceptUser;    //local buffer space requested is too big
        exception BAD_SYNTAX ExceptUser;                 //illegal SQL syntax
        exception INV_CURSOR ExceptUser;                 //cursor out of bounds
    };
};
```

5.6.1.5 Summary of View Primitives

Defined in **View**:

DSM View Select Execute a SQL read statement. Fetch an initial set of result objects to client buffer space. (READER)

DSM View Read Obtain the attributes of a single row or object.

DSM View Fetch Fetch additional result rows in the context of a View Select query. (READER)

DSM View Update Execute a SQL write statement. (WRITER)

```
module DSM
    interface View {
        const AccessRole Select_ACR = READER;
        const AccessRole Read_ACR = READER;
        const AccessRole Fetch_ACR = READER;
        const AccessRole Update_ACR = WRITER;
    };
};
```

5.6.1.6 DSM View Select

DSM View Select	Execute a SQL read statement. Fetch an initial set of result objects to client buffer space.

Client-Service IDL Syntax

```
module DSM {
    interface View {
        void Select (
                in u_long aBufSize,                     // app memory allocated
                in SQLStatement rSQLStatement,
                out ResultDescribe rResultDescribe,     // describes full result
                out ObjData rReturnBuffer)              // memory location for fetch to use
                raises (BAD_SYNTAX, BUF_TOO_SMALL, BUF_TOO_BIG);
    };
};
```

Semantics

View Select sends the SQL statement specified by rSQLstatement to the View object for execution. **aBufSize** defines the size of the local cache **rReturnBuffer**. **rResultDescribe** contains a description of the fields in the data returned in **rReturnBuffer**. **rReturnBuffer** specifies the location of the local results cache, where the DSM Library can prefetch result rows.

The client will allocate the buffer space for **rReturnBuffer** based on available memory it has at the time of invocation. The server will honor this size by not returning a sequence larger than the buffer space allocated. If the number of a single attribute value to be returned plus the overhead of _maximum and _length parameters of the sequence exceeds **aBufSize**, a **BUF_TOO_SMALL** exception will be returned. If the buffer size allocated exceeds the buffer space capability of the server, a **BUF_TOO_BIG** exception will be returned.

Privileges Required
READER

Parameters

type/variable	direction	description
u_long aBufSize	input	The client buffer space allocated for rReturnBuffer
SQLStatement rSQLStatement	input	Standard SQL statement, subject to restrictions of the View Type.
ResultDescribe rResultDescribe	output	Description of return fields.
ObjData rReturnBuffer	output	Results buffer of rows and fields.

5.6.1.7 DSM View Read

DSM View Read	Obtain the attributes of a single row or object.

Client-Service IDL Syntax

```
module DSM {
    interface View {
         void Read (
                 in u_short aCursor,                // pointer into full result list
                 out BufDescribe rObjDescribe,      // describes attributes of the row or object
                 out ObjData rRow)                  // block of one row/object's data values
                                          // described by rObjDescribe
                 raises (INV_CURSOR);
    };
};
```

Semantics

View Read is used to obtain the attributes of one object, or read a result row after a Select. aCursor identifies an object in the list of objects, either in the name context, or in the Select full result.

The Read can be done independent of a Select. If Read is done without a prior Select, then all objects of the name context are in the result set and a Remote interface Fetch is not performed. If Read follows a Select, the Cursor refers to the result created by the Select, and the DSM Library performs Fetch to prefetch a window of result rows in anticipation of the next Read.

Privileges Required
READER

Parameters

type/field	direction	description
u_short aCursor	input	Index into the full result of a Select or the list of objects in a name context.
BufDescribe rObjDescribe	output	Description of the object's attributes or the result row fields.
ObjData rRow	output	The Object's attribute values or row field values.

5.6.1.8 DSM View Fetch

DSM View Fetch	Fetch additional result rows in the context of a View Select query.

Client-Service IDL Syntax

```
module DSM {
    interface View {
        void Fetch (
                in u_short aCursor,
                out BufDescribe rObjDescribe,    // describes attributes per object row
                out ObjData rReturnBuffer)       // includes multiple rows or objects
                raises (INV_CURSOR);
    };
};
```

Semantics

View Fetch is used to fetch multiple result rows, if applicable, from the server after the successful execution of a **View Select**. Results are placed in the same buffer as the original **View Select**.

Privileges Required
READER

Parameters

type/variable	direction	description
u_short aSQLcurser	input/output	Cursor location within View query context.
BufDescribe rObjDescribe	output	Description of the object's attributes or the result row fields.
ObjData rReturnBuffer	output	The Object's attribute values or row field values.

5.6.1.9 DSM View Update

DSM View Update	Execute a SQL write statement.

Client-Service IDL Syntax

```
module DSM {
    interface View {
          void Update (
                   in SQLStatement rSQLStatement)
                   raises (BAD_SYNTAX);
    };
};
```

View Update sends the SQL statement specified by rSQLstatement to the View object for execution of SQL inserts, deletes, and updates.. **View Select** returns Success upon successful execution of the SQL statement string, else it will return an error indicating the completion status.

Use of **View Update** is not permitted for the **NON_DB** View Type. For NON_DB Views, Directory commands such as **bind** and **unbind** should be used instead.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
SQLStatement rSQLStatement	input	Standard SQL statement, subject to restrictions of the View Type.

5.6.2 Service

The Service is an entity that provides function(s) and interface(s) in support of applications. Services will typically include interfaces defined in this standard and may optionally extend the interface by defining new operations not covered by DSM. Services register with the ServiceGateway using the ServiceGateway bind operation(s). In the context of a Session, a Client requests connection to a Service through **Directory Open** (specifying service name)sent to the ServiceGateway. After authorization and resolution of the Service instance, the Client becomes associated with a Service through **Service Launch** from the ServiceGateway.

5.6.2.1 Summary of Service Primitives

DSM Service Launch	Activate an end user context for a service object. (BROKER)
DSM Service Unlaunch	Deactivate or suspend an end user context for a service object. (BROKER)

```
module DSM
    interface Service {
          const AccessRole Launch_ACR = BROKER;
          const AccessRole Unlaunch_ACR = BROKER;
    };
};
```

5.6.2.2 DSM Service Launch

DSM Service Launch	Activate an end user context for a service object. (BROKER)

Remote Interface IDL Syntax

```
module DSM {
   interface Service {
         void Launch  (
                  in ObjKey  rClientRef,     // unique identification of client over the network
                  in Profile rClientProfile,              // profile of current client configuration
                  in EndUser aEndUser,                    // identification of end user
                  in UserContext aUserContext,      // identification of application user context
                  in boolean aResume,                     // resume from previous state
                  in NetResources rNetResources,  // network resources allocated
                  out ObjKey  rServerRef)  // unique identification of server over the network
                  raises (UNK_USER, BAD_PROFILE, NO_RESUME);
   };
};
```

Semantics

A ServiceGateway may send **Service Launch** to a Service to activate a client-service connection in response to an **Open** (with specified Service Name) from the Client. This will allow the Client and Service to begin communication. It will also pass key information about the client and the connection to the Service, including unique network reference of the Client, Client Profile, an application user context, whether the end user wishes to resume a previous user context, and initial network resources assigned. The **Service Launch** reply contains the service's unique network reference, which is then passed back to the Client in the reply to the **Open**.

If, in the course of client-service interaction, the initial network resources assigned need to be modified, e.g., if bandwidth requirements change, the Service can use **ServiceGateway ModResource** to accomplish this.

Privileges Required:
BROKER

Parameters

type/variable	direction	description
ObjKey rClientRef	input	Unique identification of the client over the network.
Profile rClientProfile	input	Profile of information about the client's configuration.
EndUser aEndUser	input	Contains the EndUser system-wide identifier
UserContext aUserContext	input	User Context for this application run.
boolean aResume	input	If TRUE, resume using state from a previous UserContext. If FALSE, use initial state.
NetResources rNetResources	input	Initial network resources that the ServiceGateway has negotiated for this connection, e.g. pipes and bandwidth.
ObjKey rServerRef	output	Object reference chosen by the Service broker.

5.6.2.3 DSM Service Unlaunch

DSM Service Unlaunch	Deactivate or suspend a user context for a service object. (BROKER)

Client-Service IDL Syntax

```
module DSM {
    interface Service {
          void Unlaunch  (
                    in ObjKey  rClientRef,    // unique identification of client over the network
                    in EndUser aEndUser,                // identification of end user
                    in UserContext aUserContext,      // identification of application user context
                    in boolean aSuspend)      // suspend for later resume
                    raises (NO_SUSPEND);
    };
};
```

Semantics

A BROKER may use **Service Unlaunch** to remove an end user client context from a Service. If **aSuspend** is true, the Service is instructed to save state for a possible resumption of that Service for that client. It is up to the application to determine how it will save state.

Privileges Required:
BROKER

Parameters

type/variable	direction	description
ObjKey rClientRef	input	Unique identification of the client over the network.
EndUser aEndUser	input	Contains the EndUser system-wide identifier.
UserContext aUserContext	input	User Context for this application run.
boolean aSuspend	input	Direction to application as to whether to maintain state after the service connection is broken.

5.6.3 Interfaces

The DSM application space is constructed as a name space graph starting at the ServiceGateway. The nodes represent objects of the various types specified by DSM as well as additional types that may be implementation specific.

The Directory primitives provide browsing functions to traverse the graph. Each node has a minimum of a name and type. The node may have other browsable information such as version and date, providing the definition of the type exports these attributes. Each object type has an exported interface, which is defined through **Interfaces Define**.

5.6.3.1 Interfaces Definitions, Exceptions

```
module DSM {
    // these interface types pre-defined and reserved by DSM
    typedef u_long IntfType;
    const u_long BASE_T = 0;
    const u_long ACCESS_T = 1;
    const u_long EVENT_T = 2;
    const u_long NAMING_CONTEXT_T = 3;
    const u_long STREAM_T = 4;
    const u_long FILE_T = 5;
    const u_long DIRECTORY_T = 6;
    const u_long SERVICEGATEWAY_T = 7;
    const u_long LIFECYCLE_T = 8;
    const u_long INTERFACES_T = 9;
    const u_long SECURITY_T = 10;
    const u_long SERVICE_T = 11;
    const u_long VIEW_T = 12;
    //
     interface Interfaces {
            typedef sequence<octet, 1024> ReferenceData;
            typedef string InterfaceDef;
            exception INV_INTERFACE ExceptUser;          //invalid interface definition
     };
};
```

5.6.3.2 Summary of Interfaces Primitives

DSM Interfaces Define	Define an object interface to the System (MANAGER)
DSM Interfaces Undefine	Remove an object interface definition from the System. (MANAGER)

```
module DSM
     interface Interfaces {
            const AccessRole Define_ACR = MANAGER;
            const AccessRole Undefine_ACR = MANAGER;
     };
};
```

5.6.3.3 DSM Interfaces Define

DSM Interfaces Define	Define an object interface to the System (MANAGER)

Client-Service IDL Syntax

```
module DSM {
   interface Interfaces {
            void Define (
                        in ReferenceData id,                    // unique Identifier
                        in InterfaceDef intf,                   // IDL definition
                        // how to use version is implementation-specific
                        out Version rVersion,                   // return new version if one exists
                        out IntfType aIntfType)                 // return system-wide object type
                        raises (INV_INTERFACE);
            };
};
```

Semantics

Interfaces Define is used by an MANAGER to define an interface of an object to the system. The interface definition specifies the exported interface of the object, i.e., exported methods and attributes. The interface definition may ‘include’ other interfaces to allow new interfaces to extend the functionality of existing interfaces. The object type is specified in the IDL interface definition. If the client is redefining an existing interface, the existing type and new version are returned. If the client is defining a new interface, a new type and initial version are returned.

The interface definition can specify the **AccessRole** for each method, as well as the **Get AccessRole** and **Put AccessRole** for each exported attribute. If these are not specified, the **AccessRole** defaults to OWNER.

Following the **Define**(), the object type may be used in **Create**() to produce an object reference of a known interface type.

Privileges Required:
MANAGER

Parameters

type/variable	direction	description
ReferenceData id	input	Immutable identification information, chosen by the object implementation at object creation time, and never changed during the lifetime of the object.
InterfaceDef intf	input	Interface Repository object that specifies the set of interfaces and associated exported attributes specified by the object.
Version rVersion	output	Version assigned for this Define.
IntfType aIntfType	output	The DSM interface type.

5.6.3.4 DSM Interfaces Undefine

DSM Interfaces Undefine	Remove an object interface definition from the System. (MANAGER)

Client-Service IDL Syntax

```
module DSM {
   interface Interfaces {
           void Undefine (
                       in ReferenceData id,
                       in Version rVersion,
                       in IntfType aIntfType);
   };
};
```

Semantics

An OWNER may use Interfaces Undefine to remove the definition of an interface from the system.

Privileges Required:
OWNER

Parameters

type/variable	direction	description
ReferenceData id	input	Immutable identification information, chosen by the object implementation at object creation time, and never changed during the lifetime of the object.
Version rVersion	output	Version assigned for this Define.
IntfType aIntfType	output	The DSM interface type.

5.6.4 LifeCycle

All objects have a life cycle. They are created. They may be non-persistent, e.g. created and accessed during a Session, and then destroyed. They may be persistent, e.g. accessed over the span of many Sessions. They are ultimately destroyed. The LifeCycle defines the basic Create, for entities which have this capability.

5.6.4.1 DSM LifeCycle Create

DSM LifeCycle Create	Create an object instance from an object definition.

Client-Service IDL Syntax

```
module DSM {
    interface LifeCycle {
            const AccessRole Create_ACR = OWNER;
            void Create (
                        in IntfType aIntfType,
                        out ObjRef rObjRef);
    };
};
```

Semantics

LifeCycle Create is used to create an instance of an object type which was defined by **Interfaces Define**. All of the DSM types in this document are pre-defined and therefore well-known. An object reference for the instance is returned. This object reference can be used to bind this object to a Name Context, e.g. using **Directory bind**.

Privileges Required:
OWNER

Parameters

type/variable	direction	description
IntfType aIntfType	input	A DSM defined object type, e.g. Directory, File, Stream, for narrowing the type of object.
ObjRef rObjRef	output	A reference to an object instance.

5.6.5 Security

5.6.5.1 DSM Security Authenticate

DSM Security Authenticate	Request authentication with password or decryption key. (READER)

Client-Service IDL Syntax

```
module DSM {
    const AccessRole Authenticate_ACR = READER;
    interface Security {
          void Authenticate (
                   in Password rPassword,
                   in EncryptData rData);
    };
};
```

Semantics

The purpose of **Security Authenticate** is to enable the client to identify itself for the purposes of obtaining access to (i.e., opening) an object. The Authenticate must be given with **Directory Open** if the object has either a non-Null **rPassword** or **EncryptData** with length greater than 0 in its **Perms** attribute. Authenticate must be followed immediately by another command. If an Open is received without a corresponding authenticate preceding it, and authentication is required as described above, a **NO_AUTH** exception will be given. The client is expected to know the reason for the **NO_AUTH**, and will respond accordingly. If an encrypted data response is required, the exception will also carry **EncryptData** which must be successfully processed by the client. The client must then send an Authenticate followed by the Open (repeated), with proper password or processed encrypt data in the Authenticate.

This standard does not specify an encryption algorithm. It does enable the following sequence: a) the service passes **EncryptData** to the Client, b) the Client processes the received **EncryptData** via the encryption algorithm, c) the Client returns transformed **EncryptData** back to the Service, and d) the Service verifies it via the encryption algorithm.

The atomic operation of the authenticate with the following operation from the client is implicit. That is, the Service will wait for the next command from the application and execute the authenticate and the next command together. If the authenticate generates an exception, the next command will generate an exception.

Privileges Required:
READER

Parameters

type/variable	direction	description
Password rPassword	input	A character string password for authentication to open an object.
EncryptData rData	input	A sequence of bytes used by an encryption mechanism.

5.7 Application Portability Interfaces

The application portability interfaces provide a true API with a language mapping. It can be generated using an OMG or DSM IDL compiler from the client-service IDL. If ANSI C is the language of choice, the mapping shown in this section will be used. Applications can link with this interface as a set of simple function calls which in turn invoke the remote procedure calls to the service via the DSM library.

The application portability interfaces define a library of functions calls that can by used by client applications to invoke the DSM-CC Client-Service interface and local DSM Library functionality.

Applications that use these functions will be portable between clients that contain the DSM Library for those operations that result in remote network access.

5.7.1 Consumer Client

The consumer client will typically use a limited set of DSM-CC functions. Utilizing download options, it may keep only those function groups it needs for the application to be run. It may also store a basic set of functions on a more permanent basis. The profile of a client’s DSM-CC User-to-User primitives at any given time is summarized not only by the interface name, but by the privilege given the client. One client can have Core READER interfaces, which include the following DSM-CC primitives:

- Base: IsA, Close
- Directory: Open, Close, Get, resolve, list
- Stream: Pause, Resume, Status, Reset, Play, Jump, Next
- Event: Subscribe, Unsubscribe
- File: Read
- ServiceGateway: Attach, Detach

Another client may have READER privileges for all Core interfaces, and WRITER privileges for the File interfaces on some Services. A client in this case would have the following 13 DSM-CC primitives:

- Base: IsA, Close
- Directory: Open, Close, Get, resolve, list
- Stream: Pause, Resume, Status, Reset, Play, Jump, Next
- Event: Subscribe, Unsubscribe
- File: Read, Write
- ServiceGateway: Attach, Detach

Home settop devices are usually classified as minimal clients. While the Service Gateway complex must implement the complete Core set of interfaces, a settop client need only implement those groups (by privilege) that it needs to run an application.

5.7.2 Information Provider Client

An information provider client, on the other hand, will be accorded OWNER privileges. In addition to the READER privileges listed above, the OWNER can perform the create, put, write, bind, destroy functions, which enable loading of content to the service, and ultimate removal from the service. One kind of information provider is the author. The author, using one of many available multimedia authoring tools, will at times act in the capacity of loading content to the Service, and at other times will act as a consumer in the capacity of viewing and testing the application.

The author will see one stream service or file service which offers OWNER, WRITER and READER privileges. The DSM-CC User-to-User primitives enable MPEG multimedia content to loaded and delivered, using common navigational and access primitives, with just a few additional OWNER and WRITER primitives.

C Mapping

The IDL compiler will generate the following parameters in the C mapping equivalent of the IDL. The standard CORBA compiler generates:

- the object to which the function will be sent, and
- the environment (exception) structure.

The DSM IDL Compiler generates the above 2 parameters, and with Synchronous Deferred option on,

will also generate:

- a **RequestHandle**, which is used as an index to RPC completion status. The **RequestHandle** will be generated for those operations which have a void return value in the IDL specification. It will take the place of the void return value.

For these generated C mappings, the table describing the parameters of each primitive is augmented with these entries:

DSM_<interface name> object	input	Object reference to which the call is made.
CORBA_Environment ev	output	The resulting status of the operation. If NULL, then the operation succeeded, otherwise it points to the exception data defined by this primitive's interface. ev is a void * which is cast into either a CORBA ex_body, or one of the possible DSM exception structures, as defined by the **raises** statement.
DSM_RequestHandle	output	Synchronous Deferred completion status.

Please refer to the CORBA architecture specification for additional details on C mapping rules. In addition, the CORBA architecture specification has example code on how exceptions can be handled in C.

Below is a brief overview of types frequently used by DSM:

5.7.2.1 Basic Data Types

The basic IDL data types used in DSM map as follows:

IDL	DSM shorthand	C
short	**s_short**	`CORBA_short`
unsigned short	**u_short**	`CORBA_unsigned_short`
long	**s_long**	`CORBA_long`
unsigned long	**u_long**	`CORBA_unsigned_long`

The implementation is responsible for providing the typedefs for CORBA_short, CORBA_long, etc., consistent with the IDL requirements for these types.

5.7.2.2 Constants

Constant identifiers are #defined in the C mapping:

5.7.2.3 Struct Types

A struct in IDL maps directly to the equivalent C struct.

5.7.2.4 Sequence Types

A sequence type is converted to a struct with a maximum length, actual length and buffer pointer.

Example:

```
typedef sequence<octet, MAX_LENGTH> rSeq;
```

is converted to:

```
typedef struct {
      CORBA_unsigned_long _maximum;
      CORBA_unsigned_long _length;
      CORBA_octet * _buffer;
} rSeq;
```

5.7.2.5 Strings

IDL strings are mapped to 0-byte terminated character arrays; i.e., the length of the string is encoded in the character array itself through the placement of the 0-byte.

5.7.2.6 Any

The CORBA typedef any maps as follows in C:

```
typedef struct any {
      CORBA_TypeCode_type;
      void * _value;}
      CORBA_any;
```

TypeCodes are generated by the IDL compiler or by a CORBA Interfaces Repository. TypeCodesexpected to be commonly used by DSM are:

TC_boolean	A bit. 0= False; 1 = TRUE.
TC_char	A character.
TC_short	A signed short.
TC_ushort	An unsigned short.
TC_long	A signed long.
TC_ulong	An unsigned long.
TC_string	a 0-byte terminated string.
TC_sequence_octet	A sequence of octets.
TC_sequence_ulong	A sequence of unsigned longs.

For other CORBA TypeCodes, refer to CORBA type definitions in orb.h.

5.7.3 Application Synchronous Deferred Operations

Applications can choose to initiate synchronous deferred requests to the various services on a per process basis. DSM-CC synchronous deferred allows the application process to pipeline its function calls in a non-blocking fashion. The DSM IDL compiler can be given the option to add a RequestHandle to operations as the return value in place of void (or this can be done by hand if such a compiler is not available). The resulting C compilation will have this RequestHandle, which can be used to issue synchronous deferred requests. The following Config interface is then used by the application thread to change mode from synchronous to synchronous deferred and back.

```
// Pseudo IDL
module DSM {
    // RequestHandle is 0 if Config:: DeferredSync is FALSE (synchronous RPC)
    typedef u_long RequestHandle;
    // allow individual threads to configure dynamically as synchronous or deferred synchronous
    interface Config {
        // if TRUE RPC mode is deferred synchronous
        attribute boolean DeferredSync;
        typedef sequence<RequestHandle, 1024> RequestList;
        readonly attribute RequestList ActiveRequests;
        void Wait (in RequestHandle aRequest);   // like CORBA get_response
        void Inquire (in RequestHandle aRequest);            // inquire as to status
    };
};
```

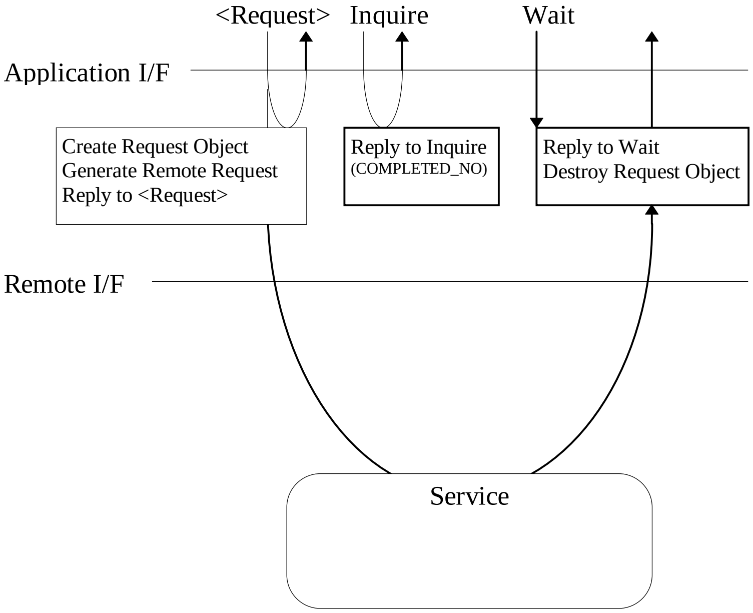


Figure 32: Application and Service I/O

DSM Application Interface primitives can compile to be either synchronous deferred or synchronous. If the client is multi-threaded, it can pipeline messages by sending them on separate threads. Each message does not block the next because they are called from separate threads, the calling thread will wait on the reply from the server. Each thread can set the mode for its RPCs through the Config interface. If DeferredSync is set to FALSE, RequestHandle will always be 0 and each invocation from that thread will block until the Remote reply is received. If DeferredSync is TRUE, the invocations will not block and the RequestHandle will advance with each invocation.

The deferred synchronous mode works as follows: The client application issues a request by calling a DSM primitive, at which time the DSM Library creates a request object for the transaction at hand. It then initiates the remote procedure call (RPC), and replies immediately to the calling application. The client application may elect to continue doing something else, issue separate requests or wait on any particular outstanding request. An Inquire primitive is provided to allow the application to check the status of the transaction. If Inquire returns without an exception, the operation is completed. If it returns an exception with either COMPLETED_NO or COMPLETED_MAYBE, the operation is not complete. The Wait operation can be invoked to block until a reply has been received. This is similar to the CORBA get_response operation.

A successful Inquire signifies that the RPC reply data is valid. If the application has pointers to reply data as a result of the request, it may now access this data. The Request Object is destroyed in the DSM Library for any of the following reasons:

- Null exception from an Inquire invocation.
- Reply to Wait upon remote reply received.
- Destruction of higher level containment object. For example, if a remote reply is received and the DSM Library determines that the parent Service is closed, the corresponding request object is destroyed.

5.7.4 API Definitions

Because nearly all of the Client-Service interfaces have a 1-1 mapping with the Application Portability interfaces, the semantics and parameter descriptions are maintained in the Client-Service Interfaces portion of this document. Please refer to that section for the descriptions of the interfaces or their operations.

5.7.4.1 C Mapping for the Synchronous Interface

The synchronous C mapping of the DSM-CC Core interfaces are shown below. These functions are generated directly from a standard CORBA IDL compiler. When called by the application, the function will always block, i.e., it will not return until after the RPC has returned.

5.7.4.1.1 Base

5.7.4.1.1.1 Base API with 1-1 Client-Service Interface Mapping

The following two functions have a 1-1 Client-Service Interface to API mapping:

```
void DSM_Base_Close (
      DSM_Base object,
      CORBA_Environment * ev)

void DSM_Base_Destroy (
      DSM_Base object,
      CORBA_Environment * ev)
```

5.7.4.1.1.2 DSM_Base_IsA

The **IsA** functionality can have varying implementations, as described in the Client Base interface. It does not necessarily result in an RPC to the Base object it is referring to.

Pseudo-IDL

```
module DSM {
  interface Base {
        void IsA (
              in IntfType aIntfType,
              out boolean aVerdict);
  };
};
```

API: C Language Mapping

```
void DSM_Base_IsA (
      DSM_Base object,
      CORBA_Environment * ev,
      DSM_IntfType aIntfType,
      CORBA_boolean * aVerdict)
```

Semantics

Base IsA enables a client to test whether an object includes(inherits) a specified interface. The interface is identified by aIntfType. If aVerdict is TRUE, the object includes that interface. If aVerdict is FALSE, it does not.

Privileges Required:
READER

Parameters

type/variable	direction	description
IntfType aIntfType	input	A DSM defined object type, e.g. Directory, File, Stream, for narrowing the type of object.
boolean aVerdict	output	TRUE or FALSE. TRUE if the object exports the interface of aIntfType.

5.7.4.1.2 Directory

The following functions have a 1-1 Client-Service Interface to API mapping:

```
void DSM_Directory_list (
      DSM_Directory object,
      CORBA_Environment * ev,
      CORBA_unsigned_long how_many,
      CosNaming_BindingList * bl,
      CORBA_Object * bi)

CORBA_Object DSM_Directory_resolve (
      DSM_Directory object,
      CORBA_Environment * ev,
      CosNaming_Name * n)

void DSM_Directory_Open (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_PathRefs * rPathRefs)

void DSM_Directory_Close (
      DSM_Directory object,
      CORBA_Environment * ev)

void DSM_Directory_Get (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_PathValues * rPathValues)

void DSM_Directory_Put (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_PathValues * rPathValues)
```

5.7.4.1.3 Stream

The following functions have a 1-1 Client-Service Interface to API mapping:

```
void DSM_Stream_Pause (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_NPT * rStop)

void DSM_Stream_Resume (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_NPT * rStart,
      DSM_Stream_Scale * rScale)

void DSM_Stream_Status (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_Stat * rAppStatus,
      DSM_Stream_Stat * rActStatus)

void DSM_Stream_Reset (
      DSM_Stream object,
      CORBA_Environment * ev)

void DSM_Stream_Jump (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_NPT * rStart,
      DSM_Stream_NPT * rStop,
      DSM_Stream_Scale * rScale)

void DSM_Stream_Play (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_NPT * rStart,
      DSM_Stream_NPT * rStop,
      DSM_Stream_Scale * rScale)

void DSM_Stream_Next (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream rNextStream)
```

5.7.4.1.4 Event

The following functions have a 1-1 Client-Service Interface to API mapping:

```
void DSM_Event_Subscribe (
      DSM_Event object,
      CORBA_Environment * ev,
      CORBA_string aEventName,
      DSM_u_short * aEventToken)

void DSM_Event_Unsubscribe (
      DSM_Event object,
      CORBA_Environment * ev,
      DSM_u_short aEventToken)
```

5.7.4.1.5 File

The following functions have a 1-1 Client-Service Interface to API mapping:

```
void DSM_File_Read (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_u_longlong * aOffset,
      DSM_u_long aSize,
      CORBA_boolean * aReliable,
      DSM_ObjData * rData)

void DSM_File_Write (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_u_longlong * aOffset,
      DSM_u_long aSize,
      DSM_ObjData * rData)
```

5.7.4.1.6 ServiceGateway

This function is called to the invoke **ClientSessionSetupRequest** and receive **ClientSessionSetupResponse** network messages:

```
void DSM_ServiceGateway_Attach (
      DSM_ServiceGateway object,
      CORBA_Environment * ev,
      DSM_ObjRef rClientRef,
      DSM_Profile rClientProfile,
      CORBA_unsigned_long * aEndUser,
      DSM_UserContext aSuspendContext,
      DSM_PathSpec * rPathSpec,
      DSM_UserContext * aResumeContext,
      DSM_PathRefs * rPathRefs,
      DSM_DateTime * rDateTime)
```

This function is called to invoke **ClientReleaseRequest** and receive **ClientReleaseConfirm** network messages:

```
void DSM_ServiceGateway_Detach (
      DSM_ServiceGateway object,
      CORBA_Environment * ev,
      CORBA_boolean aSuspend)
```

5.7.4.1.7 Security

This function has a 1-1 Client-Service Interface to API mapping.

```
void DSM_Security_Authenticate (
      DSM_Security object,
      CORBA_Environment * ev,
      DSM_Password rPassword,
      DSM_EncryptData * rData)
```

5.7.4.1.8 View

The following functions have a 1-1 Client-Service Interface to API mapping:

```
void DSM_View_Select (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_u_long aBufSize,
      DSM_View_SQLStatement rSQLStatement,
      DSM_View_ResultDescribe * rResultDescribe,
      DSM_ObjData * rReturnBuffer)
```

/* Note: Read can generate Client-Service View Fetch depending upon the aCursor position. */

```
void DSM_View_Read (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_u_short aCursor,
      DSM_View_BufDescribe * rObjDescribe,
      DSM_ObjData * rRow)

void DSM_View_Update (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_View_SQLStatement rSQLStatement)
```

5.7.4.2 C Mapping for the Synchronous Deferred Interface

When a synchronous deferred C mapping is desired, either the IDL compiler or the programmer must follow these rules:

1. The IDL operations must specify a return value of void.
2. The type DSM_RequestHandle will be substituted for the void return value.

A function with the synchronous deferred C mapping can operate either synchronously or asynchronously, using the Config interface. The following are the synchronous deferred C mappings for DMS-CC interfaces:

5.7.4.2.1 Config

These functions are used to configure the DSM Library RPC mechanism.

