INTERNATIONAL STANDARD

ISO/IEC 13818-6

First edition
1998-09-01

Information technology — Generic coding of moving pictures and associated audio information —

Part 6:
Extensions for DSM-CC

Technologies de l'information — Codage générique des images animées et des informations sonores associées —

Partie 6: Extensions pour DSM-CC

Reference number
ISO/IEC 13818-6:1998(E)

CONTENTS

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13818-6 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information.*

ISO/IEC 13818 consists of the following parts, under the general title *Information technology — Generic coding of moving pictures and associated audio information*:

— *Part 1: Systems*

— *Part 2: Video*

— *Part 3: Audio*

— *Part 4: Compliance testing*

— *Part 5: Software simulation*

— *Part 6: Extensions for DSM-CC*

— *Part 7: Advanced Audio Coding (AAC)*

— *Part 9: Extension for real time interface for systems decoders*

— *Part 10: Conformance extensions for DSM-CC*

Annex A forms an integral part of this part of ISO/IEC 13818. Annexes B to N are for information only.

0. Introduction

The Digital Storage Media Command and Control (DSM-CC) specification is an integral part of ISO/IEC 13818 (MPEG-2). It consists of a modular set of protocols that may be combined or used individually to provide a wide range of functionality which may be used to support emerging multimedia technologies.

The concepts and protocols of DSM-CC provide the general capability to browse, select, download, and control a variety of bit stream types. DSM-CC also provides a mechanism to manage network and application resources through the concept of a "session". A Session is an associated collection of resources required to deliver a Service. Examples of resources are MPEG-2 Transport Stream packet identifiers and network bandwidth. The Session complements a "Service Domain", which is a collection of interfaces to browse and select services, and control the delivery of bit streams.

One of the strengths of DSM-CC is in its abstraction from underlying networks; a suite of uniform interfaces are visible to the application, shielding it from the details of inter-working among heterogeneous networks – e.g., Hybrid Fiber Coax (HFC), Asynchronous Transfer Mode (ATM), Asymmetric Digital Subscriber Loop (ADSL), Internet Protocol (IP), and combinations of these technologies as part of an end-to-end multimedia system. In other words, a server may simultaneously and uniformly interact through a single network interface with clients connected to different network types, without requiring a separate network interface to each client.

The session signaling layer provides a uniform, flexible, and extensible method for managing heterogeneous resource types. In addition to the network and service types described in this specification, DSM-CC may be extended to support other networks and services through the definition of new resource types.

In DSM-CC, a bit stream is sourced by a Server and delivered to a Client. Both the Client and the Server are logical embodiments and do not imply a singular device in an actual implementation.

Application/service examples are interactive multimedia retrieval (including video-on-demand), Internet access, digital video broadcasting, data downloading, and audio/video/graphics conferencing.

0.1 Guiding Factors in the Formulation of DSM-CC

The DSM-CC specification was influenced by the following factors:

- A wide range of network topologies may be used to deliver DSM-CC.
- Resources are finite and need to be managed.
- Latencies need to be minimized to provide (interactive) services.
- DSM-CC applications need to be supported by an underlying protocol that facilitates communications between a Server-side application and a corresponding Client-side application.

0.2 DSM-CC Client-Network-Server Model

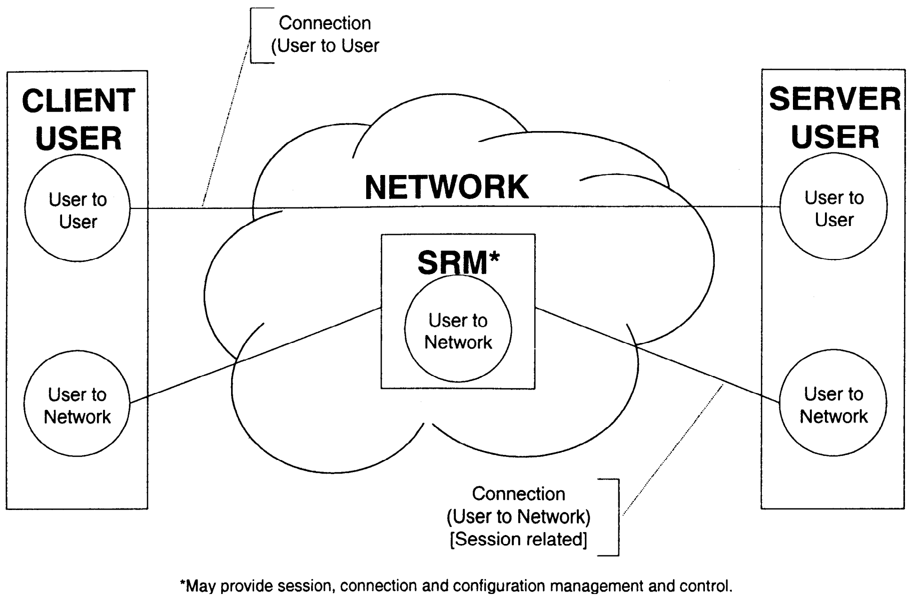


Figure 0-1 DSM-CC basic Client-Network-Server model

Figure 0-1 depicts the basic model used in DSM-CC. A Session and Resource Manager (SRM) provides logically centralized management of the DSM-CC Sessions and Resources. DSM-CC User-to-Network (U-N) messages flow between the Client and SRM and the Server and SRM. Both the Client and the Server are called Users of DSM-CC. The U-N session protocol establishes a Session and groups all the resources required for delivering a service. The service interactions are carried between the Client and the Server participating in the Session using DSM-CC User-to-User (U-U) messages. The SRM also does U-N configuration management and control of both Clients and Servers to allow their participation in the DSM-CC environment.

DSM-CC supports network topologies which consist of multiple Clients and multiple Servers. Any Client-Server pair can communicate together by establishing a Session. Each Client can have multiple simultaneous Sessions with any specific Server or any combination of Servers. For this phase of DSM-CC, a Session is typically limited to one Client, one Server, and one SRM. The exception is the case of Continuous Feed Sessions (CFS). A CFS may be used by, e.g., a stream broadcasting application, where broadcast "feeds" are established with the network with no particular Client specified. Clients may "attach" to a CFS by setting up a Session with the network to connect to the CFS and, optionally, to establish Client-unique resources (such as a return control channel that may be needed by an interactive application which shares a downstream feed, e.g., game show voting). Alternatively, Clients may "attach" to a CFS or another broadcast "feed" by using the U-N Switched Digital Broadcast Channel Change Protocol (SDB-CCP), when no Client-unique Resources are needed by the application (such as with traditional "pay-per-view").

0.3 Outline of the DSM-CC Specification

DSM-CC consists of a set of User-to-Network and User-to-User protocols. These protocols are described in the clauses listed below.

0.3.1 User-to-Network

- DSM-CC Message Header, clause 2
- U-N Configuration messages, clause 3
- U-N Session messages and flow diagrams for Session and Resource management, clause 4
- U-N Download messages, clause 7
- U-N Switched Digital Broadcast Channel Change Protocol, clause 10
- U-N Pass Thru messages, clause 12
- The transport of DSM-CC U-N messages using MPEG-2 Systems (ISO/IEC 13818-1), clause 9
- The transport of generic IP messages using DSM-CC Sections and MPEG-2 Systems, clause 9

0.3.2 User-to-User

- U-U Remote Procedure Call (RPC), clause 5
- U-U Session interface, clause 5
- U-U Download interface, clause 5
- U-U Object Carousel interface, clause 11
- U-U Local Object interface, clause 5
- U-U Stream Descriptors, clause 8

0.4 Supported Network Technologies

DSM-CC does not specify the underlying physical, data link, transport, or RPC layers of the overall protocol stack. However, DSM-CC does specify requirements for these layers in clause 9.

0.5 Supported Connection Types

DSM-CC supports the following connection types:

- Point-to-point
- Point-to-multi-point (broadcast)

User-to-User application and service exchanges are carried over point-to-point type connections.

The point-to-multi-point type connections are used to feed a single stream to multiple Clients. In this case, no single Client has control (e.g., for the purpose of pause, fast forward) of the received bit stream. However, in the case where the network (as opposed to the Client) does stream switching such as with Switched Digital Broadcast (SDB) applications, a means is provided for Clients to switch between streams using the SDB channel change protocol (SDB-CCP). The latter is useful for applications such as the so-called "enhanced pay-per-view" or "near video on demand".

0.6 DSM-CC Interfaces

The DSM-CC model (Figure 0-1) consists of three Subsystems:

- Client
- Server
- Session & Resource Manager (SRM)

Each subsystem is a logical embodiment within a DSM-CC System. It does not map directly to physical equipment. The SRM represents the DSM-CC functionality within a DSM-CC network (the Network).

In order to define interfaces, a DSM-CC System Reference Model is used to subdivide the DSM-CC environment into a hierarchy of several levels (see Figure 0-2):

- System
- Subsystem
- Entity
- Sub-entity

A Subsystem may contain more than one Entity. The types of Entities are:

- Client User-to-User Entity
- Client User-to-Network Entity
- Server User-to-User Entity
- Server User-to-Network Entity
- SRM User-to-Network Entity

DSM-CC signaling is always exchanged between specific Subsystem Entities.

From the normative perspective, the System Reference Model does not show any more detail below the Entities. However, for informative reasons, the Entities have in some cases been further divided into Sub-entities. For example, the Client and Server U-N Entities include a Session Manager, Resource Manager, and a Configuration Manager. On the Server side, the U-U Entity includes a Service Gateway and an Object Access Manager.

DSM-CC recognizes that Subsystems will require other Entities which are not specified by this part of ISO/IEC 13818. Examples are a Connection Management Entity and Application Entities.

Figure 0-2 shows the different entities and sub-entities of the DSM-CC system.

To show the boundaries between Entities, Figure 0-2 is divided into four layers:

- The Application Entity layer, which is outside the scope of this part of ISO/IEC 13818
- The User-to-User Entity, which is an Application/Service Control/Management Layer
- The User-to-Network Entity, which is a Session and Resource Control/Management Layer
- The Connection Control Entity, which is (typically part of) a Transport Layer, and is outside the scope of this part of ISO/IEC 13818.

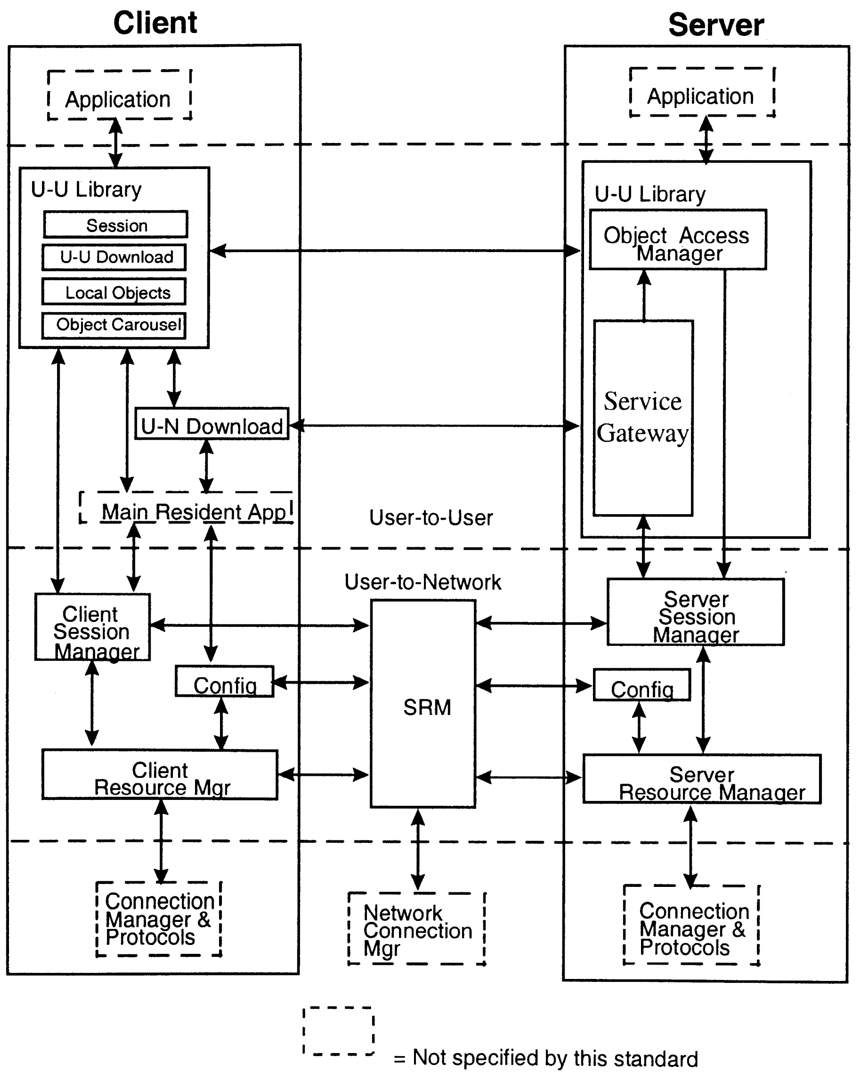


Figure 0-2 DSM-CC System Reference Model

Each of the lines with arrows in Figure 0-2 represents a logical interface. DSM-CC defines three types of interfaces:

- Inter-Entity

- Intra-Entity
- Intra-Subsystem

The Inter-Entity interfaces are between peer Entities in different Subsystems. The interfaces between the Sub-entities within a common Entity are called Intra-Entity interfaces. The interfaces between Entities within a common Subsystem are called Intra-Subsystem interfaces.

The DSM-CC System Reference Model specifies three communication paths over which DSM-CC messages are exchanged. The communication between U-U Entities are represented as the DSM-CC U-U Protocol. The communication between U-N Entities are represented as the DSM-CC U-N Protocol.

- Client U-U Entity to Server U-U Entity (U-U)
- Client U-N Entity to SRM U-N Entity (U-N)
- Server U-N Entity to SRM U-N Entity (U-N)

Table 0-1 summarizes Inter-Entity interfaces and Intra-Subsystem interfaces within the scope of DSM-CC.

Table 0-1 DSM-CC Interface Scope Summary

Peer 1	Peer 2	Protocol	Inter-Entity	Intra-Subsystem
Client U-U Library	Server Service Gateway	U-U	X	
Client U-U Library	Server Object Access	U-U	X	
Client Session Gateway	SRM	U-N	X	
Client Resource Manager	SRM	U-N	X	
Server Session Manager	SRM	U-N	X	
Server Resource Manager	SRM	U-N	X	
Client Configuration	SRM	U-N Config	X	
Server Configuration	SRM	U-N Config	X	
Server DSM Source (e.g. MPEG-2 Transport / Video / Audio)	Client DSM Consumer	(MPEG)	X[1]	
Download Server (Source)	U-N Download Client (Consumer)	Down-load	X[1]	
Object Carousel Server	Object Carousel Client	Object Carousel / Down-load	X[1]	
SDB Server	(SDB) Client	SDB-CCP	X[1]	
Client Application	Client U-U Library	U-U		X

Note 1: Interface not shown on Figure 0-2.

0.7 DSM-CC Interface Protocols

Figure 0-3 depicts DSM-CC protocols used at DSM-CC interface points. The top section of the figure contains some applications which may use DSM-CC. The middle section of the figure contains all of the DSM-CC specified protocols. The specific Transport Layers, the bottom section, are not specified by this part of ISO/IEC 13818.

Note that Figure 0-3 applies to the case where the full suite of DSM-CC protocols (except for the extended protocol groups) are employed. DSM-CC allows each protocol to be implemented without the others (see subclause 1.2 Profiles and Compliance). If the U-U Library is not used, then the implementation will not have an Application Portability Interface specified by DSM-CC.

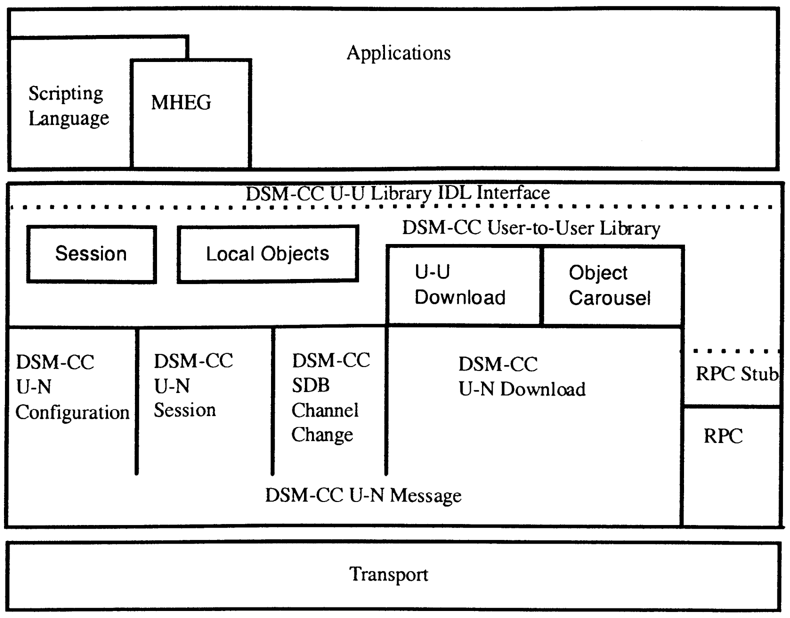


Figure 0-3 DSM-CC Interface Protocols

DSM-CC provides access to Stream and Data objects for applications (e.g., MHEG applications and scripting language applications). The primary application interface layer is the DSM-CC U-U Library Interface Definition Language (IDL), or Application Portability Interface. The U-U Library may in turn make use of the U-N Session Management, U-N Download, and U-U Object Carousel layer to establish and manage Sessions and Resources required for the management and delivery of the Stream and Data objects.

Table 0-2 lists the DSM-CC protocols. The protocols which use the DSM-CC message format are U-N Configuration, U-N Session, U-N SDB-CCP, U-N Download, and U-U Object Carousel (because it, in turn, uses U-N Download). In some cases, the use of a message passing interface is needed because the Client device may not have higher layer protocols (e.g., RPC) resident.

The U-U Library uses the services of the U-N protocols, but also adds its own on-the-wire protocol, the U-U RPC Stub Library, which are based on existing Remote Procedure Call (RPC) interfaces. The protocols which use RPC do so because it provides sophisticated object based services.

The third category is IDL, which is used in communicating within the Subsystem to applications.

Table 0-2 The DSM-CC Protocols used on the Interfaces

DSM-CC Protocols	Peer 1	Peer 2	U-N Message Format	IDL/RPC
U-N Configuration	Client / Server	SRM	x	
U-N Session	Client / Server	SRM	x	
U-N Download	Client	Download server	x	
U-N Switched Digital Broadcast Channel Change	Client	SDB server	x	
U-N Pass Thru	Client	Server	x	
U-U RPC	Client	RPC server		RPC
U-U Session	Client Application	Client U-U Library		IDL
U-U Download	Client Application	Client U-U Library		IDL
U-U Object Carousel	Object Carousel Client	Object Carousel	x	
U-U Local Objects	Client Application	U-U Library		IDL

The transport layer in Figure 0-3 may consist of any protocol which meets the transport requirements described in clause 9. Examples are, TCP or UDP over IP, AAL-5 over ATM, or DSM-CC/private_sections over MPEG-2 Transport Stream.

0.8 Communications Requirements

The DSM-CC U-N Configuration, U-N Session, U-N SDB-CCP and U-N Download messages all use the DSM-CC Message Format and are implemented using a simple message passing method; therefore, all have similar Transport Layer requirements. The U-U Object Carousel uses the U-N Download protocol and its associated transport requirements. The U-U RPC Stub Library uses RPC and its associated transport requirements.

The requirements for the underlying Transport services for all DSM-CC protocols are provided in detail in clause 9, Transport.

0.9 Methods of Specification

0.9.1 Messages

U-N messages are described in tables which list the bit or byte level assignment for all of the fields in each message. The syntactical structure of the messages are defined by Syntax Tables like the example below. Field names are shown in bold and always have an associated number of bytes indicated. All numeric values are unsigned big-endian (most significant byte first, most significant bit first) unless otherwise specified. The method of syntax description supports loops and 'procedures' using a pseudo-C syntax. In the example below, a for() loop, in normal font, indicates that the field **uuDataByte** repeats uuDataCount times. Also, the structure has been named UserData(), which now can in turn be referenced in other larger structures.

```
                         Syntax                          Num. of Bytes
UserData(){
        uuDataLength                                                2
        for(i=0;i<uuDataCount;i++) {
                uuDataByte                                          1
        }
        privateDataLength                                           2
        for(i=0;i<privateDataLength;i++) {
                privateDataByte                                     1
        }
}
```

Figure 0-4 Example of U-N message syntax

The messages for U-N Configuration and U-N Session flow between Client and Network (SRM), and Server and Network (SRM). For consistency, the suffix of each of these messages use the following terminology:

Request - A message sent from a User (Client or Server) to the Network to begin a scenario.

Confirm - A message being sent from the Network to a User (Client or Server) in response to a Request message.

Indication - A message which is sent from the Network to a User.

Response - A message from a User to the Network in response to an Indication message.

Clause 9, Transport, defines the communications requirements (reliability, addressing etc.) for the delivery of these messages.

A standard programming API for the use of these messages is outside the scope of this part of ISO/IEC 13818.

0.9.2 Message Flow Diagram Scenarios

Flow diagrams have been provided to help explain the use of the DSM-CC message protocols. These diagrams show the sequence and direction of flow for the messages of a specific scenario. In these diagrams, the time axis runs vertically, with messages lower on the diagrams representing later transmission. The selected scenarios are the most typical ones and do not represent the exhaustive list of examples of scenarios. The Specification and Description Language (SDL) representations provide a more exhaustive representation, including exception cases.

0.9.3 Specification and Description Language

The SDL-language is officially defined in ITU-T recommendation Z.100. For the translation of the DSM-CC specification into SDL, SDL-88 (Z.100 blue book) is used. There are several advantages to using SDL:

- Contrary to the textual part, usage of SDL in the specification makes it unambiguous due to the fact that SDL is a formal language.
- One representation of SDL is the graphical one. This makes the language more comprehensible.
- The SDL specification can be analyzed for completeness and correctness.
- It is easy to generate executable code in order to simulate and validate the specification.
- The specification can also be used for conformance test purposes.

For simulation purposes, Message Sequence Charts (MSC), as defined in ITU-T recommendation Z.120, are used.

A model described in SDL consists of three different types of levels.

1. System level
2. Block level
3. Process level

The highest level of the SDL model is the system level. The system is surrounded by the environment represented by a rectangle in the graphical representation. On the system level, the model of the system is described in a very rough shape divided into one or more blocks. The blocks can contain either new blocks or processes. At some block level, the content is one or several processes in each block. The process level could then describe logical parts of the system related to each other with the signals exchanged between them.

A static process is created at start-up time for the system. A dynamic process is created during runtime by another process. The number of dynamic processes which may be created is set by a constant value. A process can be stopped by the process itself at any point in time.

A process is a state machine and the only way to move from one state to another state is via a transition. One or several possible transitions can be connected to a state. A transition is always initiated by either an input signal or an enabling condition. An input signal can be generated by an output signal from an outside process, from within the same process, or by an expired timer. Here, the environment is also regarded as a process. The input signal is put in an input queue which is a common queue for the process.

When an input signal is consumed, a transition is started and the actual code defined between the state and the next state is executed. In the graphical representation, the code consists of one or several graphical symbols with some additional plain text; variables may be assigned new values in a task, questions may be answered in a decision, an output signal may be sent to another process, etc.

Figure 0-5 shows some common SDL symbols. Complete specification of SDL is outside of the scope of this specification, but may be found in ITU-T Z.100 and Z.120.

The intent is to have the message flow diagrams and prose be consistent with the SDL tables. Since the SDL is more exhaustive, if there is any form of contradiction between the prose and SDL, the SDL shall take precedence.