```
void DSM_Config_Inquire (
      DSM_Config object,
      CORBA_Environment * ev,
      RequestHandle aRequest)
```

```
void DSM_Config_Wait (
      DSM_Config object,
      CORBA_Environment * ev,
      RequestHandle aRequest)
```

5.7.4.2.2 Base

The following functions have a 1-1 Client-Service Interface to API mapping with a substituted RequestHandle as the return value:

```
DSM_RequestHandle DSM_Base_Close (
      DSM_Base object,
      CORBA_Environment * ev)

DSM_RequestHandle DSM_Base_Destroy (
      DSM_Base object,
      CORBA_Environment * ev)
```

/* Note: see the synchronous function DSM_Base_IsA for semantics */

```
DSM_RequestHandle DSM_Base_IsA (
      DSM_Base object,
      CORBA_Environment * ev,
      DSM_IntfType aIntfType,
      CORBA_boolean * aVerdict)
```

5.7.4.2.3 Directory

The following functions have a 1-1 Client-Service Interface to API mapping with a substituted RequestHandle as the return value:

```
DSM_RequestHandle DSM_Directory_list (
      DSM_Directory object,
      CORBA_Environment * ev,
      CORBA_unsigned_long how_many,
      CosNaming_BindingList * bl,
      CORBA_Object * bi)

DSM_RequestHandle DSM_Directory_Open (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_PathRefs * rPathRefs)

DSM_RequestHandle DSM_Directory_Close (
      DSM_Directory object,
      CORBA_Environment * ev)

DSM_RequestHandle DSM_Directory_Get (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_PathValues * rPathValues)
```

```
DSM_RequestHandle DSM_Directory_Put (
      DSM_Directory object,
      CORBA_Environment * ev,
      DSM_PathType aPathType,
      DSM_PathSpec * rPathSpec,
      DSM_PathValues * rPathValues)
```

5.7.4.2.4 Stream

The following functions have a 1-1 Client-Service Interface to API mapping with a substituted RequestHandle as the return value:

```
DSM_RequestHandle DSM_Stream_Pause (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_NPT * rStop)

DSM_RequestHandle DSM_Stream_Resume (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_NPT * rStart,
      DSM_Stream_Scale * rScale)

DSM_RequestHandle DSM_Stream_Status (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_Stat * rAppStatus,
      DSM_Stream_Stat * rActStatus)

DSM_RequestHandle DSM_Stream_Reset (
      DSM_Stream object,
      CORBA_Environment * ev)

DSM_RequestHandle DSM_Stream_Jump (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_NPT * rStart,
      DSM_Stream_NPT * rStop,
      DSM_Stream_Scale * rScale)

DSM_RequestHandle DSM_Stream_Play (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream_NPT * rStart,
      DSM_Stream_NPT * rStop,
      DSM_Stream_Scale * rScale)

DSM_RequestHandle DSM_Stream_Next (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_Stream rNextStream)
```

5.7.4.2.5 Event

The following functions have a 1-1 Client-Service Interface to API mapping:

```
DSM_RequestHandle DSM_Event_Subscribe (
      DSM_Event object,
      CORBA_Environment * ev,
      CORBA_string aEventName,
      DSM_u_short * aEventToken)

DSM_RequestHandle DSM_Event_Unsubscribe (
      DSM_Event object,
      CORBA_Environment * ev,
      DSM_u_short aEventToken)
```

5.7.4.2.6 File

The following functions have a 1-1 Client-Service Interface to API mapping with a substituted RequestHandle as the return value:

```
RequestHandle DSM_File_Read (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_u_longlong * aOffset,
      DSM_u_long aSize,
      CORBA_boolean * aReliable,
      DSM_ObjData * rData)

RequestHandle DSM_File_Write (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_u_longlong * aOffset,
      DSM_u_long aSize,
      DSM_ObjData * rData)
```

5.7.4.2.7 ServiceGateway

This function is called to the invoke **ClientSessionSetupRequest** and receive **ClientSessionSetupResponse** network messages:

```
RequestHandle DSM_ServiceGateway_Attach (
      DSM_ServiceGateway object,
      CORBA_Environment * ev,
      DSM_ObjRef rClientRef,
      DSM_Profile rClientProfile,
      CORBA_unsigned_long * aEndUser,
      DSM_UserContext aSuspendContext,
      DSM_PathSpec * rPathSpec,
      DSM_UserContext * aResumeContext,
      DSM_PathRefs * rPathRefs,
      DSM_DateTime * rDateTime)
```

This function is called to invoke **ClientReleaseRequest** and receive **ClientReleaseConfirm** network messages:

```
RequestHandle DSM_ServiceGateway_Detach (
      DSM_ServiceGateway object,
      CORBA_Environment * ev,
      CORBA_boolean aSuspend)
```

5.7.4.2.8 Security

This function has a 1-1 Client-Service Interface to API mapping with a substituted RequestHandle as the return value:

```
DSM_RequestHandle DSM_Security_Authenticate (
      DSM_Security object,
      CORBA_Environment * ev,
      DSM_Password rPassword,
      DSM_EncryptData * rData)
```

5.7.4.2.9 View

The following functions have a 1-1 Client-Service Interface to API mapping with a substituted RequestHandle as the return value:

```
DSM_RequestHandle DSM_View_Select (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_u_long aBufSize,
      DSM_View_SQLStatement rSQLStatement,
      DSM_View_ResultDescribe * rResultDescribe,
      DSM_ObjData * rReturnBuffer)
```

/* Note: Read can generate Client-Service View Fetch depending upon the aCursor position. */

```
DSM_RequestHandle DSM_View_Read (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_u_short aCursor,
      DSM_View_BufDescribe * rObjDescribe,
      DSM_ObjData * rRow)

DSM_RequestHandle DSM_View_Update (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_View_SQLStatement rSQLStatement)
```

6. User Capabilities

The userCapabilities information element has a special form that facilitates both compact descriptions and a general approach toward extensibility. The precise definition of this information will be shown below after the introduction of two key concepts which will be used in this section.

The first concept is to use referential rather than literal data whenever possible. This means that a tag can be set which indicates an entire capability set which will be known to the server. This concept is similar to the terminfo concept used in UNIX®, where for example, “VT100” indicates a set of terminal characteristics. This example is also illustrative of the migration of a term from a particular manufacturer’s product to a generally useful term which summarizes a known set of attributes. By specifying the first item in this information element to be an identifier associated with a manufacturer, the meaning of the rest of the data in the message is “anchored” to the definitions used by that manufacturer. The name used to refer to the manufacturer is the OUI, Organizational Unique Identifier, which is registered with the IEEE as specified in IEEE-802.1990.

The second concept which will be used here is that all tags and associated data will be ASCII characters organized as “TAG” or “TAG=VALUE” items. All items will be terminated by a comma (ASCII 0xc2). Thus, the entire client configuration information element will be expressed as a single ASCII string.

This approach has several advantages:

- This document will define a set of tags thought to be generally useful. However, manufacturers can extend this list indefinitely with items of interest to their own products.
- In many if not most cases, a simple set of referntial tags can be sent which will resort in a compact message (an example will be shown below). Additional tags need to be sent only to indicate capabilities which are not summarized by the simple referential tags.

6.1 userCapabilities Message Structure

Syntax	Value	Octets
Text Terminating_null_octet	0x00	1
		2

Text - sequence of non-null octets with an embedded field structure as explained below
Terminating_null_octet - Required to terminate the capabilities list and always 0x00 in value.

The Text section is comprised of a sequence of comma (0xc2) terminated logical fields. The first field shall be the **Manufacturer_OUI_code** which is the manufacturer's Organizational Unique Identifier code as specified by IEEE-802.1990 concatenated with the **Manufacturer_id** which is the manufacturer assigned name for the client configuration. Both the value and semantics of this field are determined by the manufacturer. All remaining fields in this section are optional and may be used to modify the configuration indicated by the **Manufacturer_id**. The following rules apply to the syntax of the text section:

1. The first field shall be the **Manufacturer_OUI_code** concatenated with the **Manufacturer_id**.
2. The **Manufacurer_OUI_code** shall be encoded as two hexidecimal digits per byte with no delimiters. For example, if the OUI code were the decimal value 12951793 then it would be expressed as as three hexidecimal bytes concatenated together, C5A0F1. Each hexidecimal byte must be encoded as two upper case ASCII characters.
3. All remaining fields shall consist of tags and their arguments if they take any.
4. Tags and arguments shall be ASCII strings which can contain any non-blank characters excluding the characters reserved by the syntax (see Table 106)
5. Arguments shall be separated from the tag by an "=" sign (0xd3).
6. A reserved list of tags is defined in Table 107, Table 108 and Table 109 below.
7. A semicolon (0xb3) in the argument can be used to indicate an embedded list.
8. Numeric arguments to tags shall be expressed as ASCII representations of decimal values.
9. ASCII characters as used here shall refer to 7 bit ASCII restricted to the values 32_{10} to 127_{10} inclusive.

Any other tags may be specified by the manufacturer. The tags in Table 108 and Table 109 require arguments. that must be expressed using this syntax. In summary, the syntax of a field can be expressed as:

TAG[=ARGUMENT[**;**ARGUMENT[...]]]**,**

The following characters are reserved by the message syntax and may not appear as part of a tag name or an argument.

Table 106: Client Configuration syntax reserved characters

Character	Hex Value	Usage
comma ‘,’	0xc2	Field terminator
semi-colon ‘;’	0xb3	Argument list separator
equal ‘=‘	0xd3	Tag assignment operator
Null	0x00	Message terminator

6.2 Tag_value_list: predefined tags.

Device descriptors fall into three categories:

1. Boolean - indicating the presence or absence of a capability. By default all undeclared booleans are false.
2. Declarative - these are strings that assign or declare a value to a configuration attribute.
3. Scope - these are strings used to bound or declare a range to a configuration attribute.

Table 107: Boolean Descriptor Tags

descriptor_tag	Meaning
11172-2	The device is able to decode constrained parameter bit streams as defined in ISO/IEC 11172-2..
11172-3	The device is able to decode MPEG-1 compliant audio syntax.
13818-2	The device is able to decode MPEG-2 compliant video syntax. By virtue of backwards compatibility setting this attribute implies setting 11172-2. There is no need for 13818-2 compliant devices to assert 11172-2 as well, this would be redundant.
13818-3	The device is able to decode MPEG-2 compliant audio syntax. By virtue of backwards compatibility setting this attribute implies setting 11172-3. There is no need for 13818-3 compliant devices to assert 11172-3 as well.
IRTx	An infrared transmitter is supported
PGMS	The device is able to demultiplex MPEG-2 program streams or MPEG-1 systems streams.
RS232	An RS-232 device is available.
SAP	Ability to handle secondary audio program streams. This capability is usually software that allows the user to select which audio stream to decode based on the associated stream descriptor.
TS	The device is able to demultiplex MPEG-2 compliant transport streams.
MCR	The device is equipped with a magnetic card reader.
P1394	A P1394 consumer electronics serial expansion capability is present.
MHEG	An ASN.1 MHEG engine is installed.
MHEGA	An alternate SGML MHEG engine is installed.

Table 108: Declarative Descriptor Tags

descriptor_tag	Meaning
CPU	Identifies the CPU part used for application execution in the client device.
OS	Declares the operating system that will be available to application programs.
GOVP	Identifies the hardware graphics processor available in the device. The convention is to supply the chip part number as defined by the manufacturer (e.g. CD-I compatible graphics would be MCD210).
CLRSP	Color space. Common values are RGB, YUV and DYUV.
MSD	The device has is a mass storage peripheral. This tag takes arguments which include the following: • SRO serial read only • SRW serial read write • RRO random read only • RRW random read write
NBCAUDIO	An ISO/IEC 13818-7 Non Backward Compatible audio decoder is installed.
AT	The named Anti Taping technology is available in the client device.
NETIF	A description of client network interfaces. The value is defined in section 6.3.
NETSTACK	A description of protocol layers of the client networking stack. The value is defined in section 6.4.
DOWNLOAD	The networking technology used for client download. The value is defined in section 6.5.
SOFTWARE	A description of the currently loaded software in the client device.
CLIENT	Allows you to include another definition by reference. This tag has two distinct uses. One is to declare a particular model from a manufacturer's family of products Another use is for a manufacturer to indicate that his product has similar capabilities as another manufacturer's product. In either the case, the data in this field will be the concatenation of the **Manufacturer_OUI_code** and the **Manufacturer_id**. Care should be taken when using this descriptor_tag, The declared values of the referenced client will supersede previously declared values of the corresponding descriptor_tags of the current settings. e.g. If you specifically declared RAM="4096000" and then use a CLIENT reference that contained a different RAM declaration the referenced CLIENT's RAM figure would supersede the explicit RAM declaration.

Table 109: Scope Descriptor Tags

descriptor_tag	Meaning
NVRAM	The total amount of Non Volatile RAM (in bytes), the device is equipped with.
GHRES	The horizontal graphic resolution in pixels.
GVRES	The vertical resolution in pixels.
CLRBITS	The number of bits comprising each color component.
ALPHA	The number of bits in the alpha channel.
PLANES	The number of graphic planes supported
PCMCIA	The number of PCMCIA ports present.
OSVER	The version and revision of the operating system.
RAM	The total amount of RAM (in bytes) available to application programs.
VRAM	The total amount of Video RAM (in bytes) in the device.
SNUM	The serial number of the client device.
MSDSIZE	The storage capacity of an attached mass storage device (in.bytes). Note: If an MSD tag used an embedded list to declare multple devices, then the MSDSIZE must list the associated sizes in the same order as the MSD embedded list.
MSDBLK	The block size (in bytes) of the mass storage device . Note: If an MSD tag used an embedded list to declare multple devices, then the MSDBLK must list the associated sizes in the same order as the MSD embedded list.
CRYPT	The name of the en/de/cryption algorithm supported. (e.g. RSA).
SCRAM	The name of the conditional access scrambler. Typical value might be “DVBSS” (Digital Video Broadcaster’s Super Scrambler.
AUDLYRS	The MPEG audio layers (expressed in digits) supported by the device’s MPEG audio decoder.
MPEGPROF	The profile level supported by an MPEG-2 compliant decoder. This is expressed as profile and level separated by the ‘@’ sign. (e.g. MP@ML would equate to Main Profile / Main Level). This tag only applies to the video decoder.

6.3 The NETIF Descriptor Tag

The NETIF descriptor tag describes the client network interfaces. The value of this tag is a string with the following syntax:

```
    NETIF=“IDENT:<id>&MAXBW:<bw>&DIR:<UP,DOWN,BIDIR>&UU”
```

where the subdescriptors are described in Table 110. The ampersand character “&” is used to separate subdescriptors. Multiple network interfaces may be defined by including the semicolon “;” argument separator and adding another complete value string.

Table 110: NETIF Subdescriptors

Subdescriptor	Meaning
IDENT	A client assigned integer identifier for this network interface.
MAXBW	The maximum rate at which the client is capable of receiving data over this interface expressed as the number of bits per second.
DIR	An enumerated value of the direction of data transfer capable over this interface. Possible values are UP for data from the client, DOWN for data to the client, and BIDIR for data both to and from the client.
UU	A boolean descriptor whose presence indicates the User to User RPC may be delivered to the client over this interface.

An example of the use of the NETIF tag is:

```
NETIF=“IDENT:1&MAXBW:10000000&DIR:DOWN;IDENT:2&MAXBW:56000&DIR:DOWN&UU;I
DENT:3&MAXBW:56000&DIR:UP”
```

6.4 The NETSTACK Descriptor Tag

The NETSTACK descriptor tag is used to identify the networking stack present on the client. The value is a string of a name and version, separated by the letter “v”, of the presentation (6), session (5), transport (4) and network (3) layers of the protocol stack separated by the ampersand “&” character. For the value of the NETSTACK descriptor, protocol names are forbidden from using the lower case letter “v” so the separation between name and version is unambiguous. If no version number of a protocol is specified, then only the name is included in the value and the “v” separator is not used. An example describing the OMG IIOP protocol stack which uses CDR version 1.0 for presentation, UNO version 1.0 for session, TCP with an unspecified version number for transport, and IP version 4 for network would be:

```
    NETSTACK=“CDRv1.0&UNOv1.0&TCP&IPv4”
```

If multiple protocol stacks are operational, then additional stacks would be specified using the semicolon “;” argument separator.

6.5 The DOWNLOAD Descriptor Tag

The DOWNLOAD descriptor tag is used to identify the networking stack which is present on the client to perform software download. The value is a string equivalent in format with the NETSTACK descriptor defined in section 6.4.

6.6 Example Tag_list.

This section does not include new normative text. An example is shown of the use of the Tag_list defined in section 3.1.8.13.2 above. Note that the actual message would not include any space or newline characters separating fields as shown here.

> SNUM=“1234567”, 13818-2, 13818-3, TS, AUDLYRS=“12”,
> CPU=“MCD68331”, OS=“OS9”, OSVER=“1.20”, NVRAM=“1024000”, RAM=“2048000”,
> NTSC, GOVP=“MCD210”, GHRES=“360”, GVRES=“240”, PLANES=“2”,
> CLRSP=“RGB”, CLRBITS=8

Please refer to informative Annex G for a more detailed discussion of how one might fully implement this section in both the client and the server.

7. Download

1. *There is still an issue as to the meaning of the word Server as used in this section; i.e., is there a separate scenario of Client - Network download?*
2. *“Broadcast” scenario name should be replaced with “Non-Flow Controlled” name or some similar such thing since the presence of flow control seems to be the distinguishing factor*
3. *Delete term superblock throughout*
4. *Need to make sure the download type is added to the general message header dsmccType list*
5. *Its not clear how all scenarios map into the protocol, e.g. use of special meanings for window size, ackperiod, etc. This needs more review for technical purity by the group.*

7.1 Overview

The download protocol is intended to be a very lightweight yet fast data or software download from a Server to a Client or from the Network to a Client. The protocol is designed to support both a more traditional flow controlled download as well as a broadcast option that are both based on a similar message set.

A complete download operation transfers a download ‘Image’ to the Client. The image is sub-divided into one or more ‘Modules’. The entire image and each module are divided into ‘Blocks’. All blocks within a download image other than the last block of a module are of the same size.

Modules are a delineation of logically separate groups of data within the overall image. A typical, but not normative, use of this feature is to indicate groups of data that need to be loaded into continuous memory. This example allows the Client to fragment the allocated memory chunks by module size rather than having to allocate a memory chunk of the image size.

The block size is negotiated to meet requirements for efficiencyand effective error detection performance. Each block contains data from only one module.

The download protocol addresses two scenarios: one in which flow control is used and one in which there is no flow control of the downloaded image. An example of the latter case would be where the downloaded data is broadcasted to multiple Clients simultaneously.

In the flow controlled scenario the Client and Server negotiate a window size for a one way sliding window protocol. The sliding window applies only to DownloadDataBlock() messages and not other control message exchanges. The complexity of the sliding window protocol is on the Server side only because of the one way nature.. The size of the window can be negotiated by the Client and Server. When the window is negotiated, the Server also selects an ack period that is equal or smaller than the window size. The ack period is a simple way to limit the rate that the Client sends acks back to the Server and therefore limits the network traffic and protocol stack processing caused by the acks. The window size is irrelevent for the broadcast (non flow controlled) case because there is no flow control.

7.2 Preconditions and Assumptions

7.2.1 User-to-User Download *This should go into the U-U section and the following should only be generic p’s and a’s*

7.2.1.1 Flow Control Case

1. A Session has been established between the Client and Server
2. A Connection between the Client and Server has been established and identified for use for Download messages from the Client to the Server.
3. A Connection between the Client and Server has been established and identified for use for Download messages from the Server to the Client. This connection may be for all messages or all messages except DownloadDataBlock().
4. If necessary a Connection between the Client and Server has been established and identified for use for DownloadDataBlock() messages from the Server and Client.
5. These connections provide a datagram service that delivers packets that are reliably delineated. The payload data of these packets may have non zero bit error probability. The datagrams may not have flow control and may be delivered out of order or dropped.

These requirements should be moved to U-U download section

7.2.1.2 Broadcast case

1. A broadcast channel has been identified to the Client for delivery of DownloadDataBlock() and DownloadInfoResponse() messages to the Client.
2. A Connection between the Client and Server may have been established and identified delivery of a DownloadInfoRequest() messages from the Client to the Server.

7.2.2 User-to-Network Download (Config)

It has not yet been fully agreed that we will support the use of this download protocol from the Network to the Client during or in substiture for the UN Config.

7.2.2.1 Flow Control Case

7.2.2.2 Broadcast Case

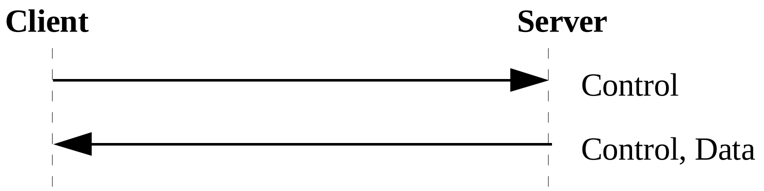
7.3 Download Methods

7.3.1 Flow Controlled

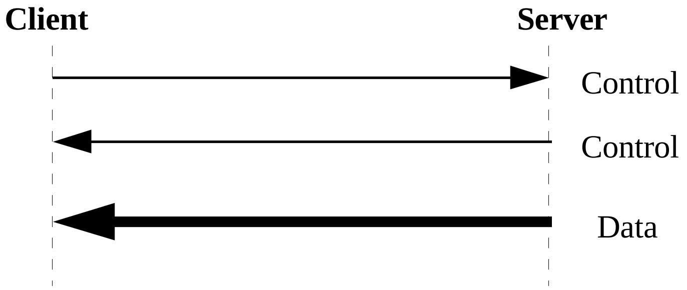
7.3.1.1 Network Models

In these network models the thin lines represent point to point connections between the Server and a single Client. The fat lines represent a separate logical connection that provides higher bandwidth data delivery than the ‘thin line’ connections.

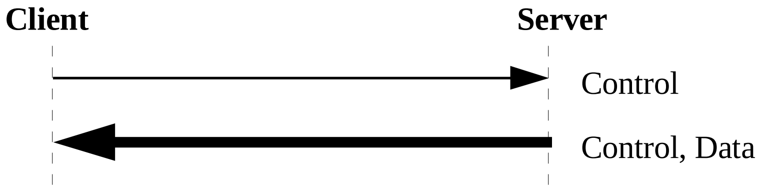
The first model has a single connection in each direction. The Server-to-Client connection carries all the DownloadDataBlock() messages as well as all the other download signalling and control messages.



The second model uses separate Server-to-Client connections for data and control. The DownloadDataBlock() messages travel over the high bandwidth connection but all other messages travel over the other connections.

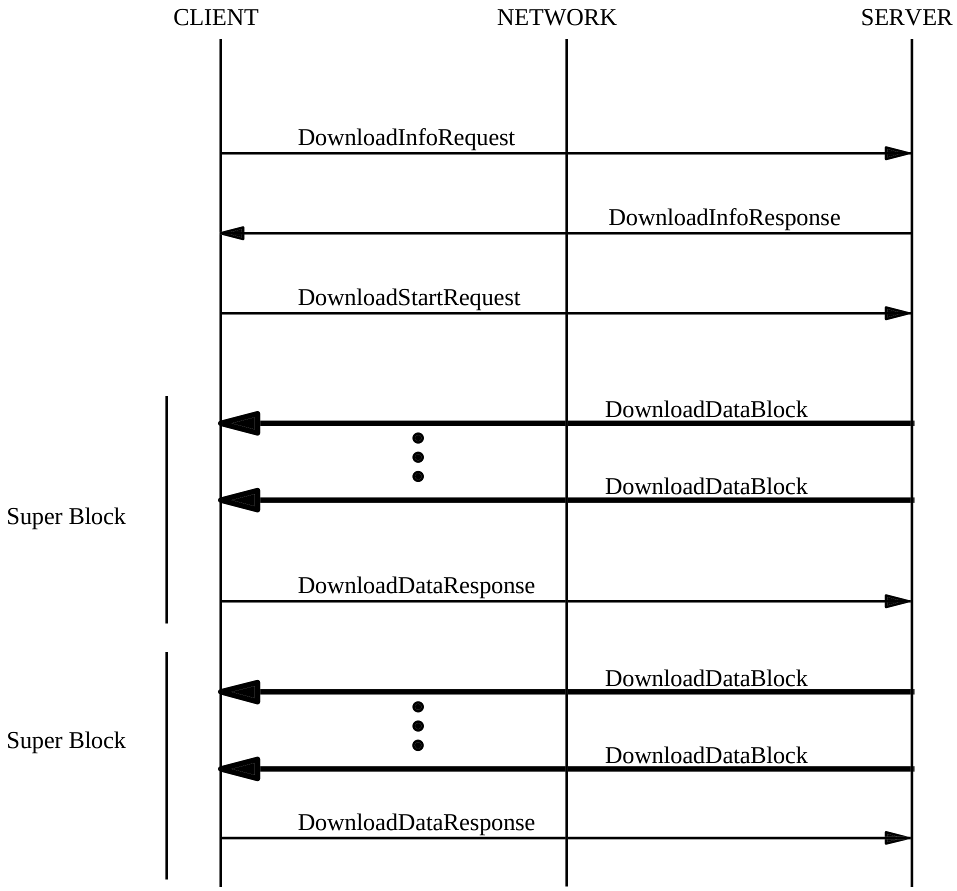


The third network model has a single high bandwidth connection from the Server-to-Client which carries all of the Server-to-Client messages. This model is logicaly the same as the first model but is worth pointing out since the protocol stack for the ‘fat line’ may be different than for the ‘thin line’.



7.3.1.2 Scenarios

Needs to have some introductory text. Network is not needed in this discussion and shouldn’t be shown in diagram.



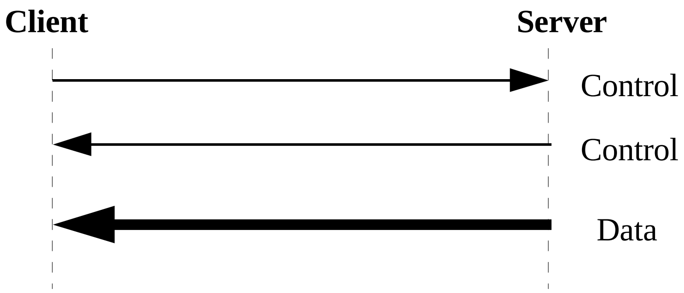
7.3.2 Non-Flow Control

In this section, no mechanism for controlling the downloaded image data flow is used. The DownloadDataBlocks are sent repeatedly in a carousel fashion over the high speed channel. A common example of this case is a broadcast carousel.

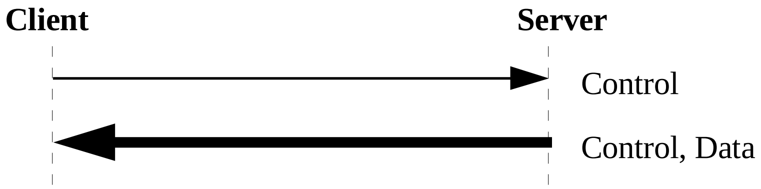
7.3.2.1 Network Models

In these network models the thin lines represent point to point connections between the Server and a single Client. The fat lines represent a separate logical connection that provides higher bandwidth data delivery than the ‘thin line’ connections and carry the download image information. In all cases the DownloadDataBlock() messages travel over the ‘fat line’.

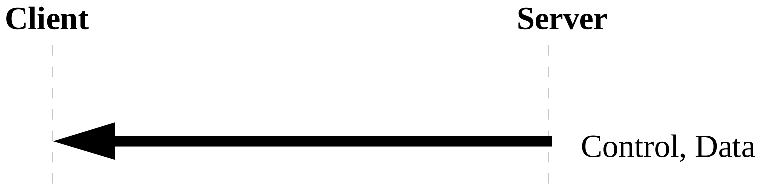
In the first model a full bidirectional pair of connections is available between the Client and Server. The DownloadInfoResponse() message is probably delivered over the ‘thin line’ Server-to-Network connection in this case.



The second model has a Client-to-Server connection but only has (or only uses) the broadcast Server-to-Client path. In this case both the DownloadInfoResponse() and DownloadDataBlock() messages are delivered over the broadcast path.



The third model is a purely broadcast case. In this model it is assumed the Client has already determined where to find the broadcast channel. Both the DownloadInfoResponse() and DownloadDataBlock() messages are delivered over the broadcast path.



7.3.2.2 Scenarios

Left as an exercise to the reader!

7.4 Messages

7.4.1 Use of DSMCCMessageHeader()

(NEW) DSMCC General Message Header

Protocol Discriminator	*1 byte*
dsmccType	*1 byte*
messageId	*1 byte*
messageLength	*2 bytes*
CALength	*1 byte*
{CA bytes}	
{Message bytes}	

1. *General Message Header format above belongs in Section 2, DSM-CC Message Header.*
2. *This format still requires review by the full DSM-CC group.*

messageId - see Table below

Message Name	messageId	Description
DownloadInfoRequest	0x0001	Client request download parameters
DownloadInfoResponse	0x0002	Server provides download parameters
DownloadStartRequest	0x0003	Client asks Server to start sending data
DownloadDataBlock	0x0004	Server sends one download data block
DownloadDataResponse	0x0005	Client acks or naks downloaded data blocks
DownloadCancel	0x0006	Client or Server cancels or aborts download due to severe error or entitlement failure
DownloadServerInitiate	0x0007	Server requests Client to initiate a download.

7.4.2 Other Common Message Fields

downloadTransactionId - Used to associate the messages from one each download sequences. This is useful for differentiating old versus new sequences that could temprorarily coexist due to abort and retry scenarios. The upper two bits of this sixteen bit field are used to indicate who allocated the transaction id (Client, Network, Server). It is up to each entity to assign unique transaction ids (unique within the retirement period of the id).

bits	field values	description
bit 15,14	00	Client
	01	Server
	10	Network
	11	Broadcast Server ???
bit 13-0	xx xxxx xxxx xxxx	transaction sequence number

checksum - A 32 bit checksum calculated over the entire message including the DSMCCMessageHeader(). The checksum is calculated by seeding a 32 bit register with zero and then doing a 32 bit XOR with all bytes of the message 32 bits (MSB first) at at time. If the message length is not a multiple of four bytes then the message is padded out with zero bytes at the end for the purpose of the checksum calculation only.

7.4.3 DownloadInfoRequest

```
DownloadInfoRequest() {
        DSMCCMessageHeader(messageId = 0x0001)
        downloadTransactionId                    2 byte
        maximumBlockSize                         2 bytes
        bufferSize                               4 bytes
        capabilitiesLength                       2 bytes
        userCapabilities                         capabilitiesLength
        privateDataLen                           1 byte
        privateData                              privateDataLen
        checksum                                 4 bytes
}
```

maximumBlockSize - is the maximum block size in number of bytes that the Client agrees to support. The server will select an actual blockSize which is no larger than this size. A value of zero means that the Client places no restrictions on the maximum block size.

bufferSize - informs the server the maximum number of bytes the Client can receive before requiring flow control (ack). The Server would then select a window size no larger than FLOOR[bufferSize / blockSize]. bufferSize must be equal to or larger than maximumBlockSize. A buffer size of zero means there is unlimited buffer size available, or equivilently the Client can absorb data at a rate greater than the maximum physical network can deliver.

capabilitiesLength -

userCapabilities -

privateDataLen -

privateData -

7.4.4 DownloadInfoResponse

```
DownloadInfoResponse() {
        DSMCCMessageHeader(messageId = 0x0002)
        downloadTransactionId                   2 bytes
        blockSize                               2 bytes
        windowSize                              1 byte
        ackPeriod                               1 byte
        tCDownloadWindow                        4 bytes
        tCDownloadScenario                      4 bytes
        numberOfModules                         1 byte
        for(i=0; i<NumberOfModules; i++) {
                moduleSize                      4 bytes
                modulePrivate                   3 bytes
                moduleDescriptorLength          1 byte
                moduleDescriptor                moduleDescriptorLength
        }
        checksum                                4 bytes
}
```

blockSize - is in units of bytes and will be less than or equal to the maximumBlockSize sent in the RequestInfo message. This is the payload size of every Block carried in the DownloadDataBlock() message, except for the last Block of each module which may be smaller than blockSize.

windowSize - is the number of blocks in the sliding window. A value of zero means that the window is the size of the entire image and that no acks are to be sent by the Client. A window size of zero can be used only if the Client set the bufferSize to zero in the DownloadInfoRequest() message. The Client does not need to know the windowSize to implement the protocol but is provided as a sanity check.

ackPeriod - is the number of blocks the Client would normally be required to receive before sending an Ack (DownloadDataResponse). The ackPeriod does not limit when a Nak can be sent. The Client also always sends an Ack for last block in the image (except for broadcast).

tCDownloadWindow - is in units of microseconds and is the time out period for each ack.

tCDownloadScenario - is in units of microseconds and is the time out period for the entire download.

numberOfModules - is the number of modules in the download image. The first module in the list is module number zero.

moduleSize - is the length of each module in bytes.

modulePrivate - *not clear both modulePrivate and moduleDescriptor are needed. Can they be combined?*

moduleDescriptorLength -

moduleDescriptor -

7.4.5 DownloadStartRequest

```
DownloadStartRequest() {
        DSMCCMessageHeader(messageId = 0x0003)
        downloadTransactionId                   2 bytes
        checksum                                4 bytes
}
```

7.4.6 DownloadDataBlock

```
DownloadDataBlock() {
        DSMCCMessageHeader(messageId = 0x0004)
        downloadTransactionId                   2 bytes
        moduleNumber                            1 byte
        blockNumber                             2 bytes
        checksum                                4 bytes
        {block data}
}
```

moduleNumber - module number that this Block belongs to. Module number zero is the first module in the module list in the DownloadInfoResponse() message.

blockNumber - block number of this module.

block data - blockSize bytes of Block data, except for the last Block of each module which may have fewer than blockSize bytes.

7.4.7 DownloadDataResponse

```
DownloadDataResponse() {
        DSMCCMessageHeader(messageId = 0x0005)
        downloadTransactionId                   2 bytes
        reason                                  1 byte
        moduleNumber                            1 byte
        blockNumber                             2 bytes
        checksum                                4 bytes
}
```

reason - indicates the reason for the response. In the Flow Controlled case the Ack is sent when each complete Super Block is received as well as when the last Block in the Image has been received.

0x01	Ack
0x02	Nak (checksum error)
0x03	Nak (missing block)
0x04	Nak (tCDownloadWindow timeout)

moduleNumber -

blockNumber -

7.4.8 DownloadCancel

```
DownloadCancel() {
        DSMCCMessageHeader(messageId = 0x0006)
        downloadTransactionId                    2 bytes
        reason                                   1 byte
        moduleNumber                             1 byte
        blockNumber                              2 bytes
        checksum                                 4 bytes
}
```

reason - A reason code to explain the cancel

0x01	tCDownloadScenario timeout
0x02	

moduleNumber - The module number at the time of the cancel.

blockNumber - The block number at the time of the cancel.

7.4.9 DownloadServerInitiate

This message is sent by the Server to the Client. It is a request for the Client to initiate a download by first sending the DownloadInfoRequest() message.