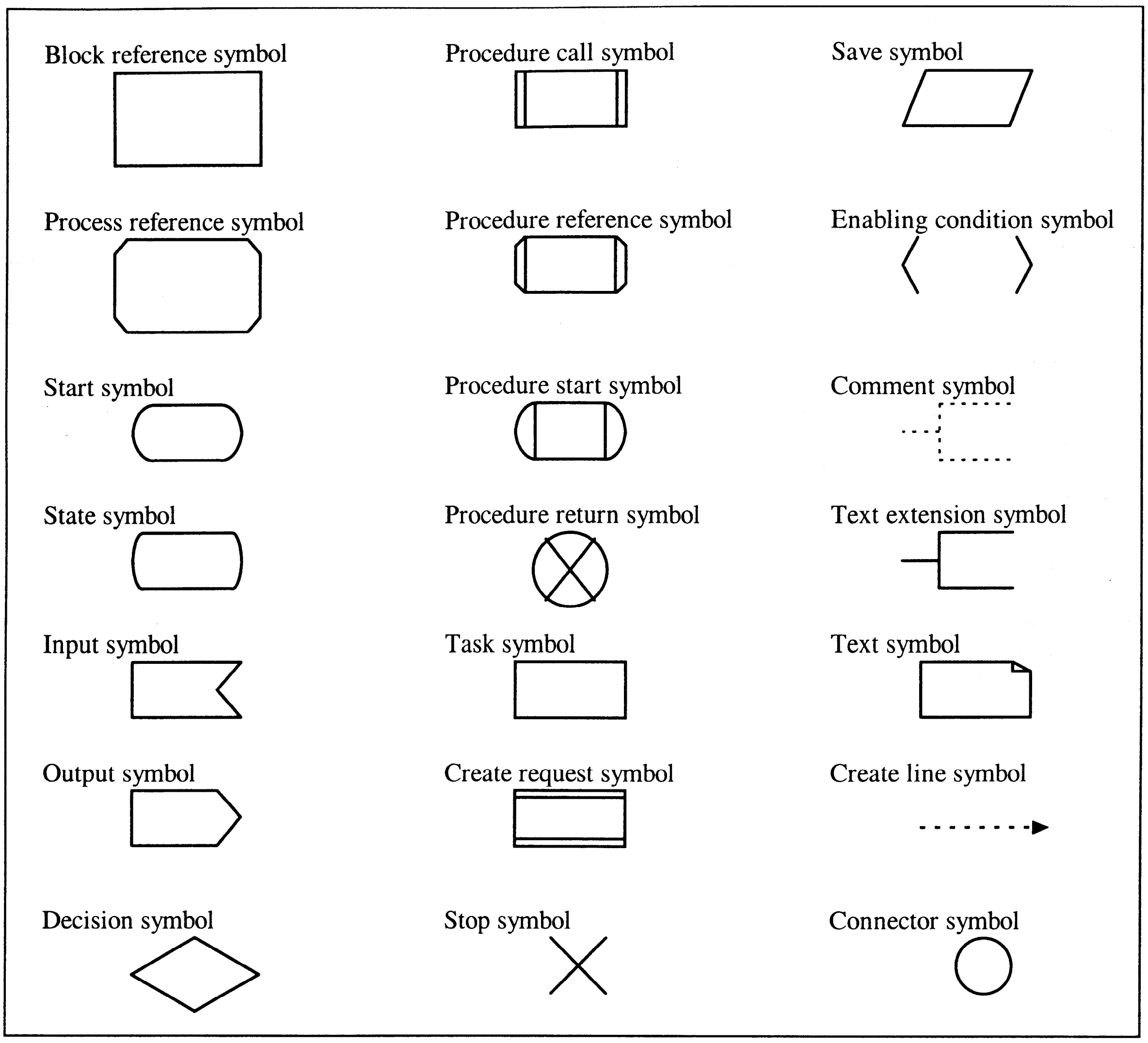


Figure 0-5 SDL Symbols

0.9.4 Interface Definition Language (IDL)

The U-U API primitives that use RPC are defined in terms of OMG Interface Definition Language (IDL), defined by ISO/IEC 14750. 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.

0.9.5 Remote Procedure Call (RPC)

U-U functionality exploits a Remote Procedure Call (RPC) protocol. A 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. For those U-U API primitives that use the RPC, the RPC and data encoding defines the actual bits that are exchanged as primitives are executed.

The downstream reply from the Server can be delivered via encapsulation within a MPEG-2 Transport Stream. Although this part of ISO/IEC 13818 specifies how to encapsulate common protocols (e.g., IP) over MPEG-2 Transport, there is no requirement that control messages or RPC messages be delivered within MPEG-2 Transport Streams.

0.9.5.1 Independence of RPC

DSM-CC may be implemented using any RPC which utilizes 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: for example, Common Data Representation (CDR) or External Data Representation (XDR).

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.

0.9.5.2 Preferred and Default RPC

DSM-CC User-to-User has designated OMG Universal Networked Objects (UNO) RPC as the default and preferred RPC (see clause 5). The preferred and default data representation is Common Data Representation (CDR).

In the absence of prior arrangement, the default RPC between two DSM-CC Users is the UNO RPC. Note that the UNO RPC supports the ability to subsequently negotiate a change to a different RPC.

0.9.5.3 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.

Information technology — Generic coding of moving pictures and associated audio information Part 6: Extensions for Digital Storage Media Command and Control (DSM-CC)

1. General

1.1 Scope

The concepts and protocols of this part of ISO/IEC 13818 (DSM-CC) provide the general capability to browse, select, download, and control a variety of bit stream types. DSM-CC also provides a mechanism to manage network and application resources through the concept of a Session, an associated collection of resources required to deliver a Service. The Session complements a "Service Domain", a collection of interfaces to browse and select services, and control the delivery of bit streams.

DSM-CC defines the syntax and semantics for a set of User-to-Network and User-to-User protocols:

- DSM-CC Message Header
- U-N Configuration messages
- U-N Session messages and flow diagrams for Session and Resource management
- U-N Download messages
- U-N Switched Digital Broadcast Channel Change Protocol
- U-N Pass Thru messages
- The transport of DSM-CC U-N messages using ISO/IEC 13818-1.
- The transport of generic IP messages using DSM-CC sections and ISO/IEC 13818-1, clause 9
- U-U Remote Procedure Call
- U-U Session interface
- U-U Download interface
- U-U Object Carousel interface
- U-U Local Object interface
- U-U Stream Descriptors

1.2 Profiles and Compliance

The DSM-CC protocols are modular and may be used individually or together to provide the needed range of features and functionality. In other words, an embodiment of DSM-CC is not required to implement every functional category (see below). However, if an embodiment implements operations that would be analogous to a functional category, then the embodiment shall implement the complete syntax and semantics of the corresponding DSM-CC functional category.

1.2.1 Functional Categories of the DSM-CC protocols

- User-to-Network Configuration
- User-to-Network Core Session Messages
- User-to-Network Extended Session Messages
- User-to-Network Flow Controlled Download
- User-to-Network Non-Flow Controlled Download
- User-to-Network Data Carousel Download
- User-to-Network Pass Thru
- User-to-Network Pass Thru Receipt

- User-to-Network Switched Digital Broadcast Channel Change
- User-to-User Core Interfaces
- User-to-User Extended Interfaces

1.2.2 User-to-Network Session Messages

The U-N Session Message protocol has been divided into Core and Extended categories. DSM-CC U-N implementations shall completely support all groups within the Core category (see below). The Extended category has been further divided into independent functional message groups. Implementation of the features/functions provided by the Extended category is not required. However, if an Extended category group is implemented, the complete set of messages within that group shall be implemented.

1.2.2.1 U-N Core Session Message Functional Groups

- Session Setup Group
- Session Release Group
- Add Resource Group
- Delete Resource Group
- Continuous Feed Session Setup Group
- Status Request Group
- Reset Group
- Session Proceeding Group (optional send by Network, required receive by User)
- Session Connect Group (optional send by User, required forward by Network, required receive by both)

1.2.2.2 U-N Extended Session Message Functional Groups

- Session Transfer Group
- Session in Progress Group

1.2.3 User-User Interfaces

The U-U IDL Interfaces have been divided into two categories: Core Interfaces and Extended Interfaces. Each of these is further divided into Consumer and General. The Consumer interfaces are designed for applications where data transfers are primarily from the Server to the Client. Consumer interfaces include file read and video stream control. General interfaces, on the other hand, extend the Consumer interfaces to include author and writer functions. A Client with a General interface can, for example, create Directories and store multimedia objects in them.

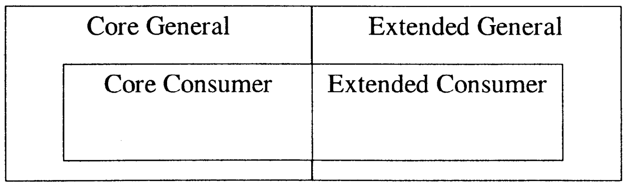


Figure 1-6 U-U IDL Interface Groups

DSM-CC U-U Client implementations shall fully support the Core Consumer Interfaces. DSM-CC U-U Server implementations shall fully support the Core General Interfaces. Each interface within the Extended set of interfaces may be implemented separately; however, if any interface is implemented, it shall be implemented as either the complete Extended Consumer Interface or the complete Extended General Interface.

1.2.3.1 U-U Core Interfaces

• Base	common close() and destroy() operations
• Access	common permissions, size, history attributes
• Stream	to control continuous streams, e.g. video

• File	to read and write files
• Directory	to create and browse multimedia directories
• Session	function calls to perform U-N Session message sequences
• ServiceGateway	combined Directory and Session
• First	to obtain initial ServiceGateway and application objects

1.2.3.2 U-U Extended Interfaces

• Download	function calls to perform U-N Download
• Event	to subscribe to data events synchronized with audio/video
• Composite	for grouping several application objects in a set
• View	to query and store databases
• State	to suspend and resume applications
• Interfaces	to define and verify new interfaces
• Security	to post authentication data for a subsequent request
• Configuration	to configure for synchronous or asynchronous RPC
• Life cycle	to create an Inter-operable Object Reference
• Kind	to test the kind of an object

1.3 Definitions

For the purposes of this part of ISO/IEC 13818, the definitions given in ISO/IEC 13818, ISO/IEC 11172, and the following definitions apply.

1.3.1.	Application	Software that executes in a client environment
1.3.2.	Association Tag	In the case of connection resources, an Association Tag identifies the groups of resources or shared resources that together make up a User to User connection. An Association Tag is unique within a session and has end-to-end significance.
1.3.3.	Client	A consumer of a service from one or more Servers.
1.3.4.	Connection	A transport link that provides the capability to transfer information between two or more end points.
1.3.5.	Downstream	Data delivery from the Server to the Client
1.3.6.	Entity	A functional module within a Subsystem, e.g. a Client Subsystem has a U-N Entity and an U-U Entity.
1.3.7.	Inter-Entity Interface	An interface between two Entities which are in different Subsystems
1.3.8.	Intra-Entity Interface	An interface between two Sub-entities which are both within the same Entity.
1.3.9.	Intra-Subsystem Interface	An interface between two Entities which are in the same Subsystem.
1.3.10.	Main Resident Application	The application (or process) on the Client which is present before execution of any DSM-CC protocols, and is the initiator of the first DSM-CC protocol exchange, e.g. U-N Configuration.
1.3.11.	Network	A collection of communicating elements that provides connections and may provide session control and/or connection control to User(s).
1.3.12.	Primary Service	The first service with which the Client interacts in a related collection of services.

1.3.13. Resident Download	A resident library capable of performing the DSM-CC Download protocols.
1.3.14. Resource Descriptor	A resource descriptor stores information for a particular resource associated with a session. It contains enough information for the Network to allocate the resource, track the resource once it is allocated, and de-allocate it once it is no longer needed.
1.3.15. Resource Sharing	In a situation where two or more resources are contained within another resource, then that resource is a shared resource. The sharing of the resource is indicated by the shared resource descriptor which identifies the resource number of the shared resource.
1.3.16. Server	A provider of a service to one or more Clients.
1.3.17. 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.
1.3.18. Service Gateway	The interface which provides a directory of services and enables a Client to attach to a service domain.
1.3.19. Session	An association between two Users providing the capability to group together the resources needed for an instance of a service.
1.3.20. Session and Resource Manager	A DSM-CC subsystem which provides a logically centralized management of DSM-CC Sessions and Resources over one or more underlying network technologies.
1.3.21. Sub-entity	An internal functional partition of an Entity.
1.3.22. Subsystem	A unit of logical 'equipment' within a DSM-CC System, e.g. Client, Server, or SRM.
1.3.23. System	The embodiment of the entire scope of DSM-CC, including all Subsystems and their interfaces.
1.3.24. Tap	An application-visible object bound to a lower layer communications channel.
1.3.25. Upstream	Data delivery from the Client to the Server.
1.3.26. 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.

1.4 Acronyms

ADSL	Asymmetric Digital Subscriber Loop
AFI	Authority and Format identifier
API	Application Programming Interface
ASN.1/BER	Abstract Syntax Notation 1/Basic Encoding Rules
ATM	Asynchronous Transfer Mode
BIOP	Broadcast Inter-ORB Protocol
CDR	Common Data Representation
CFS	Continuous Feed Session
CORBA	Common Object Request Broker Architecture
CPDU	Common Protocol Data Unit
CRC	Cyclic Redundancy Check/Code
CCP	Channel Change Protocol
DCE	Distributed Computing Environment
DSM	Digital Storage Media

DSM-CC	Digital Storage Media - Command and Control
FCS	Frame Check Sequence
FTTC	Fiber To The Curb
GIT	Generic Identifier Transport (ITU-T)
GPDU	General Protocol Data Unit
HFC	Hybrid Fiber Coax
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IOR	Inter-operable Object Reference
ITU	International Telecommunications Union
IP	Internet Protocol
IWU	Inter-Working Unit
LLC	Logical Link Control
MAC	Medium Access Control
MHEG	Multimedia/Hypermedia Experts Group
MPEG	Moving Picture Experts Group
MSL	Multimedia Scripting Language
NDR	Network Data Representation (DCE)
NPT	Normal Play Time
NSAP	Network Service Access Point
OMG	Object Management Group
ONC	Open Networked Computing
OPE	Other Protocol Element (MHEG)
ORB	Object Request Broker
OSI	Open Systems Interconnection
OUI	Organization Unique Identifier
PA	Physical Address
PCR	Program Clock Reference (ISO/IEC 13818-1)
PDU	Protocol Data Unit
PES	Packetized Elementary Stream (ISO/IEC 13818-1)
PID	Packet Identifier (ISO/IEC 13818-1)
PIN	Personal Identification Number
PVC	Permanent Virtual Connection (ATM Forum)
PMT	Program Map Table (ISO/IEC 13818-1)
PS	Program Stream (ISO/IEC 13818-1)
PSI	Program Specific Information (ISO/IEC 13818-1)
RPC	Remote Procedure Call
SDB	Switched Digital Broadcast
SDL	Specification and Description Language
SDV	Switched Digital Video
SE	SubElement
SNAP	SubNetwork Attachment Point
SQL	Structured Query Language
SRM	Session and Resource Manager
STC	System Time Clock (ISO/IEC 13818-1)
SVC	Switched Virtual Connection (ATM Forum)
TCP	Transport Control Protocol
TS	Transport Stream (ISO/IEC 13818-1)
UDP	User Datagram Protocol
U-N	User-to-Network
UNI	User to Network Interface (ITU-T / ATM Forum)
UNO	Universal Network Objects
U-U	User-to-User
VOD	Video On Demand
VCR	Video Cassette Recorder
VCI	Virtual Channel Identifier (ITU-T / ATM Forum)
VPI	Virtual Path Identifier (ITU-T / ATM Forum)

XDR	External Data Representation

1.5 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 13818. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 13818 are encouraged to investigate the possibility of applying the most recent edition of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

- American National Standards Institute X3.1351 (1992), *Database Language.* [also known as SQL 92]
- ISO/IEC 8824:1990, *Information technology – Open systems interconnection – Specification of Abstract Syntax Notation One (ASN.1).*
- ISO/IEC 8825:1990, *Information technology – Open systems interconnection – Specification of basic encoding rules for Abstract Syntax Notation One (ASN.1).*
- ISO/IEC 11172-1:1993, *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 1: Systems.*
- ISO 11578:1996, *Information technology – Open systems interconnection – Remote Procedure Call.*
- ISO/IEC 13818-1:1996, *Information technology – Generic coding of moving pictures and associated audio information: Systems.* [corresponds to ITU-T Rec. H.222.0 (1995)]
- ITU-T Recommendation E.164 (05/97), *The international public telecommunication numbering plan..*
- ITU-T Recommendation Q.931 (03/93), *ISDN user-network interface layer 3 specification for basic call control.*
- ITU-T Recommendation Q.2931 (02/95, 06/97 amendment), *Digital Subscriber Signalling System No.2 – User-network interface (UNI) layer 3 specification for basic call/connection control.*
- ITU-T Recommendation Q.2932.1 (7/96), *Digital Subscriber Signalling System No.2 – Generic functional protocol: Core functions.*
- ITU-T Recommendation Q.2957 (02/95), *Stage 3 description for additional transfer supplementary services using B-ISDN digital subscriber Signalling System No.2 (DSS 2) – Basic Call.*
- ITU-T Recommendation Q.2971 (10/95), *Broadband integrated services digital network (B-ISDN) – Digital subscriber signalling system No.2 (DSS 2) User-network interface layer 3 specification for point-to-multipoint call/connection control.*
- ITU-T Recommendation Z.100 (03/93, 10/96 addendum), *CCITT Specification abd description language (SDL).*
- ITU-T Recommendation Z.120 (10/96), *Message Sequence Chart (MSC).*
- Internet Engineering Task Force RFC 1014, *XDR: External Data Representation standard*, 06/01/1987.
- Internet Engineering Task Force RFC 1057, *RPC: Remote Procedure Call Protocol specification version 2*, 06/01/1988.
- Object Management Group, *Common Object Request Broker: Architecture and Specification*, Version 2.1, August 1997. [also known as OMG CORBA/IIOP 2.1. Includes definition of the Common Data Representation, Remote Procedure Call mechanism, and Interface Definition Language syntax and semantics]

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 interfaces 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 2-1 defines the format of a DSM-CC message header.

Table 2-1 MPEG-2 DSM-CC Message Header Format

Syntax	Num. of Bytes
dsmccMessageHeader() {	
protocolDiscriminator	1
dsmccType	1
messageId	2
transactionId	4
reserved	1
adaptationLength	1
messageLength	2
if(adaptationLength>0) {	
dsmccAdaptationHeader()	
}	
}	

The **protocolDiscriminator** field is used to indicate that the message is a MPEG-2 DSM-CC message. The value of this field shall be 0x11.

The **dsmccType** field is used to indicate the type of MPEG-2 DSM-CC message. Table 2-2 defines the possible dsmccTypes.

Table 2-2 MPEG-2 DSM-CC dsmccType values

dsmccType	Description
0x00	ISO/IEC 13818-6 Reserved
0x01	Identifies the message as an ISO/IEC 13818-6 IS User-to-Network configuration message.
0x02	Identifies the message as an ISO/IEC 13818-6 IS User-to-Network session message.
0x03	Identifies the message as an ISO/IEC 13818-6 IS Download message.
0x04	Identifies the message as an ISO/IEC 13818-6 IS SDB Channel Change Protocol message.
0x05	Identifies the message as an ISO/IEC 13818-6 IS User-to- Network pass-thru message.
0x06-0x7F	ISO/IEC 13818-6 Reserved.
0x80-0xFF	User Defined message type.

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

The **transactionId** field is used for session integrity and error processing and shall remain unique for a period of time such that there will be little chance that command sequences collide. The transactionId is of local significance (i.e., Server-Network or Client-Network) only; the transactionId contained in the request-confirm command pair shall be identical, and the transactionId contained in the related indication-response command pair shall be identical; however, the transactionId contained in the request-confirm command pair may differ from the transactionId in the indication-response command pair. Refer to the specific subclause relating to the messageType for usage. The download protocol applies a different semantic to this field. Refer to clause 7 "U-N Download messages" for these semantics.

The transactionId is constructed of a 2 bit transactionId originator indication and a 30 bit transaction number. Figure 2-1 describes the format of the transactionId field:

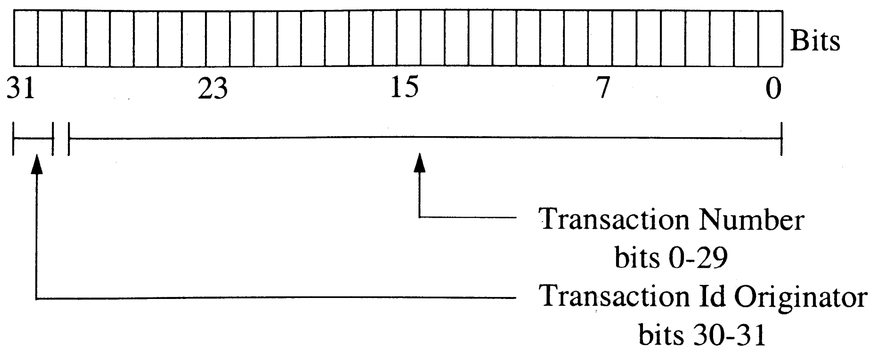

Bits
31
23
15
7
0
Transaction Number bits 0-29
Transaction Id Originator bits 30-31

Figure 2-1 Format of transactionId field

The coding of the transactionId originator indication is described in Table 2-3:

Table 2-3 MPEG-2 DSM-CC transactionId originator

originator	Description
0x00	transactionId is assigned by the Client.
0x01	transactionId is assigned by the Server.
0x02	transactionId is assigned by the Network.
0x03	ISO/IEC 13818-6 Reserved.

The **reserved** field is ISO/IEC 13818-6 reserved. This field shall be set to 0xFF.

The **adaptationLength** field indicates the total length in bytes of the adaptation header.

The **messageLength** field is used to indicate the total length in bytes of the message following this field. This length includes any adaptation headers indicated in the adaptationLength and the message payload indicated by the messageId field.

2.1 DSM-CC Adaptation Header Format

Adaptation headers are used to facilitate any network specific requirements. The general format of the DSM-CC Adaptation Header is defined in Table 2-4. Adaptation header use is optional.

Table 2-4 General format of the DSM-CC Adaptation Header

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

The **adaptationType** field is used to indicate the type of adaptation header. Table 2-5 defines the possible values of the adaptationType field.

Table 2-5 DSM-CC adaptationTypes

Adaptation Type	Description
0x00	ISO/IEC 13818-6 Reserved.
0x01	DSM-CC Conditional Access adaptation format.
0x02	DSM-CC User ID adaptation format.
0x02-0x7F	ISO/IEC 13818-6 Reserved.
0x80-0xFF	User Defined adaptation type.

The adaptationLength is included in the dsmccMessageHeader (see Table 2-1) and indicates the length of the adaptation header including the adaptationType field. The content of the adaptation data depends on the adaptationType.

2.1.1 DSM-CC Conditional Access Adaptation Format

Table 2-6 indicates the format of the conditional access adaptation fields.

Table 2-6 DSM-CC Conditional Access Adaptation Format

Syntax	Num. of Bytes
dsmccConditionalAccess() {	
reserved	1
caSystemId	2
conditionalAccessLength	2
for(i=0;i<conditionalAccessLength;i++) {	
conditionalAccessDataByte	1
}	
}	

The **reserved** field is included for alignment purposes and shall be set to 0xFF.

The **caSystemId** field specifies the type of conditional access system being used, as defined by the corresponding **CA_system_ID** field in ISO/IEC 13818-1, "MPEG2 Systems".

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.

2.1.2 DSM-CC User ID Adaptation Format

Table 2-7 indicates the format of the User ID adaptation fields.

Table 2-7 DSM-CC User ID Adaptation Format

Syntax	Num. of Bytes
dsmccUserId() {	
reserved	1
userId	20
}	

The **reserved** field is included for alignment purposes and shall be set to 0xFF.

The **userId** field identifies the address of the User which sends a request or response message, or which will receive an indication or confirm message. This field contains either the clientId or serverId fields, which are defined in clause 4, depending upon context.

3. User-to-Network Configuration Messages

3.1 Overview and the General Message Format

The User-to-Network Configuration messages provide User devices (Clients or Servers) with the configuration parameters that are required for the device to operate on the Network. Implementation of the User-to-Network Configuration messages is not required if a particular Network implementation uses some other means to provide these parameters.

The User-to-Network Configuration messages shall be only used to transfer configuration parameters to a User device. If large amounts of data are to be transferred to a User device, the U-N Download protocol described in clause 7 may be used for this purpose. The User-to-Network Configuration parameters may be used to pass the information needed to perform the download function.

The User-to-Network Configuration messages use the DSM-CC Message Format defined in clause 2. The dsmccType in the message header shall be set to 0x01 for User-to-Network Configuration messages. Table 3-1 defines the User-to-Network Configuration Message format. This format is called the unConfigurationMessage().

Table 3-1 General Format of DSM-CC User-to-Network Configuration Message

Syntax
unConfigurationMessage () {
dsmccMessageHeader()
MessagePayload()
}

The **dsmccMessageHeader** is defined in clause 2 “DSM-CC Message Header”.

The **MessagePayload** is constructed from data fields and differs in structure depending on the function of the particular message as defined by the messageId. subclause 3.3 defines the DSM-CC User-to-Network Configuration Messages.

3.2 User-to-Network configuration parameters

There are three sections of configuration parameters which are contained in the User-to-Network Configuration messages:

- DSM-CC specific configuration parameters
- Network specific configuration parameters
- User defined configuration parameters

3.2.1 DSM-CC specific configuration parameters

The DSM-CC specific configuration messages are used to define parameters which are used by the User-to-Network Messages. Table 3-2 defines the format of the dsmccConfigurationParameters section of the User-to-Network Configuration messages.

Table 3-2 Format of dsmccConfigurationParameters section

Syntax	Num. of Bytes
dsmccConfigurationParameters(){	
messageTimer	4
sessionInProgressTimer	4
messageRetryCount	1
sessionIdAssignor	1
resourceIdAssignor	1
maximumForwardCount	1
}	

The **messageTimer** field contains the value that should be used by the User's state machine in the messageTimer parameter. This value is represented in milliseconds. This value is used in the User-to-Network Session messages to set the tMsg timer.

The **sessionInProgressTimer** field contains the value that should be used by the User's state machine in the sessionInProgressTimer parameter. This value is represented in milliseconds. This value is used in the User-to-Network Session messages to set the tSIP timer. If this value is set to 0, this is an indication that Session In Progress messages are not used.

The **messageRetryCount** field contains the value that should be used by the User's state machine in the messageRetryCount parameter.

The **sessionIdAssignor** field indicates whether the sessionId is assigned by the User requesting the session or by the Network. A value of 0 indicates that the Network will assign the sessionId. A value of 1 indicates that the User shall assign the sessionId.

The **resourceIdAssignor** field indicates whether the resourceId is assigned by the User requesting the resource or by the Network. A value of 0 indicates that the Network will assign the resourceId. A value of 1 indicates that the User shall assign the resourceId.

The **maximumForwardCount** field indicates the maximum number of times that a session request may be forwarded before it is rejected. If this count is set to 0, this is an indication that session requests shall not be forwarded.

3.2.2 Network specific configuration parameters

Table 3-3 defines the format of the networkConfigurationParameters section of the User-to-Network Configuration messages. This table includes specified network-related parameters, such as the OSI NSAP addresses, that are used by the DSM-CC Session messages. Other parameters which are specific to the particular network implementation may also be included in this section. These other parameters are outside of the scope of this part of ISO/IEC 13818.

Table 3-3 Format of networkConfigurationParameters section

Syntax	Num. of Bytes
networkConfigurationParameters(){	
userId	20
primaryServerId	20
networkParameterLength	2
for(i=0;i<networkParameterLength;i++) {	
networkParameterDataByte	1
}	
}	

The **userId** field contains the OSI NSAP that will be used to uniquely identify the User on the Network. In the case of a Client, the userId becomes the Client's clientId used in clause 4 "User-to-Network Session Messages". Similarly, the userId becomes the Server's serverId when assigned to a Server by the U-N Configuration protocol.

The **primaryServerId** field contains the OSI NSAP that will be used to uniquely identify the address of the primary server on the Network. In the case of the Client device, this may be the address of the Server which supplies the initial download, applications, etc. to the User.

The **networkParameterLength** field defines the total number of networkParameterDataBytes.

The **networkParameterDataByte** field defines the configuration parameters specific to the network implementation. The content of this field is outside of the scope of this part of ISO/IEC 13818.

3.2.3 User defined configuration parameters

Table 3-4 defines the format of the userDefinedConfigurationParameters section of the User-to-Network Configuration messages. The use of these configuration parameters are outside of the scope of this part of ISO/IEC 13818.

Table 3-4 Format of userDefinedConfigurationParameters section

Syntax	Num. of Bytes
userDefinedConfigurationParameters(){	
userDefinedParameterLength	2
for(i=0;i<userDefinedParameterLength;i++) {	
userDefinedParameterDataByte	1
}	
}	

The **userDefinedParameterLength** field defines the total number of userDefinedParameterDataBytes.

The **userDefinedParameterDataByte** field contains configuration information which is outside of the scope of this specification.

3.3 User to Network Configuration Messages

Table 3-5 lists the messages which have been defined for the User-to-Network Configuration Messages.

Table 3-5 DSM-CC U-N Configuration messageId's

messageId	Message Name	Description
0x0000	Reserved	ISO/IEC 13818-6 Reserved.
0x0001	UNConfigRequest	Sent from a User to the Network to request configuration from the network.
0x0002	UNConfigConfirm	Sent from the Network to the User in response to the UNConfigRequest.
0x0003	UNConfigIndication	Sent from the Network to the User to configure a User device.
0x0004	UNConfigResponse	Sent from a User to the Network in response to a UNConfigIndication message.
0x0005 - 0x7FFFF	Reserved	ISO/IEC 13818-6 Reserved.
0x8000 - 0xFFFF	User Defined	User Defined U-N Configuration message.

3.3.1 UNConfigRequest message definition

This message is sent from a User to the Network to request that the Network return the configuration parameters for that User. Table 3-6 defines the syntax of the UNConfigRequest message.

Table 3-6 DSM-CC UNConfigRequest message

Syntax	Num. of Bytes
UNConfigRequest(){	
dsmccMessageHeader()	
deviceId	6
reserved	2
compatibilityDescriptor()	
}	

The **deviceId** field is defined in Table 3-11. It is a globally unique number which defines a User. The Network uses this field to configure the User device.

The **compatibilityDescriptor** structure is defined in clause 6. This structure contains the current configuration parameters for the User device.

3.3.2 UNConfigConfirm message definition

This message is sent from the Network to the User in response to a UNConfigRequest message. Table 3-7 defines the syntax of the UNConfigConfirm message.

Table 3-7 DSM-CC UNConfigConfirm message

Syntax	Num. of Bytes
UNConfigConfirm(){	
dsmccMessageHeader()	
deviceId	6
response	2
dsmccConfigurationParameters()	
networkConfigurationParameters()	
userDefinedConfigurationParameters()	
}	

The **deviceId** field is defined in Table 3-11. It is a globally unique number which defines a User. The value of the deviceId in the UNConfigConfirm message is set by the Network to the value received from the User in the UNConfigRequest message.

The **response** field shall be set by the Network to indicate the status of the configuration request. Values for this field are defined in Table 3-12. If the value returned in this field is a value other than rspOk, then all data following the response field shall not be processed by the User.

The **dsmccConfigurationParameters** contain parameters which are specific to DSM-CC User-to-Network. These parameters are defined in subclause 3.2.1.

The **networkConfigurationParameters** contain parameters which are specific to a particular network implementation as well as some parameters which are common to all DSM-CC User-to-Network implementations. These parameters are defined in subclause 3.2.2

The **userDefinedConfigurationParameters** contains parameters which are outside of the scope of this specification. The format of this section is defined in subclause 3.2.3

3.3.3 UNConfigIndication message definition

This message is sent from the Network to a User device to configure the device. Table 3-8 defines the syntax of the UNConfigIndication message.

Table 3-8 DSM-CC UNConfigIndication message

Syntax	Num. of Bytes
UNConfigIndication(){	
dsmccMessageHeader()	
deviceId	**6**
reason	**2**
compatibilityDescriptor()	
dsmccConfigurationParameters()	
networkConfigurationParameters()	
userDefinedConfigurationParameters()	
}	

The **deviceId** field is defined in Table 3-10. It is a globally unique number which defines a User. The Network uses this field to configure the User device.

The **reason** field shall be set by the Network to indicate the reason that the configuration indication is being sent. Codes for this field are defined in Table 3-11. Reason code values are also provided to allow the Network to inform the User that a UNConfigResponse message is not to be sent in cases where the Network does not want the User to reply (e.g., a one-way broadcast scenario).

The **compatibilityDescriptor** structure is defined in clause 6. This structure indicates the devices to which the UNConfigIndication message applies.

The **dsmccConfigurationParameters** contain parameters which are specific to DSM-CC User-to-Network protocols. These parameters are defined in subclause 3.2.1.

The **networkConfigurationParameters** contain parameters which are specific to a particular network implementation as well as some parameters which are common to all DSM-CC User-to-Network implementations. These parameters are defined in subclause 3.2.2.

The **userDefinedConfigurationParameters** contains configuration parameters which are outside of the scope of this specification. The format of this section is defined in subclause 3.2.3.

3.3.4 UNConfigResponse message definition

This message is sent from the User to the Network in response to a UNConfigIndication message. Table 3-9 defines the syntax of the UNConfigResponse message.

Table 3-9 DSM-CC UNConfigResponse message

Syntax	Num. of Bytes
UNConfigResponse(){	
dsmccMessageHeader()	
userId	**20**
response	**2**
reserved	**2**
compatibilityDescriptor()	
}	

The **userId** field contains the OSI NSAP that was assigned to the User by the UNConfigIndication message. When the UNConfigResponse message is received from a Client, the userId field contains the same value as the clientId. Similarly, when received from a Server, the userId field is the same value as the serverId.

The **response** field shall be set to indicate the User's response to the UNConfigIndication message. Codes for this field are defined in Table 3-12.

The **compatibilityDescriptor** structure is defined in clause 6. This structure contains the current configuration parameters for the User device.

3.4 User-to-Network Configuration Message Field Data Types

Table 3-10 defines the data fields used in the User-to-Network Configuration Messages.

Table 3-10 User-to-Network Configuration Message Field Data Types

Field Name	**Length (Bytes)**	**Range**	**Description**
deviceId	6	0x0000000000 - 0xFFFFFFFFFFFF	This field is used to identify a network device. This value shall be unique on the Network.
Reason	2	0x0000 - 0xFFFF	This field indicates the reason that a configuration message is being sent. Table 3-11 defines the possible values for this field.
Response	2	0x0000 - 0xFFFF	This field indicates the response to a Configuration message. Table 3-12 defines the possible values for this field.
UserId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a User. This address must be a specific address or be able to be resolved to a specific address by the Network.

3.5 User Initiated UNConfigRequest message Sequence

The User device must know its address and the address of the Network in order to operate on the Network. To obtain these addresses and other configuration parameters, a User device sends a UNConfigRequest message to the Network. The User must send this message over the network using a pre-defined mechanism provided by the Network. This mechanism exists at a lower protocol layer and is outside the scope of DSM-CC.

When the Network device that processes U-N Configuration messages receives a configuration request from a User device, it determines the appropriate configuration parameters for that device and sends those parameters to the User device in the UNConfigConfirm message with the response field set to indicate that the configuration request was accepted. If the Network cannot configure the User device or denies the UNConfigRequest, it shall set the response field in the message to indicate that the configuration request failed.

Figure 3-1 illustrates the sequence of events that occur for a User device to request its address and configuration information from the Network using the User-to-Network Configuration protocol. The Network optionally uses the external directory service to obtain address, configuration and authentication information about the User device.

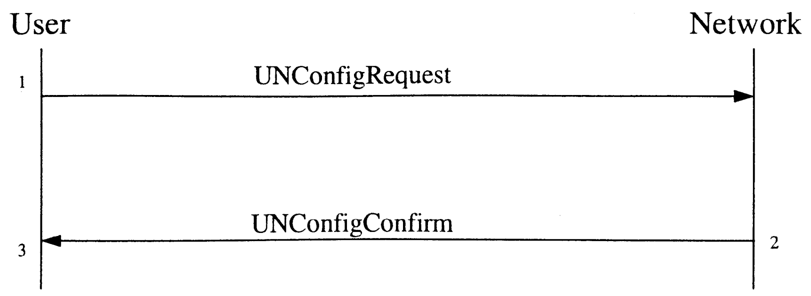


Figure 3-1 Sequence of events for User initiated UNConfigRequest

3.6 Network Initiated UNConfigIndication message Sequence

To configure a User device, the Network sends a UNConfigIndication message to that User. Since the User-to-Network Configuration sequence may be used to configure a device which does not yet know its network address, the network shall send the indication over the network using a pre-defined mechanism provided by the Network. This mechanism exists at a lower protocol layer and is outside the scope of DSM-CC. The Network may also send the indication to a specific User address to re-configure the User device if the device has been previously configured. The reason field in the indication shall be set to indicate the reason that the device is being configured.

When the User device receives a configuration indication from a Network, it updates the appropriate configuration parameters and sends a the UNConfigResponse message with the response field set to indicate that the configuration request was accepted. If the User does not accept the configuration indication, it shall set the response field in the message to indicate that the configuration indication was rejected.

Figure 3-2 illustrates the sequence of events that occur for a Network initiated configuration of a User device.

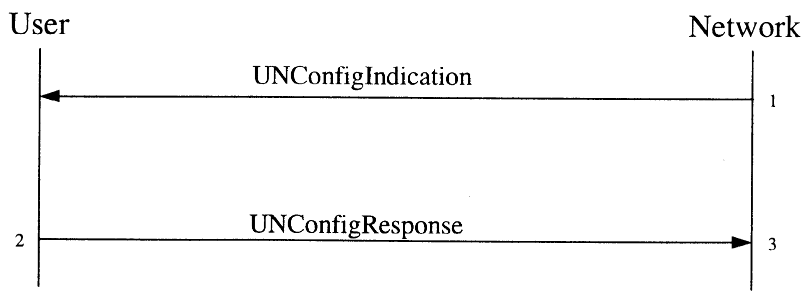


Figure 3-2 Sequence of events for Network initiated UNConfigIndication

3.7 Broadcasting of UNConfigIndication messages

In a situation where the User device is accessed using a broadcast mechanism, the User-to-Network configuration messages may be sent to the User device over the broadcast link in which many other Users are also listening. To configure such a User device, the Network sends a UNConfigIndication message to the User using a pre-defined broadcast mechanism provided by the Network. In this case, it is possible that there is no return path from the User to the Network, in which case the UNConfigIndication message cannot be confirmed. In this case, the Network may choose to send the UNConfigIndication message periodically to ensure that devices receive the configuration information. The reason code in the UNConfigIndication message can also be set by the Network to indicate that a UNConfigResponse message is not to be sent by the User.

When the User device receives a configuration indication from a Network that matches its deviceId, it updates the appropriate configuration parameters.

Figure 3-3 illustrates the sequence of events that occur when configuration is broadcast.

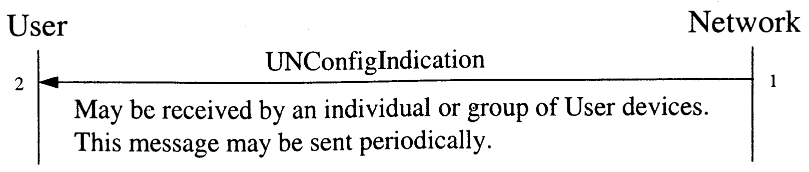

User
Network
UNConfigIndication
2
1
May be received by an individual or group of User devices.
This message may be sent periodically.

Figure 3-3 Sequence of events for Network initiated broadcast UNConfigIndication

3.8 Mixed User/Network Initiated Configuration Sequences

In some network implementations, it is possible for the User to initiate a UNConfigRequest message at the same time the Network initiates a UNConfigIndication message. For example, this could occur if a Client is attempting to recover from a power failure at the same time the Network is attempting to update a configuration parameter of the Client. In such cases, the actions taken by the User and Network are as follows:

User: When the User receives a UNConfigIndication message from the Network while it has an outstanding UNConfigRequest, the User shall process the UNConfigIndication message normally and reply to the Network with a UNConfigResponse message if a reply has been requested by the Network. When the User subsequently receives the UNConfigConfirm message from the Network in response to its original UNConfigRequest message, the User shall process the UNConfigConfirm message normally and again configure itself according to the parameters received from the Network.

Network: When the Network receives a UNConfigRequest message from the User while it has an outstanding UNConfigIndication request, the Network shall process the UNConfigRequest message normally and send a UNConfigConfirm message to the User containing the requested configuration parameters. When the Network subsequently receives a UNConfigResponse message from the User in response to its original UNConfigIndication request, the Network shall process the received message normally and note that the User has been configured successfully.

3.9 User-to-Network Configuration Reason Codes

Table 3-11 defines the reason codes that are defined for use by the U-N Configuration messages:

Table 3-11 User-to-Network Configuration reason codes

Response	Value	Description
rsnNormal	0x0000	Indicates that this is a normal U-N Config Indication message and that the User shall send a reply.
RsnNoReply	0x0001	Indicates that the U-N Configuration Indication message is being sent unsolicited to a User or group of User and no reply is required from the Users.
Reserved	0x0002 - 0x7FFF	ISO/IEC 13818-6 reserved.
User defined	0x8000 - 0xFFFF	These response codes are defined by the User and are outside of the scope of this specification.

3.10 User-to-Network Configuration Response Codes

Table 3-12 defines the response codes that are defined for use by the U-N Config messages:

Table 3-12 User-to-Network Configuration message response codes

Response	Value	Description
rspOK	0x0000	Indicates that the message was processed successfully.
RspNotAvailable	0x0001	Indicates that the U-N configuration server in the Network is not available at this time.
RspInvalid	0x0002	Indicates that the configuration request was rejected by

		the U-N configuration server in the Network.
RspRejected	0x0003	Indicates that the User rejected the UNConfigIndication message.
Reserved	0x0004 - 0x7FFF	ISO/IEC 13818-6 reserved.
User defined	0x8000 - 0xFFFF	These response codes are defined by the User and are outside of the scope of this specification.

4. User-to-Network Session Messages

4.1 Overview and the General Message Format

The User-to-Network (U-N) session messages are used for setting up, tearing down, and performing other operations which are session related. These messages are assumed to be part of a larger protocol stack and are designed to be carried on a lower layer transport protocol (e.g., UDP/IP, TCP/IP, AAL5, or Serial). Constraints on specific lower level protocols are given in clause 9 of this part of ISO/IEC 13818.

All U-N session messages are sent between a User (either a Client or a Server) and the Network. This clause describes which messages are available, the format of these messages, scenarios describing how these messages are used, and the use of Resource Descriptors which define the Network resources allocated to a session.

The syntax of these messages is extensible beyond those defined in this part of ISO/IEC 13818. Additional messages, resource descriptors used within those messages, and resource data elements which make up those resource descriptors may all be defined for a specific implementation which is not covered by this part of ISO/IEC 13818. If any of the messages, scenarios, or resource descriptors defined in this part of ISO/IEC 13818 are used, then these shall be implemented exactly as defined in this part of ISO/IEC 13818.

All messages between the Network and Users have a common message format. Table 4-1 defines the User-to-Network Session Message format. This format is called the userNetworkSessionMessage().

Table 4-1 General Format of DSM-CC User-Network Session Message

Syntax

UserNetworkSessionMessage (){
    dsmccMessageHeader()
    MessagePayload()
}

The **dsmccMessageHeader** is defined in clause 2 of this part of ISO/IEC 13818.

The **MessagePayload** is constructed from resource descriptors and data fields and differs in structure depending on the function of the particular message. Subclause 4.2 defines the DSM-CC User-to-Network Session Messages.

4.2 Session Messages

This subclause defines the User-to-Network Session Messages. Each message is identified by a specific messageId which is encoded to indicate the class and direction of the message. The messageId is carried in the dsmccMessageHeader which is defined in clause 2. Figure 4-1 defines the encoding of the messageId fields used in User-to-Network session messages. Bit 0 is the least significant bit and bit 15 is the most significant bit.

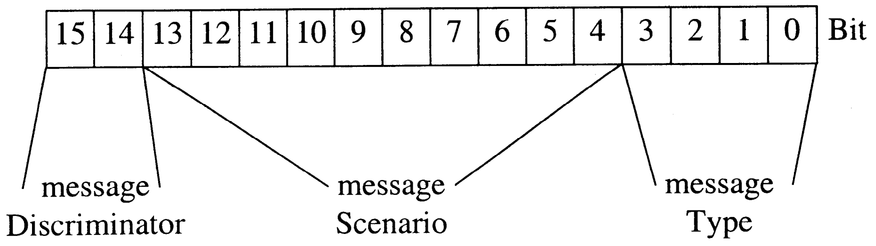


Figure 4-1 Format of DSM-CC session messageId

The **messageDiscriminator** bits are used to indicate if the message flow is between the Network and the Client or between the Network and the Server. Table 4-2 defines the possible values for the messageDiscriminator bits.

Table 4-2 messageDiscriminator bit values

messageDiscriminator	Message Flow
00	ISO/IEC 13818-6 Reserved.
01	Client and Network
10	Server and Network
11	ISO/IEC 138181-6 Reserved.

The **messageScenario** bits are used to indicate the message sequence that the message is in. Table 4-3 defines the possible values for the messageScenario bits.

Table 4-3 messageScenario bit values

messageScenario	Description
00 0000 0000	ISO/IEC 13818-6 Reserved.
00 0000 0001	Session Set-Up
00 0000 0010	Session Release
00 0000 0011	Add Resource
00 0000 0100	Delete Resource
00 0000 0101	Continuous Feed Session Set-Up
00 0000 0110	Status
00 0000 0111	Reset
00 0000 1000	Session Proceeding
00 0000 1001	Session Connect
00 0000 1010	Session Transfer
00 0000 1011	Session In Progress
00 0000 1100 – 01 1111 1111	ISO/IEC 13818-6 Reserved.
10 0000 0000 – 11 1111 1111	User Defined Message Scenario

Most of the control messages in this part of ISO/IEC 13818 use a confirmation mechanism. When a Network, Client or Server issues a request or indication message, the receiver of that message issues a definite response to that message. There are some cases of U-N session messages which do not use this mechanism. Specifically, these cases include:

- Session Proceeding Messages
- Session Connect Messages
- Session In Progress Messages

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

Request and Indication messages may contain a reason code which indicates the reason for the message. These reason codes are defined in Table 4-59. Response and Confirm messages may contain a response code which indicates the response to a Request or Indication message. These response codes are defined in Table 4-60.

The **messageType** bits are used to indicate the directionality of the message. Table 4-4 defines the possible values for the messageScenario bits.

Table 4-4 messageType bit values

messageType	Description
0000	Request Message. This indicates that the message is being sent from the User to the Network to begin a scenario.
0001	Confirm Message. This indicates that the message is being sent from the Network to the User in response to a Request message.
0010	Indication Message. This indicates that the message is being sent from the Network to the User.
0011	Response Message. This indicates that the message is being sent from the User to the Network in response to an Indication Message.
0100-1111	ISO/IEC 13818-6 Reserved.

Table 4-5 defines the messageId's which are used in the DSM-CC User-to-Network Session Messages.