```
DownloadServerInitiate() {
        DSMCCMessageHeader(messageId = 0x0007)
        downloadTransactionId                    2 bytes
        privateDataLength                        1 byte
        privateData                              privateData
        checksum                                 4 bytes
}
```

downloadTransactionId -

privateDataLength -

privateData -

checksum -

7.5 Flow Controlled Scenario

7.5.1 Getting Download Protocol Parameters

Before starting the download procedure the Client and Server must exchange basic parameter information to be used during the download. The Client initiates this information exchange by sending the DownloadInfoRequest.

The Client provides a maximum supported block size.

The Client provides an available buffer size for receiving download data without requiring the Server to pause while waiting for an ack. The Server must select a blockSize and windowSize that meet the requirements:

blockSize	<=	maximumBlockSize
windowSize * blockSize	<=	bufferSize

The server responds with the DownloadInfoResponse message. This message includes the blockSize, windowSize, ackPeriod, download protocol timers, and a module size table.

The ackPeriod must be less than or equal to the windowSize. A larger ackPeriod reduces the ack traffic back to the Server. An ackPeriod less than the windowSize allows the Client to send an ack before the Server stalls due to a full window. Two suggestions for choosing windowSize and ackPeriod are:

1. (windowSize - ackPeriod) < (ackLatency * transferRate / blockSize)
2. ackPeriod > (ackLatency * transferRate / blockSize)

The ackLatency represents the delay (seconds) to send an Ack through the network and the Client and Server protocol stacks. The transferRate is the expected average delivery rate (bytes/second) that the Server would provide if it did not have to wait for Acks. The term (ackLatency * transferRate / blockSize) is the expected number of blocks that the Server could send during the period that an Ack is delivered. The first suggestion says that an Ack should be sent in advance of the window being filled such that the ack is received before the Server stalls because the window is full. The second suggestion says that Acks should not be sent any more often than the period that it takes to deliver an Ack. There are other reasons, such as burden on the Server and network to handle Acks, to make the ackPeriod even larger than the second suggestion.

The module table is a list of modules. The table is needed to know how many modules are in the image and the number of blocks that are in each module. This information is needed in order to interpret the moduleNumber and blockNumber fields in the DownloadDataBlock and DownloadDataResponse messages. In particullar this information is needed in order to know when the download is complete (for either flow controlled or broadcast case).

tCDownloadScenario is selected by the Server such that it is larger than the longest period required for a successful download. The expected download time is the expected bit rate times the Image size. The Server should use some conservative estimate of bit rate such that the download will not timeout needlessly during an otherwise successful download.

7.5.2 Starting Download

Once the Client receives the DownloadInfoResponse() message the Client typically would allocate memory for each of the modules in the image. Since in some systems the allocation of memory can take a substantial amount of time the Server will not start the download until the DownloadStartRequest() is sent by the Client and received by the Server.

7.5.3 Acks and Naks

Acks and Naks are both coded in the DownloadDataResponse() message by way of the reason field. When a DownloadDataResponse(Nak) is sent the moduleNumber and blockNumber fields are set to point to the first Block that has not yet been received correctly and therefore needs to be re-sent. When a DownloadDataResponse(Ack) is sent the moduleNumber and blockNumber fields are set to point to the next Block that is expected to be received.

The Client would normally send a DownloadDataResonse(Ack) after it has received and stored ackPeriod Blocks. Note the importance of having stored the Blocks in their end destination in memory since the Ack implies that the Client is ready to receive windowSize Blocks beyond the Acked Block. In other words an Ack advances the window to windowSize Blocks beyond the Acked Block.

A Nack must be sent if a Block is received that is not the next expected (in sequence) Block of the image. The moduleNumber and blockNumber fields of the DownloadDataResponse(Nack) point to the next extected Block of the image. The Server responds to a Nack by resending starting at the Block that was pointed to by the Nack. A Nack implicitly is also an Ack for the Block just before the Nacked Block.

Messages that are received but fail the checksum are to be dropped. In this case the Client would typically send a Nack message that points to the next expected Block since the dropped message probably was the next DownloadDataBlock(). This assumes that the logical connection over which the message datagrams are delivered to the Client is being used exclusively for the download protocol at that moment. If for some reason this assumption is not true the Client would need to wait for the first DownloadDataBlock() message that had an out of sequence moduleNumber, blockNumber before a Nack could be sent.

When the last Block of the last Module has been received by the Client it must send an Ack immediately regardless of the ackPeriod. This Ack should point to blockNumber zero of the next logical moduleNumber. This module number is one larger than the last moduleNumber in the module list.

7.5.4 Timers and Retransmission

Client

- tCDownloadInfoReq
- tCDownloadWindow
- tCDownloadScenario

Server

- tSDownloadScenario

When the Client sends the DownloadInfoRequest() message the Client starts the tCDownloadInfoReq timer. If the DownloadInfoResponse() message is received this timer is cancelled. If this timer expires the Client may retransmit the DownloadInfoRequest() message. The tCDownloadInfoReq value is provided by U-N Config. The number of allowed retransmissions is implementation specific. (do transactionId's change on retransmission of DownloadInfoRequest - probably not)

Server needs timeout while waiting for DownloadStartRequest()... (rely on server scenario timer?)

The tCDownloadScenario timer is used to timeout the entire download procedure. The value for this timer is provided by the Server in the DownloadInfoResponse() message. This timer is useful for the Client to detect severe failures such as the Server hanging during the download. When the Client sends the DownloadStartRequest() message the Client starts the tCDownloadScenario timer. If this timer expires then the Client shall abort the download, and send a DownloadCancel(reason = scenarioTimeout) message to the server, and cancel the tCDownloadWindow timer. After the download has timed out the Client shall reject any download messages from the server that have the downloadTransactionId of the canceled download. The Client may retry the download but with a new downloadTransactionId.

Server needs scenario timer too... (start on DownloadInfoRequest?)

The tCDownloadWindow timer is used to timeout Acks/Nacks used to advance the sliding window. The value for this timer is provided by the Server in the DownloadInfoResponse() message. This timer is useful for the Client to detect that an Ack or Nack was not received by the Server. When the Client sends the DownloadStartRequest() or a DownloadDataResponse(Ack/Nack) message the Client starts or restarts the tCDownloadWindow timer. If this timer expires then the Client shall send a DownloadDataResponse(Nack/tCDownloadWindow, Module/Block = next expected) to the Server and also restarts the tCDownloadWindow again. The timer can timeout repeatedly and each time the Nack would be resent. If there is a severe problem the tCDownloadScenario timer would eventually expire.

7.5.5 Abort

Either the Client or the Server can abort the download by sending a DownloadCancel() message to the other. Potential reasons for an abort are:

1. Server responds with an illegal blockSize, windowSize or ackPeriod in DownloadInfoResponse().
2. Client runs out of memory during download.
3. Client receives modules which are inconsistent with the module table in the DownloadInfoResponse().
4. The downloadTransactionId changes during a broadcast download. *{this should be in broadcast section}*

7.6 Broadcast Scenario

7.6.1 Image Assembly

The blockSize and the module table in the DownloadInfoResponse() message enables a Client to start receiving data right out of sequence during a broadcast download. This also enables the Client to start receiving data even if it starts listening in the middle of the download.

The first step is that the Client must receive the DownloadInfoResponse() message either from a point-to-point signaling channel or from the broadcast carousel. This message includes the blockSize and module table. Before starting to receive data the Client should allocate memory for each module and initialize a scoreboard to track which Blocks from the Image have been received.   Below is some simple pseudo code for the initialization and for the what do with each received Block.

Variables:

M	numberOfModules
S[m]	size in bytes of module m
SB[m]	size in blocks of module m
A[m]	address in memory where block is to be stored
R[m][b]	one bit received flag for each block of image (one bit flag)
SBI	total number of Blocks in the Image
RBI	number of received unique Blocks

Initialization:

```
M = numberOfModules;
for(i=0; i<M; I++) {
      S[i] = moduleSize[i];
      A[i] = malloc(S[i]);
      SB[i] = (S[i] + blockSize - 1) / blockSize;
      for(j=0; j<SB[i]; j++)
            R[i][j] = 0;
}
```

For each block received (m=moduleNumber, b=blockNumber):

```
RECEIVE_BLOCK(m,b);
if( !R[m][b] ) {
      R[m][b] = 1;
      STORE_BLOCK(A[m] + b * blockSize);
      RBI++;
      if(RBI == SBI)
            DOWNLOAD_DONE;
}
else
      DISCARD_BLOCK;
```

The array SB[m] used in the example above is also useful for the flow controlled case in order to track what is the next expected Block to be received in order to Ack or Nack.

7.6.2 Timers

The tCDownloadScenario is used in a similar manner as in the flow controlled case, except that the timer is started when the Client first starts looking for the broadcast data.

7.6.3 Image Coherency

There needs to be protection against the case when the image is updated on the carousel. This could potentially happen during the period that a user is doing a broadcast download. This situation is detected by monitoring the downloadTransactionId in each DownloadDataBlock() message. If the image is ever updated the downloadTransactionId should be changed. If the Client detects that this ID changes during the download then a coherency problem has been detected and the download should be aborted and possibly retried.

The downloadTransactionId is used slightly differently in the broadcast case versus the flow controlled case. In the broadcast case it is more of a version number. For broadcast the downloadTransactionId used for control messages will probably be different from the downloadTransactionId in the DownloadDataBlock() messages.

When the Client first receives a Block from the broadcast channel the downloadTransactionId of that Block becomes the current downloadTransactionId or version for that image. If a subsequent block is received with a different downloadTransactionId then there is a coherency problem (due to update of the carousel) and the download should be aborted.

Note that the downloadTransactionId is not intended to be used as a multiplex address. It is assumed that some other method exists for the Client to efficiently filter the Blocks from the desired Image from the broadcast carousel. For example, if MPEG-2 Transport Streams are used then all the Blocks of one Image would be in one unique PID.

7.7 MPEG-2 Transport Streams

7.7.1 Encapsulation of Messages

When the Download messages are to be carried in MPEG-2 Transport Streams the encapsulation is the same as for U-N messages as described in Section 6 of Working Draft 2. This note belongs in that section.

7.7.2 Broadcast Carousel Directories

In the broadcast case there probably needs to be a way of carrying some form of directory in the broadcast channel. In the case of Transport Stream the most logical solution is to use a separate PID for each Image. A Program Map Table (PMT) could carry the list of Download PIDs with an appropriate list of descriptors for each PID/Image.

These descriptors are TBD. We may want to assign of the DSMCC stream_type values to identify a download stream.

8. Normal Play Time, Stream Mode and Stream Events

8.1 Purpose

Normal Play Time (NPT) is a continuous timeline over the duration of an Event (Event defined in 13818-1). The NPT refers to the real time of the event regardless of ‘trick mode’ presentations. For example, when a video is played in reverse NPT counts down rather than up and when a video is played at 10x speed NPT progresses at 10x regular rate. The NPT provides an absolute timeline to which references can be made for operations such as a jump.

A timeline of some sort already exists in an MPEG Program - the System Time Clock (STC) which is recovered from the stream via the Program Clock Reference (PCR) timestamps. This timeline by itself does not satisfy the requirements for the NPT. First, it always moves forward at a normal rate regardless of the presentation direction and speed of the event. Second, it may be discontinuous.

Besides the Normal Play Time concept, this standard describes two additional mechanisms to communicate the status of a stream: the streamMode descriptor and the streamEvent descriptor.

8.2 Conversion between NPT and STC Timebase

[This section has not been verified for technical accuracy yet. There may be some issues with units conversion. There should not be any reference to STC ‘frequency’ here.]

Example of NPT_descriptor usage. At any time during the course of stream playback there is a valid System Time Clock (STC) value (as defined in section D.0.3 of this Recommendation/International Standard) and a valid NPT value. Translation between these two timebases is required in the client device. For example, to display the NPT time to the viewer the client device can examine the STC, compute the corresponding NPT value and display the result.

The following linear equation describes the relationship between an NPT time value corresponding to an STC value, $STC_{i,}$ in terms of the fields in the NPT descriptor:

$$NPT = (300*(STC_i - STC\_reference)/system\_clock\_frequency) \times (scale\_numerator/scale\_denominator) + NPT_r$$

with

$$NPT_r = 27 \times (NPT\_reference\_seconds \times 10^6 + NPT\_reference\_microseconds) / system\_clock\_frequency$$

$$NPT = (300 \times (STC_i - STC\_reference) \times (scale\_numerator/scale\_denominator) + (27 \times (NPT\_reference\_seconds \times 10^6 + NPT\_reference\_microseconds)) / system\_clock\_frequency$$

From NPT the parameters NPT_seconds and NPT_microseconds can be calculated as follows:

$NPT\_seconds=$

$(300 \times (STC_i - STC\_reference) \times (scale\_numerator/scale\_denominator) +$

$(27 \times (NPT\_reference\_seconds \times 10^6 + NPT\_reference\_microseconds)) DIV(27\times 10^6)$

$NPT\_microseconds=$

$((300(STC_i - STC\_reference) \times (scale\_numerator/scale\_denominator) +$

$$(27 \times (NPT\_reference\_seconds \times 10^{6} + NPT\_reference\_microseconds)) -$$

$$NPT\_ seconds \times 10^{6}) / 27$$

Note that this equation is only valid in the context of a single STC interval (being defined as a continuous set of STC values with no discontinuities). For example assume that the current value of the NPT parameters are STC_reference = 12000, NPT_reference_seconds = 0.3, NPT_reference_microseconds=0, scale_numerator=2, scale_denominator=1 (2x fast-forward). Then given that the current value of the STC, STC_i is 1000, the corresponding NPT value is

$NPT\_seconds = 300 \times (1000 - 12000) \times 2 + (27 \times 0.3 \times 10^{6})) \text{ DIV } (27 \times 10^{6}) = 0$

$NPT\_microseconds = ((300 \times (1000\text{-}12000) \times (2) + (27\times (90\times 10^{6}))\text{-}0)/27=55555$

Note that the values used for STC_reference and NPT_reference do not have to actually be realized by the NPT clock or STC at any given point in time. That is to specify the linear translation one only needs to specify one point on the translation line and it is immaterial where that point lies.

8.3 NPT Uncertainty

There are three situations where the NPT time may be unknown or invalid.

The first case is a stream where an NPT_reference_descriptor has not yet been received for the first time. In some streams the NPT may never be included. *[In this case we could say that the STC should be somehow used as a default NPT time?].*

The second case is the period between a STC (PCR) discontinuity and the delivery of a new NPT_reference_descriptor. This is referred to as the ‘NPT gray area’. It is possible to solve this problem in two ways. First, an NPT_reference_descriptor can be sent in advance of the discontinuity by using the post_discontinuity_indicator. Second, if the NPT gray area is detected a projection of the old STC timeline can be made to approximate the needed STC time for the NPT calculation. The second method assumes that there is no change in the speed scale at the discontinuity. Generally discontinuities of the PCR due to edits and changes in presentation are two different kinds of events.

Using transport stream, when a system time-base discontinuity occurs it invalidates the current NPT parameters. In order to maintain accurate NPT in the client an NPT_descriptor with the post_discontinuity_indicator set to ‘1’ must be send prior to the system time-base discontinuity.

8.4 Descriptors

8.4.1 NPT Reference Descriptor

The NPT descriptor contains information allowing the client to maintain NPT for a specific program.

Table 8-1 -- NPT Reference Descriptor

Syntax	No. of bits	Mnemonic
NPT_reference_descriptor() {		
descriptor_tag	**8**	**uimsbf**
descriptor_length	**8**	**uimsbf**
post_discontinuity_indicator	**1**	**bsblf**
reserved	**6**	**bslbf**
STC_reference	**33**	**uimsbf**
scale_numerator	**16**	**tcimsbf**
scale_denominator	**16**	**uimsbf**
NPT_reference_seconds	**32**	**tcimsbf**
NPT_reference_microseconds	**32**	**uimsbf**
}		

Semantic definition of fields in NPT descriptor

post_discontinuity_ indicator -- the post_continuity_indicator is a 1bit field indicating when set to '1' that the NPT descriptor will become valid at the next system time-base discontinuity *(as defined in section 2.4.3.5 of part 1 of this Recommendation/International Standard)*. If the post_continuity_indicator is set to '0' the NPT descriptor is valid until the next system time-base discontinuity or a new NPT_descriptor has been received. *[This was NOT agreed to in the group. We already have a current_next_indicator. The introduction of this new feature needs some more thought into how it fits into the big picture]*

[The following definition is derived from the definition of PTS in section 2.4.3.6 of ISO/IEC 13818-1]

STC_reference -- The STC_reference is a 33 bit unsigned integer. It indicates the STC time for which the NPT equals the NPT_reference as coded in the NPT_reference_field, $tstc_r(k)$. The value of STC_reference is specified in units of the period of the system clock frequency *(as defined in section 2.4.2.1 of part 1 of this Recommendation/International Standard)* divided by 300 (yielding 90 kHz). The STC_reference_time is derived from the STC_reference according to equation x-x+1 below.

$$\text{STC\_reference}(k)=((\text{system\_clock\_frequency} \times tstc_r(k)) \text{ DIV } 300) \% 2^{33}$$

(x-x+1)

where

$tstc_r(k)$ is the time for which the NPT equals the NPT_reference as coded in the NPT_reference_field.

scale_numerator, scale_denominator -- The signed 16 bit integer scale_numerator and the unsigned 16 bit integer scale_denominator indicate the rate and direction at which the stream is currently playing. A negative numerator indicates reverse direction, whereas a positive number indicates forward direction. 1/1 indicates normal play speed.

NPT_reference_seconds, NPT_reference_microseconds -- The signed 32 bit integer NPT_reference_seconds and the unsigned 32 bit integer NPT_reference_microseconds indicate the value of the NPT, NPT_r, corresponding to the STC_reference value coded in the STC_reference field. [NPT_r has the dimension of seconds.] The value of NPT_reference_seconds is specified in units of the period of the system clock divided by 27000000. The value of NPT_reference_microseconds is specified in units of the period of the system clock divided by 27.

$$\text{NPT\_reference\_seconds}(k)= (\text{system\_clock\_frequency} \times NPT_r(k)) \text{ DIV}(27\times 10^6)$$

$$\text{NPT\_reference\_microseconds}(k)=$$

$$((\text{system\_clock\_frequency} \times NPT_r (k))\text{-NPT\_reference\_seconds}\times 10^6)/ 27$$

[DIV and % as defined in section 2.2.1 of this Recommendation/International Standard]

Issues: How often shall NPT_descriptors be sent?

8.4.2 NPT Endpoint Descriptor

The NPT descriptor contains information allowing the client to maintain NPT for a specific Event.

Table 8-2 -- NPT Endpoint Descriptor

Syntax	No. of bits	Mnemonic
NPT_endpoints_descriptor() {		
descriptor_tag	**8**	**uimsbf**
descriptor_length	**8**	**uimsbf**
NPT_start_seconds	**32**	**tcimsbf**
NPT_start_microseconds	**32**	**uimsbf**
NPT_stop_seconds	**32**	**tcimsbf**
NPT_stop_microseconds	**32**	**uimsbf**
}		

Semantic definition of fields in NPT descriptor

NPT_start_seconds, NPT_start_microseconds -- The signed 32 bit integer NPT_start_seconds and the unsigned 32 bit integer NPT_start_microseconds indicate the value of the NPT corresponding to the begin of the current event *[as defined in section 2.1.25 of part 1 of this Recommendation/International Standard]*. NPT_start_seconds and NPT_start_microseconds are specified and calculated in the same way as NPT_reference_seconds and NPT_reference_microsecons as specified in section x.x.

NPT_end_seconds, NPT_end_microseconds -- The signed 32 bit integer NPT_end_seconds and the unsigned 32 bit integer NPT_end_microseconds indicate the value of the NPT corresponding to the end of the current event *[as defined in section 2.1.25 of part 1 of this Recommendation/International Standard]*. NPT_end_seconds and NPT_end_microseconds are specified and calculated in the same way as NPT_reference_seconds and NPT_reference_microsecons as specified in section x.x.

8.4.3 Stream Mode Descriptor

The StreamMode_descriptor contains information about the mode of the stream allowing the client to request the status of a stream in progress.

Table 8-3 -- Stream Mode Descriptor

Syntax	No. of bits	Mnemonic
StreamMode _descriptor() {		
descriptor_tag	**8**	**uimsbf**
descriptor_length	**8**	**uimsbf**
StreamMode	**8**	**uimsbf**
}		

Semantic definition of fields in StreamMode_descriptor

StreamMode -- This 8 bit field indicates the current mode of the stream.

Table 8-4 -- Stream Mode Values

Value	Meaning
0	Pause
1	SearchPlay
2	SearchPlayPause
3	PauseSearchPlay
4-255	ITU-T Rec. H.222.X |ISO/IEC 13818-6 Reserved

8.4.4 Stream Event Descriptor

The Stream Event Descriptor contains information allowing the transmission of application-specific events (events as defined in Section ???, to be distinguished from those defined in section 2.1.25 of part 1 of this Recommendation/International Standard) in a way so that they are synchronous with the Transport Stream packets if a Transport Stream is used or that they are synchronous with the Program Stream PES packets if a Program Stream is used.

Table 8-5 -- Stream Event Descriptor

Syntax	No. of bits	Mnemonic
StreamEvent _descriptor() {		
descriptor_tag	**8**	**uimsbf**
descriptor_length	**8**	**uimsbf**
event_id	**16**	**uimsbf**
NPT_event_seconds	**32**	**tcimsbf**
NPT_event_microseconds	**32**	**uimsbf**
for (i=0; i<N; i++) {		
private_data	**8**	**uimsbf**
}		
}		

Semantic definition of fields in StreamEvent_descriptor

event_id -- -- The unsigned 16 bit integer event_id indicates the type of the application-specific event to which the descriptor refers to.

NPT_event _seconds, NPT_event_microseconds -- The signed 32 bit integer NPT_event_seconds and the unsigned 32 bit integer NPT_event_microseconds indicate the value of the NPT corresponding to the point in time when the event will occur. NPT_event_seconds and NPT_event_microseconds are specified and calculated in the same way as NPT_reference_seconds and NPT_reference_microsecons as specified in section x.x. A value of NPT_event_seconds of '1000 0000 0000 0000 0000 0000 0000 0000' indicates that the event occurs at the present time.

9. Transport

9.1 U-N Messages Transport Requirements

The transport mechanism for U-N messages is user defined. The transport mechanism includes the transport layer and all underlying layers. The table below summarizes the minimum requirements for this transport mechanism.

Table 9-1

Transport Function	Requirement
Reliability of Data	Error detection must be provided. Corrupted messages should be discarded.
Reliability of Delivery	The delivery of the message need not be guaranteed.
Flow Control	Transport need not regulate the rate of transmission of messages.
Fragmentation and Reassembly	Transport is responsible for any required fragmentation and reassembly of messages.
Delivery Order of Messages	Transport need not be responsible for in order delivery of messages.
Addressing	

An example of a suitable transport is UDP/IP.

9.2 U-U Initial Download Transport Requirements

The transport mechanism for U-U Initial Download messages is user defined. The transport mechanism includes the transport layer and all underlying layers. The table below summarizes the minimum requirements for this transport mechanism.

Table 9-2 -- U-U Initial Download Transport Requirements

Transport Function	Requirement
Reliability of Data	
Reliability of Delivery	
Flow Control	
Fragmentation and Reassembly	
Delivery Order of Messages	
Addressing	

9.3 U-U Remote Procedure Call Transport Requirements

The transport mechanism for U-U RPC Primitives is user defined. The transport mechanism includes the transport layer and all underlying layers. The table below summarizes the minimum requirements for this transport mechanism.

Table 9-3 -- U-U RPC Transport Requirements

Transport Function	Requirement
Reliability of Data	Defined by RPC requirements (?)
Reliability of Delivery	Defined by RPC requirements (?)
Flow Control	Defined by RPC requirements
Fragmentation and Reassembly	Defined by RPC requirements
Delivery Order of Messages	Defined by RPC requirements
Addressing	

9.4 Encapsulation within MPEG-2 Transport Streams

9.4.1 Role of MPEG-2 Transport Stream in Protocol Stack

None of the DSM-CC messages are required to be carried within an MPEG-2 Transport stream, with the exception of the NPT Time Stamps. However, if MPEG-2 Transport Streams are used to deliver DSM-CC messages the encapsulation of these messages is defined by DSM-CC.

An MPEG-2 'Transport Stream' provides Data Link Layer services. It can play a similar role in a protocol stack as the ATM Layer in an ATM network.

MPEG-2 Systems 13818-1 includes a structure called Private Sections which DSM-CC uses to provide reassembly of Transport Packets into larger DSM-CC messages, and optionally provides a CRC for reliability. The Sections can play a similar role in a protocol stack as the ATM Adaptation Layer in ATM networks.

9.4.2 DSM-CC Sections

Encapsulation of all DSM-CC messages in MPEG-2 Transport Streams use the DSMCC_section() structure which inherits all of the Private Section syntax and semantics as defined by MPEG-2 Systems (ISO/IEC 13818-1). The mapping of DSMCC_section() into MPEG-2 Transport Packets and the maximum length of a DSMCC_section() is also governed by the semantics for Private Sections defined by MPEG-2 Systems.

In some implementations it is very desirable to use the CRC_32 available in sections. Because some systems may have difficulty calculating a CRC_32 it has been made optional. To be consistent with 13818-1, if the section_syntax_indicator is set to '1' the CRC_32 must be present and correct. In the case that section_syntax_indicator is '0' the payload of the section will be exactly the same except that the CRC_32 is replaced with the NO_CRC field which must be all '1''s (0xFFFFFFFF). The private_indicator bit must always be set as the complement of the section_syntax_indicator.

Implementations should be careful not to use the section_syntax_indicator as the only indication of the presence of the CRC_32 since the section_syntax_indicator bit itself may be subject to a bit error. If the section_syntax_indicator is '0' then the indication is that the CRC is not present but the private_indicator and NO_CRC_32 fields should be verified to be all '1' and if they are not the section has suffered an error. If the section_syntax_indicator is '1' then private_indicator should be '0' and the CRC_32 should be correct and if they are not the section has suffered an error.

In the case that section_syntax_indicator is '0' the syntax for Private Sections allows payload to begin immediately after the section length field. The DSMCC_section() structure defines that section structure to be identical regardless of the state of the section_syntax_indicator, with the exception of the validity of the CRC.

Table 9-4 -- DSM-CC Section Format

```
DSMCC_section() {
        table_id                          8      uimsbf
        section_syntax_indicator          1      bslbf
        private_indicator                 1      bslbf
        reserved                          2      bslbf
        DSM_section_length                12     uimsbf
        table_id_extension                16     uimsbf
        reserved                          2      bslbf
        version_number                    5      uimsbf
        current_next_indicator            1      bslbf
        section_number                    8      uimsbf
        last_section_number               8      uimsbf

        if(table_id == 0x0A) {
                DSMCC_LLCSNAP()
        }
        else if (table_id == 0x0B) {
                DSMCCMessageHeader()
        }
        else if (table_id == 0x0C) {
                DSMCC_Descriptor_List()
        }

        if(section_syntax_indicator == ‘0’) {
                NO_CRC_32                 32     bslbf
        }
        else {
                CRC_32                    32     rpchof
        }
}
```

Semantic definition of fields in DSMCC_section

table_id -- This is an 8 bit field, which is the case of a DSMCC_section shall always be set to identify the type of DSM-CC structure in the section payload. *Note that it is the intent to set this to the same value as the stream_type.*

Table 9-5 -- DSM-CC Table IDs

table_id	DSMCC Section Type
0x0A	Multiprotocol
0x0B	DSM-CC Messages Header (U-N)
0x0C	DSM-CC Descriptors Loop
0x0D	TBD

section_syntax_indicator -- This is a 1 bit indicator. When set to '1', it indicates the use of a valid CRC_32. When set to ‘0’ the NO_CRC_32 field must be set to all ‘1’’s.

private_indicator - Set to complement of section_syntax_indicator.

DSMCC_section_length-- This 12 bit field specifies the number of remaining bytes in the DSMCC_section. That is the number of bytes following the this field.

table_id_extension -- *This is currently reserved by MPEG.*

version_number --

current_next_indicator --

section_number --

last_section_number --

9.4.3 DSM-CC Stream Types

There are three different DSM-CC section types: Multiprotocol, DSMCCMessageHeader, and DSMCC_Descriptor_List. DSM-CC has defined stream_type values defined so that the Program Map Table (PMT) can point to separate PIDs for each of the different DSM-CC section types.

Is it o.k. for two or more different DSM-CC section types to share the same PID? The table_id values are different for each so they are compatible.

Table 9-6 -- DSM-CC Stream Types

stream_type	Description
0x0A	Multiprotocol (U-U RPC)
0x0B	DSM-CC Messages Header (U-N)
0x0C	DSM-CC Descriptors (NPT)
0x0D	TBD

9.4.4 Data Link Flow Control

Flow control for messages is generally managed by the transport layer protocol. Because MPEG-2 Transport can provide the ability to deliver data at extremely high bit rates the System Layer specification (ISO/IEC 13818-1) provides a mechanism to set rules to regulate the burst delivery rate of MPEG-2 Transport Packets of the same PID. The mechanism is based on a model of a 512 byte Transport Buffer (TB).

We need to say whether any of the PIDs of DSM-CC stream_type are subject to any TB rules. If so we need to specify the leak rate.

9.4.5 DSM-CC Multiprotocol Encapsulation

The DSM-CC U-U RPC messages can be delivered over a transport layer selected by the implementor. However, if these messages are delivered over an MPEG-2 Transport Stream the encapsulation of the message must use the DSMCC_LLCSNAP() structure within a DSMCC_section. This multiprotocol encapsulation is available for applications other than U-U RPC Primitives.

The DSMCC_LLCSNAP() structure provides logical link control. This structure allows the encapsulation of a wide selection of network layer protocols, including the Internet Protocol (IP). The selected network layer protocol encapsulates a selected transport layer such as TCP or UDP.

The LLC/SNAP is an ISO standard - what is the number? The structure below is taken from RFC 1483 - we need to verify it from the ISO standard.

Comments on Maximum MTU.... We agreed that it would be based on 4Kbyte limit for private sections. Exact semantics are needed.

Table 9-7 -- DSM-CC Multiprotocol Encapsulation

```
DSMCC_LLCSNAP() {
        logical_link_control                                            24   bslbf

        /* Routed ISO PDUs */
        if(logical_link_control == 0xFEFE03) {
                for(i=0; i<PES_packet_length-3; I++)
                        ISO_PDU_data_byte                               8    bslbf
        }

        /* Sub Network Attachment Point (SNAP) */
        /* encapsulated network layer routing */
        else if(logical_link_control == 0xAAAA03) {
                /* SNAP header: */
                organizationally_unique_id                              24   bslbf
                protocol_id                                             16   bslbf

                 /* Routed Non-ISO */
                if(organizationally_unique_id == 0x000000) {
                        /* Routed IP */
                        if(protocol_id == 0x0800) {
                                for(i=0; i<PES_packet_length - 8; I++)
                                        NONISO_IP_PDU_data_byte         8    bslbf
                                }
                        /* Routed other non-ISO */
                        else {
                                for(i=0; i<PES_packet_length - 8; i++)
                                NONISO_PDU_data_byte            8  bslbf
                        }
                }

                else if(organizationally_unique_id == 0x0080C2) {
                        /* Link Layer Control (LLC) bridging */
                        /* 802.3 (ethernet) with FCS */
                        if(protocol_id == 0x0001) {
                                pad                                     16   bslbf
                                for(i=0; i<PES_packet_length-14; i++)
                                        IEEE802_3_PDU_data_byte         8      bslbf
                                IEEE802_3_FCS                           32   bslbf
                        }

                        /* 802.4 with FCS */
                        if(protocol_id == 0x0002) {
                                pad                                     24   bslbf
                                for(i=0; i<PES_packet_length-15; i++)
                                        IEEE802_4_PDU_data_byte         8     bslbf
                                IEEE802_4_FCS                           32   bslbf
                        }

                         /* 802.5 (token ring) with FCS */
                        if(protocol_id == 0x0003) {
                                pad                                     24   bslbf
                                for(i=0; i<PES_packet_length-15; i++)
                                        IEEE802_5_PDU_data_byte         8    bslbf
                                IEEE802_5_FCS                           32   bslbf
                        }

                        /* FDDI with FCS */
                        if(protocol_id == 0x0004) {
                                pad                                     24   bslbf
                                for(i=0; i<PES_packet_length-15; i++)
                                        FDDI_PDU_data_byte              8    bslbf
                                FDDI_FCS                                32   bslbf
                        }

                        /* 802.3 (ethernet) without FCS */
                        if(protocol_id == 0x0007) {
                                pad                                     16   bslbf
                                for(i=0; i<PES_packet_length-14; i++)
                                        IEEE802_3_PDU_data_byte         8    bslbf
                        }

                        /* 802.4 without FCS */
                        if(protocol_id == 0x0008) {
                                pad                                     24   bslbf
                                for(i=0; i<PES_packet_length-15; i++)
                                        IEEE802_4_PDU_data_byte         8    bslbf
                        }