Table 4-5 MPEG-2 DSM-CC U-N Messages

Client Command	Client messageId	Server messageId	Server Command
ISO/IEC 13818-6 reserved	0x0000 - 0x400f	0x8000 - 0x800f	ISO/IEC 13818-6 reserved
ClientSessionSetUpRequest	0x4010	0x8010	ISO/IEC 13818-6 reserved
ClientSessionSetUpConfirm	0x4011	0x8011	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4012	0x8012	ServerSessionSetUpIndication
ISO/IEC 13818-6 reserved	0x4013	0x8013	ServerSessionSetUpResponse
ISO/IEC 13818-6 reserved	0x4014 - 0x401f	0x8014 - 0x801f	ISO/IEC 13818-6 reserved
ClientSessionReleaseRequest	0x4020	0x8020	ServerSessionReleaseRequest
ClientSessionReleaseConfirm	0x4021	0x8021	ServerSessionReleaseConfirm
ClientSessionReleaseIndication	0x4022	0x8022	ServerSessionReleaseIndication
ClientSessionReleaseResponse	0x4023	0x8023	ServerSessionReleaseResponse
ISO/IEC 13818-6 reserved	0x4024 - 0x402f	0x8024 - 0x802f	ISO/IEC 13818-6 reserved

Client Command	Client messageId	Server messageId	Server Command
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
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
ISO/IEC 13818-6 reserved	0x4050	0x8050	ServerContinuousFeedSession Request
ISO/IEC 13818-6 reserved	0x4051	0x8051	ServerContinuousFeedSession Confirm
ISO/IEC 13818-6 reserved	0x4052 - 0x405f	0x8052 - 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
ClientResetRequest	0x4070	0x8070	ServerResetRequest
ClientResetConfirm	0x4071	0x8071	ServerResetConfirm
ClientResetIndication	0x4072	0x8072	ServerResetIndication
ClientResetResponse	0x4073	0x8073	ServerResetResponse
ISO/IEC 13818-6 reserved	0x4074 - 0x407f	0x8074 - 0x807f	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x4080 - 0x4081	0x8080 0x8081	ISO/IEC 13818-6 reserved
ClientSessionProceedingIndication	0x4082	0x8082	ServerSessionProceedingIndication
ISO/IEC 13818-6 reserved	0x4083 - 0x408f	0x8083 - 0x808f	ISO/IEC 13818-6 reserved

Client Command	Client messageId	Server messageId	Server Command
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 - 0x409f	0x8093 - 0x809f	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x40a0	0x80a0	ServerSessionTransferRequest
ISO/IEC 13818-6 reserved	0x40a1	0x80a1	ServerSessionTransferConfirm
ClientSessionTransferIndication	0x40a2	0x80a2	ServerSessionTransferIndication
ClientSessionTransferResponse	0x40a3	0x80a3	ServerSessionTransferResponse
ISO/IEC 13818-6 reserved	0x40a4 - 0x40af	0x80a4 - 0x80af	ISO/IEC 13818-6 reserved
ClientSessionInProgressRequest	0x40b0	0x80b0	ServerSessionInProgressRequest
ISO/IEC 13818-6 reserved	0x40b1 - 0x40bf	0x80b1 - 0x80bf	ISO/IEC 13818-6 reserved
ISO/IEC 13818-6 reserved	0x40c0 - 0x5fff	0x80c0 - 0x9fff	ISO/IEC 13818-6 reserved
User definable messageId's. Private use, outside of the scope of this part of ISO/IEC 13818.	0x6000 - 0x7fff	0xa000 - 0xffff	User definable messageId's. Private use, outside of the scope of this part of ISO/IEC 13818.

4.2.1 U-N Functional groups

In this subclause, the U-N session messages are divided into functional groups. A functional group is a grouping which allows a specific implementation not to implement every group. However, if a specific implementation implements operations of a group type, then the embodiment shall implement the complete syntax and semantics of the corresponding functional group.

4.2.1.1 U-N Core Group

The following U-N session messages belong to the core group of messages required for basic U-N operation:

Session Set-Up Group:
- ClientSessionSetUpRequest/Confirm
- ServerSessionSetUpIndication/Response

Session Release Group:
- ClientSessionReleaseRequest/Confirm/Indication/Response
- ServerSessionReleaseRequest/Confirm/Indication/Response

Add Resource Group:
- ClientAddResourceIndication/Response
- ServerAddResourceRequest/Confirm

Delete Resource Group:
- ClientDeleteResourceIndication/Response
- ServerDeleteResourceRequest/Confirm

Continuous Feed Session Set-Up Group:
- ServerContinuousFeedSessionRequest/Confirm

Status Group:
- ClientStatusRequest/Confirm/Indication/Response
- ServerStatusRequest/Confirm/Indication/Response

Reset Group:
- ClientResetRequest/Confirm/Indication/Response
- ServerResetRequest/Confirm/Indication/Response

The following message groups are optional within the core group. A particular implementation of DSM-CC may choose whether or not to implement these groups. The requirement to implement these functions is determined by using U-N Configuration messages or through a private agreement between the Network and the Users.

Session Proceeding Group:
- ClientSessionProceedingIndication
- ServerSessionProceedingIndication

> The above messages may be optionally sent by the Network; however, if sent, the User shall be required to receive and process these messages.

Session Connect Group:
- ClientConnectRequest
- ServerConnectIndication

> The ConnectRequest message may be optionally sent by a Client; however, if sent, the Network shall be required to receive and process this message, then send a ConnectIndication message to the Server. The Server shall be required to receive and process a ConnectionIndication message.

4.2.1.2 Extended Functional groups

The following functions are considered to be normative but optional. If a specific implementation implements these functions, then they shall be implemented as defined in this part of ISO/IEC 13818. The requirement to implement these functions is determined by using U- N Configuration messages or through a private agreement between the Network and the Users.

Session Transfer Group:
- ClientSessionTransferIndication/Response
- ServerSessionTransferRequest/Confirm/Indication/Response

Session In Progress Group:
- ClientSessionInProgressRequest
- ServerSessionInProgressRequest

4.2.2 Use of UserData() structure in session messages

U-N Session messages sent from a User to the SRM which result in a corresponding message to be sent from the SRM to another User may contain a UserData() field. DSM-CC User-to-Network does not make use of this data, but if it is present, the Network passes it on transparently to the Server or Client as the case may be. DSM-CC User-to-User clause does define data which is transported in the uuData portion of these fields. The UserData() also contains a section for including privateData, the content of which is outside of the scope of this part of ISO/IEC 13818. Table 4-6 defines the format of the UserData() structure which is transported in U-N session messages.

Table 4-6 DSM-CC U-N UserData format

Syntax	Num. of Bytes
UserData(){	
uuDataLength	2
for(i=0; i<uuDataLength; i++) {	
uuDataByte	1
}	
privateDataLength	2
for(i=0; i<privateDataLength; i++) {	
privateDataByte	1
}	
}	

The **uuDataLength** field shall indicate the total number of uuDataBytes.

The **uuDataBytes** contain the uuData which is defined by the DSM-CC User-to-User clause of this part of ISO/IEC 13818. The total number of uuDataBytes shall be padded to a multiple of four bytes.

The **privateDataLength** field shall indicate the total number of privateDataBytes.

The **privateDataBytes** contain privateData. The format and usage of this data is outside of the scope of this part of ISO/IEC 13818. The total number of privateDataBytes shall be padded to a multiple of four bytes.

4.2.3 Use of Resources() structure in session messages

A Session consists of a relationship between the Network and one or more Users. At least one of the Users is acting in the role of a Server. Network resources are allocated to a session either by the Server or by requesting the allocation from the Network. In either case, the Network shall be aware of resources which are allocated to a session. Messages which are used to request resources from the Network or to inform the Network of resources which have been allocated by the Server contain a Resources() data structure. This structure contains a count and a list of resource descriptors which are defined in subclause 4.7. Table 4-7 defines the format of the Resources() data structure:

Table 4-7 DSM-CC U-N Resources format

Syntax	Num. of Bytes
Resources(){	
resourceDescriptorCount	2
for(i=0;i<resourceDescriptorCount;i++) {	
ResourceDescriptor()	
}	
}	

The **resourceDescriptorCount** field shall be set to indicate the total number of ResourceDescriptor() structures which are included in the list.

The **ResourceDescriptor()** structure shall define the type, status, and values of a resource which is being requested or which has been assigned to a session. Refer to subclause 4.7 for resource descriptor definitions.

4.2.4 Session Set-Up group message definitions

4.2.4.1 ClientSessionSetUpRequest

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 shall send 0 or more ClientSessionProceedingIndication messages. Table 4-8 defines the syntax of the ClientSessionSetUpRequest message.

Table 4-8 DSM-CC U-N ClientSessionSetUpRequest message

Syntax	Num. of Bytes
ClientSessionSetUpRequest(){	
dsmccMessageHeader()	
sessionId	**10**
reserved	**2**
clientId	**20**
serverId	**20**
UserData()	
}	

The **sessionId** is used to identify a session throughout its life cycle. If the Network configuration indicates that the User which is the originator of the command sequence is responsible for generating the sessionId, this field shall be generated by the Client. If the Network configuration indicates that the Network is responsible for generating the sessionId, this field shall be set to all 0's and the Network shall assign the sessionId in the ClientSessionSetUpConfirm message. Both the Network and the Client shall use the identical sessionId in all messages which refer to this session.

The **clientId** field shall be set by the Client and shall contain a value which uniquely identifies 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 attempting to establish a session.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.4.2 ClientSessionSetUpConfirm

This message is sent from the Network to a Client in response to a ClientSessionSetUpRequest message. Table 4-9 defines the syntax of the ClientSessionSetUpConfirm message.

Table 4-9 DSM-CC U-N ClientSessionSetUpConfirm message

Syntax	Num. of Bytes
ClientSessionSetUpConfirm(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
serverId	**20**
Resources()	
UserData()	
}	

The **sessionId** is used to identify a session throughout its life cycle. If the Network configuration indicates that the User is responsible for assigning the sessionId, the Network shall set this 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, 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 Client that the requested service has been established.

The **serverId** field shall be set by the Network to the value of the serverId field which was received in the ServerSessionSetUpResponse.

The **Resources()** structure defines 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. Refer to subclause 4.2.3 for more information on Resources.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.4.3 ServerSessionSetUpIndication

This message is sent from the Network to a Server to establish a session which was requested by a Client. Table 4-10 defines the syntax of the ServerSessionSetUpIndication message.

Table 4-10 DSM-CC U-N ServerSessionSetUpIndication message

Syntax	Num. of Bytes
ServerSessionSetUpIndication(){	
dsmccMessageHeader()	
sessionId	**10**
reserved	**2**
clientId	**20**
serverId	**20**
forwardCount	**2**
for(i=0;i<forwardCount;i++){	
forwardServerId	**20**
}	
UserData()	
}	

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.

The **forwardCount** field is used to indicate the number of session forwards that have occurred.

The **forwardServerId** is a list of serverIds of servers that have forwarded this session. There shall be exactly forwardCount number of forwardServerIds in the list.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.4.4 ServerSessionSetUpResponse

This message is sent from a Server to the Network in response to a ServerSessionSetUpIndication message. Table 4-11 defines the syntax of the ServerSessionSetUpResponse message.

Table 4-11 DSM-CC U-N ServerSessionSetUpResponse message

Syntax	Num. of Bytes
ServerSessionSetUpResponse(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
serverId	**20**
nextServerId	**20**
Resources()	
UserData()	
}	

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.

The **serverId** field shall be set by the Server to the value of the serverId field which was received in the ServerSessionSetUpIndication message.

The **nextServerId** specifies the serverId of the Server to which this session is to be forwarded if the response field indicates that the request is to be forwarded to another Server. If the response field does not indicate that this message is to be forwarded, this field shall be set to 0.

The **Resources()** structure shall indicate all resources which are assigned to this session. This does not include any resources which were previously negotiated with the Network. It includes only resources which were allocated to the session by the Server.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data. This data is supplied by the Server.

4.2.5 Session Release group message definitions

4.2.5.1 ClientSessionReleaseRequest

This message is sent from a Client to the Network to request that a session be torn-down. The Network responds with a ClientSessionReleaseConfirm message. Before sending the ClientSessionReleaseConfirm message, the Network shall also release the session between the Network and the Server. Table 4-12 defines the syntax of the ClientSessionReleaseRequest message.

Table 4-12 DSM-CC U-N ClientSessionReleaseRequest message

Syntax	Num. of Bytes
ClientSessionReleaseRequest(){	
dsmccMessageHeader()	
sessionId	**10**
reason	**2**
UserData()	
}	

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 **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.5.2 ClientSessionReleaseConfirm

This message is sent from the Network to a Client in response to a ClientSessionReleaseRequest message. Table 4-13 defines the syntax of the ClientSessionReleaseConfirm message.

Table 4-13 DSM-CC U-N ClientSessionReleaseConfirm message

Syntax	Num. of Bytes
ClientSessionReleaseConfirm(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
UserData()	
}	

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

The **response** field shall be set by the Network to indicate the status of the session release request.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.5.3 ClientSessionReleaseIndication

This message is sent from the Network to a Client initiate a session release. Table 4-14 defines the syntax of the ClientSessionReleaseIndication message.

Table 4-14 DSM-CC U-N ClientSessionReleaseIndication message

Syntax	Num. of Bytes
ClientSessionReleaseIndication(){	
dsmccMessageHeader()	
sessionId	**10**
reason	**2**
UserData()	
}	

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 release was initiated by the Server, this field shall be identical to the reason field which was received in the ServerSessionReleaseRequest message.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.5.4 ClientSessionReleaseResponse

This message is sent from a Client to the Network in response to a ClientSessionReleaseIndication message to indicate the Clients response to the request. Table 4-15 defines the syntax of the ClientSessionReleaseResponse message.

Table 4-15 DSM-CC U-N ClientSessionReleaseResponse message

Syntax	Num. of Bytes
ClientSessionReleaseResponse(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
UserData()	
}	

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

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

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.5.5 ServerSessionReleaseRequest

This message is sent from a Server to the Network to request that a session be torn-down. The Network responds with a ServerSessionReleaseConfirm message. Before sending the ServerSessionReleaseConfirm message, the Network shall also release the session between the Network and the Client. Table 4-16 defines the syntax of the ServerSessionReleaseRequest message.

Table 4-16 DSM-CC U-N ServerSessionReleaseRequest message

Syntax	Num. of Bytes
ServerSessionReleaseRequest(){	
dsmccMessageHeader()	
sessionId	**10**
reason	**2**
UserData()	
}	

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 **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.5.6 ServerSessionReleaseConfirm

This message is sent from the Network to a Server in response to a ServerSessionReleaseRequest message. Table 4-17 defines the syntax of the ServerSessionReleaseConfirm message.

Table 4-17 DSM-CC U-N ServerSessionReleaseConfirm message

Syntax	Num. of Bytes
ServerSessionReleaseConfirm(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
UserData()	
}	

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

The **response** field shall be set by the Network to indicate the status of the session release request.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.5.7 ServerSessionReleaseIndication

This message is sent from the Network to a Server to initiate a session release which was requested by the Client. Table 4-18 defines the syntax of the ServerSessionReleaseIndication message.

Table 4-18 DSM-CC U-N ServerSessionReleaseIndication message

Syntax	Num. of Bytes
ServerSessionReleaseIndication(){	
dsmccMessageHeader()	
sessionId	**10**
reason	**2**
UserData()	
}	

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 release was initiated by the Client, this field shall be identical to the reason field which was received in the ClientSessionReleaseRequest message.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.5.8 ServerSessionReleaseResponse

This message is sent from a Server to the Network in response to a ServerSessionReleaseIndication message. Table 4-19 defines the syntax of the ServerSessionReleaseResponse message.

Table 4-19 DSM-CC U-N ServerSessionReleaseResponse message

Syntax	Num. of Bytes
ServerSessionReleaseResponse(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
UserData()	
}	

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

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

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.6 Add Resource group message definitions

4.2.6.1 ClientAddResourceIndication

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 4-20 defines the syntax of the ClientAddResourceIndication message.

Table 4-20 DSM-CC U-N ClientAddResourceIndication message

Syntax	Num. of Bytes
ClientAddResourceIndication(){	
dsmccMessageHeader()	
sessionId	**10**
Resources()	
UserData()	
}	

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

The **Resources()** structure defines any new resources which have been added to the Client view 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.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data. This field shall be passed from the Server.

4.2.6.2 ClientAddResourceResponse

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 4-21 defines the syntax of the ClientAddResourceResponse message.

Table 4-21 DSM-CC U-N ClientAddResourceResponse message

Syntax	Num. of Bytes
ClientAddResourceResponse(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
Resources()	
UserData()	
}	

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.

The **Resources()** structure contains the client view of the new resource descriptors which were added to the Session.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data. This field shall be set by the Client.

4.2.6.3 ServerAddResourceRequest

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 4-22 defines the syntax of the ServerAddResourceRequest message.

Table 4-22 DSM-CC U-N ServerAddResourceRequest message

Syntax	Num. of Bytes
ServerAddResourceRequest(){	
dsmccMessageHeader()	
sessionId	**10**
Resources()	
UserData()	
}	

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

The **Resources()** structure indicates the resources that the server is requesting that the Network add to the session or resources which the Server has allocated to the session.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data. If the sessionId is for a Continuous Feed Session, there shall be no uuData supplied within UserData().

4.2.6.4 ServerAddResourceConfirm

This message is sent from the Network to a Server in response to a ServerAddResourceRequest message. Table 4-23 defines the syntax of the ServerAddResourceConfirm message.

Table 4-23 DSM-CC U-N ServerAddResourceConfirm message

Syntax	Num. of Bytes
ServerAddResourceConfirm(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
Resources()	
UserData()	
}	

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.

The **Resources()** structure shall be set by the Network to values which were assigned to the requested resources.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data. This data shall be passed from the Client. If the sessionId is for a Continuous Feed Session, there shall be no uuData supplied within UserData().

4.2.7 Delete Resource group message definitions

4.2.7.1 ClientDeleteResourceIndication

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 4-24 defines the syntax of the ClientDeleteResourceIndication message.

Table 4-24 DSM-CC U-N ClientDeleteResourceIndication message

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

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 resourceNum in combination with the sessionId shall be used to identify a unique resource descriptor.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.7.2 ClientDeleteResourceResponse

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 4-25 defines the syntax of the ClientDeleteResourceResponse message.

Table 4-25 DSM-CC U-N ClientDeleteResourceResponse message

Syntax	Num. of Bytes
ClientDeleteResourceResponse(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
UserData()	
}	

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.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data. This data shall be set by the Client.

4.2.7.3 ServerDeleteResourceRequest

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 4-26 defines the syntax of the ServerDeleteResourceRequest message.

Table 4-26 DSM-CC U-N ServerDeleteResourceRequest message

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

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 shall be used to identify a unique resource descriptor.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.7.4 ServerDeleteResourceConfirm

This message is sent from the Network to a Server in response to a ServerDeleteResourceRequest message. Table 4-27 defines the syntax of the ServerDeleteResourceConfirm message.

Table 4-27 DSM-CC U-N ServerDeleteResourceConfirm message

Syntax	Num. of Bytes
ServerDeleteResourceConfirm(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
UserData()	
}	

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 **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.8 Continuous Feed Session group message definitions

4.2.8.1 ServerContinuousFeedSessionRequest

This message is sent from a Server to the Network to request that a continuous feed session (CFS) be established. A CFS does not connect to a specific client. Instead, the resources of a CFS may be shared among multiple clients. The Network responds with a ServerContinuousFeedSessionConfirm message. Table 4-28 defines the syntax of the ServerContinuousFeedSessionRequest message.

Table 4-28 DSM-CC U-N ServerContinuousFeedSessionRequest message

Syntax	Num. of Bytes
ServerContinuousFeedSessionRequest(){	
dsmccMessageHeader()	
sessionId	**10**
reserved	**2**
serverId	**20**
Resources()	
}	

sessionId 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 ServerContinuousFeedSessionConfirm message.

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

The **Resources()** structure identifies any resources required for the session. These resources may be either resources added by the Server or resources which are being requested from the Network.

4.2.8.2 ServerContinuousFeedSessionConfirm

This message is sent from the Network to a Server in response to a ServerContinuousFeedSessionRequest message. Table 4-29 defines the syntax of the ServerContinuousFeedSessionConfirm message.

Table 4-29 DSM-CC U-N ServerContinuousFeedSessionConfirm message

Syntax	Num. of Bytes
ServerContinuousFeedSessionConfirm(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
Resources()	
}	

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.

The **Resources()** structure is used to confirm Server assigned resources and to notify the Server of the status and value of any resources which were requested from the Network.

4.2.9 Status group message definitions

4.2.9.1 ClientStatusRequest

This message is sent from a Client to the Network to request a status message. Table 4-30 defines the syntax of the ClientStatusRequest message.

Table 4-30 DSM-CC U-N ClientStatusRequest message

Syntax	Num. of Bytes
ClientStatusRequest(){	
dsmccMessageHeader()	
reason	**2**
clientId	**20**
statusType	**2**
statusCount	**2**
for(i=0;i<statusCount;i++) {	
statusByte	**1**
}	
}	

The **reason** field shall be set by the client to indicate the reason that the status is being requested.

The **clientId** field shall be set by the client to its own id to identify itself to the Network.

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.

4.2.9.2 ClientStatusConfirm

This message is sent from the Network to a Client in response to a ClientStatusRequest message. Table 4-31 defines the syntax of the ClientStatusConfirm message.

Table 4-31 DSM-CC U-N ClientStatusConfirm message

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

The **response** field shall be set by the Network to indicate if the Network accepted the status request.

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.

4.2.9.3 ClientStatusIndication

This message is sent from the Network to a Client to request a status message from the Client. Table 4-32 defines the syntax of the ClientStatusIndication message.

Table 4-32 DSM-CC U-N ClientStatusIndication message

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

The **reason** field shall be set by the Network to indicate the reason that the status is being requested.

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.

4.2.9.4 ClientStatusResponse

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 4-33 defines the syntax of the ClientStatusResponse message.

Table 4-33 DSM-CC U-N ClientStatusResponse message

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

The **response** field shall be set by the Client to indicate if the Client accepted the status request.

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

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

4.2.9.5 ServerStatusRequest

This message is sent from a Server to the Network to request a status message. Table 4-34 defines the syntax of the ServerStatusRequest message.

Table 4-34 DSM-CC U-N ServerStatusRequest message

Syntax	Num. of Bytes
ServerStatusRequest(){	
dsmccMessageHeader()	
reason	2
serverId	20
statusType	2
statusCount	2
for(i=0;i<statusCount;i++) {	
statusByte	1
}	
}	

The **reason** field shall be set by the server to indicate the reason that the status is being requested.

The **serverId** field shall be set by the server to its own id to identify itself to the Network.

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.

4.2.9.6 ServerStatusConfirm

This message is sent from the Network to a Server in response to a ServerStatusRequest message. Table 4-35 defines the syntax of the ServerStatusConfirm message.

Table 4-35 DSM-CC U-N ServerStatusConfirm message

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

The **response** field shall be set by the Network to indicate if the Network accepted the status request.

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.

4.2.9.7 ServerStatusIndication

This message is sent from the Network to a Server to request a status message from the Server. Table 4-36 defines the syntax of the ServerStatusIndication message.

Table 4-36 DSM-CC U-N ServerStatusIndication message

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

The **reason** field shall be set by the Network to indicate the reason that the status is being requested.

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.

4.2.9.8 ServerStatusResponse

This message is sent from a Server to the Network in response to a ServerStatusIndication. Table 4-37 defines the syntax of the ServerStatusResponse message.

Table 4-37 DSM-CC U-N ServerStatusResponse message

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

The **response** field shall be set by the Server to indicate if the Server accepted the status request.

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

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

4.2.10 Reset group message definitions

4.2.10.1 ClientResetRequest

This message will be sent from the Client to the Network to initiate clearing of all sessions active for a Client. Table 4-38 defines the syntax of the ClientResetRequest message.

Table 4-38 DSM-CC U-N ClientResetRequest message

Syntax	Num. of Bytes
ClientResetRequest(){	
dsmccMessageHeader()	
clientId	20
reason	2
}	

The **clientId** field shall identify the Client which is being reset.

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

4.2.10.2 ClientResetConfirm

This message is sent from the Network to the Client in response to the ClientResetRequest message. Table 4-39 defines the syntax of the ClientResetConfirm message.

Table 4-39 DSM-CC U-N ClientResetConfirm message

Syntax	Num. of Bytes
ClientResetConfirm(){	
dsmccMessageHeader()	
clientId	20
response	2
}	

The **clientId** field identifies the Client which is being reset.

The **response** field is used to indicate the result of the reset operation.

4.2.10.3 ClientResetIndication

This message will be sent from the Network to the Client to initiate clearing of all sessions active for a Client. Table 4-40 defines the syntax of the ClientResetIndication message.

Table 4-40 DSM-CC U-N ClientResetIndication message

Syntax	Num. of Bytes
ClientResetIndication(){	
dsmccMessageHeader()	
clientId	**20**
reason	**2**
}	

The **clientId** field shall be set by the Network to identify the Client which is being reset.

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

4.2.10.4 ClientResetResponse

This message is sent from the Client to the Network in response to the ClientResetIndication message. Table 4-41 defines the syntax of the ClientResetResponse message.

Table 4-41 DSM-CC U-N ClientResetResponse message

Syntax	Num. of Bytes
ClientResetResponse(){	
dsmccMessageHeader()	
clientId	**20**
response	**2**
}	

The **clientId** field identifies the Client which is being reset.

The **response** field is used to indicate the result of the reset operation.

4.2.10.5 ServerResetRequest

This message will be sent from the Server to the Network to initiate clearing of all sessions. Table 4-42 defines the syntax of the ServerResetRequest message.

Table 4-42 DSM-CC U-N ServerResetRequest message

Syntax	Num. of Bytes
ServerResetRequest(){	
dsmccMessageHeader()	
serverId	**20**
reason	**2**
}	

The **serverId** field identifies the Server which is being reset.

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

4.2.10.6 ServerResetConfirm

This message will be sent from the Network to the Server in response to the ServerResetRequest message. Table 4-43 defines the syntax of the ServerResetConfirm message.

Table 4-43 DSM-CC U-N ServerResetConfirm message

Syntax	Num. of Bytes
ServerSessionResetConfirm(){	
dsmccMessageHeader()	
serverId	**20**
response	**2**
}	

The **serverId** field identifies the Server which is being reset.

The **response** field is used to indicate the result of the reset operation.

4.2.10.7 ServerResetIndication

This message will be sent from the Network to the Server to initiate clearing of all sessions. Table 4-44 defines the syntax of the ServerResetConfirm message.

Table 4-44 DSM-CC U-N ServerSessionResetIndication message

Syntax	Num. of Bytes
ServerSessionResetIndication(){	
dsmccMessageHeader()	
serverId	**20**
reason	**2**
}	

The **serverId** field identifies the Server which is being reset.

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

4.2.10.8 ServerResetResponse

This message will be sent from the Server to the Network in response to the ServerResetIndication message. Table 4-45 defines the syntax of the ServerResetResponse message.

Table 4-45 DSM-CC U-N ServerResetResponse message

Syntax	Num. of Bytes
ServerSessionResetResponse(){	
dsmccMessageHeader()	
serverId	**20**
response	**2**
}	

The **serverId** field identifies the Server which is being reset.

The **response** field is used to indicate the result of the reset operation.

4.2.11 Session Proceeding group message definitions

4.2.11.1 ClientSessionProceedingIndication

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 tMsg to its initial value upon receipt of this message. Table 4-46 defines the syntax of the ClientSessionProceedingIndication message.

Table 4-46 DSM-CC U-N ClientSessionProceedingIndication message

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

The **sessionId** field shall be set by the Network to indicate the session which the proceeding message is being sent for. If the network is assigning the sessionId, this field may contain either the assigned sessionId or may be set to 0 to indicate that the Network has not yet assigned a sessionId to the request.

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

4.2.11.2 ServerSessionProceedingIndication

This message is sent from the Network to a Server in response to a ServerContinuousFeedSessionRequest to inform the Server that the request is being processed. The Network may send this message 0 or more times before sending the session set-up confirm message. The Server shall reset timer tMsg to its initial value upon receipt of this message. Table 4-47 defines the syntax of the ServerSessionProceedingIndication message.

Table 4-47 DSM-CC U-N ServerSessionProceedingIndication message

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

The **sessionId** field shall be set by the Network to indicate the session which the proceeding message is being sent for. If the network is assigning the sessionId, this field may contain either the assigned sessionId or may be set to 0 to indicate that the Network has not yet assigned a sessionId to the request.

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

4.2.12 Connect group message definitions

4.2.12.1 ClientConnectRequest

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 4-48 defines the syntax of the ClientConnectRequest message.

Table 4-48 DSM-CC U-N ClientConnectRequest message

Syntax	Num. of Bytes
ClientConnectRequest(){	
dsmccMessageHeader()	
sessionId	**10**
UserData()	
}	

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

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data. This data shall be supplied by the Client.

4.2.12.2 ServerConnectIndication

This message 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 ClientConnectRequest message. Table 4-49 defines the syntax of the ServerConnectIndication message.

Table 4-49 DSM-CC U-N ServerConnectIndication message

Syntax	Num. of Bytes
ServerConnectIndication(){	
dsmccMessageHeader()	
sessionId	**10**
UserData()	
}	

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

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data. This data shall be supplied by the Client.

4.2.13 Session Transfer group message definitions

4.2.13.1 ClientSessionTransferIndication

This message is sent from the Network to a Client to indicate that a transfer has occurred. Table 4-50 defines the syntax of the ClientSessionTransferIndication message.

Table 4-50 DSM-CC U-N ClientSessionTransferIndication message

Syntax	Num. of Bytes
ClientSessionTransferIndication(){	
dsmccMessageHeader()	
sessionId	**10**
reserved	**2**
clientId	**20**
oldServerId	**20**
newServerId	**20**
Resources()	
UserData()	
}	

The **sessionId** identifies the session being transferred.

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 **oldServerId** field contains the serverId for the Server from which the session is being transferred.

The **newServerId** field contains the serverId for the Server to which the session is being transferred.

The **Resources()** structure defines 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.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.13.2 ClientSessionTransferResponse

This message is sent from a Client to a Network to response to a ClientSessionTransferIndication. Table 4-51 defines the syntax of the ClientSessionTransferResponse message.

Table 4-51 DSM-CC U-N ClientSessionTransferResponse message

Syntax	Num. of Bytes
ClientSessionTransferResponse(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
UserData()	
}	

The **sessionId** identifies the session being transferred.

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

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.13.3 ServerSessionTransferRequest

This message is sent from a Server to a Network to request to transfer a session. Table 4-52 defines the syntax of the ServerSessionTransferRequest message.

Table 4-52 DSM-CC U-N ServerSessionTransferRequest message

Syntax	Num. Of Bytes
ServerSessionTransferRequest(){	
dsmccMessageHeader()	
sessionId	10
reserved	2
destServerId	20
baseServerId	20
UserData()	
}	

The **sessionId** identifies the session being transferred.

The **destServerId** field contains the serverId for the Server to which the session is being transferred.

The **baseServerId** contains the first server involved in this session. It is the starting point for the session. This field is used in case the Client wants to transfer back to the start of the transfer stack.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.13.4 ServerSessionTransferConfirm

This message is sent from a Network to a Server to response to a ServerSessionTransferRequest. Table 4-53 defines the syntax of the ServerSessionTransferConfirm message.

Table 4-53 DSM-CC U-N ServerSessionTransferConfirm message

Syntax	Num. of Bytes
ServerSessionTransferConfirm(){	
dsmccMessageHeader()	
sessionId	10
response	2
UserData()	
}	

The **sessionId** identifies the session being transferred.

The **response** field shall be set by the Server to a value which indicates the Client's or SRM's response to the ServerSessionTransferRequest message.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.13.5 ServerSessionTransferIndication

This message is sent from the Network to a Server to indicate that a transfer is requested to that Server. Table 4-54 defines the syntax of the ServerSessionTransferIndication message.

Table 4-54 DSM-CC U-N ServerSessionTransferIndication message

Syntax	Num. of Bytes
ServerSessionTransferIndication(){	
dsmccMessageHeader()	
sessionId	**10**
reserved	**2**
clientId	**20**
srcServerId	**20**
baseServerId	**20**
Resources()	
UserData()	
}	

The **sessionId** identifies the session being transferred.

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 **srcServerId** field contains the serverId for the Server from which the session is being transferred.

The **baseServerId** contains the first server involved in this session. It is the starting point for the session. This field is used in case the Client wants to transfer back to the start of the transfer stack.

The **Resources()** structure defines any resources which are being transferred to the new Server.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.13.6 ServerSessionTransferResponse

This message is sent from a Server to a Network to response to a ServerSessionTransferIndication. Table 4-55 defines the syntax of the ServerStatusResponse message.

Table 4-55 DSM-CC U-N ServerSessionTransferResponse message

Syntax	Num. of Bytes
ServerSessionTransferResponse(){	
dsmccMessageHeader()	
sessionId	**10**
response	**2**
Resources()	
UserData()	
}	

The **sessionId** identifies the session being transferred.

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

The **Resources()** structure describes all resources which are active on the session. This includes transferred resources, new resources which were negotiated with the network, and resources which were added by the Server.

The **UserData()** structure contains uuData which is defined by the User-To-User clause of this part of ISO/IEC 13818 and privateData which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 4.2.2 for more information on this data.

4.2.14 Session In Progress group message definitions

4.2.14.1 ClientSessionInProgress

This message is sent from a Client to a Network periodically to inform the Network of the sessions which are active on the Client. The period that this message is sent, if ever, is determined by timer tSip which may be set from the sessionInProgressTimer field in the U-N Configuration protocol or some private means which is outside the scope of this part of ISO/IEC 13818. Table 4-56 defines the syntax of the ClientSessionInProgress message.

Table 4-56 DSM-CC U-N ClientSessionInProgress message

Syntax	Num. of Bytes
ClientSessionInProgress(){	
dsmccMessageHeader()	
sessionCount	2
for(i=0;i<sessionCount;i++) {	
sessionId	
}	10
}	

The **sessionCount** indicates the number of sessions which are active on the Client. This count shall equal the number of sessionId's included in the message.

The **sessionId** field indicates the session which is active on the Client. This field shall be repeated for each session which is active on the Client.

4.2.14.2 ServerSessionInProgress

This message is sent from a Server to a Network periodically to inform the Network of the sessions which are active on the Server. The period that this message is sent, if ever, is determined by timer tSip which may be set from the sessionInProgressTimer field in the U-N Configuration protocol or some private means which is outside the scope of this part of ISO/IEC 13818. Table 4-57 defines the syntax of the ServerSessionInProgress message.

Table 4-57 DSM-CC U-N ServerSessionInProgress message

Syntax	Num. of Bytes
ServerSessionInProgress(){	
dsmccMessageHeader()	
sessionCount	2
for(i=0;i<sessionCount;i++) {	
sessionId	
}	10
}	

The **sessionCount** indicates the number of sessions which are active on the Server. This count shall equal the number of sessionId's included in the message.

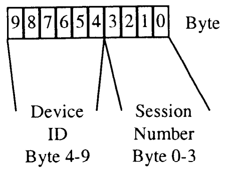
The **sessionId** field indicates the session which is active on the Server. This field shall be repeated for each session which is active on the Server.

4.3 User-to-Network Session Message Field Data Types

Table 4-58 defines the data fields used in the User-to-Network Session Messages.

Table 4-58 User-to-Network Session Message Field Data Types

Field Name	**Length (Bytes)**	**Range**	**Description**
BaseServerId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Server. This address must be a specific address or be able to be resolved to a specific address by the Network. This is the identifier of the first Server involved in a transferred session.
ClientId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Client. This address must be a specific address or be able to be resolved to a specific address by the Network.
DestServerId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Server. This address must be a specific address or be able to be resolved to a specific address by the Network. This is the server to which a session is being transferred.
DeviceId	6	0x000000000000 - 0x3FFFFFFFFFFF	A globally unique number which defines a User or Network device. For networks which use the ATM B-HLI 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).
ForwardCount	2	0x0000 - 0xFFFF	Defines the number of forwardServerId's that are included in the message.
ForwardServerId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Server. This address must be a specific address or be able to be resolved to a specific address by the Network. This is the server to which a session is being forwarded.
NewServerId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Server. This address must be a specific address or be able to be resolved to a specific address by the Network. Indicates the Server to which the session is being transferred.
NextServerId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Server. This address must be a specific address or be able to be resolved to a specific address by the Network. This is the Server to which a session is to be forwarded.

Field Name	Length (Bytes)	Range	Description
OldServerId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a Server. This address must be a specific address or be able to be resolved to a specific address by the Network. This identifies the Server that a session is being transferred from.
PrivateDataByte	privateDataCount	0x00 - 0xFF	Private data which is outside of the scope of this part of ISO/IEC 13818.
PrivateDataCount	2	0x0000 - 0xFFFF	Indicates the number of privateDataBytes which are included in a message.
Reason	2	enumerated	This field indicates the reason for a failure. Refer to Table 4-59 for possible values for this field.
ResourceCount	2	0x0000 - 0xFFFF	This field contains the number of resource descriptors which follow the resourceCount field in the message.
ResourceNum	2	0x0000 - 0xFFFF	Defines a specific resource descriptor within the scope of a resource number assignor.
Response	2	enumerated	This field indicates the response to a requested action. Refer to Table 4-60 for possible values for this field.
SdbId	6	0x000000 - 0xffffff	A network unique ID which identifies a SDB service. The procedure of how the network operator assigns sdbId's is outside the scope of DSM-CC.
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.
SessionCount	2	0x0000 - 0xFFFF	Indicates the number of sessionId's included in the message.
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. 

Field Name	Length (Bytes)	Range	Description
SessionNum	4	0x00000000 - 0xFFFFFFFF	A number which uniquely identifies a session on the device which assigns the sessionId.
StatusByte	statusCount	variable	Status data that depends on the statusType field.
StatusCount	2	0x0000 - 0xFFFF	This field indicates the number of status records that are being returned in a status response.
StatusType	2	enumerated	This field indicates the type of status that is being requested or returned in a status transaction.
UserId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a User. This address must be a specific address or be able to be resolved to a specific address by the Network.
UuDataByte	uuDataCount	0x00 - 0xFF	This data is specified in the DSM-CC User-to-User clause.
UuDataCount	2	0x0000 - 0xFFFF	This indicates the number of uuDataBytes included in a message.

4.4 Reason Codes

The following reason codes are defined for use by the U-N session messages.

Table 4-59 User-to-Network session message reason codes

Reason	Value	Description
RsnOK	0x0000	Indicates that the command sequence is proceeding normally
RsnNormal	0x0001	Indicates normal conditions for releasing the session.
RsnClProcError	0x0002	Indicates that the condition is due to procedure error detected at the Client
RsnNeProcError	0x0003	Indicates that the condition is due to procedure error detected at the Network
RsnSeProcError	0x0004	Indicates that the condition is due to procedure error detected at the Server
RsnClFormatError	0x0005	Indicates that the condition is due to invalid format (e.g., missing parameter) detected at the Client
RsnNeFormatError	0x0006	Indicates that the condition is due to invalid format (e.g., missing parameter) detected at Network
RsnSeFormatError	0x0007	Indicates that the condition is due to invalid format (e.g., missing parameter) detected at Server
RsnNeConfigCnf	0x0008	Indicates that this is a confirmed configuration sequence (i.e., Client must respond)
RsnSeTranRefuse	0x0009	Indicates that the Session transfer was refused by the destination Server
RsnSeForwardOvl	0x000A	Indicates that the session forwarding is due to overload conditions
RsnSeForwardMnt	0x000B	Indicates that the session forwarding is due to overload maintenance conditions

Reason	Value	Description
RsnSeForwardUncond	0x000C	Indicates that the session forwarding is sent as an unconditional request
RsnSeRejResource	0x000D	Indicates that the server rejected the assigned resources.
RsnNeBroadcast	0x000E	Indicates that a message is being broadcast and does not require a response.
RsnSeServiceTransfer	0x000F	Server indicates that the Client shall establish a session to another serverId based on the context provided in the PrivateData().
RsnClNoSession	0x0010	Client indicates that the sessionId indicated in a message is not active.
RsnSeNoSession	0x0011	Server indicates that the sessionId indicated in a message is not active.
RsnNeNoSession	0x0012	Network indicates that the sessionId indicated in a message is not active.
RsnRetrans	0x0013	Indicates that a message is a retransmission.
RsnNoTransaction	0x0014	Indicates that the message was received without a transactionId.
RsnClNoResource	0x0015	Indicates that a requested resource is not supported.
RsnClRejResource	0x0016	Indicates that the Client rejected the assigned resources.
RsnNeRejResource	0x0017	Indicates that the Network rejected the resources assigned by the Server.
RsnNeTimerExpired	0x0018	Indicates that the message is being sent as the result of an expired timer.
RsnClSessionRelease	0x0019	Indicates that the Client initiated a session release.
RsnSeSessionRelease	0x001A	Indicates that the Server initiated a session release.
RsnNeSessionRelease	0x001B	Indicates that the Network initiated a session release.
Reserved	0x001C - 0x7FFF	ISO/IEC 13818-6 reserved
User Defined	0x8000 - 0xFFFF	User defined reason codes.

4.5 Response Codes

The following response codes are defined for use by the U-N session messages.

Table 4-60 User-to-Network session message response codes

Response	Value	Description
RspOK	0x0000	Indicates that the requested command completed with no errors.
RspClNoSession	0x0001	Indicates that the Client rejected the request because the requested sessionId is invalid.
RspNeNoCalls	0x0002	Indicates that the Network is unable to accept new sessions.
RspNeInvalidClient	0x0003	Indicates that the Network rejected the request due to an invalid clientId.
RspNeInvalidServer	0x0004	Indicates that the Network rejected the request due to an invalid serverId.
RspNeNoSession	0x0005	Indicates that the Network rejected the request because the requested sessionId is invalid.
RspSeNoCalls	0x0006	Indicates that the Server is unable to accept new sessions.
RspSeInvalidClient	0x0007	Indicates that the Server rejected the request due to an invalid clientId.
RspSeNoService	0x0008	Indicates that the Server rejected the request because the requested service could not be provided.

Response	Value	Description
RspSeNoCFS	0x0009	Indicates that the Server rejected the request because the requested Continuous Feed Session could not be found.
RspClNoResponse	0x000A	Indicates that the Network timed out before the Client responded to an Indication message.
RspSeNoResponse	0x000B	Indicates that the Network timed out before the Server responded to an Indication message.
Reserved	0x000C - 0x000F	ISO/IEC 13818-6 reserved.
RspSeNoSession	0x0010	Indicates that the Server rejected the request because the requested sessionId is invalid
RspNeResourceContinue	0x0011	Indicates that a resource request completed with no errors but, an indicated resource was assigned an alternate value by the Network.
RspNeResourceFailed	0x0012	Indicates that a resource request failed because the Network was unable to assign the requested resources.
RspNeResourceOK	0x0013	Indicates that the requested command completed with no errors.
RspResourceNegotiate	0x0014	Indicates that the Network was able to complete a request but has assigned alternate values to a negotiable field.
RspClSessProceed	0x0015	Indicates that the Network is waiting on a response from the server.
RspClUnkRequestID	0x0016	Indicates that the Client received a message which contained an unknown resourceRequestId.
RspClNoResource	0x0017	Indicates that the Client rejected a session set-up because it was unable to use the assigned resources.
RspClNoCalls	0x0018	Indicates that the Client rejected a session set-up because it was not accepting calls at that time.
RspNeNoResource	0x0019	Indicates that the network is unable to assign one or more resources to a session.
Reserved	0x001A - 0x001F	ISO/IEC 13818-6 reserved.
RspSeNoResource	0x0020	Indicates that the server is unable to complete a session set-up because the required resources are not available.
RspSeRejResource	0x0021	Indicates that the server rejected the assigned resources.
RspClProcError	0x0022	Indicates that the condition is due to procedure error detected at the Client
RspNeProcError	0x0023	Indicates that the condition is due to procedure error detected at the Network
RspSeProcError	0x0024	Indicates that the condition is due to procedure error detected at the Server
RspNeFormatError	0x0026	Indicates that the condition is due to invalid format (e.g., missing parameter) detected at Network
RspSeFormatError	0x0027	Indicates that the condition is due to invalid format (e.g., missing parameter) detected at Server
RspSeForwardOvl	0x0028	Indicates that the session forwarding is due to overload conditions
RspSeForwardMnt	0x0029	Indicates that the session forwarding is due to overload maintenance conditions
RspClRejResource	0x002A	Indicates that the client rejected a resource assigned to a session.
Reserved	0x002B - 0x002F	ISO/IEC 13818-6 reserved.
RspSeForwardUncond	0x0030	Indicates that the session forwarding is sent as an unconditional request
RspNeTransferFailed	0x0031	Indicates that the session transfer failed at the network

Response	Value	Description
RspClTransferReject	0x0032	Indicates that the session transfer was rejected by the Client
RspSeTransferReject	0x0033	Indicates that the session transfer was rejected by the transferred to server
RspSeTransferResource	0x0034	Indicates that the transferred to server rejected the session transfer due to insufficient resource
RspResourceCompleted	0x0035	Indicates that the Server has accepted the resources assigned by the Network.
RspForward	0x0036	This indicates that the Server is requesting a Session Forward.
RspNeForwardFailed	0x0037	Indicates that the Network is unable to process a Session Forward.
RspClForwarded	0x0038	Indicates that the session was forwarded to the indicated clientId.
Reserved	0x0039 - 0x0040	ISO/IEC 13818-6 reserved.
RspSeTransferNoRes	0x0041	The transfer to Server could not get enough resources, so it rejected the transfer.
RspNeNotOwner	0x0042	An action was requested on a session by a User which was not the owner of that session.
Reserved	0x0043 - 0x7FFF	ISO/IEC 13818-6 reserved
User Defined	0x8000 - 0xFFFF	User defined Response Code

4.6 MPEG-2 DSM-CC statusTypes

The **statusType** field is used to indicate the type of status being requested or returned. Table 4-61 defines the possible statusTypes. In this table, resourceId is the combination of a sessionId and a resourceNum which uniquely identify a resource descriptor within a session.

Table 4-61 MPEG-2 DSM-CC statusType values

statusType	Description	Required Fields for User Status Request / Indication	Required fields for User Status Response / Confirm
0x0000	ISO/IEC 13818-6 Reserved.	N/A	N/A

0x0001	Identify Session List. This status indicates the sessions which are active.	None	sessionCount
			for(sessionCount)
			{
			sessionId
			}

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

0x0003	Identify Configuration. This status indicates the current configuration.	None	Refer to clause 6 "User Compatibility". The Compatibility Descriptor() is returned for this request.

0x0004	Query Resource Descriptor. This status message returns the resource descriptors for the requested resourceIDs	resourceIdCount
		for(resourceIdCount)
		{
		resourceId
		}
			resourceIdCount
			for(resourceIdCount)
			{
			sessionId,
			ResourceDescriptor()
			}

0x0005	Query Resource Status. This status message returns the status of the requested resourceIDs	resourceIdCount
		for(resourceIdCount)
		{
		resourceId
		}
			resourceIdCount
			for(resourceIdCount)
			{
			sessionId,
			resourceNum,
			resourceStatus
			}

0x0006 - 0x7FFF	ISO/IEC 13818-6 Reserved.	N/A	N/A
0x8000 - 0xFFFF	User Defined statusType.	Private Use.	Private Use.

4.7 Resource Descriptors

Resource descriptors contain the information required for the Network to allocate a resource, track the resource once it has been allocated, and de-allocate the resource once it is no longer needed. A resource is an atomic entity which is uniquely identified within a session by its resourceNum.

Session Set-Up and Resource Allocation message classes use resource descriptors to request, assign, and control the various network resources of a session. Status Response/Confirm messages use resource descriptors to describe the resources of a session and their status.

4.7.1 DSM-CC User-to-Network Resource Descriptor

Table 4-62 describes the general format of a Resource Descriptor:

Table 4-62 General format of the DSM-CC Resource Descriptor

Syntax	Num. of Bytes
dsmccResourceDescriptor {	
commonDescriptorHeader()	
resourceDescriptorDataFields()	
}	

The **commonDescriptorHeader** is normative and shall be included with every resource descriptor definition. Table 4-63 defines the format of the commonDescriptorHeader:

Table 4-63 DSM-CC User-to-Network commonDescriptorHeader

Syntax	Num. of Bytes
commonDescriptorHeader() {	
resourceRequestId	2
resourceDescriptorType	2
resourceNum	2
associationTag	2
resourceFlags	1
resourceStatus	1
resourceDescriptorDataFieldsLength	2
resourceDataFieldCount	2
if (resourceDescriptorType == 0xffff) {	
typeOwnerId	3
typeOwnerValue	3
}	
}	

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.