                        /* 802.5 (token ring) without FCS */
                        if(protocol_id == 0x0009) {
                                pad                                     24   bslbf
                                for(i=0; i<PES_packet_length-15; i++)
                                        IEEE802_5_PDU_data_byte         8   bslbf
                        }

                        /* FDDI without FCS */
                        if(protocol_id == 0x000A) {
                                pad                                     24   bslbf
                                for(i=0; i<PES_packet_length-15; i++)
                                        FDDI_PDU_data_byte              8      bslbf
                        }

                        /* 802.6 with Common PDU Header */
                        if(protocol_id == 0x000B) {
                                /* Common PDU Header */
                                IEEE802_6_CPDU_reserved                 16   bslbf
                                IEEE802_6_CPDU_BEtag                    16   bslbf
                                IEEE802_6_CPDU_BAtag                    16   bslbf
                                for(i=0; i<PES_packet_length-14; i++)
                                        IEEE802_6_PDU_data_byte         8      bslbf
                        }

                        /* 802.1 BPDU */
                        if(protocol_id == 0x000E) {
                                for(i=0; i<PES_packet_length-8; i++)
                                        IEEE802_1_BPDU_data_byte        8   bslbf
                        }
                }
                }
}
```

9.4.6 U-N Messages

DSM-CC U-N messages are encapsulated in the DSMCCMessageHeader() (defined in section 2.3) within the DSMCC_section().

The reference to section 2.3 is 'hardwired'. Is there a trick in Word to make the reference automatically track.

9.4.7 U-U Initial Download

TBD

9.4.8 U-U Remote Procedure Calls

For use of DSM-CC U-U RPC Primitives over MPEG-2 Transport Stream, the DSM-CC Multiprotocol Encapsulation is required. The use of this encapsulation for the delivery of other user defined messages is allowed but optional. For example, if TCP/IP is encapsulated in this manner to carry U-U RPC messages, the same method can also be used to deliver IP for other user defined applications. In this example the IP packets for both U-U RPC and user defined purposes may be delivered with the same PID.

9.4.9 NPT Time Stamps and DSM-CC Descriptors in MPEG-2 Transport Streams

NPT time stamp descriptors must be carried in a DSMCC_Descriptor_List() within a DSMCC_section() when carried in a MPEG-2 Transport Stream. The DSM-CC StreamMode and StreamEvent descriptors may be carried in this same descriptor list. The number of descriptors in the loop is determined by the stream_info_length field.

Table 9-8 -- DSM-CC Descriptor List

```
DSMCC_Descriptor_List() {
        for(i=0; i<N; i++) {
                reserved                        3               uimsbf
                stream_PID                      13              uimsbf
                stream_info_length              16              uimsbf
                for (i=0;i<N1;i++) {
                        descriptor()
                }
        }
}
```

Semantic definition of fields in DSM-CC Descriptor List

stream_PID -- This is a 13 bit field specifying the PID of the Transport Stream packets which carry the stream to which the following descriptor loop refers to.

stream_info_length -- This is a 16 bit field specifying the total length, in bytes, of the descriptors immediately following the stream_info_length field.

9.5 Encapsulation within MPEG-2 Program Streams

TBD.

When we used PES Packets the delivery of DSM-CC messages in Program Streams was more clear. We certainly can define a way to carry DSM-CC messages in a PES Packet using the DSMCC stream_id for use in Program Streams but we will need at least a one byte adaptation header to prevent emulation of 13818-1 Annex A DSM-CC. Need to refresh our memory as to how PSI sections are carried in Program Streams.

This issue will be resolved in a later meeting

9.5.1 NPT Time Stamps and DSM-CC Descriptors in MPEG-2 Program Streams

This section was not yet agreed upon by the committee

NPT time stamp descriptors when carried in a MPEG-2 Program Stream must be carried in a DSMCC_program_stream_Descriptor_List() as a PES packet as defined in Section 2.4.3.6 of part 1 of this Recommendation/International Standard. The DSM-CC StreamMode and StreamEvent descriptors may also be carried in this descriptor list. The number of descriptors in the loop is determined by the elementary_stream_info_length field.

Table 9-9 -- DSM-CC_program_stream_Descriptor List

```
DSMCC_program_stream_Descriptor_List() {
        packet_start_code_prefix                        24              bslbf
        stream_id                               8               uimsbf
        packet_length                           16              uimsbf
        dsmcc_discriminator                     8               uimsbf
        for(i=0; i<N; I++) {
                elementary_stream_id            8               uimsbf
                elementary_stream_info_length   16              uimsbf
                for (i=0;i<N1;i++) {
                        descriptor()
                }
        }
}
```

Semantic definition of fields in DSM-CC_program_stream_Descriptor List

stream_id -- This is a 8-bit field specifying the bitstream identification that takes the value '1111 0010' for the DSMCC bitstream.

packet_length -- A 16-bit field specifying the number of bytes in the DSM-CC_program_stream_Descriptor List following the last byte of the field.

dsmcc_discriminator -- This is an 8-bit unsigned integer that takes the value '1000 0000' for the DSM-CC_program_stream_Descriptor List.

elementary_stream_id -- This is an 8-bit field indicating the value of the stream_id field in the PES packet headers of PES packets in which the elementary stream is stored to which the descriptor loop following the elementary_stream_id refers to.

elementary_stream_info_length -- This is a 16-bit field specifying the total length, in bytes, of the descriptors immediately following the elementary_stream_info_length.

9.6 Encapsulation within MPEG-1 System Streams

The delivery of DSM-CC messages is not supported in MPEG-1 System Streams.

10. INFORMATIVE ANNEX A

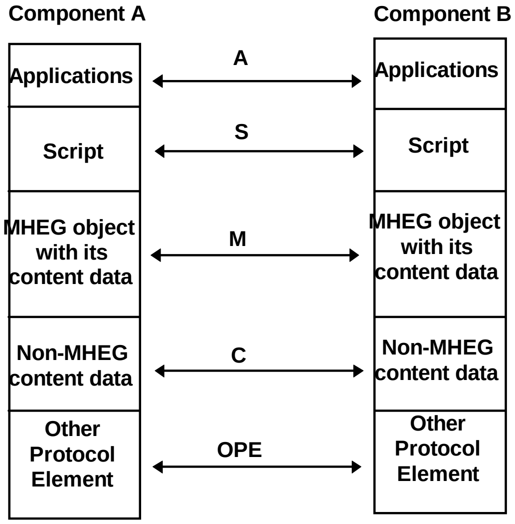
(This annex does not form an integral part of this International Standard)

RELATIONSHIP OF DSM-CC and MHEG

10.1 Overview of MHEG

MHEG provides a model and constructs for interchange of Multimedia objects between applications. It assumes a requirements model in which data are exchanged in a layered model between different components of an implementation. These components may have a symmetric or asymmetrical relationship to each other and may be viewed as separate (client-server) or the same application. This model is shown in Figure A-1.

Fig. A-1: MHEG Reference Model



MHEG explicitly supports scripting languages through a Script Class, but it does not specify a scripting language of its own nor requires the use of any particular scripting language. Likewise, application-level data exchange is not covered by MHEG.

At the lower levels of the model, the C-layer (non-MHEG content data) and OPE-layer (Other Protocol Element) are also outside of the scope of MHEG. For C-layer data, MHEG looks to other monomedia recommendations and standards specifications. One example of this layer would be MPEG (even though MPEG has video, audio and systems components, it can be abstracted as a single class of data). The OPE-layer is for the exchange of messages and acknowledgments which may be required by the application, but not by MHEG itself.

MHEG offers support for multimedia interchange through an object-oriented approach. It provides for the definition of MHEG objects with attributes for:

- identification of standard and standard version
- identification of class of MHEG object
- MHEG identifier of the MHEG object

- general object information (name, owner, …)

It also supports generic referencing mechanisms (logical name, head, tail, etc.) and a consistent approach to generic types of attribute values (Boolean, Integer, etc.). Additionally, each MHEG object has a behavior set associated with it. These include presentation behavior, preparation behavior, and some behavior classes (action, link, and script) which may be used to modify other intrinsic behaviors. In particular, the script class may be used to exchange scripts which describe very complex modifications to the intrinsic behaviors of objects. The action object is used to exchange ordered sets of actions without reference to the objects or related presentable objects.

Via its object model, expressed as various MHEG classes, MHEG acts as a container for the interchange of multiple media types. It can encapsulate both, other standard interchange formats such as MPEG or JPEG, and private encoding techniques. It also supports the final form of presentation of multiple media. It provides for identification of the coding techniques used to enable the use of the appropriate presentation resources on a specific platform. It also explicitly supports multimedia presentation by providing structures for the composition of different media types into a presentation. These include time-sequencing, spatial positioning and logical interaction between media.

MHEG's class set can be used to specify:

- objects containing monomedia information
- relationship between objects
- dynamic behavior between objects
- information to optimize the real-time handling of objects

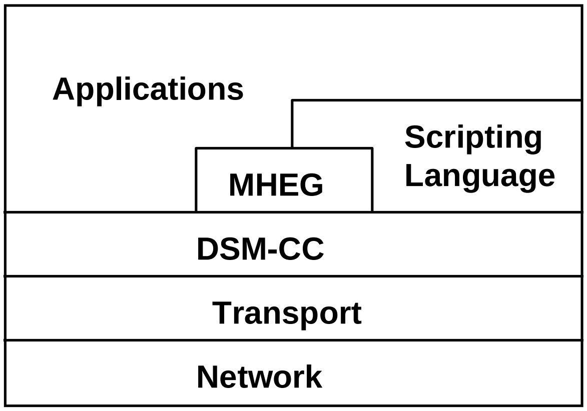
MHEG's classes include, the content class, composite class, link class, action class, script class, descriptor class, container class, and result class.

It should be noted that the MHEG standard does NOT define an API for the handling of objects on its classes. Nor does it define methods on its classes. Thus, the use of object model is limited to attribute inheritance between classes.

10.2 Detailed relationship of DSM-CC to MHEG

based on this cursory review, DSM-CC may be seen to fit into the OPE-layer of the MHEG model. The overall relationship of MHEG, DSM-CC and scripting languages is shown in Figure A-2.

Fig. A-2: DSM-CC-centric view of MHEG, scripting language, and networks



This figure shows that applications may access DSM-CC directly or through an MHEG layer. Also, scripting languages may be supported through an MHEG layer on top of DSM-CC. If an application requires an extensive object-oriented approach for object attributes and encapsulation classes, the MHEG layer may be considered.

Explicit support for database or file system access is not provided in MHEG through MHEG class definitions or APIs.

11. INFORMATIVE ANNEX B

Note: code contained in this section is for explanation purposes only. It is not intended for actual use, and is incomplete from the standpoint of being compilable source code.

11.1 Sample Port and Use of CD-ROM Multimedia Application

This code sample illustrates the porting and use of a CD-ROM application for the MPEG heterogeneous network environment. It is well known that CD-ROM applications are a popular form of multimedia entertainment, and many authoring tools are structured for creating such programs. CD-ROMs and video networks have one important thing in common: they both have a latency problem which requires the application to prefetch in advance the multimedia objects to be used in forming the presentation to the end-user. This sample shows the creation of a service which includes a Directory Interface. The Service's Directory hierarchy is constructed to match that of the equivalent CD-ROM application. When the application starts up at the settop, the Service is opened which effectively presents the CD-ROM filesystem to the client.

Prerequisites:

1. The content owner defines the CD-ROM service interface using Interfaces Define:

```
// Interfaces object reference is returned by an Open or Resolve to get
DSM_Interfaces *ifObject;
// information provider uniquely identifies the service with an id
// the id is unique for within that information provider's id space
DSM_Sequence id = {,8,0x1234FFFFFFFFF1};
// the Service Object interface is defined using IDL
string *intf =
"Module DSM {interface CDROMService : Service, Directory}";
rh DSM_Interfaces_Define(
      &ifObject,        // Interfaces object reference
      &ev,              // exception info
      &id,              // information provider's identifier of Service
      &intf,            // IDL interface definition
      aType,            // output, object type assigned
      &rVersion);       // output, initial or new version assigned
```

3. A reference for the Service is obtained using **LifeCycle_Create.**
4. The Service instance is registered with the ServiceGateway under the name "Encyclopedia" using **ServiceGateway_bind**. At this time it is bound to the ServiceGateway name context.
5. The OWNER attaches to a ServiceGateway using **ServiceGateway_Authenticate** chained to **ServiceGateway_Attach**. This reference is **sgObject.**
6. The OWNER opens the "Encyclopedia" Service:

```
// dbObject will be of type DSM_CDROMService
DSM_CDROMService *cdObject;
// create a reference Step where
// name is "Encyclopedia", type is DSM_CDROMService, and
// object ref is cdObject
DSM_RStep *node = {"Encyclopedia", DSM_CDROMService, *cdObject};
```

```
// Open the CDROMService, which resolves the reference cdObject
rh DSM_Directory_Open(
      &sgObject,              // target object this command is sent to
      &ev,                    // exception info
      DSM_DEPTH,        // path type is a single node
      &node);                 // contains name, type, ref
```

7. The Owner creates sub-Directories as needed and populates with the directory with the contents of the CD-ROM.

READER Procedure:

The READER in this example is the client (settop) application. After browsing the available applications offered by the ServiceGateway, the client open’s the “Encyclopedia” Service. Following the open, the client obtains the initial application shell through Data_Get or the Initial Application Download procedure as defined in section 3.1. The application shell, now at the client, takes charge of running the application. It will have the object reference of the Service, which represents the equivalent of the CD-ROM directory hierarchy. It can get, open and read multimedia file and data objects using the File and Directory Interfaces. It can open video and audio streams and manipulate them using the Stream Interface. It can navigate through its CD-ROM equivalent directory hierarchy using the Directory Interface.

11.2 Sample Creation and Use of a Movie Attributes Database Object

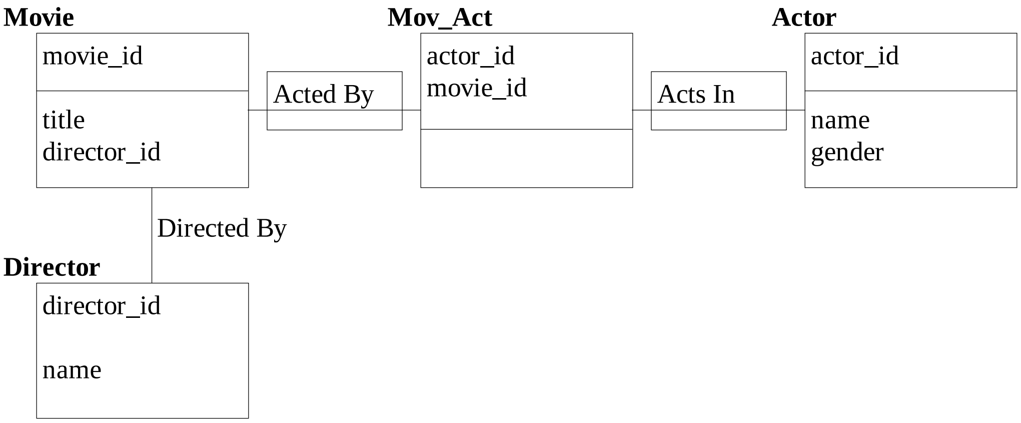
This code sample describes the creation and usage of a movie attribute database service. Such a service may be used to search for movies and information about movies using the standard SQL Structured Query Language. In a Movies on Demand scenario, prior to selecting a movie, the end-user is presented with list boxes of choices for titles, directors and actors, plus check boxes or radio boxes for other movie attributes. Since the server’s selection of movies changes from day to day, the information provider will continually update the database with the latest information on available movies. Without changing the application or requiring recompilation, the database may be updated at the server. When the end-user browses the database, graphics object selections cause the application to send database queries on the DSM interface to the database Service. The query reply can return new lists of titles, directors and actors, sorted and filtered, which are then displayed in the list boxes of the application presentation.

OWNER Procedure:

1. The content OWNER installs an empty database at a Server.
2. The database service is defined using **Interfaces Define.** This object includes includes the Service, View and Directory Interfaces in its IDL definition:

```
// ServiceGateway object reference is returned by ServiceGateway Attach
DSM_ServiceGateway *sgObject;
// information provider uniquely identifies the service with an id
// the id is unique for within that information provider’s id space
DSM_Sequence id = {,8,0x1234FFFFFFFFF2};
// the Service Object interface is defined using IDL
string *intf =
“Module DSM {interface DBService : Service, View, Directory}”;
rh DSM_Interfaces_Define(
      &sgObject         // ServiceGateway object reference
      &ev               // exception info
      &id,              // information provider’s identifier of Service
      &intf,            // IDL interface definition
      aType,            // output, object type assigned
      &rVersion);       // output, initial or new version assigned
```

3. A reference for the Service is obtained using **LifeCycle_Create.**
4. The Service instance is registered with the ServiceGateway under the name “Movie Info” using **ServiceGateway_Bind**. At this time it is bound to the ServiceGateway name context.
5. The OWNER attaches to a ServiceGateway using **ServiceGateway_Authenticate** chained to **ServiceGateway_Attach**. This reference is **sgObject.**
6. The OWNER opens the database service, then creates and populates the database tables with movie information:



```
// dbObject will be of type DSM_DBService
DSM_DBService *dbObject;
// create a reference Step where
// name is “Movie Info”, type is DSM_DBService, and
// object ref is dbObject
DSM_RStep *dbNode = {“Movie_Info”, DSM_DBService, *dbObject};
// Open the View, which resolves the reference dbObject
rh DSM_Directory_Open(
      &sgObject,              // this command is sent to ServiceGateway
      &ev,                    // exception info
      DSM_DEPTH,              // path traversal is DEPTH
      &dbNode);               // contains name, type, ref

// execute SQL statements to create and populate a movie table
string rSQLStatement =
  “CREATE TABLE movie(movie_id char(5), title char(30), director_id
   char(5))”;
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);

string rSQLStatement =
  “INSERT INTO movie VALUES (‘M3’, ‘Raiders of the Lost Ark’, ‘D2’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO movie VALUES (‘M2’, ‘BeetleJuice’, ‘D1’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO movie VALUES (‘M1’, ‘Batman’, ‘D1’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);

// execute SQL statements to create and populate a actor table
string rSQLStatement =
  “CREATE TABLE actor(actor_id char(5), name char(30), gender char(6))”;
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);

string rSQLStatement =
  “INSERT INTO actor VALUES (‘A7’, ‘Harrison Ford’, ‘male’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO actor VALUES (‘A6’, ‘Kim Bassinger’, ‘female’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO actor VALUES (‘A5’, ‘Michael Keaton’, ‘male’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO actor VALUES (‘A4’, ‘Alec Baldwin’, ‘male’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO actor VALUES (‘A3’, ‘Geena Davis’, ‘female’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO actor VALUES (‘A2’, ‘Karen Allen’, ‘female’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO actor VALUES (‘A1’, ‘Jack Nicholson’, ‘male’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);

// execute SQL statements to create and populate a director table
string rSQLStatement =
  “CREATE TABLE director(director_id char(5), name char(20))”;
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
string rSQLStatement =
  “INSERT INTO director VALUES (‘D2’, ‘Steven Spielberg’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
“INSERT INTO director VALUES (‘D1’, ‘Tim Burton’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);

// execute SQL statements to create and populate a mov_act table
string rSQLStatement =
  “CREATE TABLE mov_act(movie_id(5), actor_id char(5))”;
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
“INSERT INTO mov_act VALUES (‘M2’, ‘A4’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
“INSERT INTO mov_act VALUES (‘M2’, ‘A3’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
“INSERT INTO mov_act VALUES (‘M3’, ‘A7’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
“INSERT INTO mov_act VALUES (‘M3’, ‘A2’)”
```

```
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
“INSERT INTO mov_act VALUES (‘M1’, ‘A5’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
“INSERT INTO mov_act VALUES (‘M1’, ‘A1’)”
rh DSM_View_Update(&dbObject,&ev,&rSQLStatement);
“INSERT INTO mov_act VALUES (‘M1’, ‘A6’)”
```

READER Procedure:

The READER in this example is the client (settop) application. The READER follows a similar procedure (to Owner Procedure) for opening the View Service. A client with READER privileges only may not perform write operations such as View WriteSelect, but may perform read operations such as View Select.

To retrieve the name of the director of the movie Batman:

```
string rSQLStatement =
  “SELECT name FROM director d, movie m WHERE d.director_id =
   m.director_id AND m.title = ‘Batman’”;
rh DSM_View_Select(
      &dbObject,              // View object this is sent to
      &ev,                    // exception info
      aBufSize,               // size of memory allocated
      &rSQLStatement,         // SQL SELECT statement
      &rResultDescribe,       // description of return buffer
      &rReturnBuffer);        // sequence of result rows or objects
```

To retrieve the list of actors in Beetlejuice:

```
string rSQLStatement =
  “SELECT name FROM actor a, movie m, mov_act ma WHERE a.actor_id =
   ma.actor_id AND m.movie_id = ma.movie_id AND m.title =
   ‘BeetleJuice’”;
```

To retrieve the list of female actors in Beetlejuice:

```
string rSQLStatement =
  “SELECT a.name FROM actor a, movie m, mov_act ma WHERE a.actor_id =
   ma.actor_id AND m.movie_id = ma.movie_id AND m.title = ‘BeetleJuice’
   AND a.gender = ‘female’”;
```

DSM_**View_Read** can now be used to retrieve one actor at a time from the View.

12. INFORMATIVE ANNEX C

(This annex does not form an integral part of this International Standard)

[The format of this annex does not match that of the rest of the document and needs to be changed. This annex refers to User-User Application Download section.]

An Application Download Procedure Example Using MPEG-2 Transport

12.1 Scope

This informative annex is describes an example of what may occur during a boot sequence

12.2 Network configuration and restrictions

The network considered has the restriction that all high-speed data must be carried in a multiplexed MPEG-2 Transport Stream which is broadcast such that many client users are receiving and interpreting the same data. The Application Boot Messaging Path is a bidirectional network path exists which can only be received by the client user and server user involved in a transaction (perhaps an X.25 connection) and the unidirectional (Boot Server to Client) broadcast high-speed MPEG-2 Transport Stream connection.

The establishment of the Application Boot Messaging Path included negotion between the server and network a free MPEG-2 Program (with at least one MPEG-2 PID to use for Boot Messaging). The Application Boot Server and the Application Boot Initiator both know the Program number (and RF Frequency, if necessary) allocated to the Boot Messaging path. The Application Boot Initiator passes the Program number (and RF Frequency, if necessary) to the BLOB loader.

12.3 Application Boot Procedure chosen

The example contained in this appendix is constrained by the following Application Boot Procedure options:

One Pass BLOB download

dsmAppInfoReq and **dsmBLOBInfoResp** messages are not used

12.4 Application Boot Initiator Actions

The Application Boot Initiator performs the actions necessary to establish the preconditions specified above and in the Boot Sequence definition.

The ABI then notifies the BLOB Loader to begin the Boot Sequence, passing the appropriate network paths and information to it, then enters a sleeping or waiting state until the Boot Sequence is completed either successfully or unsuccessfully.

12.5 Application Boot Sequence

dsmAppBootReq message sent from BLOB Loader to Application Boot Server

The BLOB loader sends a valid **dsmAppBootReq** message over the Boot Messaging Path to the Application Boot Server. The BLOB buffer size allocated (as indicated in the message) is sufficient to receive the application modules which the ABS needs to send in order to support the client as described in the Client Configuration information element. The ABS has every component necessary to support the client available, and determines that it would like to respond with a **dsmBLOBDataResp**.

dsmBLOBDataResp message sent from Application Boot Server to BLOB Loader

The Application Boot Server collects all of the modules necessary to boot an initial application on the Client, then constructs and transmits the **dsmBLOBDataResp** message over the Application Boot Messaging Path using the allocated Program and one of the MPEG-2 PIDs allocated to that program (ISO/IEC 13818-1 DIS reserves stream type 0x08 for ITU-T Rec. H.222.0 | ISO/IEC 13818-1 DSM CC).

BLOB Loader actions

Using the MPEG-2 Program number (and RF frequency, if necessary), the BLOB loader receives and validates the **dspBLOBDataResp** message. If any errors have occurred, or the entire BLOB has not been received within the network timeout period, and the maximum network retry count has not been exceeded, the BLOB Loader restarts the Application Boot Sequence.

BLOB Loader notifies Application Boot Initiator of completion

The BLOB Loader notifies the ABI of successful completion of the BLOB loading process.

At this point the Application Boot Sequence is completed.

12.6 Application Boot Initiator Actions

After notifing the BLOB Loader to begin the Application Boot Sequence, the ABI entered a sleep or wait condition until notification by the BLOB Loader that a BLOB download was successful or not..

If notified by the BLOB Loader that the download failed, it is expected that the ABI would attempt to take some corrective action (if possible), or notify the Human Element or take some other unspecified action.

If the ABI was notified that the BLOB was loaded successfully, the ABI should perform the following actions:

Validate each individual downloaded module

Notify the operating system of the existance of each module

If the BLOB contained one or more applications, notify the operating system to begin execution of each of them. NOTE: it is possible for one BLOB to contain more than one application and to expect all of the applications to execute concurrently.

13. INFORMATIVE ANNEX D

(This annex does not form an integral part of this International Standard)

EXAMPLES OF USING DSM-CC USER-NETWORK MESSAGES WITH ATM

13.1 Introduction

The intent for this annex is to provide the reader an appreciation of DSM-CC operation over ATM. The normative Section of this International Standard defines the DSM-CC functional model, this is reproduced in Figure F1 below. In this annex examples are shown of how DSM-CC can be implemented over MPEG-2 based multimedia services configurations with ATM.

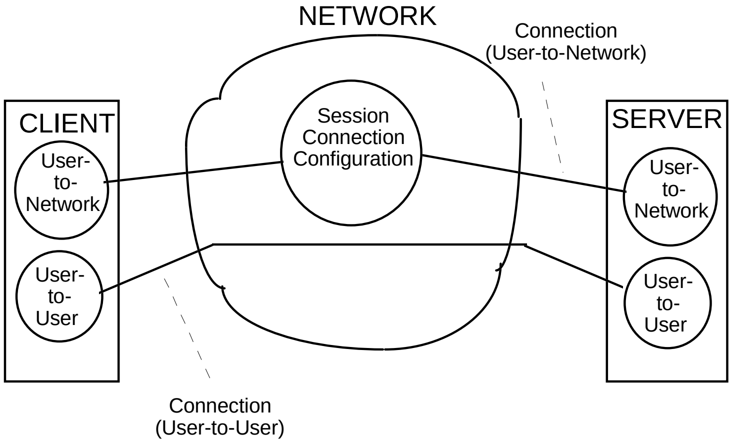


Figure F1: DSM-CC Functional Model

The following definitions taken from the normative text in this IS apply to Figure F1:

Connection: Transport link that provides the capability to transfer information between two or more end points.

Session: Association between two or more Users, providing the capability to group together the resources needed for an instance of a Service.

Service: A logical entity in the system that provides functions and interfaces in support of one or more applications. The distinction of the service from other objects is that end-user access to it is controlled by a service gateway.

Network: A collection of communicating elements that provides connections and may provide session control and/or connection control to Users.

Client: Consumer of a service from one or more Servers.

Server: Provider of a service to one or more Clients.

User: An end system that is connected to a Network and can transmit information to or receive information from other such systems by means of the Network. A User may function as a Client, Server, or both.

This annex presents a number of configurations conformant to the DSM-CC functional model. Figure F2 shows the categories of configurations. These are:

- Hybrid ATM Core-Shared Access MPEG Transport Stream (e.g., on Hybrid Fiber Coax)
- Hybrid ATM Core-Shared Access ATM (e.g., on Hybrid Fiber Coax)
- End-to-end ATM segregated with Proxy (e.g., ATM network with Client and/or Server Q.2931 proxy signaling, but Session control done outside Q.2931)
- End-to-end ATM segregated direct (e.g., ATM network with Q.2931 by Client and Server, but Session control done outside Q.2931)
- End-to-end ATM integrated (e.g., ATM network with Session control done on future "extended" Q.2931)

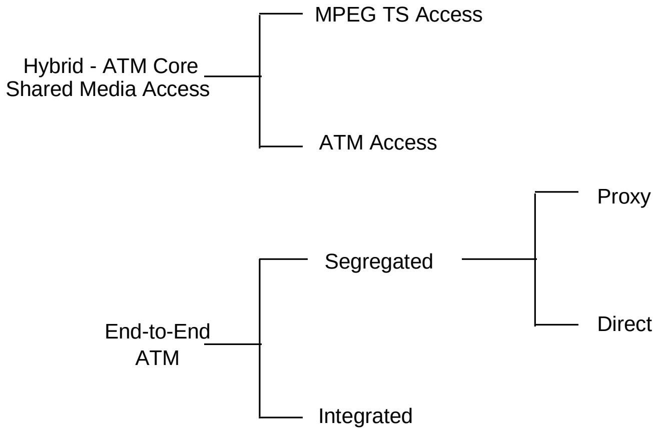


Figure F2: Configuration Taxonomy

Figure F3 shows the characteristics of the above configurations from the perspective of the ATM control plane (signaling) and the ATM user plane (ATM Bearer).

Configuration	Q.2931 Signaling End points	Client Proxy	Server Proxy	ATM Bearer Channel
Hybrid- ATM Core Shared Media (MPEG TA or ATM) Access	SM * <--> Server	Not Applicable	-	IWU*** <--> Server
	SM <--> Server PSA**		yes	
End-to-End ATM with Proxy for connection setup	Client PSA <--> Server	yes	-	Client <--> Server
	Client <--> Server PSA	-	yes	
	Client PSA <--> Server PSA	yes	yes	
End-to-End ATM Segregated Direct	Client <--> Server	-	-	Client <--> Server
End-to-End ATM Integrated	Client <--> Server	-	-	Client <--> Server

* SM = Session Manager is an implementation of the Network functions of the DSM-CC functional model in a block external to the ATM network.
**PSA = Proxy Signaling Agent
***IWU = InterWorking Unit

Figure F3: Configuration Characteristics

13.2 Associations Between ATM SVC and DSM-CC

13.2.1 Methods

From the basic principle of ATM SVC connectivity under the context of a multimedia service session, two methods that tie the Call Reference (which represents the Q.2931 call concept) and the resourceNum (which represents the DSM-CC session concept) are discussed:

1. The Session Method
2. The Network Method

It is NOT possible to use both the Session method and the Network method within the same session.

13.2.1.1 Session Method

The Session Method requires that any Server that needs an ATM SVC connectivity must inform the SM. The SM then initiates a Q.2931 signaling procedure to establish an ATM SVC connection between the Server and a Client (e.g., through a downstream headend equipment). The Q.2931 SETUP message initiated by the SM to the Server includes the resourceId (in its entirety or in a compressed form see Section 2.2) to allow the Server to associate the resources with ATM connections.

The characteristics of the Session Method are as follows:

1. It maintains the DSM-CC session layer approach which views an ATM connectivity as one of the resources to be assigned by the SM.

2. It allows the multimedia Network provider to make high layer policy decision on all resource requests, including the ATM connection resource (e.g., restrictions on the number of ATM connections per session).
3. It allows the abstraction of the connection resources so that the multimedia Network architecture can be hidden (e.g., using the concept of Class of Service instead of specifying detailed ATM parameters).
4. It allows multiple ATM connections to be requested and torn down through one session request.

13.2.1.2 Network Method

The Network Method allows any Server that needs an ATM SVC connectivity to do a Q.2931 signaling to the Client. For those Networks which have Q.2931 signaling available at the Client, the Client may also do Q.2931 signaling to the Server to initiate an ATM SVC. The Q.2931 SETUP message that the Server or Client sends to the ATM UNI is required to contain a resourceId ((in its entirety or in a compressed form see Section 2.2)) in an information element that has end-to-end significance (i.e., the ATM network will transfer it transparently between Server, Client or SM. In the case when SM is involved (because of interworking with the access media or Proxy signaling) and it receives a SETUP from its side of the UNI, it will link up the Server with the Client (for Server initiated SETUP) or it will link up the Client with the Server (for Client initiated SETUP). In the case of Client to Server connectivity for Networks which have Q.2931 signaling available at both Client and Server, the involvement of the SM is typically not required in establishing the connection.

The characteristics of the Network method are as follows:

1. It involves less signaling steps in establishing an ATM connection, while still preserving the association between a session layer's resource and the ATM connection.
2. It reduces the capability of the multimedia Network provider to make high layer policy decision on the ATM connection resource.
3. It can be used by Servers/Clients that request large volume of ATM connections to be set up with time constraints.

Session and Network are two methods which could be used on any of the configurations identified in Figure F2.

13.2.2 resourceId Mapping into B-HLI Correlation ID

As described in Section 2.1 the resourceId needs to be carried along the ATM SVC connection to allow the Users to associate a connecton with resources. The current view is that B-HLI in Q.2931 is the proper field to carry the resourceId. For this purpose a specific B-HLI type for MPEG-2 DSM-CC is required. A codepoint for ISO/IEC 13818-6 MPEG-2 DSM-CC has been requested to ITU-T SG11 Q15/11 who is responsible for providing the codepoint. However the B-HLI information field is currently limited to a maximum size of 8 bytes which is not sufficient to carry the resourceId field of 12 bytes specified in the normative body of this specification. In order to overcome this shortcoming two solutions are specified for use in case of using MPEG-2 DSM-CC with ATM SVC. These are an IMMEDIATE solution and a SUBSEQUENT solution. The Immediate solution will be applied in case of the B-HLI information field is constrained to a maximum of 8 bytes and the Subsequent solution will be applied when the B-HLI field is increased to accomodate the resourceId field or as an alternative an end-to-end information element is created to carry the full resourceId.

13.2.2.1 Immediate Mapping to B-HLI Correlation ID

In order to be able to compress the resourceId field of 12 bytes to fit within an 8 byte B-HLI information field, the deviceId values are created with local significance in implementations using MPEG-2 DSM-CC with ATM SVC. With the consequence that the DSM-CC Networks will have a relatively limited size. In addition the maximum number of Sessions is reduced from its normative value in order to maintain an overall sessionId field size of 5 bytes down from 10 bytes as shown in Figure F4. In addition the 5 bytes reduced sessionId value follows a specific format is adopted as shown below:

MSb 1,2 = 13818-6 reserved (00)

a) Network assigned
MSb 1-2 = Network assigned (11)
MSb 3-32 = reduced sessionId value

b) Server assigned
MSb 1-2 = Server assigned (10)
MSb 3-20 = reduced deviceId value of network domain significance.
MSb 21-40 = reduced session number value

c) Client assigned
MSb 1-2 = Server assigned (01)
MSb 3-32 = reduced deviceId value of network domain significance.
MSb 33-40 = reduced session number value

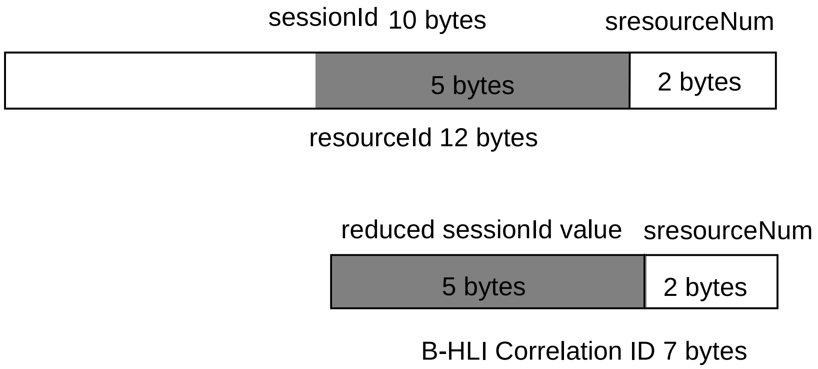


Figure F4: Relationship of Immediate Correlation ID to resourceId

13.2.2.2 Subsequent Mapping to B-HLI Correlation ID

This mapping will allow the full resourceId of 12 bytes to be carried within the enhanced B-HLI field. In this case the B-HLI Correlation ID will be identical to the resourceId.

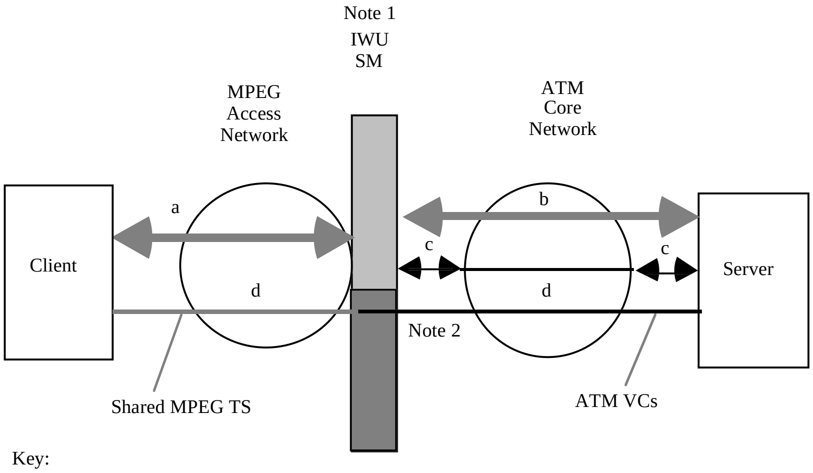
13.3 Hybrid ATM Core-Shared Access MPEG Transport Stream

This DSM-CC configuration over ATM, uses MPEG transport stream on the access network and ATM on the core network, see Figure F5. An interworking unit (IWU) operates between the core and the access network. In addition to interworking, the unit contains functions of end-to-end session management control, access network connection management control and configuration management control. The entire unit is designated as IWU SM (Session Connection, and Configuration Management Controller).

- Messages between the Client and SM consist of both session and connection control messages (a). These include DSM-CC User-to-Network messages shown in Figure F1.
- Messages between SM and the Server (b) are DSM-CC User-to-Network messages consisting mainly of session control messages.
- The connection control messages (c) are interchanged between the Server and the ATM core network and the SM and the ATM core network using Q.2931. Some of the information in these messages are

carried end-to-end between the Server and SM. At this time a minimum limit of 34 bytes has been identified but no such information elements have been standardized in Q.2931.

The User-to-User connections (d) between the Client and the Server are established across both the MPEG TS access network and the core ATM network. The establishment of these connections within the context of a session occurs through the a, b and c message exchanges. Because of the differences in the transport media of the MPEG TS access and the core ATM network, the network resources in each are different. A session on the MPEG TS access network groups resources, identified by Packet Identifiers (PIDs) to carry video program(s) shown in Figure F6 on the left side. The same session on the ATM network includes virtual channels each of which may carry an MPEG transport with a single program,



Key:

a: DSM-CC User-to-Network messages (handles Session and Connection resources)
b: DSM-CC User-to-Network messages (handles Session resources only)
c: B-ISDN Signaling
d: User-to-User messages

Note 1: An optional role of the IWU is to also manage the access network resources.
Note 2: The flow of content data is out of scope and is not shown

Figure F5: DSM-CC over ATM/Shared Media Non ATM

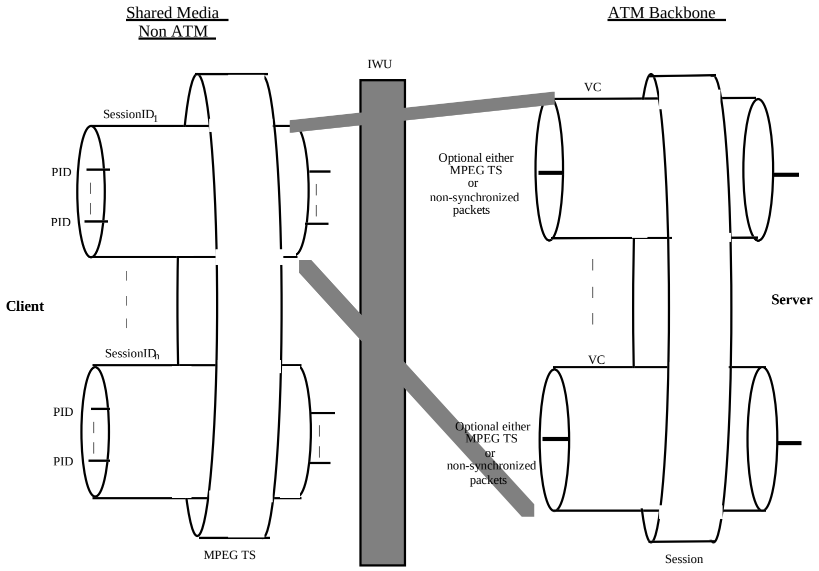


Figure F6: Session Network Resource Interworking between Shared Media and ATM Backbone

on the right side. Figure F6 illustrates the same session on both sides of IWU. Because of the difference of the network resources a mapping from one to the other is required for interworking. A session is identified with a global *sessionId*.

13.3.1 **Session Method Scenarios**

To clarify the Session Method a Session Set-up scenario will be explored in detail. Additional Session method scenarios will be provided which include the following:

- Resource Request
- Resource Deletion
- Session Tear-Down

13.3.1.1 Session Set-Up

This session is either established by a request from the Client (Client Session Set-Up scenario) or from the Server itself (Server Session Set-Up scenario).

13.3.1.1.1 Client Session Set-Up

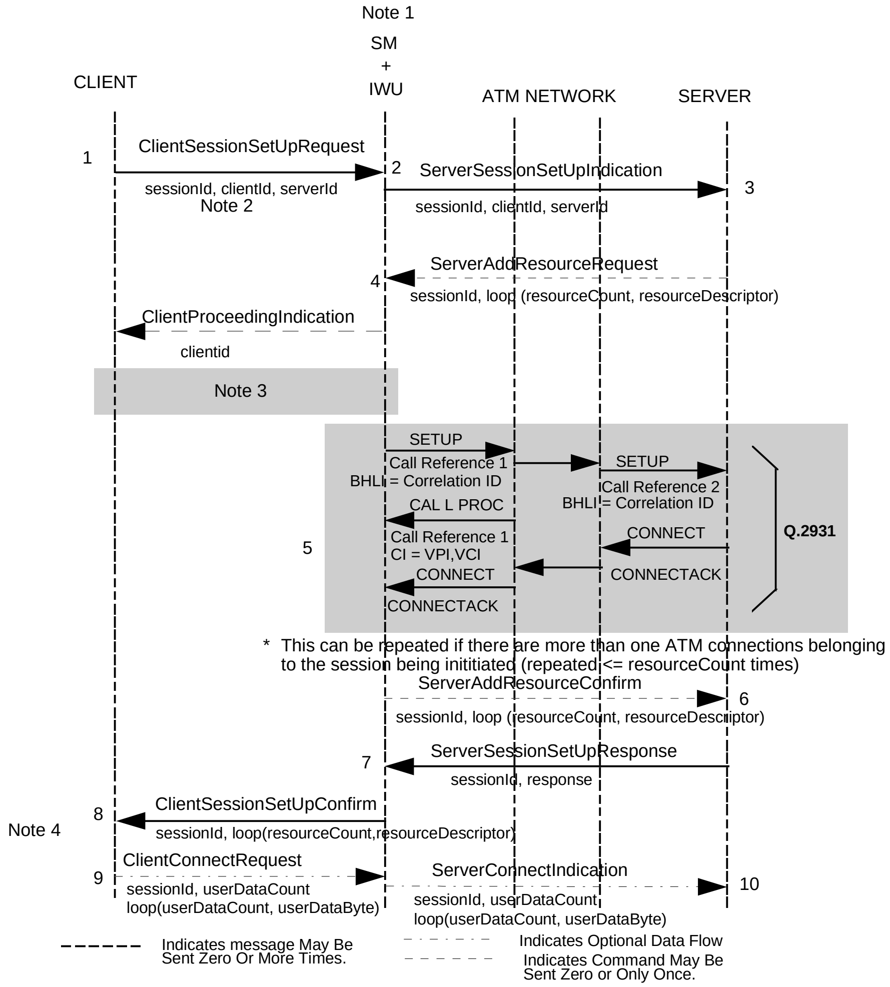

Note 1
SM
+
IWU
CLIENT
ATM NETWORK
SERVER
1
ClientSessionSetUpRequest
sessionId, clientId, serverId
Note 2
2
ServerSessionSetUpIndication
sessionId, clientId, serverId
3
ServerAddResourceRequest
4
sessionId, loop (resourceCount, resourceDescriptor)
ClientProceedingIndication
clientid
Note 3
SETUP
Call Reference 1
BHLI = Correlation ID
SETUP
Call Reference 2
BHLI = Correlation ID
CAL L PROC
Call Reference 1
CI = VPI,VCI
CONNECT
CONNECTACK
CONNECT
CONNECTACK
5
Q.2931
* This can be repeated if there are more than one ATM connections belonging
to the session being inititiated (repeated <= resourceCount times)
ServerAddResourceConfirm
6
sessionId, loop (resourceCount, resourceDescriptor)
7
ServerSessionSetUpResponse
sessionId, response
8
ClientSessionSetUpConfirm
Note 4
sessionId, loop(resourceCount,resourceDescriptor)
ClientConnectRequest
9
sessionId, userDataCount
loop(userDataCount, userDataByte)
ServerConnectIndication
10
sessionId, userDataCount
loop(userDataCount, userDataByte)
Indicates message May Be Sent Zero Or More Times.
Indicates Optional Data Flow
Indicates Command May Be Sent Zero or Only Once.

Figure F7: Typical Sequence of Events: Client Session Set-Up

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.
Note 2: Only relevant parameters in each message are shown.
Note 3: Connection Control messages with Client will be specific to the type of access network.
Note 4: Connections for the exchange of User-Network messages between Client and SM and Server and SM are assumed.

Figure F7 provides a typical sequence of events for Client Session set-up. Before a Server can request ATM SVC connectivity to the Client (i.e., before a Server may issue a ServerAddResourceRequest), it must have begun establishment of an MPEG-2 based multimedia service session (i.e., it must have issued a ClientSessionSetUpRequest).

DSM-CC protocol:

Step 1
The Client sends a ClientSessionSetUpRequest message to the SM.

Step 2
The SM verifies the *clientId* from the Network provider’s point of view, and if positive, contacts the proper Server that has the service identified by *serverId* .

Step 3
Upon receipt of the ServerSessionSetUpIndication message, the Server verifies the *clientId* from the Server's point of view, and if positive accepts the request. The Server collects all resources that it needs from the Network in order to support the selected service. If the service needs one or more ATM SVC connections, the Server can embed the proper amount of "ATM SVC" resource descriptors in the ServerAddResourceRequest message. These resource descriptors are tagged for negotiation, and contain sufficient information for the SM to later establish a connection between the Server and the IWU in SM.

The encoding of the "AtmConnection" resource descriptor uses ATM UNI containing Q.2931 ATM User Cell Rate and Quality-of-Service (QoS) parameters.

Step 4
Upon receipt of the ServerAddResourceRequest message, the SM can process the resource descriptors included in the message. For an "ATM SVC" resource, the SM can verify the requested properties (e.g., User Cell Rate and QoS) against what is available in the access network. If the "ATM SVC" resource descriptor is tagged as MANDATORY NEGOTIABLE and the available value is within the requested range, then it can be satisfied and will be recommended by the SM to the Server.

Step 5
For any "ATM SVC" resource that is not rejected, the SM initiates a connection matching an available access resource and containing both the sessionId and the resourceNum in the SETUP message.

Q.2931 protocol:

The SM initiates a Call/Connection procedure by sending a Q.2931 SETUP message to its ATM User-Network Interface (UNI), with the following information elements:

- Call reference selected by SM
- Calling Party Number = ATM address of the IWU.
- Called Party Number = ATM address of the Server as derived from the *serverId* field.
- ATM Adaptation Layer Parameters ATM User Cell Rate and Quality-of-Service parameter = values imported from the "ATM SVC" resource request descriptor.
- Broadband High Layer information (BHLI) = *Correlation ID* corresponding to this "ATM SVC" resource.

After connections are established the SM can send a ServerAddResourceConfirm message indicating the resources which were successfully allocated.

The ATM UNI acknowledges the SM with a CALL PROCEEDING message. From that message, the SM extracts the assigned VPI and VCI values. These values will be used by the SM to connect the Client to the ATM SVC connection being set up.

The exact procedure to connect the Client to the ATM SVC being set up depends on the network architecture. In a passband architecture, this procedure may involve the SM translating the VPCI/VCI value into an RF channel and other multiplexing information (such as MPEG-2 program number) which will be sent to the Client via the DSM-CC ClientSessionSetUpConfirm message. The details of this procedure is outside the scope of this annex.

When the Server is informed of the SETUP on its ATM UNI, it can access the *Correlation ID* from the BHLI information, and associates the Call Reference with this session information. The Server maintains this association until the ATM SVC connection is released.

Both the SM and the Server maintain a Call Reference and *Correlation ID* association so that one can be retrieved from the other.

DSM-CC Protocol:

Step 6
The ServerSessionSetUpResponse message will signal SM of the Server's readiness to begin using the connections.

Step 7
After all requested resources have been assigned, including the "ATM SVC" resources, the SM will inform the Client through the ClientSessionSetUpConfirm message. Both Client and Server are now ready to exchange User-to-User messages.

Step 9
Optionally the ClientConnectRequest can be used. This will be sent by the Client to indicate to SM the acceptance of the Session and pass along data to the Server.

Step 10
The SM notes the starts of the Session and informs the Server to begin the Session.

13.3.1.1.2 Server Session Set-Up

For Further Study

13.3.1.2 **Resource Request**

After a session has been set up between the Client and the Server, either the Client or the Server can later come back to the SM to request new resources within the context of the established session

13.3.1.2.1 ***Resource Request by the Server***

The message flow is shown in Figure F8.

Step 1
The Server requests the needed connection resources by sending the ServerAddResourceRequest message to SM with a list of resourceDescriptors.

Step2
If DSM-CC does not reject the requested "ATM SVC" resources, it proceeds by securing a connection on the access network portion (through vendor specific commands) and initiates an ATM core network connection matching the available access resources through Q.2931 signaling messages, containing both the sessionId and the resourceNum in the SETUP message.

Step 3
SM informs the Client of the requested Client side resources.

Step 4
On receipt of ClientAddResourceIndication message, the Client determines if it is capable of using the additional resources and if positive sends ClientAddResourceResponse to the network with the response field set to rspOK. At this point the Client shall consider the additional resources as committed to the Session.

Step 5
After all requested connection are established and a positive ClientAddResourceResponse is received, the SM sends a ServerAddResourceConfirm message indicating the additional resources were successfully allocated. After sending the message, SM shall consider the additional resources as being committed to the Session.

Step 6
On receipt of the ServerAddResourceConfirm the Server shall consider the additional resources as being committed to the Session.

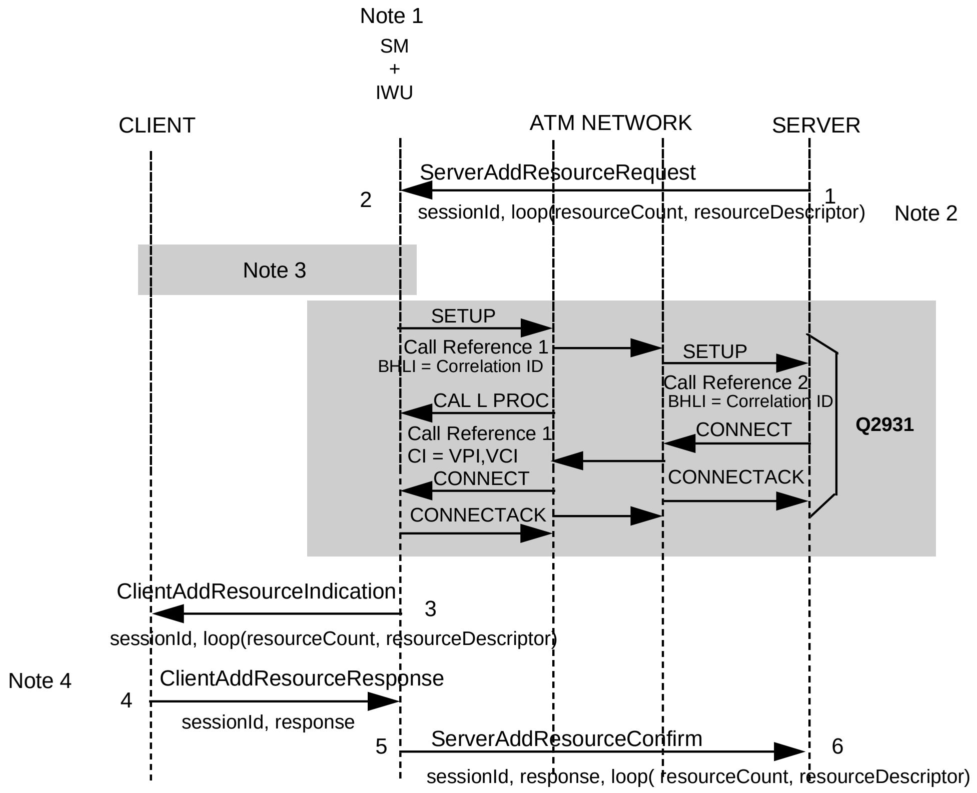


Figure F8: Typical Sequence of Events for Session Method: Server Resource Request

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.
Note 2: Only relevant parameters in each message are shown.
Note 3: Connection Control messages with Client will be specific to the type of access network.
Note 4: Connections for the exchange of User-Network messages between Client and SM and Server and SM are assumed.

*13.3.1.2.2 **Resource Request by the Client***

For Further Study

13.3.1.3 **Resource Deletion**

After a resource has been successfully requested, either through the Session Set-Up scenario or the Session Resource Request scenario, the originator of the resource request (Client or Server) can later come back to the SM to request that the resource is to be deleted.

13.3.1.3.1 ***Resource Deletion by the Server***

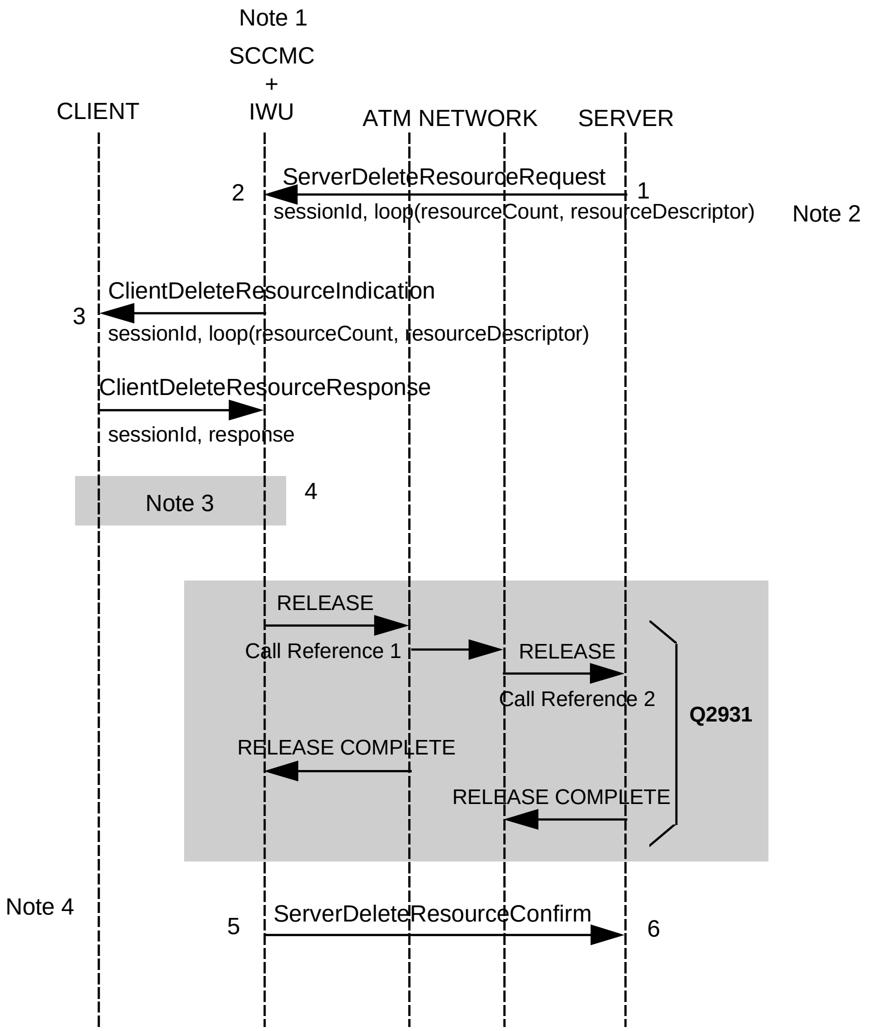

Note 1
SCCMC
+
IWU
CLIENT
ATM NETWORK
SERVER
ServerDeleteResourceRequest
2
1
sessionId, loop(resourceCount, resourceDescriptor)
Note 2
ClientDeleteResourceIndication
3
sessionId, loop(resourceCount, resourceDescriptor)
ClientDeleteResourceResponse
sessionId, response
Note 3
4
RELEASE
Call Reference 1
RELEASE
Call Reference 2
Q2931
RELEASE COMPLETE
RELEASE COMPLETE
Note 4
5
ServerDeleteResourceConfirm
6

Figure F8: Typical Sequence of Events for Server Resource Deletion

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.

Note 2: Only relevant parameters in each message are shown.

Note 3: Connection Control messages with Client will be specific to the type of access network.

Note 4: Connections for the exchange of User-Network messages between Client and SM and Server and SM are assumed.

DSM-CC protocol:

Step 1
The Server informs the SM of its request for deletion of one or multiple assigned resources via the ServerDeleteResourceRequest message. The resources are identified by their corresponding *Correlation ID*.

Step 2
The SM then informs the Client via the ClientDeleteResourceIndication message so that it can stop using the identified resources.

Step 3
The Client accepts the deletion of resources by responding with a ClientDeleteResourceResponse with the response field set to rspOK. At this point the Client shall consider the resource deletion process completed and shall not use the deleted resources.

Step 4
Upon receipt of the ClientDeleteResourceResponse message, The SM will process the deletion of the identified resources from the established session.

If the SM detects a *Correlation ID* identifying an “ATM SVC” resource, it can retrieve the corresponding Call Reference and initiates a Q.2931 Call/Connection Clearing procedure at its ATM UNI.

Q.2931 protocol:

The SM sends a Q.2931 RELEASE message to the UNI, with the following information elements:

Call Reference = Call Reference retrieved from the *Correlation ID*.

When the Server receives the corresponding RELEASE message at its UNI, it retrieves the Call Reference from the message and from the Call Reference, the associated *Correlation ID* which is then given to the session in the Server for housekeeping chores.

DSM-CC protocol:

Step 5
After all the resources have been deleted, The SM will inform the Server of the outcome of the operation via the ServerDeleteResourceConfirm message. At this point the SM may consider the resource deletion completed.

Step 6
On receipt of the ServerDeleteResourceConfirm, the Server will consider the resource deletion procedure completed.

13.3.1.3.2 ***Resource Deletion by the Client***

For Further Study

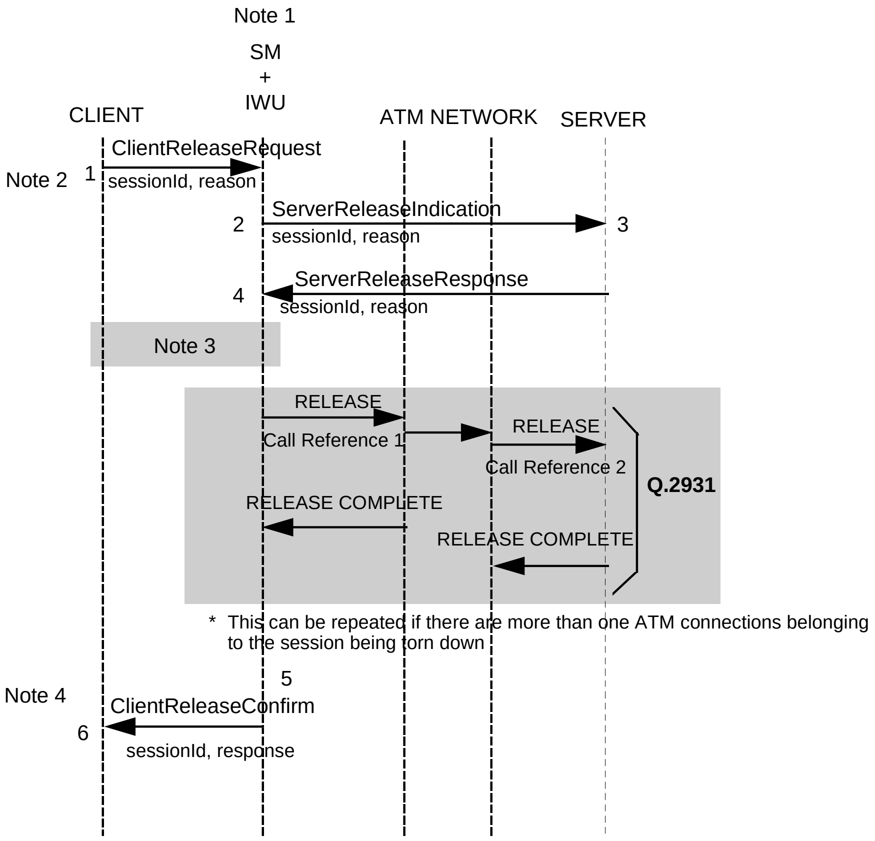
13.3.1.4 Session Tear-Down

After a session has been established through the Session Set-Up, either the Client or Server can later come back to the SM to request that session is to be torn down. For a session containing an "ATM SVC" resource, the scenarios follow below.

13.3.1.4.1 ***Session Tear-Down by Server***

The message flows in this scenario are symmetrical to the flows in the Session Tear-Down by Client shown in 3.1.4.2 below.

13.3.1.4.2 ***Session Tear-Down by Client***



* This can be repeated if there are more than one ATM connections belonging to the session being torn down

Figure F10: Typical Sequence of Events for Session Method: Client Session Tear-Down

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.
Note 2: Only relevant parameters in each message are shown.
Note 3: Connection Control messages with Client will be specific to the type of access network.

Note 4: Connections for the exchange of User-Network messages between Client and SM and Server and SM are assumed.

DSM-CC protocol:

Step 1
The Client informs the SM of its request for session tear-down via the ClientReleaseRequest message.

Step 2
The SM then informs the Server via the ServerReleaseIndication message.

Step 3