The **resourceDescriptorType** field defines the specific resource being requested. Table 4-73 in subclause 4.7.5 defines the resourceDescriptorTypes defined by DSM-CC.

The **resourceNum** field comprises a unique number and whether the resourceNum was assigned by the Network, Client or Server. Figure 4-2 defines the format of the resourceNum field:

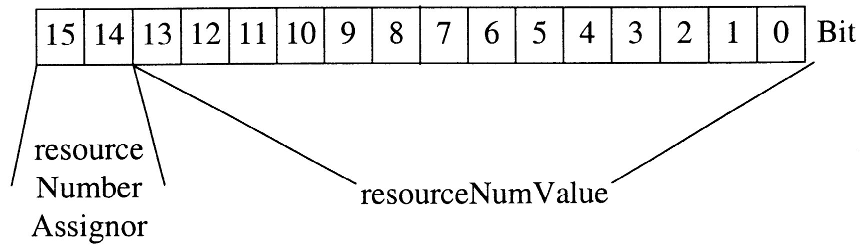


Figure 4-2 Format of DSM-CC User-to-Network resourceNumber field

The **resourceNumValue** subfield uniquely identifies the resource within the session.

The **resourceNumberAssignor** subfield indicates who assigned the resource. Table 4-64 defines the values for this field:

Table 4-64 DSM-CC User-to-Network resourceNumberAssignor values

assignor	Value	Description
Reserved	00	ISO/IEC 13818-6 reserved.
Client	01	Resource number assigned by the Client. This option is not currently supported in DSM-CC and is an item for further work.
Server	10	Resource number assigned by the Server.
Network	11	Resource number assigned by the Network.

The **associationTag** identifies the groups of resources or shared resources that together make up an end-to-end connection or other type of association (if not connection related). An associationTag is unique within a session and has end-to-end significance (i.e., is the same at both ends of the connection). Figure 4-3 defines the format of the associationTag field.

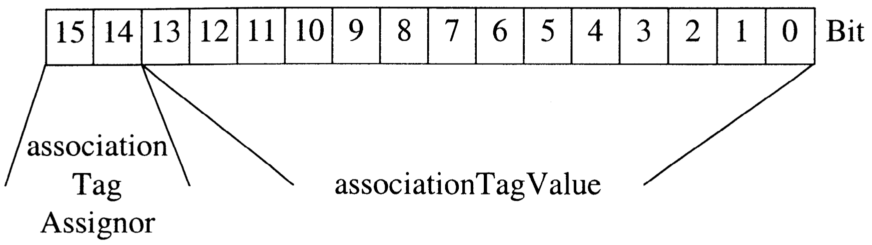


Figure 4-3 Format of DSM-CC User-to-Network associationTag field

The **associationTagValue** portion of the field uniquely identifies the association tag within the session.

The **associationTagAssignor** portion of the field indicates who assigned the resource. Table 4-65 defines the values for this field:

Table 4-65 DSM-CC User-to-Network associationTagAssignor values

assignor	Value	Description
Reserved	00	ISO/IEC 13818-6 reserved.
Client	01	Association tag assigned by the Client. This option is not currently supported in DSM-CC and is an item for further work.
Server	10	Association tag assigned by the Server.
Network	11	Association tag assigned by the Network.

The **resourceFlags** field contains definitions of the entity which is responsible for allocating the resource, the negotiation type of the resource, and which view of the resource is being presented. Figure 4-4 defines the bit definitions of the resourceFlags field:

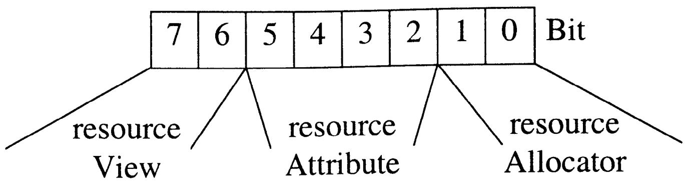


Figure 4-4 Bit definitions of the resourceFlags field

The **resourceAllocator** flag is set by the Server to make it clear which entity in the network is responsible for allocating and controlling the resource. When the resource is allocated by the Server the Network will not modify the resource descriptor. Table 4-66 defines the possible values:

Table 4-66 DSM-CC User-to-Network resourceAllocator field

resourceAllocator	Value	Description
Unspecified	00	Used when the creator of the Resource Descriptor does not know or care who will allocate the resource.
Client	01	Resource allocated by the Client. Not currently supported in DSM-CC and is a subject of future work.
Server	10	Resource allocated by the Server.
Network	11	Resource allocated by the Network.

The **resourceAttribute** field defines how the Server and Network or Client will negotiate a resource. The values for the resourceAttribute field are defined in Table 4-67:

Table 4-67 DSM-CC U-N Resource Descriptor resourceAttribute

resourceAttribute	Value	Description
Mandatory Non-Negotiable	0000	Indicates that the Network must either satisfy the requested value exactly or the entire Resource Request command sequence fails.
Mandatory Negotiable	0001	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 of all ones, then the sequence only fails if no resources are available.
Non-Mandatory Non-Negotiable	0010	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	0011	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 of all ones, then any resource value may be assigned ("don't care" condition).
Reserved	0100-1111	ISO/IEC 13818-6 reserved.

A User shall specify a resource as Mandatory Non-Negotiable when it will not consider an alternative offered by the Network in case of failure. Thus, the Network shall not propose one. The failure to assign a Mandatory Non-Negotiable resource shall cause the failure of the resource allocation procedure and any previously assigned resources in this procedure shall be released.

A User shall specify a resource as Mandatory Negotiable when it will consider an alternative offered by the Network which is within the range given in the resource descriptor. The failure to assign a Mandatory Negotiable resource (i.e., the Network cannot offer a resource within the requested range) shall cause the failure of the resource allocation procedure and any previously assigned resources in this procedure shall be released.

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

A User shall specify a resource as Non-Mandatory Negotiable when it will to consider an alternative offered by the Network which is within the range given in the resource descriptor. The failure of a Non-Mandatory Negotiable resource (i.e., the Network cannot provide a resource within the requested range) does not stop the Network from processing other resource requests within the same AddResourceRequest message.

The Network shall always process Mandatory Non-Negotiable resource requests before Mandatory Negotiable resource requests since the failure of the former aborts the resource allocation procedure. Processing of non-mandatory resources shall occur after successful processing of all mandatory resources.

A User may "re-negotiate" the Mandatory resources that fail by initiating a new AddResource command sequence with new values.

The **resourceView** flags defines from which view of the Resource Descriptor is being presented. Table 4-68 defines the allowable resource views:

Table 4-68 DSM-CC resourceView Values

View	Value	Description
Reserved	00	ISO/IEC 13818-6 reserved.
ClientView	01	Resource List as seen by the Client
ServerView	10	Resource List as seen by Server
Reserved	11	ISO/IEC 13818-6 reserved.

A resource descriptor is used to define a resource which is used by all ends of a session. Since the network technology used to carry a session may be different on each of the endpoints of a session, the resourceView is used to indicated which view of a resource descriptor is being presented.

If a session is inter-worked across network technologies, it shall be the responsibility of the network inter-working equipment to determine the proper resource descriptors and values for the view of the resource descriptor that it presents to the Users.

It is possible that a different resource descriptor type or a different number of resource descriptors must be used to represent a resource descriptor at another endpoint in the session. In this case the Network shall be responsible for mapping the original resource descriptor to the new descriptors and for maintaining all relationships and associations to other resource descriptors used in the session.

For example, when MPEG is sent over ATM into the Network from a server, TSDownstreamBandwidth, atmSvcConnection, and an mpegProgram resource descriptors may be required for the Server view of the session. In an ATM end-to-end system, these same resource descriptors may be also used for the client view. If the Client network is an HFC system, the Client view of these same resources may require a TSDownstreamBandwidth, TSUpstreamBandwidth, clientTDMAAssignment, and mpegProgram resources.

A Client view Resource List occur in the following messages types:

- ClientSessionSetUpConfirm
- ClientAddResourceIndication
- ClientAddResourceResponse
- ClientDeleteResourceIndication
- ClientStatusRequest
- ClientStatusConfirm

- ClientStatusIndication
- ClientStatusResponse

The Client view Resource List conveys to the Client the parameters and attributes of the resources available to it during a session. associationTags in the Client View Resource list will have the same value as associationTags in a ServerView Resource List if the associated resources have end-to-end significance.

A Server may request that the Network include the client view resource list in the ServerAddResourceConfirm message. The Server requests this if it has a need to know which resources are allocated by the Network for use by the Client. Certain network implementations may require that a client view resource list appear in ServerAddResourceRequest messages. In this case, the Server is responsible for determining which resources make up the Client Connection Segment.

A Server view of Resource List consists of the parameters and attributes of the resources it is requesting and those which have been assigned to it within a session. It appears in the following messages:

- ServerAddResourceRequest
- ServerAddResourceConfirm
- ServerDeleteResourceReques
- ServerStatusRequest
- ServerStatusConfirm
- ServerStatusIndication
- ServerStatusResponse

The **resourceStatus** field defines the status of the requested resource between the Server and the Network or Client. The resourceStatus field may have the following values.

Table 4-69 DSM-CC U-N Resource Descriptor resourceStatus

resourceStatus	Value	Description
Reserved	0x00	ISO/IEC 13818-6 Reserved
Requested	0x01	Indicates that the resource has been requested
InProgress	0x02	Indicates that the resource allocation is in progress
AlternateAssigned	0x03	Indicates that an alternative value within the specified range/list was assigned in response to a Negotiable resource.
Assigned	0x04	Indicates that the exact resource value was assigned
Failed	0x05	Indicates that the Network was unable to assign a resource to satisfy the resourceAttribute constraints.
Unprocessed	0x06	Indicates that the Network did not process the request because a Mandatory resource failed prior to the processing of this descriptor.
Invalid	0x07	Indicates that the resource requested is not valid or is unknown
Released	0x08	Indicates that the resource was released
Reserved	0x09 - 0x7F	ISO/IEC 13818-6 reserved.
User Defined	0x80 - 0xFF	Private Data.

The **resourceDescriptorDataFieldsLength** field defines the total length of the resourceDescriptorDataFields section which follows the commonDescriptorHeader. The value of the resourceDescriptorDataFieldsLength field depends on the particular type of the resourceDescriptor being defined and the actual data in the resource descriptor.

The **resourceDataFieldCount** field indicates the total number of data fields in the resource descriptor.

The **typeOwnerId** and **typeOwnerValue** fields are defined only if the resourceDescriptorType field is set to 0xffff. In this situation, these fields are used to indicate that the resource descriptor data fields are defined by an organization which is outside of the scope of DSM-CC.

The **typeOwnerId** field is the first three bytes of an IEEE Organization Unique Identifier (OUI).

The **typeOwnerValue** field is a resourceDescriptorType field defined by the owner of the typeOwnerId (OUI).

Resource descriptor data fields define the actual data fields associated with a particular resourceDescriptorType. The attributes for these fields is defined in Figure 4-5:

Syntax	Encoding	Variable	Number of Bytes

Figure 4-5 Attributes of DSM-CC Resource Descriptor data fields

Syntax defines the data field name and any conditionals or loops associated with the data field.

Encoding defines whether the field value may be requested as a single value (s), list of values (l), or range of values (r).

Variable is a yes or no field which defines if a data field uses the dsmccResourceDescriptorValue format of the field (value is 'Yes') or if the data field uses a simple string of bytes (value is 'No'). In the case of 'No', the Encoding attribute specified for the data field has no meaning.

Number of Bytes indicates the length of each instance of the data field. There shall be exactly 'Number of Bytes' of data for each occurrence of the data field in a resource descriptor.

Table 4-70 defines the format of the resourceDescriptorDataFields:

Table 4-70 DSM-CC User-to-Network resourceDescriptorDataFields

```
Syntax                                                      Num. Of Bytes
resourceDescriptorDataFields() {
    for(i=0;i<resourceDataFieldCount;i++) {
        if (Variable == 'Yes') {
            dsmccResourceDescriptorValue()
        } else {
            for(i=0;i<resourceLength;i++) {
                resourceDataValueByte                       1
            }
        }
    }
}
```

The **resourceDescriptorDataFields** structure contains a list of data fields which are specific to the resourceDescriptorType defined in the commonResourceHeader. The **resourceDataFieldCount** field is defined in the commonResourceHeader. The Variable attribute is defined for each field in the resource descriptor data field for the specific resource descriptor type. If a data field is defined with the Variable attribute defined as 'Yes', then the data field shall use the dsmccResourceDescriptorValue format defined in subclause 4.7.2. If the Variable attribute is defined as 'No', then the content of the resourceDataValueByte field shall contain exactly the number of bytes specified for that data field.

4.7.2 Specifying Ranges and Lists of values in resource descriptors

When requesting that the network assign a value to a field in a resource descriptor, it is possible that the field may have more than one acceptable value. DSM-CC permits the use of a range or list of values when requesting a resource value. For example, if a User is requesting a connection resource, it may have several possible ports from which the session may be delivered. In this case, the request may contain a list of ports from which the network may choose. In another example, a Server may be able to deliver the service at a variable rate. In this case, the resource request may specify an upper and lower range of bandwidth values and the network may choose an appropriate value within this range.

If a resource descriptor data field is variable, as defined in Figure 4-5 Attributes of DSM-CC Resource Descriptor data fields, then that field shall be encoded using the dsmccResourceDescriptorValue() format (see Table 4-71). If a resource descriptor data field is not defined as variable, that resource descriptor value shall not use the dsmccResourceDescriptorValue() format.

Table 4-71 dsmccResourceDescriptorValue() field format

Syntax	Num. Of Bytes
dsmccResourceDescriptorValue() {	
resourceValueType	2
if (resourceValueType = singleValue) {	
resourceValue()	
}	
else if (resourceValueType = listValue) {	
resourceListCount	2
for(i=0;i<resourceListCount;i++) {	
resourceValue()	
}	
}	
else if (resourceValueType = rangeValue) {	
mostDesiredRangeValue()	
leastDesiredRangeValue()	
}	
}	

The **resourceValueType** field indicates the format of the value which is being requested. This field corresponds to the encoding definition (see Figure 4-5) as specified in the data field definitions for the specific resource descriptor. Table 4-72 defines the possible resourceValueTypes:

Table 4-72 DSM-CC Resource Value Types

resourceValueType	Encoding	Value	Description
Reserved		0x0000	ISO/IEC 13818-6 reserved.
singleValue	s	0x0001	Indicates that a single value is being requested for this resource. In this case, the resource value field shall contain only one element.
listValue	l	0x0002	Indicates that the requested value contains a list of possible values that the requester will accept. In this case, the resource value field shall contain a resourceValueCount field and exactly resourceValueCount elements. The list of values shall be ordered with the most desired value as the first entry and the least desired value as the last entry.
rangeValue	r	0x0003	Indicates that the requested value contains a range of values that the requester will accept. In this case, the resource value field shall contain two elements. The first value shall specify the most desired end of the range of values that will be accepted and the second value shall specify the least desired end of the range that will be accepted.
Reserved		0x0004 - 0x7fff	ISO/IEC 13818-6 reserved.
User Defined		0x8000 - 0xffff	Resource value types in this range are user definable.

The **resourceListCount** field is included as the first field of the resource value when the resourceValueType field is set to 'listValue'. This field indicates the number of resource values which follow this field.

resourceValue() indicates a group of resource descriptor fields for the particular resource descriptor type. These fields are defined in subclause 4.7.5 for resource descriptors which are defined in this specification. In the case of a **singleValue**, the number and format of the fields shall match exactly those defined for a particular resource descriptor type. In the case of a **listValue**, the list of fields for that type of resource shall be repeated **resourceListCount** times to indicate the acceptable combinations of fields for that resource descriptor.

In the case of a **rangeValue**, the **mostDesiredRangeValue()** and **leastDesiredRangeValue()** shall each specify a resource descriptor field for the particular resource descriptor type. The SRM shall attempt to allocate resources to the request, starting with the values defined in the most desired values and working towards the least desired values. Each field in the resource descriptor is treated in this manner. There shall be no explicit interaction between fields. The SRM may allocate any value in the range to an individual field with no consideration to the values allocated to other fields in the descriptor. If specific interactions between fields is required, the listValue method shall be used to indicate which combination of field values is acceptable for this resource.

A resource descriptor data field which is defined as using the dsmccResourceDescriptorValue() format shall be encoded with the resourceValueTypes which are possible for that field. A descriptor data field may be encoded in different formats depending on the application.

4.7.3 Horizontal Association of Resource Descriptors

Resource descriptors convey information about connections and other Network resources between a User and the Network. When more than one resource type is required to specify an end-to-end association, horizontal association of resources shall be used. Horizontal association of resource descriptors provides the ability to correlate the resources from one User's view to the resources which appear in another User's view.

An example of horizontal association of resource descriptors is the case where a resource may be of a type that provides an end-to-end connection between Server and Client. In many network implementations, several resources of different types are required to jointly provide a complete connection. The connection from the Server to the Network may use a different technology than the connection from Network to the Client. In this case, each time the network technology changes, the association tag is preserved in the resource descriptor which describes that link of the connection. In this manner, the applications at either end of the connection are able to refer to this association tag for the connection without knowing what type of network technology is being used at the other end.

Association Tags are used in resource related U-N messages:

- ClientSessionSetUpConfirm
- ClientAddResourceIndication
- ClientAddResourceResponse
- ClientDeleteResourceIndication
- ClientStatusRequest
- ClientStatusConfirm
- ClientStatusIndication
- ClientStatusResponse
- ServerAddResourceRequest
- ServerAddResourceConfirm
- ServerDeleteResourceRequest
- ServerStatusRequest
- ServerStatusConfirm
- ServerStatusIndication
- ServerStatusResponse

4.7.4 Vertical Resource Sharing

In situations where two or more resources are contained within a resource descriptor (e.g., as a result of multiplexing, as is the case of MPEG Transport Streams), then that resource is a shared resource. The vertical resource sharing is indicated by the shared resource descriptor which identifies the resource number of the shared resource. The association tag of the shared resource is used to indicate the resource which is being shared.

4.7.5 Resource Descriptor Definitions

This subclause defines the data fields which are specified for 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 resourceDescriptorType in a session.

The resources descriptors listed in this subclause are defined by DSM-CC to be normative optional. This means that an implementation of this part of ISO/IEC 13818 may choose to implement none, any, or all of the descriptors. However, if any of these descriptors are implemented, then they shall comply to the syntax and semantics of the descriptor as defined in this subclause.

Table 4-73 lists the resource descriptors which are defined in this part of ISO/IEC 13818. It is not required that an implementation of DSM-CC contain any or all of these descriptors. If any of these descriptors are used however, it is required that they be implemented exactly as described in this part of ISO/IEC 13818.

Table 4-73 DSM-CC User-to-Network resourceDescriptorTypes

resourceDescriptorType	Value	Description
Reserved	0x0000	ISO/IEC 13818-6 reserved.
ContinuousFeedSession	0x0001	Describes resources already allocated in a continuous feed session.
AtmConnection	0x0002	Describes either an ATM PVC, or a pre-allocated SVC, connection resource.
MpegProgram	0x0003	Provides a method of delivering the MPEG-2 Systems Program Map Table (PMT) information 'out of band'.
PhysicalChannel	0x0004	Indicates the use of a specific transport stream. (e.g. the tuner channels on a Hybrid Fiber Coax (HFC) system).
TSUpstreamBandwidth	0x0005	Describes the total upstream bandwidth in bits/second required to deliver session data from the Client to the Server.
TSDownstreamBandwidth	0x0006	Describes the downstream bandwidth in bits/second required to deliver session data from the Server to the Client.
AtmSvcConnection	0x0007	Provides ATM SVC SETUP parameters. This is used when the Network or Client is responsible for initiating a call.
ConnectionNotify	0x0008	This is sent between the User and the Network to indicate that a connection has been established outside of the scope of DSM-CC. This resource descriptor contains no fields and is used as a correlation between the session and the connection.
IP	0x0009	This is requested from the Server to indicate that data will be exchanged between the Server and the Client using IP protocol.
ClientTDMAAssignment	0x000a	Sent from the Network to the Client to assign slots in the upstream TDMA channel to a session.
PSTNSetup	0x000b	Enables a DSM-CC User to set-up a PSTN call.

resourceDescriptorType	Value	Description
NISDNSetup	0x000c	Enables a DSM-CC User to set-up an N-ISDN call.
NISDNConnection	0x000d	Describes a N-ISDN connection and marks which B channel is in use for this connection.
Q.922Connections	0x000e	Allows one or more data link connections to be multiplexed into a single Q.922 connection.
HeadEndList	0x000f	Indicates a list of head-ends which are to be connected to a Continuous Feed Session.
AtmVcConnection	0x0010	Indicates the VPI / VCI of an ATM virtual connection.
SdbContinuousFeed	0x0011	Enables a SDB Management Server to identify program feeds to SDB Servers and obtain broadcastProgramId's
SdbAssociations	0x0012	Enables an SDB management server to set association tags for SDB control and SDB program channels, and enables the Network to identify to the Client the connection resources for SDB control and SDB program channels.
SdbEntitlement	0x0013	The SDB service provider uses this descriptor to enable a client to access the SDB service.
Reserved	0x0014 - 0x7ffd	ISO/IEC 13818-6 reserved.
SharedResource	0x7ffe	Sent by the Server to the Network, or by the Network to the Client to instruct the recipient to share an already used resource (the shared resource).
SharedRequestId	0x7fff	Sent by the Server to the Network along with the resource descriptors in a request message and is used by the Server to associate the resource numbers assigned by the network to the specific resource descriptors.
UserDefined	0x8000 - 0xfffe	Resource descriptors in this range are user definable.
TypeOwner	0xffff	Defines owner of the UserDefined resource type

4.7.5.1 ContinuousFeedSession resource descriptor definition

The **ContinuousFeedSession** resource descriptor is requested by the Server to connect a Client session to a continuous feed session which has been previously established. When the Network receives this resource in a response from the Server to a session set-up indication or in a session set-up request from the server, then the Network shall connect the indicated resources from that continuous feed session to the session which is being established. Table 4-74 defines the format of the ContinuousFeedSession descriptor.

Table 4-74 ContinuousFeedSession resource descriptor

Syntax	Encoding	Variable	Num. of Bytes
sessionId	**s,l**	**Yes**	10
resourceNumCount	**s**	**No**	2
for(i=0; i<resourceNumCount; i++){			
resourceNum	**s,r,l**	**Yes**	2
}			

The **sessionId** 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.

The **resourceNumCount** is the number of resource descriptors in the cfSession to be connected to the new session.

The **resourceNum** field is used to indicate the resourceNum of the resources from the continuous feed session to be connected to the new session.

4.7.5.2 AtmConnection resource descriptor definition

The **AtmConnection** resource descriptor describes an ATM PVC, or a pre-allocated ATM SVC connection. This type of connection is most common in third-party and proxy signaling methods. Table 4-75 defines the format of the AtmConnection resource descriptor.

Table 4-75 AtmConnection resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
atmAddress	s,r,l	Yes	20
atmVci	s,r,l	Yes	2
atmVpi	s,r,l	Yes	2

The **atmAddress** field indicates the ATM address of the User. This address is supplied by the requester of the resource. The Network may use this address to establish a connection between the Network and the User or to verify the address of a User. This value may be sent to the appropriate entity in the Network to be used in the ATM connection establishment procedure.

The **atmVci** and **atmVpi** parameters may be supplied by either the User or the Network. If the User is requesting a connection, it shall set these fields to 0. After the Network has established an ATM connection, these fields shall be set to values assigned by the Network. If the User is establishing the connection, it shall set these fields to the values used for the connection. The VPI value shall be right justified in the atmVpi field and this field shall be 0 filled to the left of the VPI value.