The Server acknowledges the request with a ServerReleaseResponse message. At this point the Server shall consider the Session terminated.

Step 4
Upon receipt of the ServerReleaseResponse message, the SM will retrieve all resources allocated to the identified session and proceed to delete those resources.

If the SM detects an “ATM SVC” resource belonging to the session, it can retrieve the corresponding Call Reference from the *Correlation ID* and initiates a Q.2931 Call Connection Clearing at its ATM UNI.

Q.2931 protocol:

The description of this Call/Connection Clearing procedure is similar to the one in the Resource Deletion scenario.

DSM-CC protocol:

Step 5
After all the resources have been deleted, the SM will inform the Client of the outcome of the operation via the ClientReleaseConfirm message.

Step 6
On receipt of the ClientReleaseConfirm the Client shall release all resources assigned to the Session. At this point the Client shall consider the Session terminated.

13.3.2 Network Method Scenarios

To clarify the Network Method a Session Set-up scenario will be explored in detail. Additional Network method scenarios will be provided which include the following:

- Resource Request
- Resource Deletion
- Session Tear-Down

13.3.2.1 Session Set-Up

In the Network Method, before a Server can request an ATM SVC connection to the Client or a Client can request an ATM connection to a Server, a multimedia session must have been previously established. This session is established via either a Client or Server Session Set-Up scenario as described by the DSM-CC session protocol.

13.3.2.1.1 Client Session Set-Up

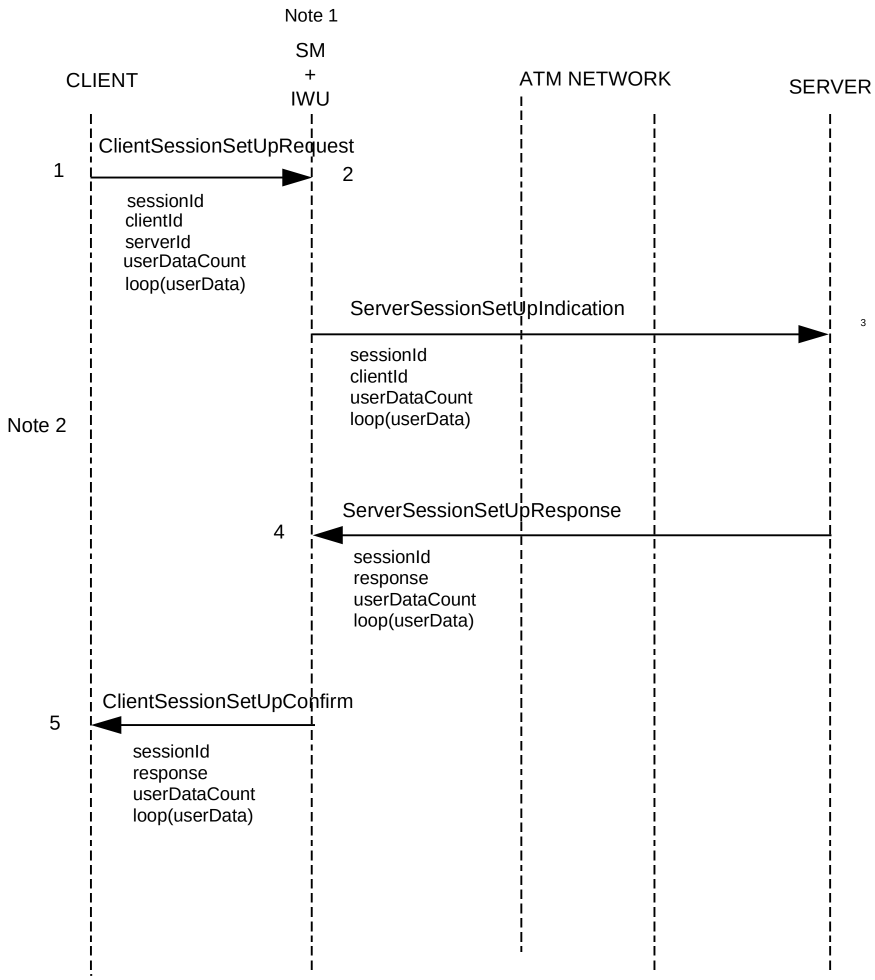


Figure F11: Network Method: Client Session Set -Up

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.

Note 2: Connections for the exchange of User-Network messages between Client and SM and Server and SM are assumed.

During this Session Set-Up phase, the Client does not request any ATM SVC resources. Instead, all the ATM SVC connections will be initiated later from the Server or Client using Q.2931 associated Call/Connection procedures. In order to tie these future ATM connections to the session layer protocol, the SM assigns a sessionId as part of the session establishment sequence. This sessionId must be included in all subsequent Resource Requests. The SM is limited in the manner in which it may audit the resulting ATM connections. Auditing will need to be done at the Q.2931 layer (using Call Reference values) as opposed to the DSM-CC layer using Correlation IDs.

13.3.2.1.2 Server Session Set-Up

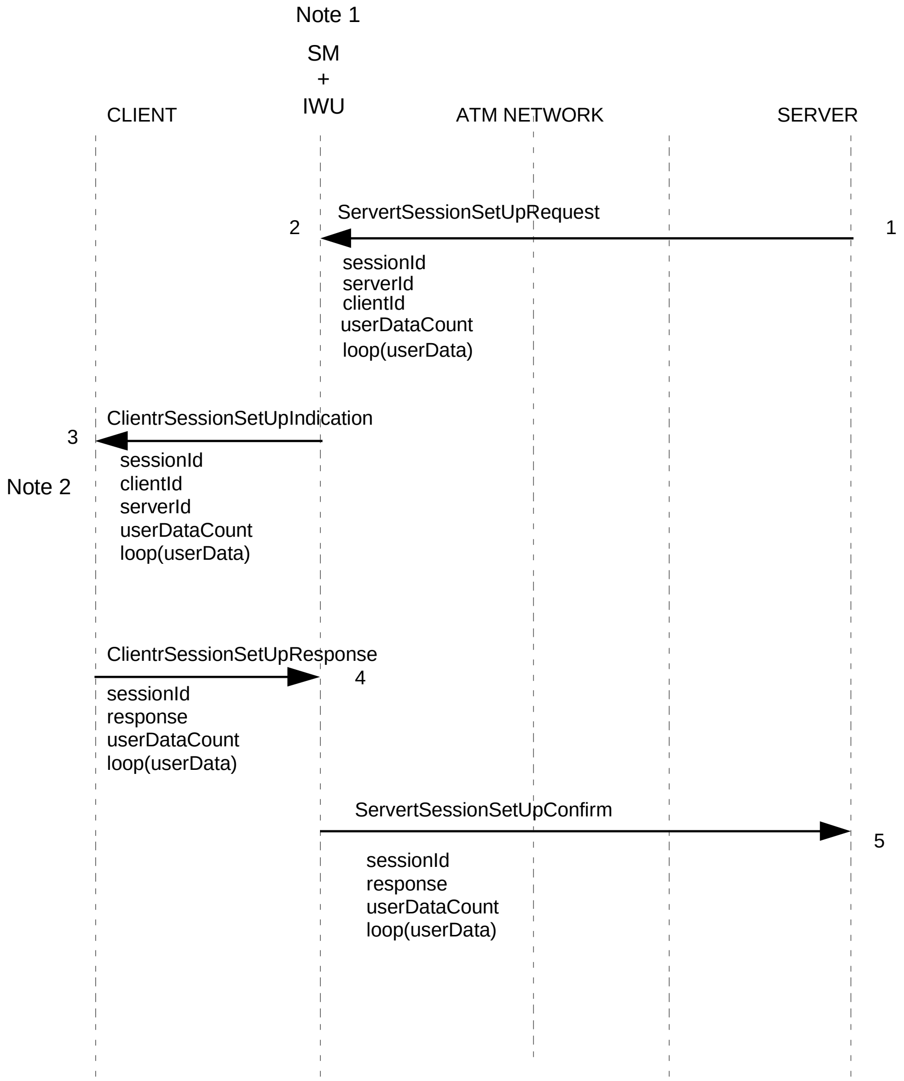


Figure F12: Network Method: Server Session Set -Up

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.

Note 2: Connections for the exchange of information between Client and SM and Server and SM are assumed.

13.3.2.2 **Resource Add Request**

13.3.2.2.1 ***Resource Request by the Server***

Whenever the Server needs an ATM SVC connection, it will initiate a Q.2931 Connection Set-Up procedure as shown below:

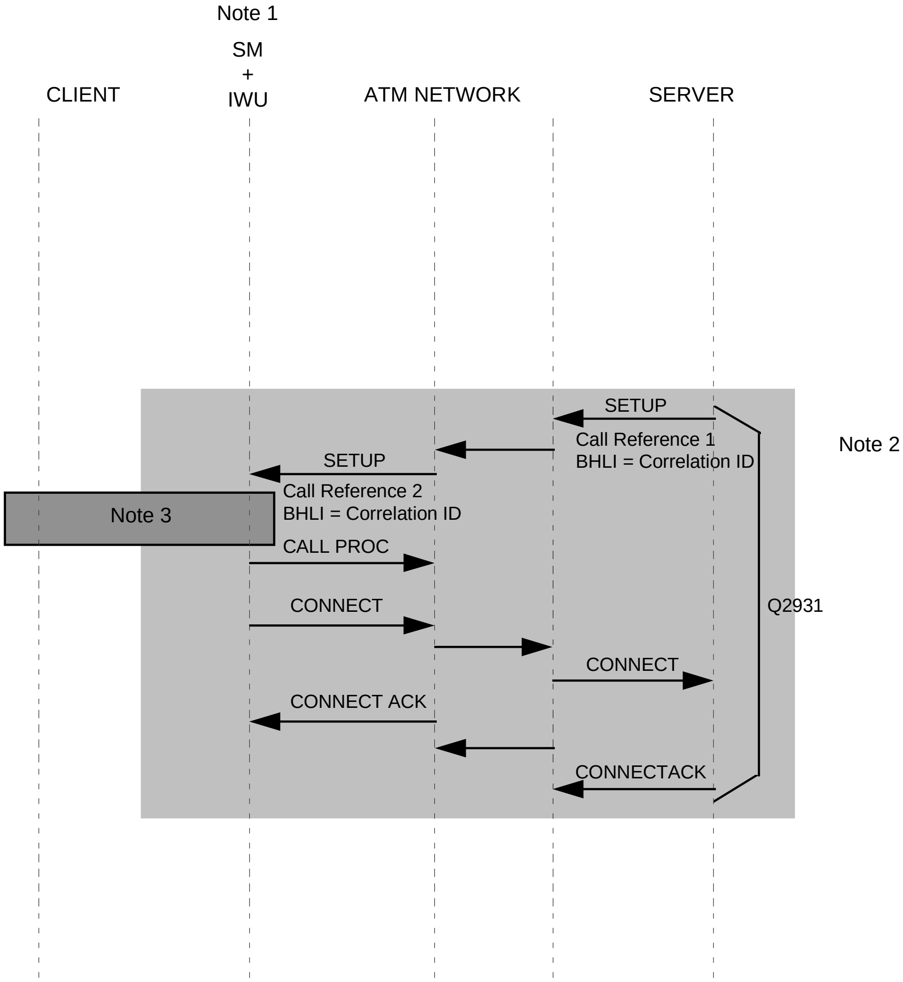


Figure F13: Network Method: Server Session Set -Up

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.
Note 2: Only relevant parameters in each message are shown.
Note 3: Connection Control messages with Client will be specific to the type of access network.

The Server sends a Q2931 SETUP message to its ATM User-Network Interface (UNI), with the following information in elements:

Call Reference = selected by Server

Calling Party Number = ATM address of the Server
Called Party Number = ATM address of SM
Broadband High Layer information (BHLI) = sessionId

When the SM receives the SETUP message on its UNI, it recovers the Call Reference information element and the sessionId from the SETUP message. From the sessionId, the SM knows who is the Client and proceeds to establish a path from the IWU to the Client and connect that path with the ATM connection that the Server just established.

13.3.2.2.2 ***Resource Request by the Client***

A Client may also request an ATM resource using the Network method. Whenever the Client needs an ATM SVC connection, it will initiate a Q.2931 Connection Set-Up procedure.

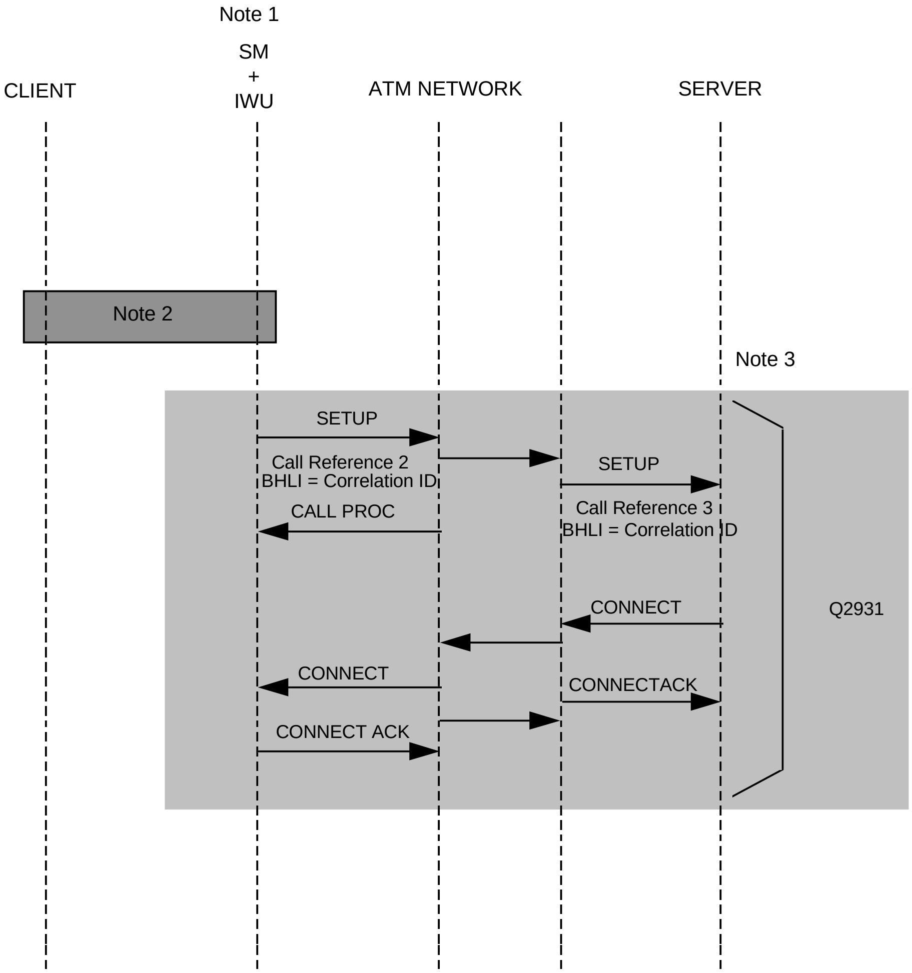


Figure F14: Network Method: ATM Resource Request - Client Initiated

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.
Note 2: Connection Control messages with Client will be specific to the type of access network.
Note 3: Only relevant parameters in each message are shown.

The Client requests SM a connection to the Server. The SM sends a Q2931 SETUP message to its ATM User-Network Interface (UNI), with the following information in elements:

Call Reference = selected by SM
Calling Party Number = ATM address of SM
Called Party Number = ATM address of the Server
Broadband High Layer information (BHLI) = sessionId

When the SM sends the SETUP message on its UNI, it associates the Call Reference information element with the sessionId. From the sessionId, the SM knows who is the Server and proceeds to establish a path to the Server and connects that path with the ATM connection that the Client just established.

13.3.2.3 **Connection Clearing**

13.3.2.3.1 ***Connection Clearing by the Server***

The Server can delete an established ATM SVC connection by initiating the Q.2931 Connection Clearing procedure:

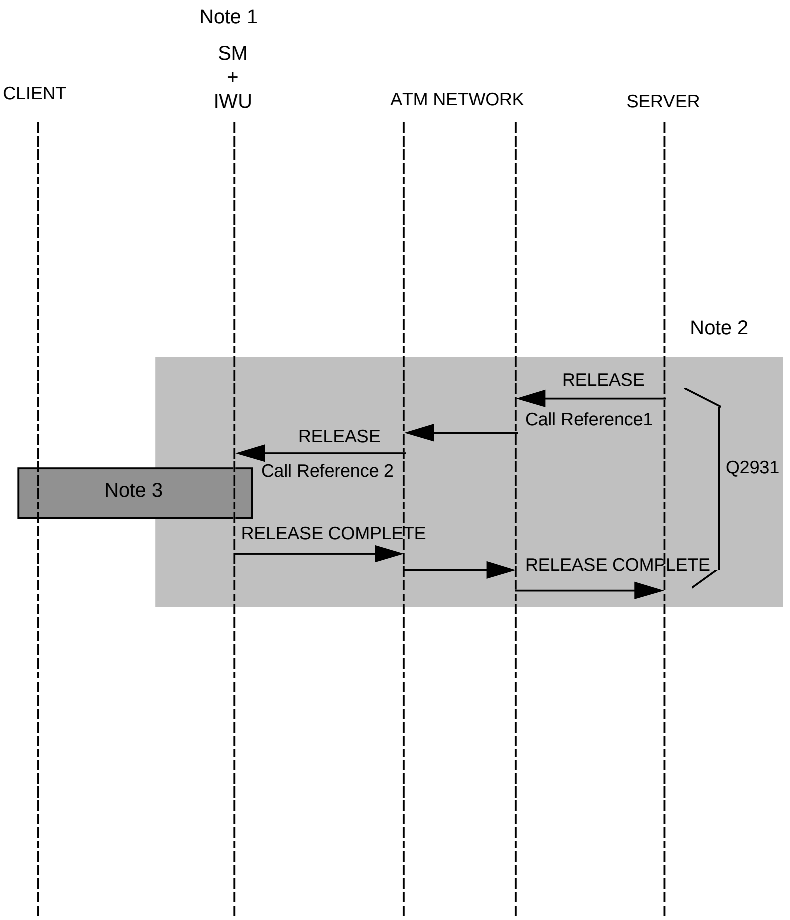


Figure F16: Network Method: Resource Deletion via Call/connection Release - Initiated by Server

Note 1: Optional role of the IWU is to manage the access network resources in case of Hybrid ATM Network- MPEG TS.
Note 2: Only relevant parameters in each message are shown.
Note 3: Connection Control messages with Client will be specific to the type of access network.

When the SM receives the RELEASE message on its UNI, it keys off the Call Reference to retrieve the sessionId.

The SM then releases the access network path and closes the usage record of the resource associated with this Call Reference.

When the Server receives the RELEASE COMPLETE message on its UNI, it keys off the Call Reference to retrieve the sessionId.

13.3.2.3.2 *Connection Clearing by the Client*

The Client can delete an established ATM SVC connection by initiating the Q.2931 Connection Clearing procedure through SM:

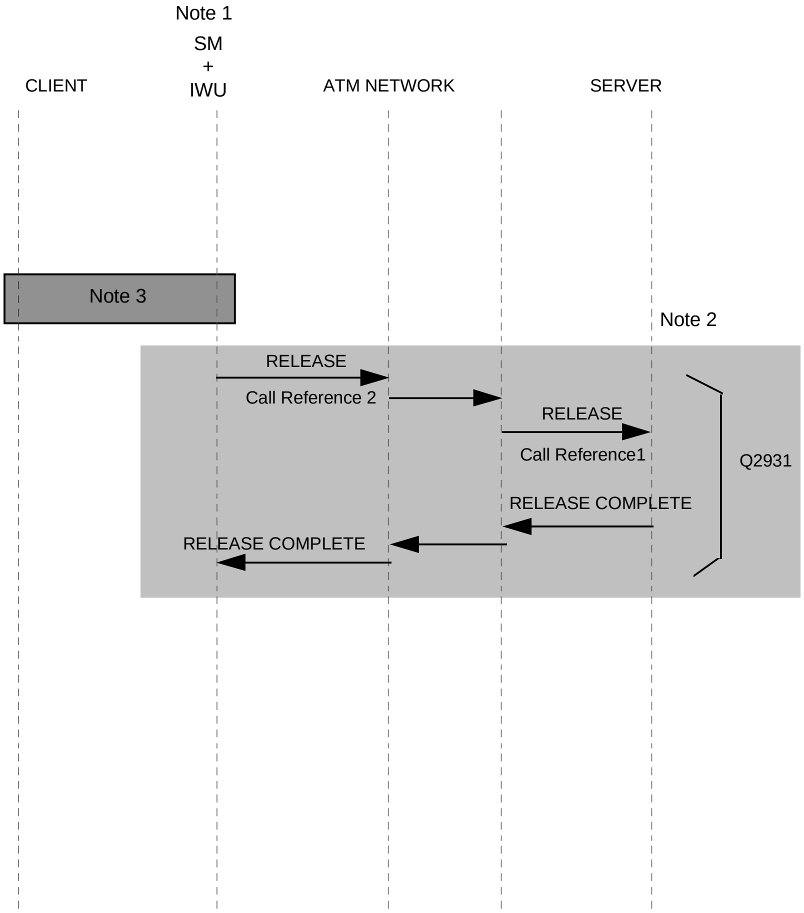


Figure F16: Network Method: Resource Deletion via Call/connection Release -Client Initiated

When the Server receives the RELEASE message on its UNI, it keys off the Call Reference to retrieve the sessionId.

When the SM receives the RELEASE COMPLETE message on its UNI, it keys off the Call Reference to retrieve the sessionId.

The SM then closes the usage record of the resource associated with this Call Reference.

13.3.2.4 Session Tear-Down

A session can only be torn down through the DSM-CC Session Tear-Down scenario, initiated either by the Client, Server or SM. Even if all the ATM SVC connections requested through the Network method have been cleared, the session is still up if the Session Tear-Down scenario has not been invoked.

With the Session Tear-Down scenario, the SM will delete all the resources associated with the session.

13.3.2.4.1 *Session Tear-Down by Server*

The ATM SVC connections that have been established through the Network method can be cleared through the Session method. See Section 3.1.4.1.

13.3.2.4.2 *Session Tear-Down by Client*

The ATM SVC connections that have been established through the Network method can be cleared through the Session method. See Section 3.1.4.1.

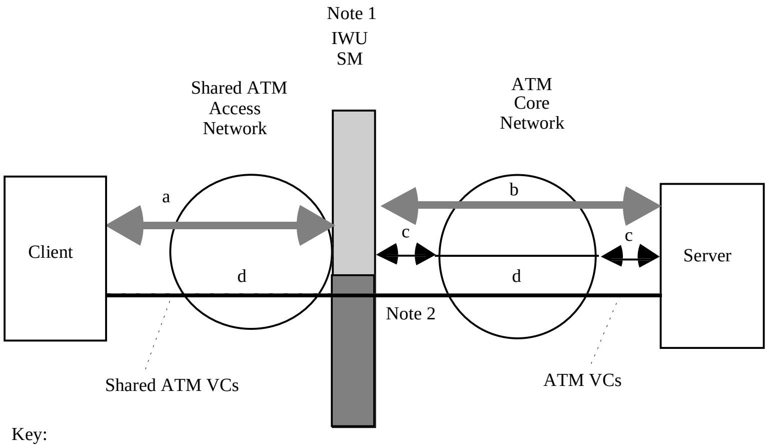
13.4 Hybrid ATM Core-Shared ATM Access

This DSM-CC configuration over ATM, uses ATM on the shared access network and ATM on the core network, see Figure F16. As in Section 3, an interworking unit (IWU) operates between the core and the access network. In addition to interworking, the unit contains functions of end-to-end session management control, access network connection management control and configuration management control. The entire unit is designated as IWU SM (Session Connection, and Configuration Management Controller).

- Messages between the Client and SM consist of both session and connection control messages (a). These include DSM-CC User-to-Network messages shown in Figure F1.
- Messages between SM and the Server (b) are DSM-CC User-to-Network messages consisting mainly of session control messages.
- The connection control messages (c) are interchanged between the Server and the ATM core network and the SM and the ATM core network using Q.2931. Some of the information in these messages are carried end-to-end between the Server and SM. At this time a minimum limit of 34 bytes has been identified but no such information elements have been standardized in Q.2931.

The User-to-User connections (d) between the Client and the Server are established across both the shared ATM access network and the core ATM network. The establishment of these connections within the context of a session occurs through the a, b and c message exchanges. Figure F17 illustrates the same session on both sides of IWU. Even though both sides are ATM Networks, because of the different assignment of network resources a mapping of VCs from one to the other is required for interworking. A session is identified with a global *sessionId*.

The message flows for this scenario are the same as the Session Method and Network Method scenarios in Sections 3.1 and 3.2 for hybrid ATM Core-Shared Access MPEG Transport Stream.



Key:

a: DSM-CC User-to-Network messages (handles Session and Connection resources)
b: DSM-CC User-to-Network messages (handles Session resources only)
c: B-ISDN Signaling
d: User-to-User messages

Note 1: An optional role of the IWU is to also manage the access network resources.
Note 2: The flow of content data is out of scope and is not shown

Figure F17: DSM-CC over ATM/Shared Media Non ATM

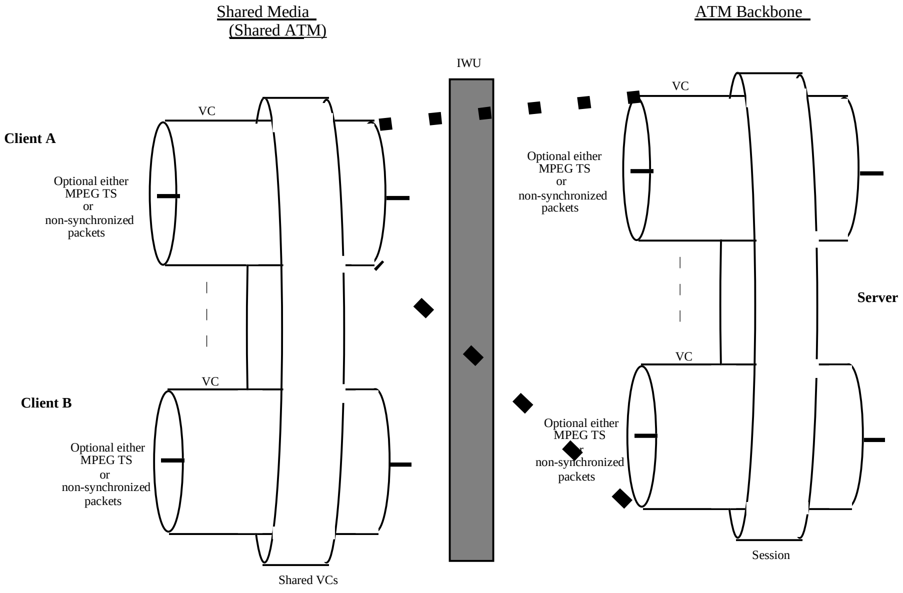


Figure F18: Session Network Resource Interworking between Shared Media and ATM Backbone

13.5 End-to-end ATM segregated with Proxy

This DSM-CC configuration is shown in Figure F19. In this configuration, Q.2931 signaling termination for the User (Client or Server side) could use a proxy signaling agent. This would allow for simplification at the Client or the Server by it not having to implement Q.2931. In this contribution, the Client is shown to have a Proxy Signaling Agent (PSA). In Figure F19, the PSA is shown co-located with the Session Configuration Management Controller (SCMC). The particular functional grouping has no specific implications and no conclusion should be drawn based on this Figure.

The Session Configuration Management Controller (SCMC) shown in Figure F19 does not indicate whether the SCMC is centralized or distributed.

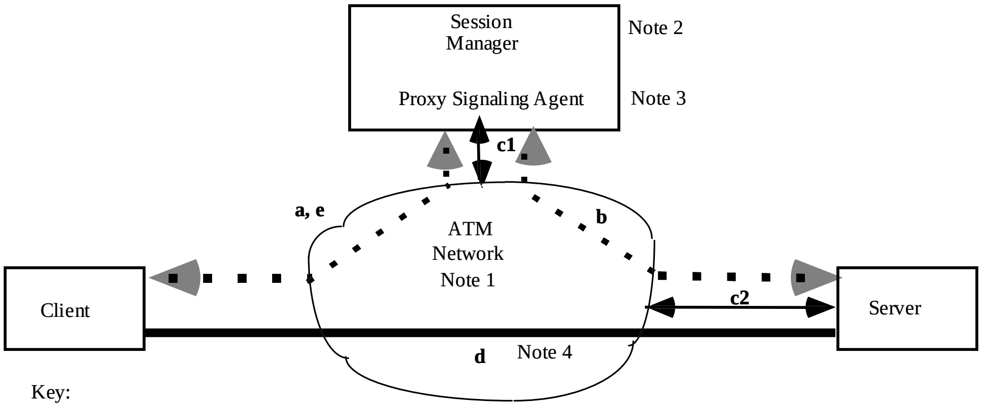
- Both the Client and the Server exchange session control messages (a,b) with the SCMC.
- Connection control messages (c1) are exchanged between the PSA and the ATM network
- Connection control messages (c2) are exchanged between the Server and the ATM network using Q.2931.
- PSA messages (e) are exchanged between the SCMC and the Client.

Some of the information are needed to be carried on Q.2931 end-to-end between the PSA and the Server. At this time no such information elements have been standardized on Q.2931.

Messages (a,e,b) can be categorized as being on the control or the user plane depending on the network perspective taken. If the perspective of the ATM network is taken, messages (a,e,b) will be on the user plane, however if the perspective of the DSM-CC network is taken, then messages (a,e,b) will be on the control plane.

MPEG TSs are carried over end-to-end ATM Switched Virtual Channels. Each MPEG TS may carry a combination of video, audio and synchronized data. Any parallel non-synchronized data may be carried over separate virtual channels, see Figure F20.

Figure F20 shows session resources on the ATM network. These VCs are identified by virtual path and virtual connection identifiers(VPI/VCIs). As stated above each VC may contain either an MPEG TS with the appropriate PIDs or non-synchronized data.



Key:

a,b: Session Management Control Messages (over PVC or pre-established SVC)
c1, c2: Connection Signaling, Q.2931
d: User-to-User Primitives
e: Proxy Signaling Agent messages

Note 1: The ATM Network consists of ATM core, access and premise networks
Note 2: Session Control can be implemented on an external platform and may be centralized or distributed.
Note 3: In this figure, the Proxy Signaling Agent (PSA) is co-located with the SCMC. This is just ONE possibility for the PSA's location in the Network
Note 4: The flow of content data is out of scope and is not shown

Figure F19: DSM-CC over End-to-End ATM Architecture with Client Proxy Signaling, but Session Control Done Outside Q.2931

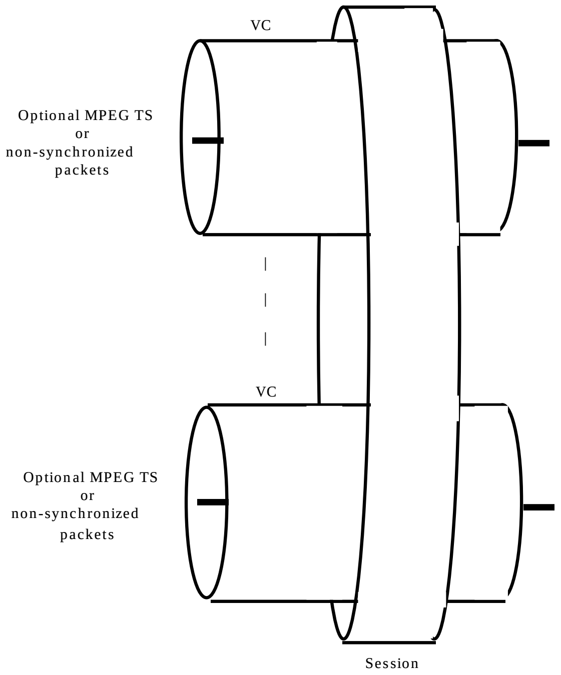


Figure F20: Session Network Resources in End-to-End ATM Architecture

To clarify DSM-CC over end-to-end ATM, with Client Q.2931 Proxy Signaling where Session control done outside Q.2931, a Session Set-up scenario will be explored in detail.

13.5.1 **Session Method Scenario**

13.5.1.1 **Session Set-Up**

Before a Client can request an ATM switched virtual channel connection to the Server, it must have a multimedia session. This session is either established by a request from the Client (Client Session Set-Up scenario) or from the Server itself (Server Session Set-Up scenario).

13.5.1.1.1 Client Session Set-Up, Client ATM Connection Set-Up

A Session is initiated by a Client sending to the SM a ClientSessionSetUpRequest message. This will initiate the Session and a sessionId will be generated by the Client (or the SM if the Network assigns it). All subsequent messages will contain the sessionId.

A session is identified with a global *sessionId*. A resource is identified by the globally unique identifier consisting of *Correlation ID*.

The Server then decides whether or not the Client or the Server is to initiate Q.2931 signaling. If the Server is to be the initiator, it just goes ahead and initiates Q.2931 signaling because it knows the initial set of connections needed for the Session. If the Client is to be the initiator, the Server sends a ServerAddResourceRequest message telling the Client what resources it should set up. The Client then initiates Q.2931 signaling through the Proxy Signaling Agent (PSA), for the resources indicated in the ServerAddResourceRequest.

The Q.2931 SETUP messages include the *Correlation ID* to allow the Server to associate the resources with the incoming ATM connection set-ups over Q.2931.

The message flows in this scenario are the same as in Section 3.1.1.1 for hybrid ATM Network-MPEG Transport Stream Configuration. The only exception is that the IWU is replaced by the PSA and the Calling Party number is the Client address as derived from the *clientId* field as opposed to the address of IWU.

DSM-CC protocol:

The steps 1 through 4 are consistent with the steps in Section 3.3.1 for hybrid ATM Network-MPEG Transport Stream Configuration.

Step 5

Q.2931 protocol :

The PSA initiates a Call/Connection procedure by sending a Q.2931 SETUP message across its ATM User-Network Interface (UNI), with the following information elements:

- Call reference selected by Client
- Calling Party Number = ATM address of the Client.
- Called Party Number = ATM address of the Server as derived from the *serverId* field.
- ATM Adaptation Layer Parameters ATM User Cell Rate and Quality-of-Service parameter = values derived from the “ATM SVC” resource request descriptor.
- Broadband High Layer Information (BHLI)= *sessionId+resourceNum* corresponding to this “ATM SVC” resource.

When the Server is informed of the SETUP on its ATM UNI, it can access the *sessionId+resourceNum* from the BHLI information element. It associates the Call Reference and its ATM connection with this session and resourceNum. The Server maintains this association until the ATM SVC connection is released.

Both the Client and the Server maintain a Call Reference and *sessionId/resourceNum* association so that one can be retrieved from the other.

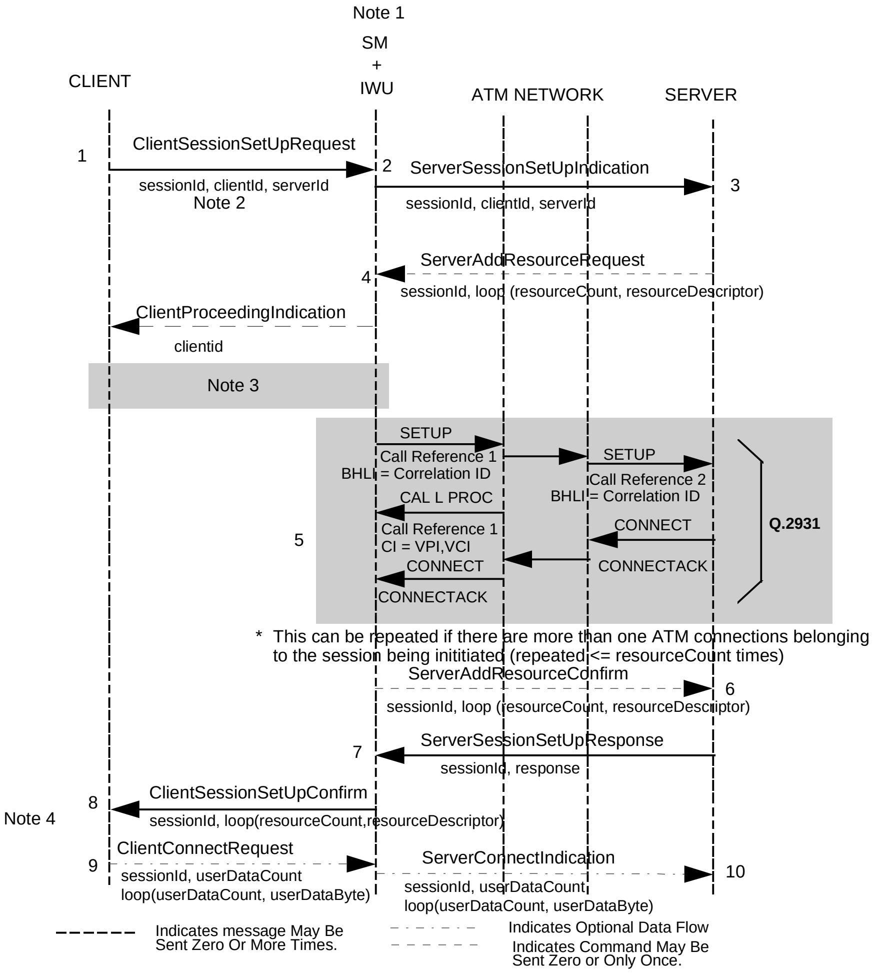

Note 1
SM + IWU
CLIENT
ATM NETWORK
SERVER
1
ClientSessionSetUpRequest
sessionId, clientId, serverId
Note 2
2
ServerSessionSetUpIndication
sessionId, clientId, serverId
3
4
ServerAddResourceRequest
sessionId, loop (resourceCount, resourceDescriptor)
ClientProceedingIndication
clientid
Note 3
5
SETUP
Call Reference 1
BHLI = Correlation ID
SETUP
Call Reference 2
BHLI = Correlation ID
CAL L PROC
Call Reference 1
CI = VPI,VCI
CONNECT
CONNECTACK
CONNECT
CONNECTACK
Q.2931
* This can be repeated if there are more than one ATM connections belonging to the session being inititiated (repeated <= resourceCount times)
ServerAddResourceConfirm
6
sessionId, loop (resourceCount, resourceDescriptor)
7
ServerSessionSetUpResponse
sessionId, response
8
ClientSessionSetUpConfirm
Note 4
sessionId, loop(resourceCount,resourceDescriptor)
9
ClientConnectRequest
sessionId, userDataCount
loop(userDataCount, userDataByte)
ServerConnectIndication
sessionId, userDataCount
loop(userDataCount, userDataByte)
10
Indicates message May Be Sent Zero Or More Times.
Indicates Optional Data Flow
Indicates Command May Be Sent Zero or Only Once.

Figure F21: End-to-end ATM segregated with Client Proxy, Client initiates Session Set-Up, Client's PSA initiates ATM connection

Note 1: The Session Manager/Proxy Signaling Agent (SM/PSA) can use proprietary Proxy Signaling Agent messages to interact with the Client on ATM Network connection control.
Note 2: Only relevant parameters in each message are shown
Note 3: Only the session control messages are shown over the MPEG access network. Proxy Signaling Agent messages between the Client and the SM can be proprietary and are not shown.
Note 4: Connections for the exchange of User-Network messages between Client and SM and Server and SM are assumed.

DSM-CC Protocol:

Steps 6 through 10 are consistent with the steps in Section 3.3.1 for hybrid ATM Network-MPEG Transport Stream Configuration.

An alternative to the above scenario is shown below where the connections are initiated by the Server. In this scenario the need for ServerAddResourceRequest and ServerAddResourceConfirm messages is obviated, but the resources are still managed by SM through Correlation ID.

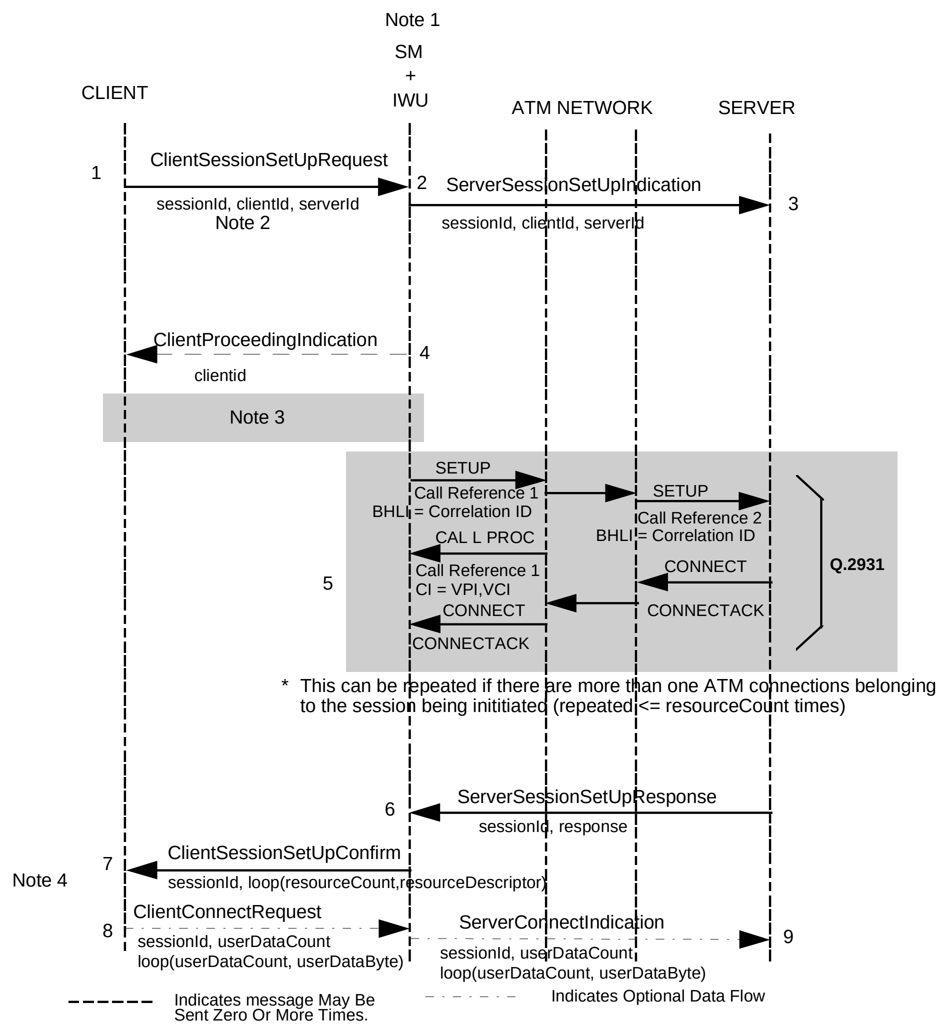

Note 1
SM
+
IWU
CLIENT
ATM NETWORK
SERVER
1
ClientSessionSetUpRequest
sessionId, clientId, serverId
Note 2
2
ServerSessionSetUpIndication
sessionId, clientId, serverId
3
ClientProceedingIndication
clientid
4
Note 3
SETUP
Call Reference 1
BHLI = Correlation ID
SETUP
Call Reference 2
BHLI = Correlation ID
CAL L PROC
Call Reference 1
CI = VPI,VCI
5
CONNECT
CONNECTACK
CONNECT
CONNECTACK
Q.2931
* This can be repeated if there are more than one ATM connections belonging to the session being inititiated (repeated <= resourceCount times)
6
ServerSessionSetUpResponse
sessionId, response
7
ClientSessionSetUpConfirm
sessionId, loop(resourceCount,resourceDescriptor)
Note 4
8
ClientConnectRequest
sessionId, userDataCount
loop(userDataCount, userDataByte)
ServerConnectIndication
sessionId, userDataCount
loop(userDataCount, userDataByte)
9
Indicates message May Be Sent Zero Or More Times.
Indicates Optional Data Flow

Figure F22: End-to-end ATM segregated with Client Proxy, Client initiates Session Set-Up, Server initiates ATM connection

Note 1: The Session Manager/Proxy Signaling Agent (SM/PSA) can use proprietary Proxy Signaling Agent messages to interact with the Client on ATM Network connection control.
Note 2: Only relevant parameters in each message are shown
Note 3: Only the session control messages are shown over the MPEG access network. Proxy Signaling Agent messages between the Client and the SM can be proprietary and are not shown.
Note 4: Connections for the exchange of User-Network messages between Client and SM and Server and SM are assumed.

DSM-CC protocol:

Steps 1 and 2 are consistent with the steps in Section 3.1.1.1 for hybrid ATM Network-MPEG Transport Stream Configuration.

Step 3

Q.2931 protocol :

The Server initiates a Call/Connection procedure by sending a Q.2931 SETUP message across its ATM User-Network Interface (UNI), with the following information elements:

- Call reference selected by Server
- Calling Party Number = ATM address of the Server.
- Called Party Number = ATM address of the Client as derived from the *clientId* field
- ATM Adaptation Layer Parameters, ATM User Cell Rate and Quality-of-Service parameter.
- Broadband High Layer Information (BHLI)= *sessionId+resourceNum*.

When the SCMC is informed of the SETUP on its ATM UNI, it can access the *sessionId+resourceNum* from the BHLI information element. It associates the Call Reference and its ATM connection with this session and resourceNum. SM maintains this association until the ATM SVC connection is released.

Both the SCMC and the Server maintain a Call Reference and *Correlation ID* association so that one can be retrieved from the other.

Step 4
SM completes the connection with the Client through Client to PSA proprietary messages not shown.

DSM-CC Protocol:

Step 5
After all the ATM SVC connections are attempted and at least the Mandatory non-negotiable connections are successfully established, the Server sends a ServerSetUpResponse message to SM.

Steps 6 through 9 are consistent with steps 7 through 10 in Section 3.1.1.1 for hybrid ATM Network-MPEG Transport Stream Configuration.

13.5.1.1.2 Server Session Set-Up, Client ATM Connection Set-Up

For Further Study

13.5.1.2 Add Resources Request

13.5.1.2.1 Add Resource Request by Server and ATM SVC connection set-up by Client

The message flows in this scenario are the same as in Section 3.1.2.1 for hybrid ATM Network-MPEG Transport Stream Configuration. The only exception is that the IWU is replaced by the PSA and the Calling Party number is the Client address as derived from the *clientId* field as opposed to the address of SM + IWU.

13.5.1.2.2 Add Resource Request by Client and ATM SVC connection set-up by Server

For Further Study

13.5.1.3 Resource Deletion

13.5.1.3.1 Resource Deletion by Server and ATM SVC connection deletion by Client

The message flows in this scenario are the same as in Section 3.1.3.1 for hybrid ATM Network-MPEG Transport Stream Configuration. The only exception is that the IWU is replaced by the PSA and the

Calling Party number is the Client address as derived from the *clientId* field as opposed to the address of IWU.

13.5.1.3.2 Resource Deletion by Client and ATM SVC connection deletion by Server

For Further Study

13.5.1.4 Session Tear-Down

13.5.1.4.1 Session Tear-Down by Server ATM SVC connections deletion by Server

The message flows in this scenario are symmetrical to the flows in the Session Tear-Down by Client shown in 5.1.4.2 below.

13.5.1.4.2 Session Tear-Down by Client ATM SVC connections deletion by SCMC

The message flows in this scenario are the same as in Section 3.1.4.2 for hybrid ATM Network-MPEG Transport Stream Configuration. The only exception is that the IWU is replaced by the PSA, the Calling Party number is the Client address as derived from the *clientId* field as opposed to the address of IWU and the connections are cleared by the Server and not DSM-CC.

13.5.2 Network Method Scenarios

13.5.2.1 Session Set-Up

The message flows in this scenario are the same as in Section 3.2.1 for hybrid ATM Network-MPEG Transport Stream Configuration

13.5.2.2 Resource Add Request

The message flows in these scenarios are the same as in Section 3.2.2 for hybrid ATM Network-MPEG Transport Stream Configuration. The only exception is that the IWU is replaced by the PSA and the Called Party number is the Client address as derived from the *clientId* field as opposed to the address of IWU.

13.5.2.3 Connection Clearing

The message flows in these scenarios are the same as in Section 3.2.3 for hybrid ATM Network-MPEG Transport Stream Configuration. The only exception is that the IWU is replaced by the PSA and the Called Party number is the Client address as derived from the *clientId* field as opposed to the address of IWU.

13.5.2.4 **Session Tear-Down**

The message flows in these scenarios are the same as in Section 3.2.4 for hybrid ATM Network-MPEG Transport Stream Configuration

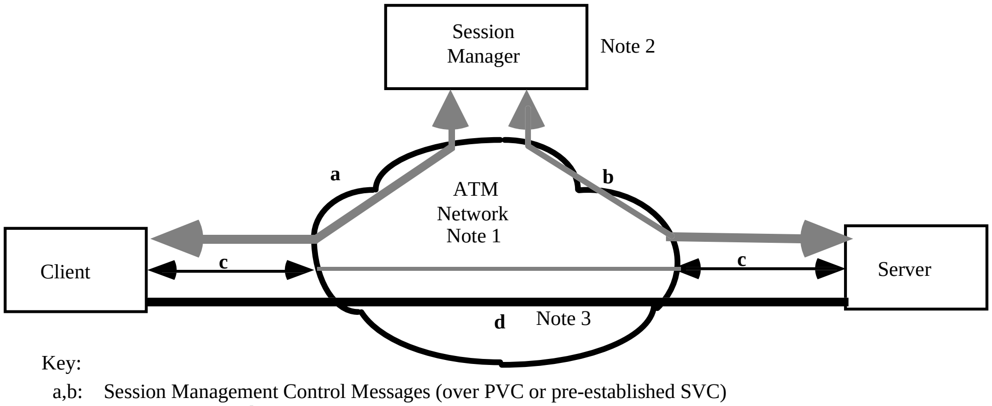
13.6 **End-to-End ATM Segregated Direct**

This DSM-CC configuration is shown in Figure F23. MPEG TSs are carried over end-to-end ATM Switched Virtual Channels. Each MPEG TS may carry a combination of video, audio and synchronized data. Any parallel non-synchronized data may be carried over separate virtual channels, see Figure F24.

- In this configuration, the Session Configuration Management Controller (SCMC) is external to the ATM network, see Figure F23. The Figure does not indicate whether the SCMC is centralized or distributed.
- Both the Client and the Server exchange session control messages (a,b) with the SCMC.
- The connection control messages (c) are exchanged between the Client and the ATM network and between the Server and the ATM network using Q.2931. Some of the information required to be carried on Q.2931 may relate to prior activities on the SCMC and could be end-to-end between the Client and the Server. At this time no such information elements have been standardized on Q.2931.

Messages (a,b) can be categorized as being on the control or the user plane depending on the network perspective taken. If the perspective of the ATM network is taken, messages (a,b) will be on the user plane, however if the perspective of the DSM-CC network is taken, then messages (a,b) will be on the control plane.

Figure F24 is the same as Figure F20 for End-to-end ATM segregated with Proxy. It shows session resources on the ATM network. These VCs are identified by virtual connections and virtual path identifiers(VC/VPIs). As stated above each VC may contain either an MPEG TS with the appropriate PIDs or non-synchronized data.



Key:
a,b: Session Management Control Messages (over PVC or pre-established SVC)
c: Connection Signaling
d: User-to-User Primitives

Note 1: ATM Network includes ATM core, access and premise networks
Note 2: Session Control is implemented on an external platform on a standalone basis. It can be centralized or distributed. Also, Configuration can be covered here, or within the network.
Note 3: The flow of content data is out of scope and is not shown

Figure F23: DSM-CC over End-to-End ATM Architecture

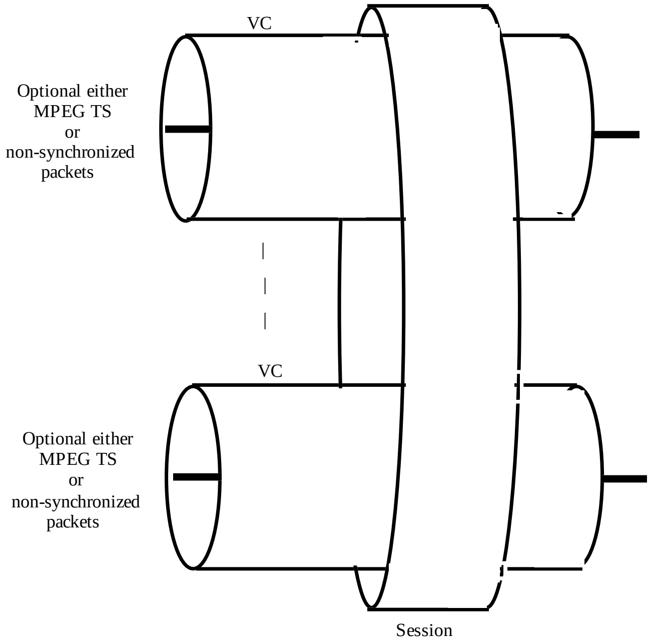
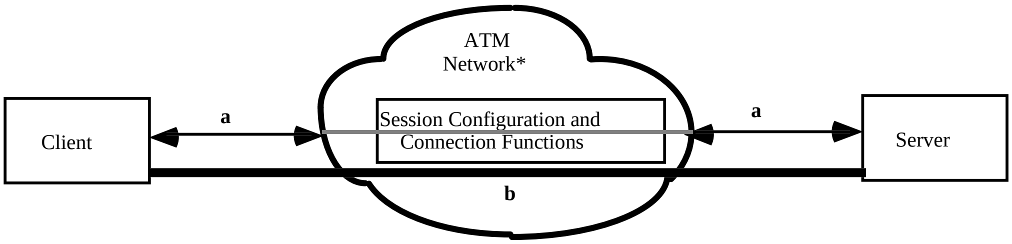


Figure F24: Session Network Resources in End-to-End ATM Architecture

The message flows for this scenario may be beyond the scope of DSM-CC and are For Further Study

13.7 End-to-End ATM Integrated

In this architecture the DSM-CC Network is no longer distinguished from the ATM Network. The session control function is integrated with the ATM Network, but its function still remains the same as with End-to-end ATM with separated session control, see Sections 4 and 5. Figure F25 shows end-to-end ATM with session configuration and connection functions integrated into the ATM Network.



Key:
a: Session Configuration and Connection Functions and Signaling: Q.2931 extended to carry DSM-CC session messages - The assumption is made that at least ONE VC (b) will be set up as part of a session, and the client knows the QoS for it.
b: User-to-User Primitives

*Note: ATM Network includes ATM core, access and premise networks

Figure F25: End-to-End ATM with integrated Signaling

The message flows for this scenario may be beyond the scope of DSM-CC and are For Further Study

14. INFORMATIVE ANNEX E

(This annex does not form an integral part of this International Standard)

Possible implementation and use of client_configuration_message Manufacturer_id

Upon receipt of the userCapabilities information element the server uses the **Manufacturer_OUI_code** and the **Manufacturer_id** field as the unique identifier for the client device. The server uses this unique identifier to access its own database of known capabilities for the declared device. Typically the server will know all the well known aspects of the device such as CPU, OS etc.. The optional device descriptors contained in the message would be used to convey options. The options may redefine the base assumptions or declare new information. One implementation would be to leverage the syntax and parsing mechanism used in UNIX's terminfo database.

In this implementation the following rules would apply to the capabilities file located on the server:

1. All lines in the entry end with a comma (hex 0x2C)
2. Each entry starts with a header line
3. The header line starts in column one
4. The header line must contain at least two entries separated by the vertical bar character (hex 0x7C). At the very least the header line contains an alias and the long description of the entry. Usually an additional abbreviated version of the concatenated **Manufacturer_OUI_code** and **Manufacturer_id** fields is also given.
5. Each capability line is indented by 2 or more white space characters (space 0x20 or tab 0x09)
6. Alias names must be unique throughout the entire database of devices.
7. Alias names must conform to the naming convention of files on the server.

Each entry in a DSM-CCinfo file starts with a header line that contains at least two entries separated by the '|' . The first entry is created by concatenating the Manufacturer_id to the end of the Manufacturer_OUI_. The second alias may be an more readable version of the first. In either case both aliases must be unique with respect to all known devices in the system. The last entry on the header line is the long name of the device. Finally the line is terminated by a comma.

Following the header line are the capability entries. Each entry is separated from the next by a comma and the line must always end with a comma. Below is an example of a DSM-CCinfo file

```
C5A0F1100|apx100|apex model 100,
    SNUM="1234567", 13818-2, 13818-3, TS, AUDLYRS="12",
    CPU="MCD68331", OS="OS9", OSVER="1.20", NVRAM="1024000", RAM="2048000",
    NTSC, GOVP="MCD210", GHRES="360", GVRES="240", PLANES="2",
    CLRSP="RGB", CLRBITS=8,

C5A0F1200|apx200|apex model 200
        CLIENT="C5A0F1-100", RAM="4096000",

C5A0F1option1|apxopt1|apex option package one
        RAM="4096000", PCMCIA="2", MCR, OSVER="1.40", RS232,

C5A0F1200a|apx200a| apex 200 with option 1 installed
```

CLIENT=“C5A0F1-200”, CLIENT=“C5A0F1-option1”

In the above example C5A0F1100 is defined very completely and serves as the base definition. The second entry defines the next model which is the same as the previous except with more memory. The third definition describes the elements of an option package. The final entry shows how the CLIENT descriptor can be used to create different variations on base descriptions.

Each information provider is its own registration authority. It is expected that each manufacturer will make available to information providers all the necessary information to construct their “DSMCCinfo database.” Ideally manufacturers would create and test definition files like the one provided in this annex. Each information provider may then wish to edit this information before placing it on their system. The purpose of this editing would be to ensure consistency in declarations, for example; the EC040, LC040 68331 and 68339 are all members of Motorola’s 68000 processor family and share a common instruction set. The information provider may wish to edit the CPU entry of all files that use any one of the aforementioned processors to simply read “68000”. This would simplify the parsing and branching software needed in the server.

15. INFORMATIVE ANNEX F

(This annex dos not form an integral part of this International Standard)

DSM-CC User to User Asynchronous RPC

15.1 Purpose

This informative annex describes the use of standard RPC to create non-blocking RPCs. It also addresses some general issues in using standard RPC in non-standard fashions.

15.2 Description

RPC provides the capability of creating distributed software systems using a programming paradigm that is familiar to programmers who are experienced in non-distributed applications programming. It supports a client-server model, where a client calls a function, as if it were in the local address space on the local machine, and the RPC software transfers the arguments to the server host, invokes the server function and returns the results to the client. To the client this appears to be a local function call (albeit a slow one).
The basic RPC model is synchronous: the client blocks until the server receives the request, process the command, and returns the results. For many systems network delays and server processing time make it impractical for the client to synchronize with the RPC server. These are the basic alternatives for creating asynchronous RPC from synchronous RPC:

- use non-blocking RPC with client call-backs
- use a multi-threaded client with blocking RPC

15.3 3. Non-Blocking RPC with Client Callbacks

Non-blocking RPC allows the client to continue execution immediately after invoking the RPC, but before the server completes the operation. This requires an RPC signature with no return values from the RPC Server Function. The client local call may return after the RPC Argument are copied to a local transport buffer or after the RPC arguments are successfully transfered to the server. The first case is often associated with local batching of non-blocking RPCs and decreases network traffic in some cases. The other case allows the client application the opportunitz to handle network errors (timeouts) when the RPC uses a reliable transport mechanism.

Data are returned to the client by an RPC invoked the RPC Server on the client. This transfers the return data to the client which is notified when the client RPC function executes. The RPC connection information needed for the client call back must be sent to the RPC Server with the function arguments.

This approach requires that the client side define its own set of call-back RPCs (for all functions returning a value) and that it either call an RPC server main loop, or recreate the RPC server functionality. The second approach is more difficult to implement, but provides greater control over the call-back execution.

In cases where the server cannot process RPCs faster than they arrive, the non-blocking RPCs may block the client. This occurs when a reliable transport is used and the server's buffers fill. New RPC message buffers will be rejected by the RPC server and the client application will block until the RPC fails with a time out or server buffers become available.

15.4 Multi-Threaded Clients

If the client operating system supports multi-threaded applications, then asznchronous RPC can be implemented using blocking RPCs in separate threads, processes or tasks. This technique relies on operating szstem services to communicate the RPC status and return valuse to the original thread:

- semaphores to wait on events from the RPC server and interface task
- shared memory spave for arguments and return values

signals to indicate completion

This technique works well in applications where few client RPCs are simultaneously called. Since each RPC requires a separate thread of execution, it is somewhat wasteful of client operating system resources.

Some operating systems may limit the number of concurrent threads of execution and this should be accounted for in design of systems using this technique.

This technique works particularly well in systems where memory space is global and shared by all tasks. Most real-time operating systems operate in this fashion. It has the advantage, over client call-backs, of isolating the main client process from problems associated with RPC errors, timeouts and network congestion.

15.5 General RPC Issues

The basic RPC mechanism selected to support asynchronous RPC operation must provide these functions:

- reliable transport
- unreliable transport
- support for non-blocking RPC
- guaranteed order of delivery of RPC buffers from the client

Optionally, the RPC mechanism should support batch transport of multiple RFCs.

Multi-threading a server is commonly employed to improve overall response time (fast functions are not required to begin execution after completion of slow functions that arrived first) and to overcome the transport buffer overflow problem mentioned previously. It is the responsibility of the server to maintain the semantical ordering of consecutive commands received from a client. For example: if a server provides an MPEG stream to a client, it is the servers responsibility to guarantee that the Pause executes before the Resume if that is the order they were received.

16. INFORMATIVE ANNEX G

(This annex does not form an integral part of this International Standard)

INTEROPERABLE RPC PROTOCOL STACK

Abstract:

The annex recommends an interoperable protocol stack for the presentation, session, transport, and network levels. The solution comprises a) a framework which allows nodes to select a protocol stack, or interpose an object which translates protocol stacks b) conventions to encode the message payload at the presentation level, c) a message set for remote procedure call at the session level, and d) a pervasive protocol solution at the transport level and network level.

Motivation:

If a client and a service distribute across a network, the question of interoperation arises. A concept which frames the question is the notion of a domain. A domain is a collection of nodes, or at a fine grain objects, which share consistent expectations about a convention which affects interoperation. While multiple distinctions between domains are valid, the questions which the annex will explore are:

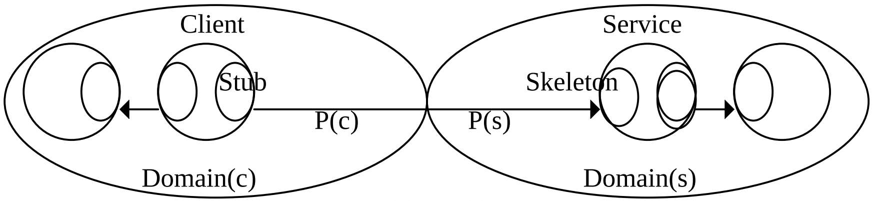
Type Domain: The objective is to encode invocation signatures which are intelligible to multiple domains. The implication is that a domain which wishes to interoperate with another domain must provide typecode space which the domain understands. A typecode is a Interface Definition Language construct which encodes an interface name into a concrete value. A domain which wishes to interoperate with another domain must either adopt the same typecode, or there must be mechanism which can translate the typecode values.

Protocol Domain: While the protocol stack can be diverse, the spectrum frustrates interoperation. For interoperationo be fesible, either a domain must share the same native protocol stack with another domian, or there must be mechanism to translate the protocol stack. The solution, in the second case, constitutes a protocol gateway.

Before the annex considers the solution, the discussion first explores the solution space. Te framework, as we shall observe later, allows nodes to detect the situation where domains share a common native protocol stack, but still converge if a more subtle solution exists.

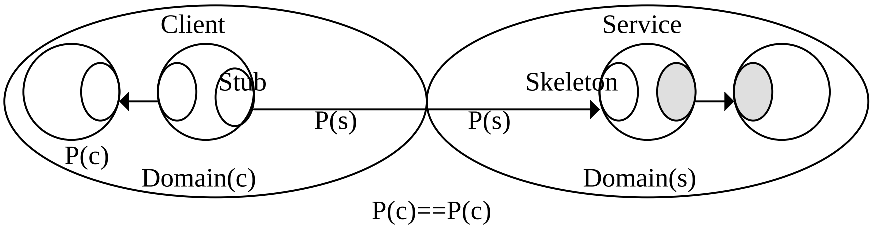
Solution Space:

The spectrum of design options is extensive. The discussion below considers the spectrum. The next section then describes a framework which, while it anticipates diverse design centers, also provides the mechanism to converge to an interoperable solution.

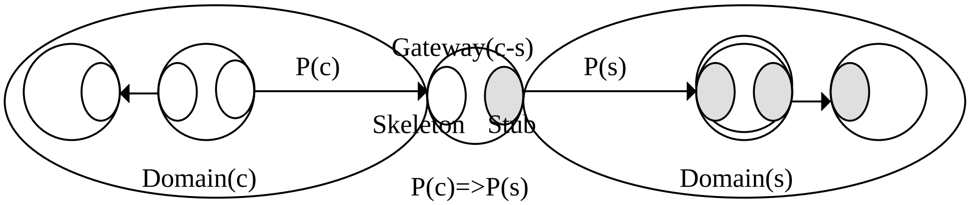


P(c)==P(s)

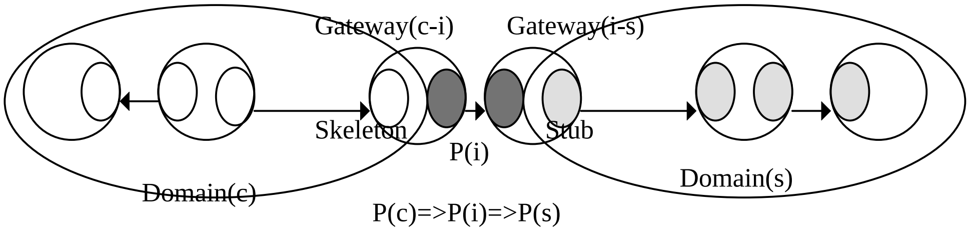
In the example above a client native protocol, P(c), matches the service native protocol, P(s). The solution should allow a node to detect that, although the node and another node reside in distinct domain, the nodes share the native protocol, and can retain the native protocol to interoperate.



In the example above the native client protocol does not match the native service protocol, but the skeleton can support the client native protocol. The solution is for the skeleton to link the client native protocol. The service, from the perspective of the client, is indistinguishable from a service in the client protocol domain. The companion solution, not shown, for the stub to link the service native protocol. The client, from the perspective of the service, is indistinguishable from a client in the service protocol domain.



In the example above, the solution is to interpose a protocol gateway, as a distinct object, between the client domain and the service domain. The protocol gateway translates the client native protocol into the service native protocol. The protocol gateway, from the perspective of the client, resides in the client domain. The protocol gateway, from the perspective of the service, resides in the service domain. The framework interposes the gateway. The technique is not visible to the neither client at the ation level nor to the service at the application level.



In the last example, the solution is to interpose two gateways. The first protocol gateway translates from the client protocol to a canonical protocol, while the second gateway translates from the canonical protocol to the service protocol. The solution accounts for the situation where a protocol gateway which directly translates from the client native protocol to the service native protocol is not available. If there is a convention with respect a protocol which a domain supports *for interdomain interoperation* it is feasible to cascade a gateway which translates into the canonical protocol with the gateway which translates from the canonical protocol to achieve interoperation. The solution does not mandate that the protocol within both domains adopt the canonical protocol.

Interoperation Framework:

The recommendation is to adopt the Universal Network Object solution of the Object Management Group. The specification resolves the basic questions of the first section. The description of the specification divides into four sections. The first section will describe the framework which allows the design options noted above, but provides the interface to converge to a solution. The second section describes conventions at the presentation level which encode data structures which result from compilation of Interface Definition Language into a concrete message payload. If the client side adopts te conventions, and a service adopts the conventions, the client operation signature and the service operation signature are consistent. The third section describes the message set of the session level. It realizes remote procedure call semantics. The last section describes the semantics at the transport level and at the service level.

Protocol Selection:

The interoperation framework allows a node which resides in some protocol domain to discover the protocol known to another domain. The interface adopts the concept of a profile interface. The profile describes the protocol decisions which the native domain must adopt to interoperate with the remote domain. The interface on which the solution builds is[1]:

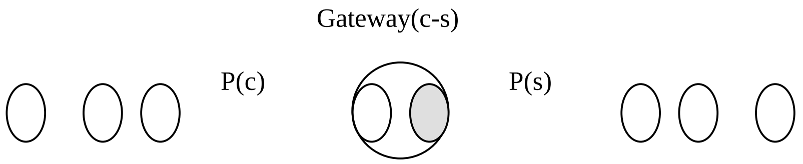
```
typedef unsigned long       ProfileId;
const ProfileId             TAG_INTERNET_IOP = 0;
struct TaggedProfile {
        ProfileId          aProfileId;
        sequence<octet>  aProfile;
};
```

The structure comprises a) the code which identifies the profile and b) an opaque value which contains profile specific data. An example of a profile is the internet protocol which is the foundation of certain protocol gateway solutions. The fields inside the sequence<octet> for the protocol are:

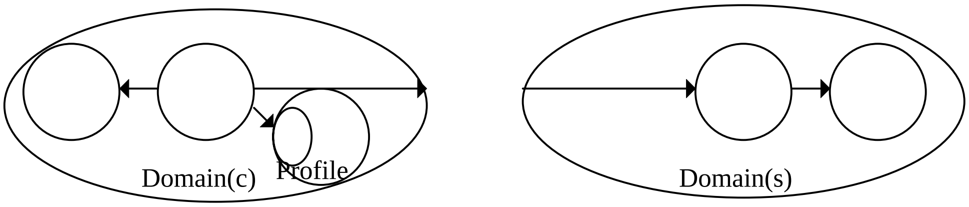
```
struct Version {
        char                major;
        char                minor;
};