4.7.5.3 MpegProgram resource descriptor definition

The **MpegProgram** descriptor is requested by the Server to request the allocation of MPEG resources (Program Number and PIDs) needed to deliver an MPEG Program to the Client. If the Server is the resourceAllocator, the Server shall fill in the mpegProgramNum and PIDs (mpegPmtPid, mpegPid) values to inform the Network of the values selected by the Server. If the resourceAllocator is the Network the mpegProgramNum and PIDs must be initially set to 0 by the Server and the Network will return Network allocated values.

Note that due to the potential for re-multiplexing of MPEG-2 Transport Streams during transport, the PIDs and Program Number from the Server view (resourceView = serverView) need not have the same values as the Client's view (resourceView = clientView). The User-to-User layer of DSM-CC uses the associationTags to refer to each of the elementary streams since the PID's themselves can not be used as an absolute reference. DSM-CC requires that the association tags assigned to the PIDs be preserved throughout the transport network.

This resource descriptor provides a method of delivering the MPEG Program Map Table (PMT) information 'out of band'. However, this shall not relieve any of the MPEG-2 Systems compliance requirements including the requirement to include the PMT in the Transport Stream.

All of the elementary stream mpegPid values listed in the MpegProgram() descriptor must be found in the PMT found in the in band stream. It is not required that all of the elementary stream PID's listed in band in the PMT need to be listed in the MpegProgram() descriptor. The MpegProgram() descriptor lists only those elementary streams (PID's) which need to be delivered to the Client. Elementary streams listed in the PMT but not listed in the MpegProgram() descriptor may be dropped by the Network (e.g., in an MPEG re-multiplexer in the network).

Table 4-76 mpegProgram descriptor

Field Name	Encoding	Variable	Field Length In Bytes
mpegProgramNum	**s,r,l**	**Yes**	**2**
mpegPmtPid	**s,r,l**	**Yes**	**2**
mpegCaPid	**s**	**No**	**2**
elementaryStreamCount	**s**	**No**	**2**
for(i=0; i<elementaryStreamCount; i++) {			
mpegPid	**s,r,l**	**Yes**	**2**
stream_type	**s,r,l**	**Yes**	**1**
reserved			**1**
associationTag	**s,r,l**		**2**
}			
mpegPcr	**s,r,l**	**Yes**	**2**

The **mpegProgramNum** field specifies the MPEG programNum value used in the Program Association Table (PAT) and Program Map Table (PMT) tables for this Program.

The **mpegPmtPid** field specifies the PID which carries the Program Map Table (PMT) for this program.

The **mpegCaPid** field contains an 13 bit MPEG Packet ID (PID) as defined in ISO/IEC 13818-1. If the Server will insert Conditional Access (CA) data then the PID to be used is requested/assigned with this field. If the Server either does not need to insert CA data or is unable to insert CA data then the Server may set this field to all 1's and the Network will not allocate a CA PID.

The **elementaryStreamCount** indicates the number of elementary streams used by the MPEG-2 program. There shall be one elementary stream described for each value in this field. The value of all 1's in reserved.

The **mpegPid** field contains an 13 bit MPEG Packet ID (PID) as defined in ISO/IEC 13818-1. The upper three bits shall be set to 0, the lower thirteen bits contain the PID. If the PIDs are to be set by the Network then the resource requester (Server) will initially set all the mpegPid fields to 0.

The **stream_type** field indicates the type of the elementary stream, e.g. video, audio etc. See Table 2-36, stream_type assignments, in ISO/IEC 13818-1.

The **reserved** field shall be set to 0.

The **associationTag** field is the same associationTag as used in the commonDescriptorHeader(). It uses the same name space as all of the other associationTags within the Session. Since the add resource initiator allocates the associationTag values, it is feasible to assign tag values within the body of a descriptor as well as in the header. The associationTag in the descriptor header tags the MPEG Program. Each elementary stream in the session shall have a unique associationTag.

The **mpegPcr** field defines which of the elementary streams, if any, in the preceding list contains the Program Clock Reference (PCR). This value is an offset into the array of elementary streams. A value of 0 indicates the first elementary stream in the array, 1 indicates the second elementary stream, etc. Setting this value to all 1's indicates that no PCR PID is used.

4.7.5.4 Physical Channel resource descriptor definition

The intended use for the PhysicalChannel resource descriptor is to allocate and/or indicate the use of a specific logical channel in a frequency division multiplexed medium, e.g. the tuner channels on a Hybrid Fiber Coax (HFC) system.

Table 4-77 PhysicalChannel resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
channelId	**s,r,l**	**Yes**	**4**
direction	**s**	**No**	**2**

channelId – example: specifies tuner channel number (either an index or in units of Hz). Details are private to the implementation.

The **direction** field defines the direction of data flow between the Server and Client.

Table 4-78 Direction Values

direction	Description
0x0000	Downstream (Server to Client)
0x0001	Upstream (Client to Server)
0x0002 - 0xffff	ISO/IEC 13818-6 Reserved.

4.7.5.5 TSUpstreamBandwidth resource descriptor definition

The TSUpstreamBandwidth resource descriptor is requested to allocate a portion of the upstream transport stream for a session. Multiple TSUpstreamBandwidth resource descriptors may be requested for a session. Table 4-79 defines the format of the TSUpstreamBandwidth resource descriptor.

Table 4-79 TSUpstreamBandwidth resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
upstreamBandwidth	s,r,l	Yes	4
upstreamTransportId	s,l	Yes	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 TSUpstreamBandwidth resource descriptor. The Network shall assign this field.

4.7.5.6 TSDownstreamBandwidth resource descriptor definition

The TSDownstreamBandwidth resource descriptor is requested to allocate a portion of the downstream transport stream for a session. Multiple TSDownstreamBandwidth resource descriptors may be requested for a session. Table 4-80 defines the format of the TSDownstreamBandwidth resource descriptor.

Table 4-80 TSDownstreamBandwidth resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
downstreamBandwidth	s,r,l	Yes	4
downstreamTransportId	s,l	Yes	4

The **downstreamBandwidth** field indicates the data rate in bits per second which have been allocated to the session for delivery of application data to the Client. The entire range of values from 0x00000000 to 0xffffffff for downstreamBandwidth are valid. The Requester 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 **downstreamTransportId** field indicates the transport stream that the Client should use to receive the downstream application data from the Server. The Server shall set this field to 0 when requesting a TSDownstreamBandwidth resource descriptor. The Network shall assign this field.

4.7.5.7 AtmSvcConnection resource descriptor definition

The **AtmSvcConnection** resource descriptor is used when requesting an ATM connection resource. The Network may pass this request to the ATM device which sets up the connection using the appropriate signaling or the Network may set up the connection on behalf of the devices. Table 4-81 describes the format of the AtmSvcConnection resource descriptor. It contains all the set up message parameters required to set up the ATM SVC connection.

Table 4-81 AtmSvcConnection3.0 resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
AtmSvcSetUp	s	No	Parameters as they appear in the SETUP message of the specific signaling specification ATM UNI 3.0, 3.1, 4.0 or ITU-T Q.2931

4.7.5.8 ConnectionNotify resource descriptor definition

The **ConnectionNotify** resource descriptor is sent to inform the SRM that a connection (i.e. ATM SVC connection or HFC MAC Layer connection) has been established outside of the Network. This descriptor has no data fields and is sent in order to allow the SRM to correlate a session with the connection using the resourceld of the descriptor.

Table 4-82 defines the format of the ConnectionNotify resource descriptor.

Table 4-82 ConnectionNotify resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes

Note: There are no data fields associated with this resource descriptor since the details of the connection are not within the scope of the Network.

4.7.5.9 IP resource descriptor definition

The IP resource descriptor is requested by the server to indicate that IP data is being transported. Table 4-83 defines the format of the IP resource descriptor.

Table 4-83 IP resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
sourceIpAddress	s	No	4
sourceIpPort	s	No	2
destinationIpAddress	s	No	4
destinationIpPort	s	No	2
ipProtocol	s	No	2

The **sourceIpAddress** field indicates the IP address of the device which is sending the IP messages.

The **sourceIpPort** field indicates the port from which the data will be transmitted.

The **destinationIpAddress** field indicates the IP address of the device to which the IP messages are sent.

The **destinationIpPort** field indicates the port to which the data will be transmitted.

The **ipProtocol** field indicates the protocol which is being carried over this IP stream. Table 4-84 defines these protocol types:

Table 4-84 IP Protocol Types

ipProtocol	Value	Description
Reserved	0x0000	ISO/IEC 13818-6 reserved.
TCP	0x0001	Indicates that TCP is being carried over IP.
UDP	0x0002	Indicates that UDP is being carried over IP.
Reserved	0x0003 - 0x7fff	ISO/IEC 13818-6 reserved.
User Defined	0x8000 - 0xffff	User defined

4.7.5.10 ClientTdmaAssignment resource descriptor definition

The ClientTdmaAssignment resource descriptor is assigned by the Network and sent to the Client to instruct the Client about how/where to transmit the upstream application data. The Server may also request this resource if it has a need for this information (for informational purposes only). Table 4-85 defines the format of the ClientTdmaAssignment resource descriptor.

Table 4-85 ClientTdmaAssignment resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
startSlotNumber	s,r,l	Yes	4
numberOfSlots	s,r,l	Yes	4
slotSpacing	s,r,l	Yes	4
upstreamTransportId	s,r,l	Yes	4

The **startSlotNumber** field indicates the first TDMA slot on which the session may transmit data.

The **numberOfSlots** field indicates the number of consecutive slots on which the session may transmit data.

The **slotSpacing** field indicates the number of slots that the session must wait after startSlotNumber + numberOfslots before it may transmit again.

The **upstreamTransportId** field indicates the transport stream on which the session should transmit data.

4.7.5.11 PSTNSetup resource descriptor definition

The PSTNSetup resource descriptor enables a DSM-CC server to set up a PSTN call to a DSM-CC Client.

Table 4-86 PSTNSetup resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
callingId	s	No	12
calledId	s	No	12

The **callingId** is a globally unique E.163 dialing string that identifies the calling party.

The **calledId** is globally unique E.163 dialing string that identifies the called party.

4.7.5.12 NISDNSetup resource descriptor definition

The NISDNSetup resource descriptor enables a DSM-CC server to set up an N-ISDN call to a DSM-CC Client.

Table 4-87 nISDNSetup resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
nISDNSetup	s	No	Parameters as they appear in the SETUP message of Q.931 for N-ISDN call setup

The **nISDNSetup** field contains parameters as they appear in the SETUP message of Q.931 for N-ISDN call setup.

4.7.5.13 NISDNConnection resource descriptor definition

This descriptor describes the N-ISDN connection, and marks which B Channel is in use for this connection.

Table 4-88 nISDNConnection resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
bChannel	s	No	2

The **bChannel** field indicates the N-ISDN B channel this N-ISDN resource is using.

4.7.5.14 Q922Connections resource descriptor definition

The Q922Connections resource descriptor assumes that Q.922 is used as the link layer over the modem connection, allowing more than one data link connection to be multiplexed over the connection this resource descriptor is associated with.

Table 4-89 Q922Connections resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
dLCICount	s	No	2
for(i=0; i<dLCICount; i++){			
dLCI	s	No	2
associationTag	s	No	2
}			

The **dLCICount** field gives the number of data link connections multiplexed over this PSTN connection

The **dLCI** field is the Data Link Connection Identifier for this data link connection.

The **associationTag** field is the association tag of the connection resource for this dLCI.

4.7.5.15 SharedResource resource descriptor definition

This descriptor is used to indicate that two or more resources are contained within another resource.

Table 4-90 SharedResource Descriptor

Field Name	Encoding	Variable	Field Length In Bytes
sharedResourceNum	s	No	2

The **sharedResourceNum** field identifies an already existing resourceNum in the session. This descriptor is used to specify that an already existing resource is going to be reused.

4.7.5.16 SharedRequestId resource descriptor definition

This descriptor is used to indicate that two or more resources are contained within another resource and is only used if the Server does not know the resourceNum.(i.e. a case where the Server has requested a resource, the Network is assigning resourceNums, and a ServerAddResourceConfirm message has not arrived).

Table 4-91 sharedResourceRequestId Descriptor

Field Name	Encoding	Variable	Field Length In Bytes
sharedResourceRequestId	s	No	2

The **sharedResourceRequestId** field identifies a previously requested resource that has not been allocated yet. This descriptor is used to specify that a already existing resource is going to be reused.

4.7.5.17 HeadEndList resource descriptor definition

This descriptor contains a list of head-end codes which indicates which head-ends a Continuous Feed Session is to be connected. Table 4-92 defines the fields in the HeadEndList resource descriptor.

Table 4-92 HeadEndList resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
headEndCount	s	No	2
for(i=0; I<headEndCount; i++){			
headEndCode	s	No	20
}			

The **headEndCount** is the number of head-end codes included in this resource descriptor.

The **headEndCode** is an OSI NSAP address which uniquely identifies a head-end throughout the Network.

4.7.5.18 AtmVcConnection resource descriptor definition

This descriptor describes an AtmVcConnection.

Table 4-93 AtmVcConnection Descriptor

Field Name	Encoding	Variable	Field Length In Bytes
atmVpi	s	No	2
atmVci	s	No	2

The **atmVpi** and **atmVci** parameters contain the VPI and VCI values for the ATM connection.

4.7.5.19 SdbContinuousFeed resource descriptor definition

The SdbContinuousFeed resource descriptor enables a SDB Management Server to identify program feeds to SDB Servers and obtain broadcastProgramIds.

Table 4-94 SdbContinuousFeed resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
sdbId	s	No	6
programCount	s	No	2
for(i=0; I<programCount; i++){			
associationTag	s	No	2
broadcastProgramId	s	No	2
}			

The **sdbId** is a network unique ID which identifies a SDB service. The procedure of how the network operator assigns sdbId's to SDB service providers is outside the scope of DSM-CC

The **programCount** is the number of programs available in the SdbContinuousFeed resource descriptor

The **associationTag** refers to the association tag of the corresponding MpegProgram descriptor.

The **broadcastProgramId** is a unique identifier for the program feed allocated by the network.

4.7.5.20 SdbAssociations resource descriptor definition

The SdbAssociations resource descriptor enables an SDB management server to set association tags for SDB control and SDB program channels, and enables the Network to identify to the Client the connection resources for SDB control and SDB program channels.

Table 4-95 SdbAssociations resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
sdbControlAssociationTag	s	No	2
sdbProgramAssociationTag	s	No	2

sdbControlAssociationTag is the association tag of the connection resource for the control information (e.g., atmConnection, AtmSvcConnection).

The **sdbProgramAssociationTag** is the association tag of the connection resource for the program (e.g., atmConnection, AtmSvcConnection).

4.7.5.21 SdbEntitlement resource descriptor definition

The SDB service provider uses the SdbEntitlement descriptor to enable a client to access the SDB service.

Table 4-96 SdbEntitlement resource descriptor

Field Name	Encoding	Variable	Field Length In Bytes
sdbId	s	No	6
excludeCount	s	No	2
for(i=0; i<excludeCount; i++){			
broadcastProgramId	s	No	2
}			
includeCount	s	No	2
for(i=0; i<includeCount; i++){			
broadcastProgramId	s	No	2
}			

The **sdbId** is a network unique ID which identifies a SDB service. The procedure of how the network operator assigns sdbId's is outside the scope of DSM-CC.

The **excludeCount** is the number of programs excluded from the set of broadcastProgramId's belonging to the sdbId.

The **includeCount** is the number of programs included from the set of broadcastProgramId's belonging to the sdbId.

If both the excludeCount and includeCount are null then all the broadcastProgramIds under the sdbId are included.

The **broadcastProgramId** is a unique identifier for the program feed allocated by the network. It is an error condition if the specified broadcastProgramIds are outside the set belonging to the sdbId.

4.8 Client Initiated Command Sequences

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

- Session set-up command sequence.
- Session release of a session command sequence.
- Status request 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 a special case of a Session which is not connected to any particular Client. Any number of Clients on the network may connect to a CFS after it has been set up. Client sessions which connect to the CFS are allocated the same downstream resources thereby sharing a single connection and MPEG program between all Clients. Actual network implementations may require some segmentation. In addition to the resources which are shared, Clients which are connected to a continuous feed session may also be allocated additional resources which are exclusive to that Client.

The Server may notify the Clients that a Continuous Feed Session has been set-up via the pass-thru messages described in clause 12, User-to-User messages described in clause 5, the Data Carousel method defined in clause 7, or by a means outside of the scope of this part of ISO/IEC 13818. The Client connects to a CFS using the User-to-Network Session Messages for session set-up. It passes the information about the CFS in the UserData clause of the set-up request.

4.8.1 Client Session Set-Up Command Sequence

Figure 4-6 illustrates the procedure for session establishment initiated by the Client.

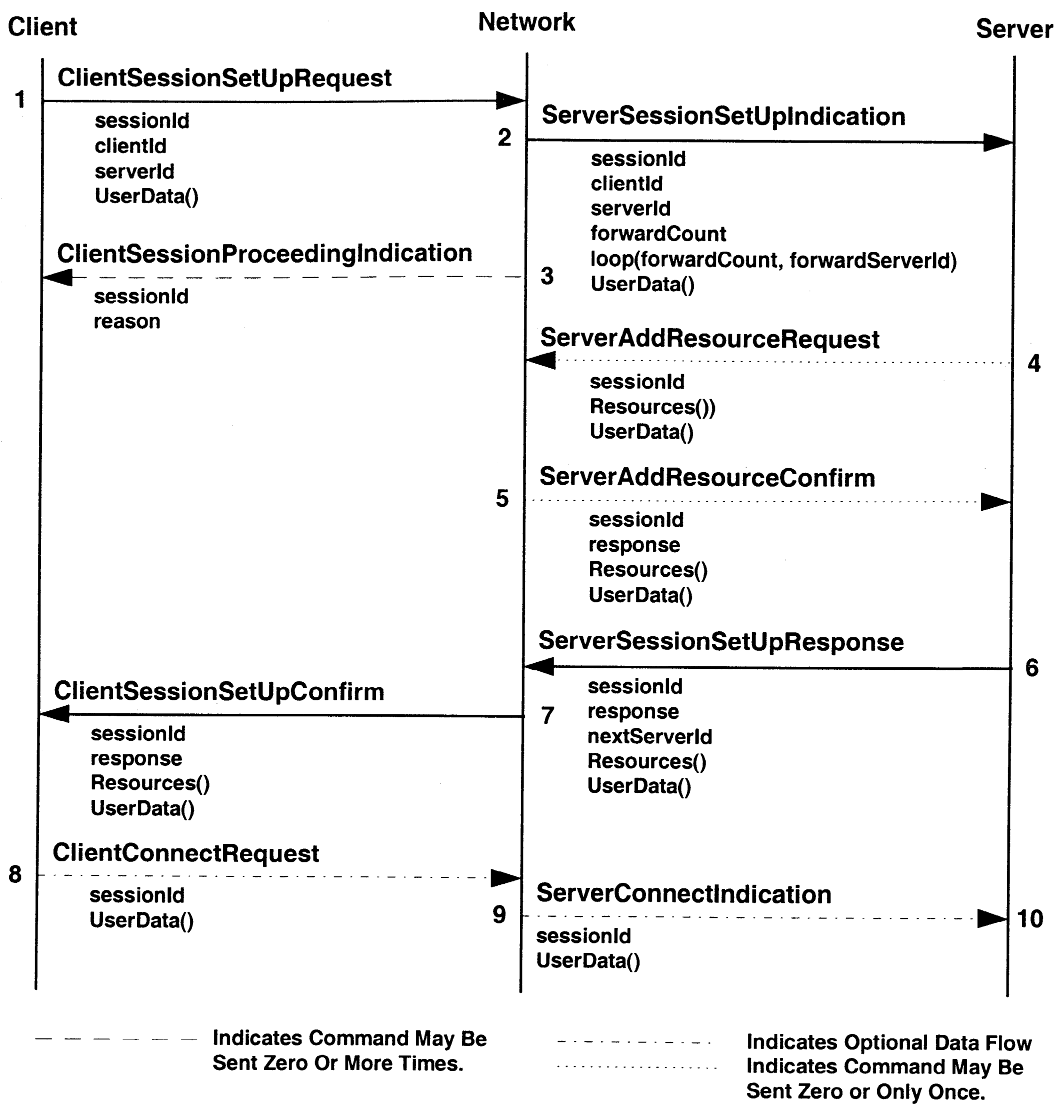


Figure 4-6 Scenario for Client Session Set-up message sequence

4.8.1.1 Client Initiates Session Set-Up Request

Step 1 (Client)

To begin establishing a new session, the Client shall send ClientSessionSetUpRequest to the Network and start timer. The value of the transactionId shall be selected by the Client and shall be used to correlate replies from the Network to this ClientSessionSetUpRequest message. If the Client is responsible for 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 session establishment. The UserData() field may be populated with additional data to be sent to the Server. Upon sending the message, the Client shall start timer tMsg.

If timer tMsg expires before the ClientSessionSetUpConfirm, ClientSessionProceeding, or ClientResetIndication message is received, flow shall shift to the "Network does not respond to ClientSessionSetUpRequest" scenario.

If a ClientSessionProceedingIndication message is received for this request, the client shall restart the tMsg timer and re-enter the wait loop.

Step 2 (Network)

On receipt of ClientSessionSetUpRequest, the Network shall verify the message and determine if the serverId field represents an entity known to the Network. If the message and parameters are valid and the Network can support a new session, the Network shall send the ServerSessionSetUpIndication to the Server indicated by the serverId field and shall start timer tMsg for the server state machine. 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 session through the life of the session. The value of the clientId field identifies the Client that requested the session and shall be identical to the clientId received in the ClientSessionSetUpRequest. The UserData() structure shall be identical to the values received in the ClientSessionSetUpRequest. The Network shall pass 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 tMsg expires before the ServerSessionSetUpResponse or ServerSessionSetUpResponse messages are received, the Network shall send ClientSessionProceedingIndication to the Client. The reason field shall be set to RsnNeTimerExpired. Timer tMsg shall then be restarted. The Network may implement policy to determine how many times the tMsg timer may expire before the ServerSessionSetUpResponse or ServerContinuousFeedSessionResponse messages are received before timing out and clearing the session. At this time, flow shall shift to the "Server does not respond to ServerSessionSetUpIndication" scenario.

Step 3 (Network Optional)

The Network may send 0 or more ClientSessionProceedingIndication messages to the client while it is waiting for a response from the Server. The purpose of these messages is to reset the tMsg timer on the Client to prevent the Client from timing out.

Step 4 (Server)

On receipt of ServerSessionSetUpIndication, the Server may validate the message parameters. If the requested parameters are acceptable, the Server may satisfy the session request by either establishing a new session for the requesting Client or, by connecting the Client to an existing Continuous Feed Session.

If it is necessary to create an exclusive session to satisfy the set-up request, the Server may request Network allocated resources for the new session by sending the ServerAddResourceRequest to the Network. If no Network allocated resources are required for this session, flow shall skip to step 7.

If the request may be satisfied by connecting the client session to an existing Continuous Feed Session, the Server shall request the ContinuousFeedSession resource from the Network which contains the sessionId from the continuous feed session which is to be connected to the Client. The Server may also allocate or request additional resources which are exclusive to the Client Session in addition to those already allocated to the Continuous Feed Session. The ContinuousFeedSession resource is considered to be allocated by the Server shall be sent in the ServerSessionSetUpResponse message. If additional Network allocated resources are required for the session, the server shall send the ServerAddResourceRequest to the Network. If no additional Network allocated resources are required for this session, flow shall skip to step 7.

Step 5 (Network Optional)

On receipt of ServerAddResourceRequest, the Network shall resolve the resource request in accordance with the procedure described in subclause 4.7.2. Upon resolving and assigning all requested resources, the Network shall send a ServerAddResourceConfirm message to the Server. This message shall contain the disposition of all resource requests and the values assigned to the resources.