struct ProfileBody {
        Version           aVersion;
        string            aAddress;
        unsigned short    aPort;
        sequence<octet>  aObjectKey;
};
```

The first field is the version as shown above. The second field, the network address, is the full internet symbolic address. The third field is the target network port. The fourth field specifies the target object behi the neywork port.



[1] In some cases a semantic filed, for example aVersion, of the annex will differ from the description of the Universal Network Object specification. The motivation for the difference is comprehension. Please refer to the Universal Network Object specification the complete interface.



The figure above illustrates how the elements of the solution integrate. The service domain installs into the client domain a profile object which articulates the protocols which the service domain supports. If the client native protocol and the service native protocol match, the objects can interoperate without a protocol gateway. If the client native protocol and the service native protocol do not match, the solution require the interposition of a protocol a gateway. The client can find the gateway through whatever is the convention for the client domain. In the case of Digital Storage Media, the client should expect to find the gateway in the service gateway. The transport code in the stub, in concept, selects the gateway. The interposition of the gateway, however, would not be visible to the client above the stub.

Common Data Representation:

The interoperation solution requires specification of how to translate an operation signature (here the type space of the Interface Definition Language) into a message payload. The conventions of the interoperation solution are known as the Common Data Representation. The annex will not describe all the conventions. The objective will be to surface the questions which the translation raises, and note that for each question there is a resolution.

The message payload can be thought of as an octet stream. The octet stream is a finite sequence of eight-bit values with a clear point at which the stream begins. The position of an octet in the stream is known as its index. The session software must understand the octet index to calculate alignment boundaries.

Encapsulation:

The interoperation solution distinguishes between two octet streams: a) a message and b) an encapsulation. A message is the basic unit of data exchange. An encapsulation is an octet stream into which data structure may be marshaled. Once a data structure has been encapsulated, the octet stream can be represented as the opaque data type sequence<octet>, which can be marshaled into a message or another encapsulation. The encapsulation allows complex constants to be pre-marshaled. Just as a message contains a field which encodes the byte order, an encapsulation contains a field which encodes the byte order.

Alignment:

The primitive data types are encoded in multiples of octets. The alignment boundary of a primitive datum is equal to the size of the datum in octets. The table below presents the alignment conventions:

TYPE	OCTET ALIGNMENT
char	1
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
float	4

double	8
boolean	1
enum	4

The alignment is relative to the beginning of the octet stream. The first octet of the stream is octet index zero. The octet stream begins at the start of the message header. In the case of encapsulation, the octet stream begins at the start of the encapsulation, even if the encapsulation is nested in another encapsulation.

Primitive Data Types:

The encoding rules of the primitive data types are intuitive and will not be shown. The figure below illustrates a few data types.

Big Endian(char)	Octet	Little Endian(char)	Octet
	0		0

Big Endian(short)	Octet	Little Endian(short)	Octet
MSB	0	LSB	0
LSB	1	MSB	1

Big Endian(long)	Octet	Little Endian(long)	Octet
MSB	0	LSB	0
	1		1
	2		2
LSB	3	MSB	3

Big Endian(float)	Octet	Little Endian(float)	Octet
S E1	0	F3	0
E2 F1	1	F2	1
F2	2	E2 F1	2
F3	3	S E1	2

Big Endian(double)	Octet	Little Endian(double)	Octet
S E1	0	F7	0
E2 F1	1	F6	1
F2	2	F5	2
F3	3	F4	3
F4	4	F3	4
F5	5	F2	5
F6	6	E2 F1	6
F7	7	S E1	7

Compound Types:

A constructed type has no alignment restrictions beyond those of its primitive components. The rules which relate to constructed type are:

Struct: The structure consists of its components encoded in the order of their declaration in the structure.

Union: The union consists of the discriminant tag of the selected type plus the representation of the corresponding member.

Array: The encoding preserves the element sequence. Since the array length is fixed, the length value is not encoded. In the case of multiple dimensions, the elements are order so that the index of the first dimension varies most slowly and the index of the last dimension varies most quickly.

Sequence: The sequence consists of an unsigned long, which encodes the sequence length, plus the elements, as encoded their type.

String: The string consists of an unsigned long, which encodes the string length, plus the individual characters. The string terminates with the null character. The length includes the null character. The character set is ISO Latin-1 (88599.1).

Enum: Each enumeration value is a unsigned long. The companion numeric values reflect the order in which the identifier appears in the declaration. The first enum identifier has the numeric value zero. The successive enum identifiers ascend in value, in order of declaration from left to right.

TypeCode:

The rules to encode typecode values build on the assignments shown below:

TCKIND	VALUE	TYPE	PARAMETER
tk_null	0	empty	none
tk_void	1	empty	none
tk_short	2	empty	none
tk_long	3	empty	none
tk_ushort	4	empty	none
tk_ulong	5	empty	none
tk_float	6	empty	none
tk_double	7	empty	none
tk_boolean	8	empty	none
tk_char	9	empty	none
tk_octet	10	empty	none
tk_any	11	empty	none
tk_TypeCode	12	empty	none
tk_Principal	13	empty	none
tk_objref	14	simple	string
tk_struct	15	complex	string(name) ulong(count) {string(memberName), TypeCode(memberType)
tk_union	16	complex	string(name) TypeCode(discrimant) long(default), ulong(count) {discriminant(labelValue) string(memberName) TypeCode(memberType)}
tk_enum	17	complex	string(name) ulong(count) {string(memberName)}
tk_string	18	simple	ulong(maxLength)
tk_sequence	19	complex	TypeCode(elementType) ulong(bounds)
tk_array	20	complex	TypeCode(elementType) ulong(count) {ulong(dimensionSize)}
-none-	Oxffffffff	simple	long(indirection)

A complete description of the TypeCode conventions is beyond the scope of the annex. Please refer to the Universal Network Object specification of the Object Management Group.

Session Interface:

The interoperation solution requires clear interface at the session level. The premise is that the session level realizes a remote procedure call. The discussion of this section will focus on the message set which a client and a service exchange. The next section describes the semantics at the transport level and the network level.

The remote procedure message set shares a common preamble with the fields shown below. (While the syntax below is identical to the standard, certain semantic names (for example aVersion below) differ to aid comprehension.)

```
struct Version {
        char            major;
        char            minor;
};

struct MessageHeader {
        char            aProtocolName[4];
        Version         aVersion;
        boolean         aEndeanBit;
        octet           aMessageCode;
        unsigned long   aMessageSize;
};
```

Note that each message describes the byte order. If the source native byte order does not match the target native byte order, the receive side is responsible for the translation.

The table below presents the message set, with the roles of the client and the service.

Message	Source of Message	Value
Request	Client	0
Reply	Service	1
CancelRequest	Client	2
LocateReply	Service	3
CloseConnection	Service	4
MessageError	Client+Service	5

A brief description of each message follows. For a complete description, refer to the Universal Network Object specification of the Object Management Group.

Request Message:

The request message includes three elements: a) the standard message header, b) the request header, and c) the message body. The request header is:

```
struct RequestHeader {
        ServiceContextList      aServiceContextList;
        unsigned long           aRequestId;
        boolean                 aExpectRequest;
        sequence<octet>         aObjectKey;
```

```
            string                  aOperation;
            Principal               aRequestPrincipal;
    };
```

The interpretation of each field is:

a) ServiceContextList: The signature of an operation allows the inclusion of a context to associate with the operation. The service context list accounts for the option. The declaration is:

```
    struct ServiceContext {
            ServiceID               aContextId;
            sequence<octet>         aContextValue;
    };
    typedef sequence<ServiceContext> ServiceContextList;
```

b) RequestId: The field allows the remote procedure software on the client side to scoreboard results.

c) ExpectRequest: There is a construct in Interface Definition Language which allows the interface to declare if an operation does not return results. The field encodes this state.

d) ObjectKey: The field identifies the target of the invocation. The description of the Internet InterOperation Protocol describes the data found in the sequence<octet> field.

e) Operation: The field specifies the operation name.

f) Principle: The field relates to the Principle concept. A principal is a trusted object on which authentication techniques build.

Reply:

The Reply message contains three elements: a) a standard message header, b) a ReplyHeader, and c) a message body. The schema for the reply header is:

```
    enum ReplyStatusType {
            NO_EXCEPTION,
            USER_EXCEPTION,
            SYSTEM_EXCEPTION,
            LOCATION_FORWARD
    };

    struct ReplyHeadrer {
            ServiceContextList      aServiceContextList;
            unigned long            aRequestId;
            ReplyStatusType         aReplyStatus;
    };
```

The interpretation of each field is:

ServiceContextList: The is the same field as found in the Request message.

RequestId: The client remote procedure software can exploit the field to scoreboard results.

ReplyStatusType: The service can indicate that the target object which realizes the operation does not now reside at the service object location. The message body then provides the object reference, cast in the interoperable signature, at which the target object resides. The infrastructure on the client side is responsible for forwarding the original request to the that target object. The feature anticipates object migration, which empowers techniques such as reaction to failures or load balance.

CancelRequest:

The CancelRequest message contains two elements: a) a standard message header and b) a CancelRequest header. The schema for the cancel request header is:

```
struct CancelRequestHeadrer {
        unsigned long     aRequestId;
};
```

The RequestId specifies the invocation to which the message applies. Because the service could be unable to reverse the operation, the service is not required to realize the request. The client could receive a standard reply to the operation.

LocateRequest:

The message allows the client to establish a) whether the object reference is valid, b) whether the current target of the operation can realize the request through the object reference, and c) to what address a request for the object reference should be sent. The message complements the status found in the Reply message. The client can discover whether the target of the operation realizes the operation before it invokes a request. The message contains two elements: a) the standard message header and b) the LocateRequest header. The schema for the locate request header is:

```
struct LocateRequestHeader {
        unsigned long     aRequestId;
        sequence<octet>   aObjectKey;
}:
```

LocateReply:

The LocateReply message is the reply to the LocateRequest message. The message contains three elements: a) a standard message header, b) a LocateReply header, and c) a LocateReply body. The schema for the locate reply header is:

```
enum LocateStatusType {
        UNKNOWN_OBJECT,
        OBJECT_HERE,
        OBJECT_FORWARD
};

struct LocateReplyHeadrer {
        unsigned long           aRequestId;
        LocateStatusType        aLocateStatus;
};
```

CloseConnection:

The message allows a service to alert a client that the service intends to close the connection. The client should not expect further responses. The message contains just the standard message header.

MessageError:

The conditions which can cause the message include a) an invalid message header b) an invalid version number or c) an invalid message type.

Session Semantics:

The session level, in combination with the transport level and the network level, implements a remote procedure call. The semantics are roughly those of a subroutine invocation, where the software which calls the subroutine would expect the subroutine to a) execute the invocation (rather than fail with no notification) b) execute the invocation just once, and c) preserve the order of successive invocations. The next section explores the semantics further

Transport and Network Semantics:

The objective of the solution is to be implementable on a wide range of transport protocols. The interoperation protocol, however, does expect certain semantics at the transport level and at the network level:

Connection Establishment: The transport is to be connectionful. The connection bounds the scope of RequestId.

Byte Stream: The transport is thought to be a byte stream. There are no restrictions on message size, fragmentation, and alignment.

Reliable Transport: The transport is to be reliable. The transport is to guarantee that the target of the message receive the byte stream in the order in which it was sent, at most once, and that the source of the message receive an positive acknowledgment.

Connection Failure: The transport is to provide some reasonable notification of connection failure. The object which establishes the connection should receive notification

Connection Establish: The connection establish phase is to translate to the connection abstraction of Transport Control Protocol (Version tbd) and Internet Protocol (Version 4.0 to Version 6.0).

17. INFORMATIVE ANNEX H

(This annex does not form an integral part of this International Standard)

Implementation model for IP over ATM with DSM-CC

17.1 Purpose

The purpose of this annex is to clarify how IP family protocols might be implemented over DSM-CC sessions, making explicit the relationship of IP to DSM-CC, and to make sure issues relevant to implementing IP are surfaced. This contribution is proposed as an informative annex to the DSM-CC specification.

17.2 Introduction

First, a caveat: the following descriptions are stripped to bare essentials for expository purposes; many details are omitted , but hopefully nothing fundamental. Also, this is by no means the only way to implement IP, nor does it pretend to be the optimal way, but hopefully it exposes all the critical issues.

To set some context, we will first outline how UDP/IP is often implemented over a connectionless data link layer such as Ethernet. The figure on the next page shows a simplified block diagram of such an implementation. Those familiar with sockets and IP can skip to section 3, which extends the model to IP over ATM; section 4 further extends the model to IP over ATM with DSM-CC.

An important requirement of the extensions is that existing applications are completely unaffected by them, so that they run unchanged over ATM and ATM with DSM-CC.

17.3 IP over Ethernet

The diagram below shows an application, the sockets layer, the UDP layer, the IP layer, the routing function, the address resolution protocol (ARP) layer, and the network interface card (NIC) device driver, which implements the data link layer.

The application talks to the UDP layer via some API -- a popular one is the *sockets* interface. The sockets interface supports many protocol families, such as IP, XNS, and IPX; we will omit details of this and focus on just IP. The sockets interface also supports many protocols within a family, such as TCP/IP and UDP/IP; we will omit details of this and focus on just UDP/IP. The crucial point for our purposes is that the sockets interface identifies the source and target systems via *sockaddrs* which consist of a protocol family ID and an opaque address string specific to that protocol family.

The UDP layer can send a packet over the network to a *port* on another *host*. The host is identified by its *IP address*, a 32 bit number unique in the network. The port is a 16 bit number, and identifies one of many possible recipients within the host. UDP can also receive packets addressed to the host it is running on and a specified port within that host.

The IP layer can send a packet to a host identified by its IP address. It can break packets too large to be sent in one data link layer packet into fragments, and reassemble them upon reception.

The ARP layer can translate an IP address into an address suitable for use by the NIC hardware. In the ISO OSI model, the IP address is a network layer address; the NIC address is a MAC address. The IP layer treats the MAC address obtained from ARP as an opaque byte string.

The routing function can translate an IP address into the IP address of the first (or only) hop of a potentially multi-hop route from the sending host to the receiving host. In the case that the target host is on the same *subnet* as the sending host (i.e., it can be reached in one hop), the output IP address will be the same input IP address, otherwise it will be the IP address of a *router* which can forward the packet to the target host or another router closer to the target host.

The NIC driver can send a packet to a destination host given its data link layer MAC address, and receive packets addressed to its MAC address. (We omit details of hosts with more than one NIC.)

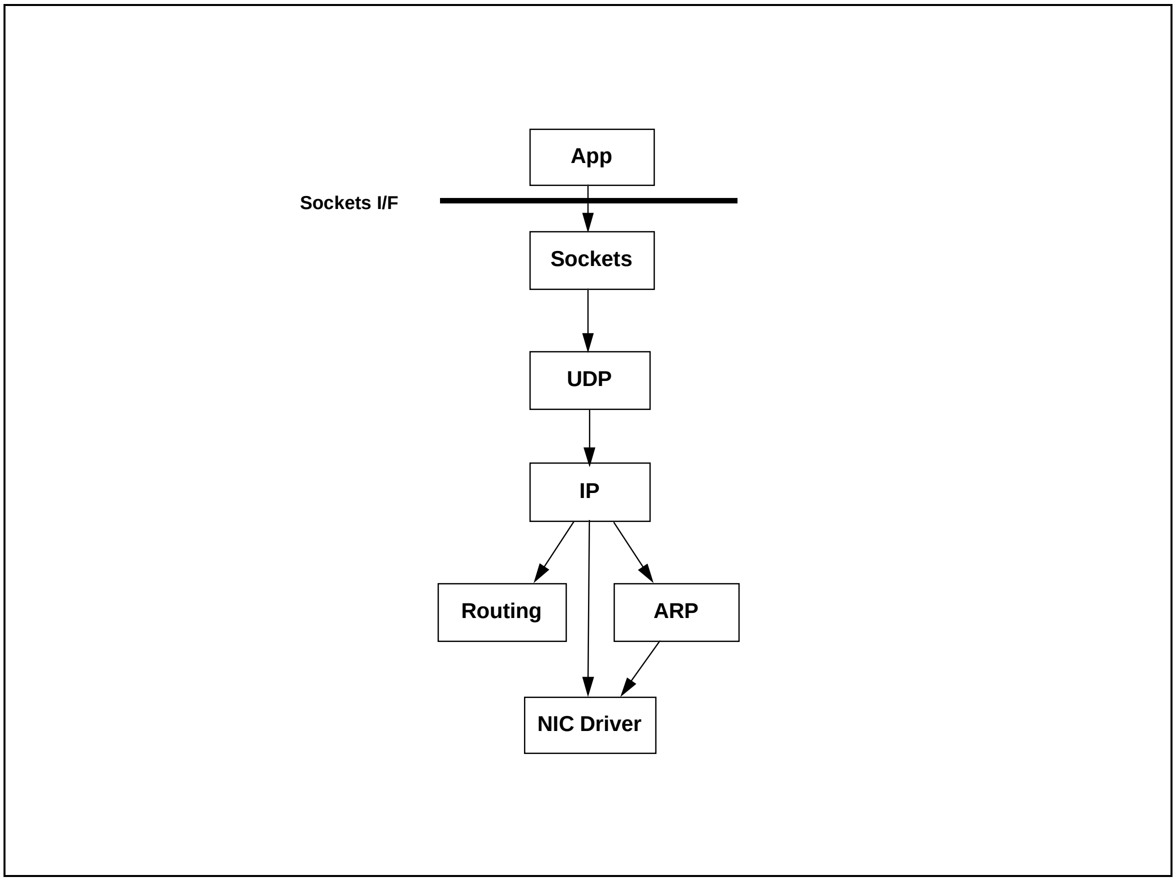


Figure 17-33. IP Model - Ethernet

17.3.1 Application interface to UDP

The application uses the *sockets* interface to send and receive UDP packets. The main APIs for this purpose are **sendto** and **recvfrom.**

To send packets, the **sendto** API is used:

int sendto(SOCKET s, char * buf, int len, sockaddr * to,
    int tolen);

s	A descriptor identifying a socket.
buf	A buffer containing the data to be transmitted.
len	The length of the data in buf.
to	A pointer to the address of the target socket.
tolen	The size of the address in to.
Returns	the number of bytes sent.

To receive packets, the **recvfrom** API is used:

int recvfrom (SOCKET s, char * buf, int len, sockaddr * from,
    int * fromlen);

s	A descriptor identifying a socket.
buf	A buffer for the incoming data.
len	The length of buf.
from	A pointer to a buffer which will hold the source address upon return.
fromlen	A pointer to the size of the from buffer.
Returns	the number of bytes in the received packet.

The “sockaddr” structure contains, for the IP protocol family, the IP address and port number of the source or destination of the packet received or to be sent (respectively). In the calls above, the length of the address is variable in order to support multiple protocol families.

17.3.2 Internal interfaces

The interfaces to UDP, IP, ARP, routing and the NIC are not public. What is presented here are not actual interfaces of any implementation, but *model* ones that capture the fundamental capabilities of the main APIs of the layers beneath sockets for IP implementations.

17.3.2.1 UDP

The model UDP layer interface is almost the same as the sockets interface, except that it knows that the addresses are IP addresses.

int UDPsendto (SOCKET s, char * buf, int len, IPaddr to, short port);

s	A descriptor identifying a socket.
buf	A buffer containing the data to be transmitted.
len	The length of the data in buf.
to	IP address of the target host.
port	Port number to send to in the target host.
Returns	the number of bytes sent.

int UDPrecvfrom (SOCKET s, char * buf, int len, IPaddr * from,
short * port);

s	A descriptor identifying a socket.
buf	A buffer for the incoming data.
len	The length of buf.
from	A pointer to the IP address which will hold the source host’s address upon return.

17.3.2.2 IP

The mode IP layer interface is almost the same as the UDP interface, except that the port number is gone, and the buffer has a UDP header (containing the port number among other things) on it in addition to the application data.

int IPsendto (SOCKET s, char * buf, int len, IPaddr to);

s	A descriptor identifying a socket.
buf	A buffer containing the data to be transmitted.
len	The length of the data in buf.
to	IP address of the target host.
Returns	the number of bytes sent.

int IPrecvfrom (SOCKET s, char * buf, int len, IPaddr * from);

s	A descriptor identifying a socket.
buf	A buffer for the incoming data.
len	The length of buf.
from	A pointer to the IP address which will hold the source host’s address upon return.

17.3.2.3 ARP

The function of ARP is to translate IP addresses into lower level addresses that can be used to actually send packets.

STATUS ARPtranslate(IPaddr to, char * DLaddr, int * DLlen);

to	IP address of target host to translate
DLaddr	A buffer containing on return the lower level address of the target host.
DLlen	The length of the data in Ladder.

17.3.2.4 Routing

The function of the routing module is to map the IP address of the target host into the IP address of the first (perhaps only) hop of a route from the source host to the target host.

STATUS Route(IPaddr to, IPaddr * first);

to	IP address of target host to translate
first	IP address of first hop

For brevity, how the routing module does this mapping is omitted. At the bare minimum, the host is configured with the IP address of a *default gateway,* which is the router to use for all hosts not on the same subnet as the sending host, and a *subnet mask* which can be used to computer whether a target host is on the same subnet. For a given *to* address, if it is on the same subnet as the sending host, it is returned; if not, the default gateway's IP address is returned.

17.3.2.5 NIC Driver

The interface to the NIC driver looks similar to the interface to IP, except that the address has been translated to a data link address, and the buffer being sent has an IP header prefixed. The underlying data link has some maximum message size that the hardware can send in one unit; this is called the maximum transmissible unit (MTU). The NIC driver will often add a data link layer checksum on transmission, and check it on reception; packets that don't check are usually discarded.

int NICsendto (SOCKET s, char * buf, int len, char * DLaddr,
int DLlen);

s	A descriptor identifying a socket.
buf	A buffer containing the data to be transmitted.
len	The length of the data in buf.
DLaddr	Opaque data link layer MAC address of the target system.
DLlen	The length of the data in DLaddr.
Returns	the number of bytes sent.

int NICrecvfrom (SOCKET s, char * buf, int len, char * DLaddr,
int * DLlen);

s	A descriptor identifying a socket.
buf	A buffer for the incoming data.
len	The length of buf.
DLaddr	Opaque data link layer MAC address of the sending system.
DLlen	The length of the data in DLaddr.
Returns	the number of bytes received.

int NICgetMTU (SOCKET s);

s	A descriptor identifying a socket.

Returns the maximum transmissible unit for the medium.

17.3.3 Packet transmission flow

The flow for packet transmission falls out pretty naturally from the above modular decomposition.

1. Applications calls `sendto` which calls `UDPsendto`.
2. UDP adds UDP header and calls `IPsendto`.
3. IP calls `Route` to get the first hop IP address
4. IP calls `ARPtranslate` with that IP address to get the data link layer address.
5. IP calls `NICgetMTU`; breaks packet into fragments if it is bigger than the MTU.
6. For each fragment, it calls `NICsendto`.

17.3.4 Packet reception flow

Is omitted for brevity.

17.4 IP over ATM

In this section, we extend the implementation model to support ATM, via the addition of a connection manager (CM) in between IP and ARP, and the NIC driver. The purpose of the CM is to create ATM connections as needed to a given ATM address. A block diagram of this configuration follows below.

The CM performs the same functions for IP and ARP, and has the same interface, as the NIC driver did above: given a data link MAC address, it can send/receive packets to/from that address. The only difference is that now the opaque addresses contain ATM addresses -- essentially, the CM makes the ATM network function as a (connection-oriented) data link as far as IP is concerned. If the CM is asked to send a packet to a system with an ATM address to which no connection currently exists, it uses Q.2931 to create one. Q.2931 messages are sent, using the NIC driver, over a permanent virtual circuit (PVC) to the ATM switch. If a connection isn't used for long enough, it releases it. When the CM initializes, it uses the ILMI (Interim Local Management Interface) protocol specified by UNI 3.1 to obtain the ATM address of the host system.

ARP performs the same function and has the same interface as in section 1 above. However, in this configuration, the address returned by ARP is an ATM address, which is transparent to IP, since it treats them as opaque byte strings. Its internal working is also different -- instead of broadcasting ARP requests, it sends them to an "ARP server" which has a database that maps IP addresses to ATM addresses. The ATM address of the ARP server is typically in a configuration database; ARP uses the CM to open a connection to it. RFC 1577 specifies how ARP works in an ATM environment.

The NIC driver's interface is unchanged from above, but the DLaddr is an ATM VPI/VCI instead of the MAC address; it can send/receive packets to/from Vcs.

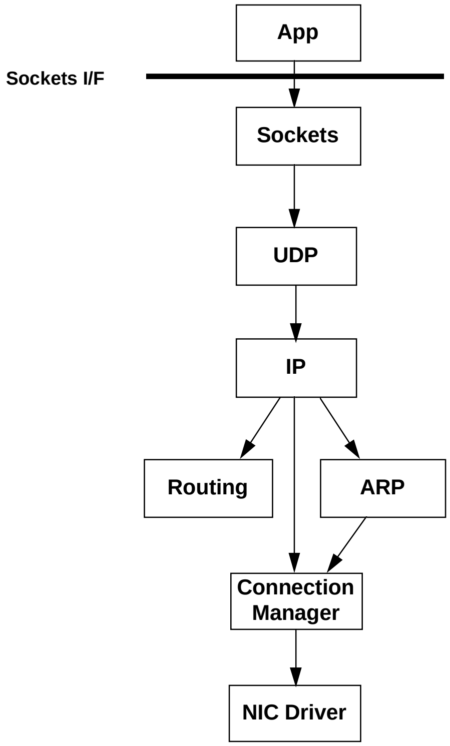


Figure 17-1. IP Model - ATM

17.4.1 Packet transmission flow

The flow for packet transmission again falls out pretty naturally from the above modular decomposition.

1. Applications calls `sendto` which calls `UDPsendto`.

2. UDP adds UDP header and calls `IPsendto`.
3. IP calls `Route` to get the first hop IP address
4. IP calls `ARPtranslate` with that IP address to get the data link layer address.
5. If needed, ARP calls CM to get connection to ATM ARP server.
6. If not cached, ARP sends request for translation to ATM ARP server.
7. IP calls `NICgetMTU` entry in CM; breaks packet into fragments if it is bigger than the MTU.
8. For each fragment, it calls `NICsendto` of CM.
9. CM creates connection to target host if one doesn't exist, using `NICSendTo` of NIC driver to send Q.2931 messages over VCI 5.
10. CM sends IP fragment using `NICSendTo` entry of NIC driver.

17.5 IP over ATM with DSM-CC U-N Session Management

In this section, we extend the implementation model to support DSM-CC networks, via the addition of a DSM-CC U-N session management module in between the CM and NIC driver. The (main) purpose of the DSM-CC module is to create sessions as needed so that connections can be properly established to a given ATM address. A block diagram of this configuration follows.

Everything is as above for ATM, except that when the CM is asked to create a connection to an ATM address, and it has no session, then it uses DSM-CC module (via the DSM-CC ClientSessionSetUp message sequence) to create one, and when a session has been used for long enough, it releases it. The CM then includes the session ID in the BHLI of the Q.2931 messages that are used to create and release connections. The DSM-CC layer sends DSM-CC messages using the NIC driver and a pre-provisioned PVC to the SCCMC.

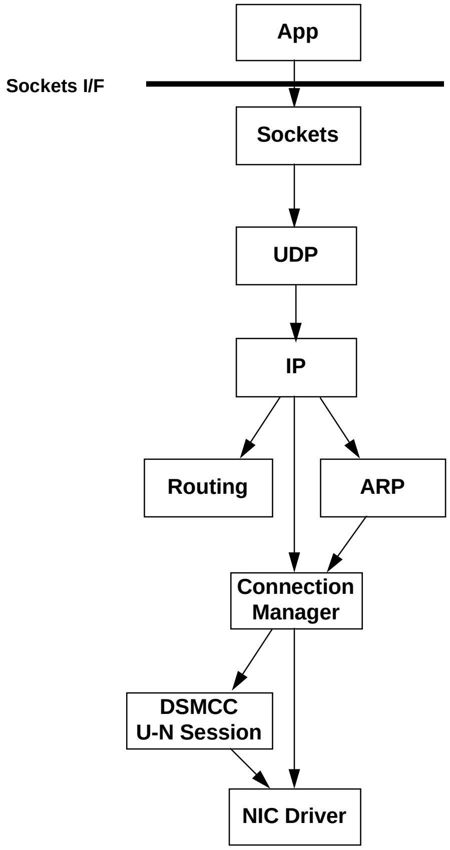


Figure 17-1. IP Model - ATM with DSM-CC U-N Session

17.5.1 Packet transmission flow

The flow for packet transmission again falls out pretty naturally from the above modular decomposition.

1. Applications calls `sendto` which calls `UDPsendto`.
2. UDP adds UDP header and calls `IPsendto`.
3. IP calls `Route` to get the first hop IP address
4. IP calls `ARPtranslate` with that IP address to get the data link layer address.
5. If needed, ARP calls CM to get connection to ATM ARP server.
6. If not cached, ARP sends request for translation to ATM ARP server.
7. IP calls `NICgetMTU` entry in CM; breaks packet into fragments if it is bigger than the MTU.
8. For each fragment, it calls `NICsendto` of CM.
9. CM calls DSM-CC U-N session manager module to create session if one doesn't exist.
10. CM creates connection to target host if one doesn't exist, using `NICSendTo` of NIC driver to send Q.2931 messages over VCI 5.
11. CM sends IP fragment using `NICSendTo` entry of NIC driver.

17.6 IP over DSM-CC U-N Session and Connection Management

In this section, we extend the implementation model to support DSM-CC networks that are not based on ATM, via the replacement of the CM with a DSM-CC U-N connection management module. In this scenario, the DSM-CC connection manager module creates connection when needed, just like the CM in previous scenarios, except that it uses DSM-CC ServerAddResources messages instead of Q.2931. A block diagram of this configuration follows.

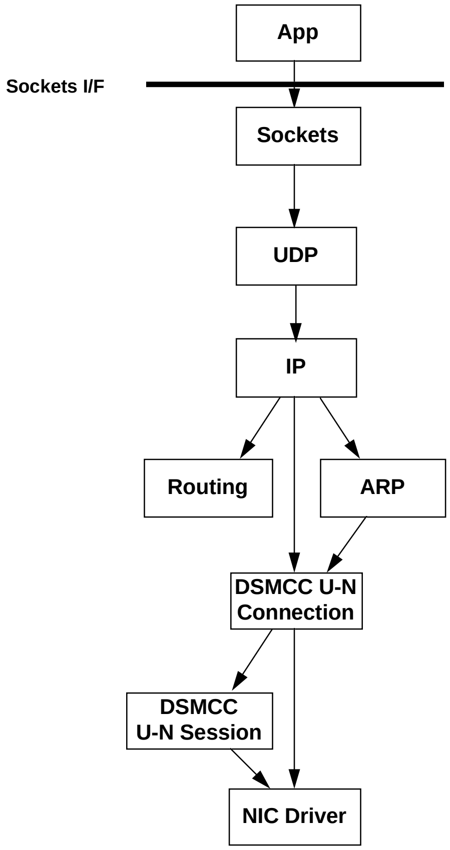


Figure 17-1. IP Model - DSM-CC Session and Connection Management

17.7 Example with DSM-CC U-U and WWW

Here is an example showing the protocol stack for two applications: an ITV application using DSM-CC U-U and a Web Browser.

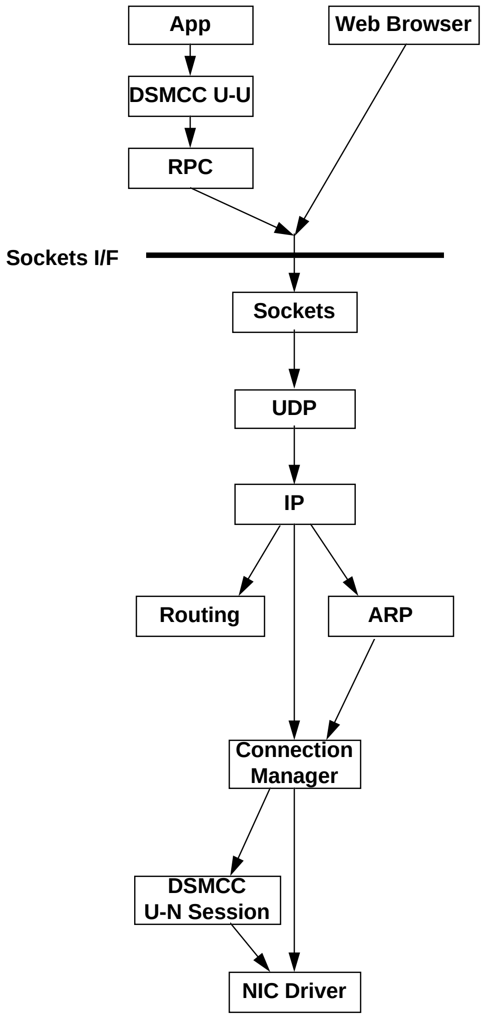


Figure 17-1. DSM-CC U-U Application and Web Bowser

17.8 Observations and Issues

Since in a pure IP/ATM environment, there are no other resources besides ATM connection resources, Addesources messages aren't needed in this configuration. Since it uses ILMI to obtain its ATM address, UNConfig messages aren't needed in this configuration.

Since IP makes no distinction between clients and servers, it always uses ClientSessionSetUp sequences to establish sessions.
In this model, sessions start at boot and end at shutdown, due to the requirement that sessions be transparent to IP. However, new applications, such as one to control session duration, are possible. With such an application, multiple sessions in a single client or server would be possible.
Similarly there is no way, without sesssion aware apps, for a server to to carry sessions over to other servers it is communicating with on behalf of a client. In order to do this, one would need `getsockopt`() and `setsockopt`() calls to get and set the sessionId being used on an IP connection. Also, it would be

necessary to establish multiple connections for multiple sessions where otherwise one might do -- classical IP over ATM only uses one connection per pair of hosts.