Step 6 (Server)

On receipt of ServerAddResourceConfirm, the Server shall send ServerSessionSetUpResponse to the Network which contains the Server's response to the session set-up request, any resources which were allocated by the Server but not included in the ServerAddResourceRequest message, and UserData() to be passed to the Client.

At this point, the Session Gateway at the Server shall consider the session established. If the Network requires that a Session-In-Progress message be sent, the Server shall start timer tSIP.

Step 7 (Network)

On receipt of ServerSessionSetUpResponse with the response field encoded as rspOK, the Network shall terminate the tMsg timer for this session. If Server allocated resources are present, the Network shall check the validity of the resource descriptors provided. Each shall relate to a resource with a resourceStatus of Assigned and shall not be a resource descriptor that appeared in a previous ServerAddResourceRequest (i.e. the resources were allocated by the Server). If the resources are not acceptable to the Network then the flow shall shift to the "Network rejects Server's resource allocation" scenario.

If a the cfSession resource descriptor is included, the Network shall attempt to connect the resources of the requested continuous feed session to the Clients view of the session.

If there are no resources in the resource list or the resources are acceptable to the Network then the Network shall 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 assigned to the Client view of the session. The resource descriptors shall be those resources visible to the Client for this session. The values of the UserData() shall be identical to those received in the ServerResourceResponse message.

Step 8 (Client)

On receipt of ClientSessionSetUpConfirm, the Client shall terminate timer tMsg and determine if it is capable of using the resources assigned to the session. If the Client can use all of the assigned resources, it shall determine if it has UserData() to be delivered to the Server. If the Client does not have UserData() to be delivered to the Server, the session shall be considered to be active. If the Client is required to send Session In Progress messages, it shall start timer tSIP.

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 uuDataBytes to be delivered to the Server, flow shall shift to the "Client Has Final UserData" scenario.

Network does not respond to ClientSessionSetUpRequest

If timer tMsg 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 using the same transactionId, the ClientSessionSetUpConfirm message is received with that transactionId (or anytime a ClientSessionSetUpConfirm is received with an unknown transactionId) the Client shall send a ClientSessionReleaseRequest 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 rsnClNoSession.

4.8.1.2 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. No uuData shall be sent. The following response codes shall apply:

- **rspNoCalls** - Indicates that the Network is unable to accept new sessions.
- **rspInvalidClientId** - Indicates that the Network rejected the request due to an invalid clientId.
- **rspInvalidServerId** - Indicates that the Network rejected the request due to an invalid serverId.

Step 8 (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.

4.8.1.3 Server Rejects Server Session Indication

Step 6 (Server):

If the Server is unable to accept the session request, it 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. UserData() may be included to be passed to the Client. The following response codes shall apply:

- **rspNoCalls**- Indicates that the Server is unable to accept new sessions.
- **rspInvalidClientId**- Indicates that the Server rejected the request due to an invalid clientId.
- **rspServiceUnavailable** - Indicates that the Server could not be provide the requested service.

Step 7 (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 UserData() received in the ServerSessionsetUpResponse shall be included in this message.

Step 8 (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 the session establishment procedure.

4.8.1.4 Client Has Final UserData()

Step 8 (Client):

If the Client has UserData() 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.

Step 9 (Network):

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

Step 10 (Server):

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

4.8.1.5 Client Initiates Early Release

Early release refers to the release procedure that is invoked prior to end-to-end establishment of the session (or the receipt by the User of the first confirm message). Early Release makes use of the ClientSessionReleaseRequest and the ServerSessionReleaseRequest messages. Because early release represents a session abort by either the Client or the Server, it is customary that the procedure may make use of the transactionId and/or the sessionId as the association reference on a link. In some cases (e.g., a session establishment request by either a Client or a Server in which the sessionId is assigned by the source) both the transactionId and the sessionId are known and are available to identify the association reference. If this is the case, the sessionId shall take precedence. If the sessionId is assigned by the Network, the Early Release messages shall contain the sessionId set to "0" and the transactionId shall be used to identify the reference.

Step 1 (Client):

After transmitting a ClientSessionSetupRequest to the Network a Client may cancel the request prematurely, prior to the receipt of the first confirm message from the Network, by transmitting a ClientSessionReleaseRequest message. The value of the transactionId will be set equal to the value in the ClientSessionSetup message. The message will contain a sessionId which value will be set to the value in the ClientSessionSetupRequest. The reason field shall indicate why the session is being canceled. On sending the ClientSessionReleaseRequest message the Client starts timer tMsg.

On expiration of timer tMsg, the ClientSessionReleaseRequest message will be re-transmitted. If no response is received after the second expiration period the Client shall release all resources and no further action will be taken.

Step 2 (Network):

On receipt of the ClientSessionReleaseRequest message the Network will do one of the following:

1. If the value of the sessionId corresponds to an existing session and if the Network has begun session establishment to the Server, the Network shall release all resources allocated to the session and send a ServerSessionReleaseIndication message to the Server. The value of the sessionId shall be identical to the sessionId received from the Client. The value of the reason field shall indicate why the session is being released. If the Network has not yet began session establishment to the Server, no indication shall be sent to the Server. After sending the ServerSessionReleaseIndication, server starts timer tMsg. On expiration of timer tMsg, the ServerSessionReleaseIndication message will be re-transmitted. If no response is received after the second expiration period the Network shall release all resources and no further action will be taken toward the Server.
2. If the value of the sessionId is "0" the Network shall analyze the transactionId. If the value of the transactionId corresponds to the value of the transactionId received in the ClientSessionSetupRequest to which the Network has not yet responded, and if the Network had began session establishment to the Server, the Network shall release all resources allocated to the session and send a ServerSessionReleaseIndication message to the Server. The value of the sessionId shall be identical to the sessionId sent in the ServerSessionSetupIndication. The value of the reason field shall indicate why the session is being released. If the Network had not yet began session establishment to the Server, no indication will be sent to the Server. After sending the ServerSessionReleaseIndication, server starts timer tMsg. On expiration of timer tMsg, the ServerSessionReleaseIndication message will be re-transmitted. If no response is received after the second expiration period the Network shall release all resources and no further action will be taken toward the Server.

The Network shall then send a ClientSessionReleaseConfirm message to the Client.

Step 3 (Client):

On receipt of the ClientSessionReleaseConfirm message timer tMsg will be stopped and the Client shall abandon the session.

4.8.1.6 Server Does not respond to serverSessionSetUpIndication

Step 7 (Network):

If the server does not respond before the tMsg timer expires, the Network shall terminate the session and send the ClientSessionSetUpConfirm message to the client with the reason field set to RspSeNoResponse.

Step 8 (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 the session establishment procedure.

4.8.1.7 Network Rejects Server's Resource AllocationStep 7 (Network):

If the server is unable to accept any of the resources for the session, it shall terminate the session and send the ClientSessionSetUpConfirm message to the client with the reason field set to RspNeResourceFailed. The Network shall also initiate a session release procedure to the server for this session.

Step 8 (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 the session establishment procedure.

4.8.1.8 Client Unable to Use Resources

Step 8 (Client):

If the Client is unable to use any of the resources assigned to the session, it shall terminate the session using the session release procedure. The reason code shall be set to RsnClRejResource.

4.8.2 Client Session Release Command Sequence

Figure 4-7 illustrates the normal procedure for session release initiated by the Client.

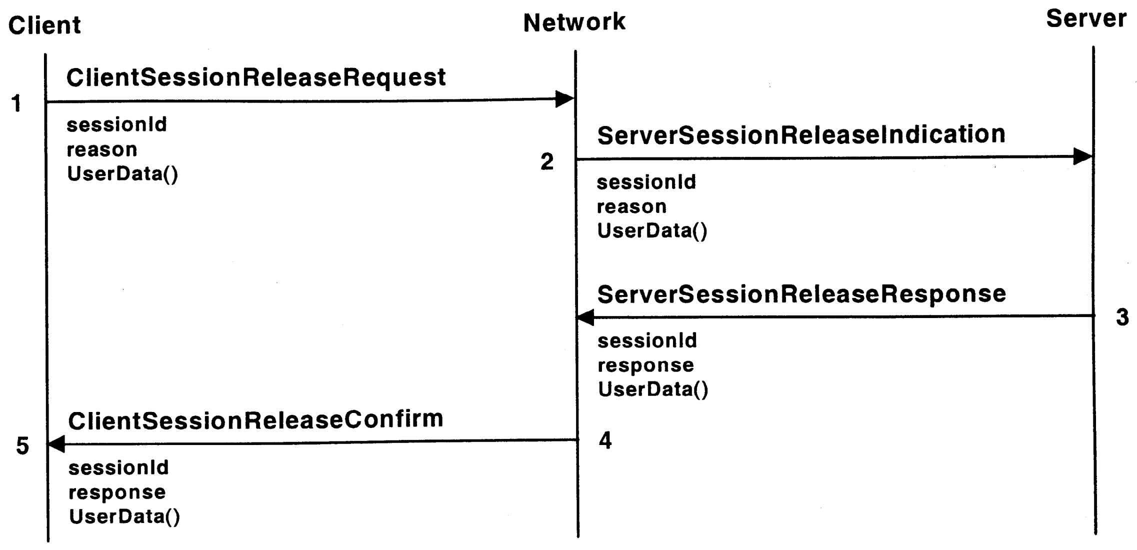


Figure 4-7 Scenario for Client initiated session release command sequence

4.8.2.1 Client Initiates Release Request

Step 1 (Client):

To start the procedure for releasing an existing session, the Client shall send ClientSessionReleaseRequest 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. Upon sending the ClientSessionReleaseRequest message, the Client shall no longer use any of the resources assigned to the session.

Step 2 (Network):

On receipt of ClientSessionReleaseRequest, the Network shall verify that the value of the sessionId field corresponds to an existing session which belongs to that Client. If the sessionId is valid and owned by the Client, the Network shall send ServerSessionReleaseIndication 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.

If the Network determines that the sessionId received in ClientSessionReleaseRequest 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 ServerSessionReleaseIndication, 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 ServerSessionReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. At this point, the Server shall consider the session to be terminated. If the session is connected to a continuous feed session, the Server shall release only any additional resources which were allocated to the session. The continuous feed session shall not be affected.

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 ServerSessionReleaseResponse, the Network shall release all resources assigned to the session and send ClientSessionReleaseConfirm to the Client. If the session is connected to a continuous feed session, the Network shall release only any additional resources which were allocated to the session. The continuous feed session shall not be affected.

Step 5 (Client):

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

4.8.2.2 Network Rejects Client Release Request

Step 2 (Network):

If the Network determines that the value of the sessionId field received in ClientSessionReleaseRequest is invalid, the Network shall send ClientSessionReleaseConfirm to the Client. The value of the sessionId field shall be identical to the value received in ClientSessionReleaseRequest, and the value of the reason parameter shall be set to indicate that the sessionId is invalid. After sending ClientSessionReleaseRequest, 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 response codes may be used in the ClientSessionReleaseConfirm message.

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

Step 3 (Client):

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

4.8.2.3 Server Rejects Server Release Indication

Step 3 (Server):

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

Step 4 (Network):

On receipt of ServerSessionReleaseResponse, the Network shall release all resources assigned to the session. The Network shall then send ClientSessionReleaseConfirm to the Client. The value of the sessionId field shall be identical to the value received in ServerSessionReleaseResponse because the Network previously validated it on receipt of ClientSessionReleaseRequest. The value of the reason field shall be set to rsnOK to indicate that the session has been released. After sending ClientSessionReleaseConfirm, the Network may initiate an audit with the Server.

Step 5 (Client):

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

4.8.3 Client Initiated Status Command Sequence

Figure 4-8 illustrates the procedure used by the Client for issuing status request messages to the Network. Refer to Table 4-61 for different types of status messages.

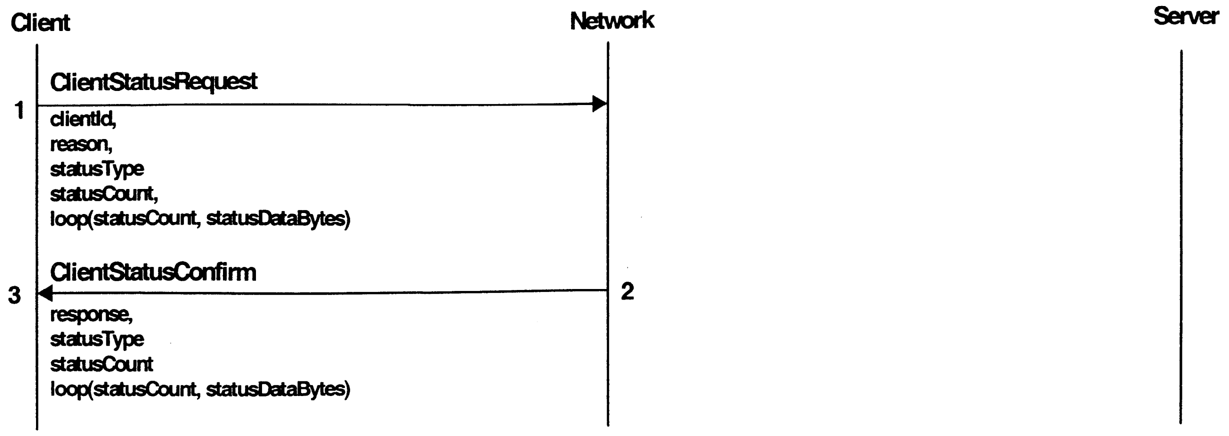


Figure 4-8 Scenario for Client-Initiated Network Status command sequence

Step 1 (Client)

The Client initiates a session status message by issuing a ClientStatusRequest message.

Step 2 (Network)

The Network receives the ClientStatusRequest message. If the Network can answer it, it generates a ClientStatusConfirm. If it wants to first get an answer from the Server, the Network issues a ServerStatusIndication message and waits for the ServerStatusResponse before issuing the ClientStatusConfirm to the Client.

Step 3 (Client)

The Client receives the ClientStatusConfirm message with the information it requested. For interpretation of the statusByte fields, which depend on the value of statusType, refer to Table 4-61.

4.9 Server Initiated Command Sequences

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

- 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 release command sequence.
- Server Initiated Continuous feed session release command sequence.

4.9.1 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 either the Network or 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 connection is assigned an individual sessionId which allows each Client connection to be tracked individually. Additional resources may be added to the individual session which is connected to the CFS. These additional resources are exclusive to the Client which owns the session.

Figure 4-9 describes the sequence of events that occur during a Server Continuous Feed Session set-up.

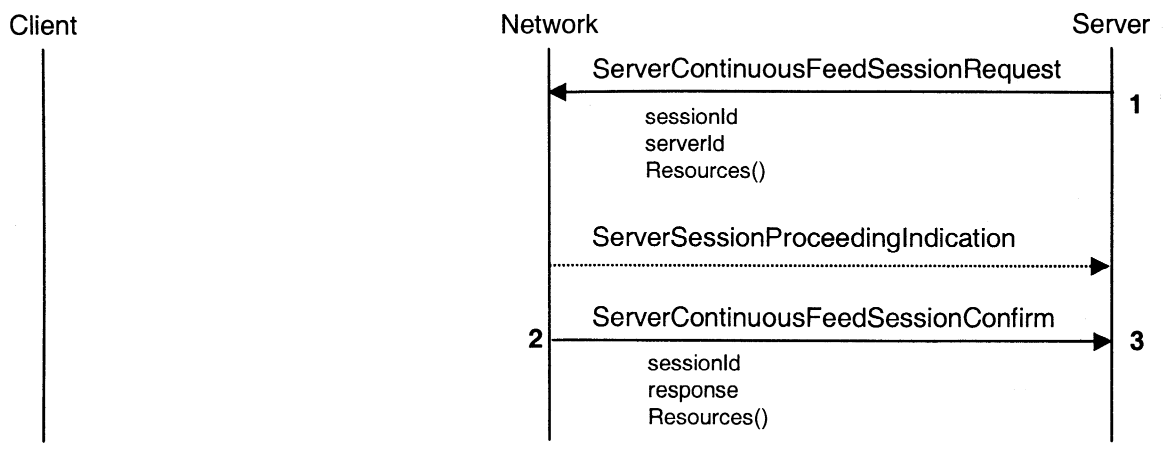

Client
Network
Server
ServerContinuousFeedSessionRequest
1
sessionId
serverId
Resources()
ServerSessionProceedingIndication
ServerContinuousFeedSessionConfirm
2
3
sessionId
response
Resources()

Figure 4-9 Scenario for Server initiated Continuous Feed Session Set-Up command sequence

4.9.1.1 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. If the User is responsible for assigning the sessionId, the value of the sessionId field shall be selected by the Server. If the Network is responsible for assigning the sessionId, this field shall be set to 0. The value of the serverId field shall identify the Server which will deliver the CFS. The Server shall set the value of the resourceCount field to the number of resources required to initially establish the session. For each resource requested, the Server shall include a resourceDescriptor in the request.

Step 2 (Network):

On receipt of the ServerContinuousFeedSessionRequest, if the Network is able to accept the new session, it shall attempt to assign any requested resources. The Network shall issue a ServerContinuousFeedSessionConfirm message with the response code set to indicate the status of the session. If the session request is rejected, the Network shall consider the session terminated. If the session is accepted, the Network shall consider the session active.

Step 3 (Server):

On receipt of ServerContinuousFeedSessionConfirm with the response set to indicate that the request was accepted, 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. If the Server cannot use any of the assigned resources, it may terminate the session by using the normal session release procedures. If the response field indicates that the session was rejected, the Server shall consider the session terminated.

4.9.2 Server Add Resource Command Sequence

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

Figure 4-10 illustrates the normal procedure for adding new resources to an existing session.

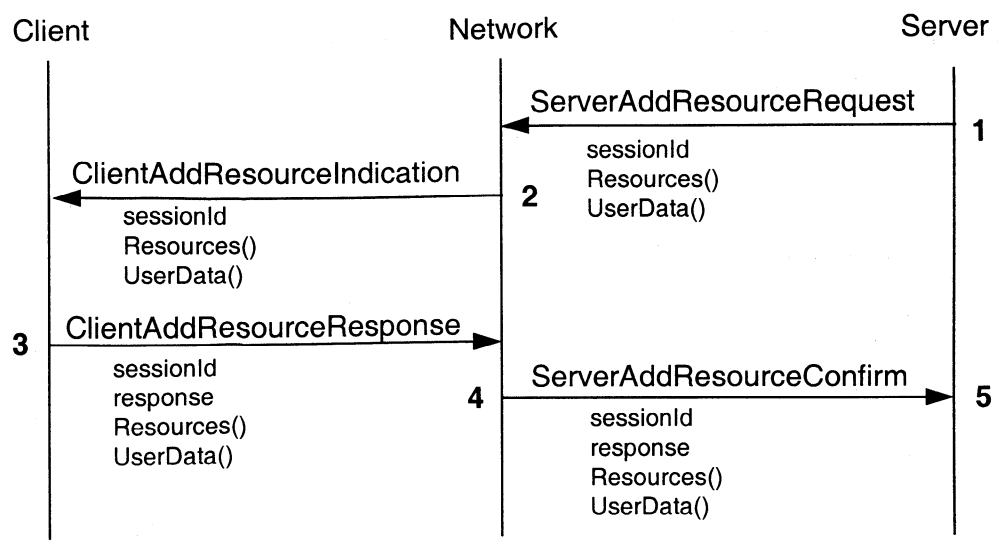


Figure 4-10 Scenario for Adding Resources to a Session command sequence

4.9.2.1 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 value of the resourceCount field shall indicate the number of resources to be added to the session. There shall be one resource descriptor for each resource to be added to the session.

Step 2 (Network):

Upon receipt of the ServerAddResourceRequest, the network shall attempt to assign the requested resources. If the network is able to complete the request, it shall send the ClientAddResourceIndication to the client. This message shall contain the sessionId of the session to which the resources are being added and the client view of the resources.

If the Network is not able to complete the request, it shall send the ServerAddResourceConfirm message to the Server with the response field set to indicate that the request failed. The Network shall consider the add resource request to be terminated at this point.

Step 3 (Client):

Upon receipt of the ClientAddResourceIndication message, the Client shall begin using the resources which are allocated for the session. The Client shall send a ClientAddResourceResponse message to the Network with the response field set to indicate if the add resource request succeeded. If the request succeeded, the Client shall begin using the resources immediately. If the request failed, the Client shall consider the add resource procedure to be terminated.

Step 4 (Network):

On receipt of ClientAddResourceResponse message, the Network shall send the ServerAddResourceConfirm message to the Server. If the response field indicates that the Client was unable to use the resources, the Network shall release the resources and consider the add resource procedure terminated. If the Client was able to use the resources, the Network shall consider the add resource procedure completed and the new resources to be active.

Step 7 (Server):

On receipt of ServerAddResourceConfirm, if the response indicates that the add resource procedure was successful, the Server shall consider the additional resources to be committed to the session. If the response indicates that the add resource procedure failed, the Server shall consider the procedure terminated and shall not use any of the requested resources.

4.9.3 Server Session Delete Resource Command Sequence

Figure 4-11 illustrates the normal procedure for deleting resources from an existing session.

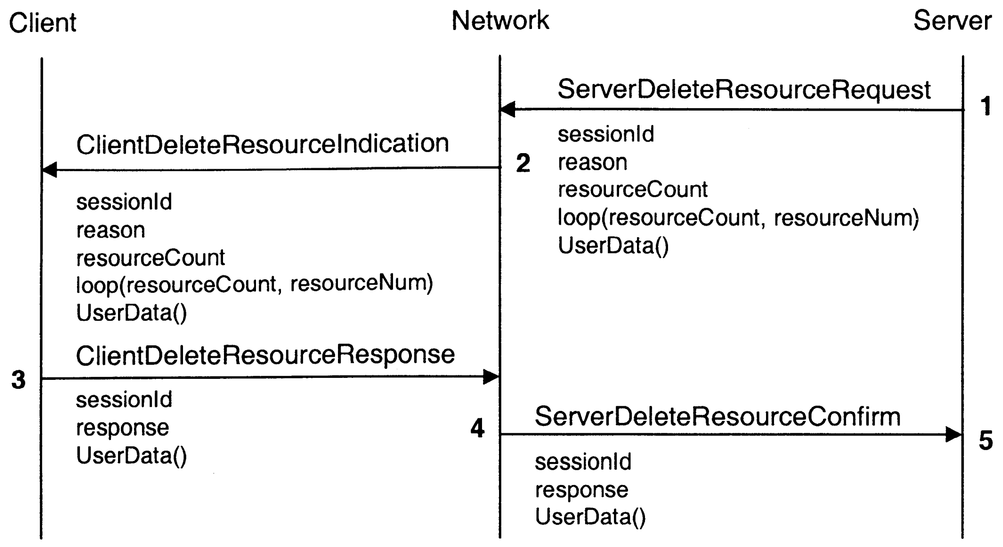

Client
Network
Server
ServerDeleteResourceRequest
1
sessionId
reason
resourceCount
loop(resourceCount, resourceNum)
UserData()
ClientDeleteResourceIndication
2
sessionId
reason
resourceCount
loop(resourceCount, resourceNum)
UserData()
ClientDeleteResourceResponse
3
sessionId
response
UserData()
4
ServerDeleteResourceConfirm
5
sessionId
response
UserData()

Figure 4-11 Scenario for Deleting Resources from a Session command sequence

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 resourceNum fields present in the 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 deactivate the resources and send the ClientDeleteResourceIndication message to the Client. 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 resourceNum 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 resourceNum fields shall be included in the message. The Network shall send only the Client view resourceNum fields in this 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, the Network shall send the ServerDeleteResourceConfirm message to the Server indicating the reason that the request was denied. At this point, the Network shall consider the delete resource procedure terminated.

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. 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, the Client shall send the ClientDeleteResourceResponse message to the Network with the response code set to indicate this. At this point, the Client shall consider the delete resource procedure to be terminated.

Step 4 (Network):

On receipt of ClientDeleteResourceResponse, the Network shall send ServerDeleteResourceConfirm to the Server. 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.

4.9.4 Server Session Release Command Sequence

Figure 4-12 illustrates the normal procedure for session release initiated by the Server.

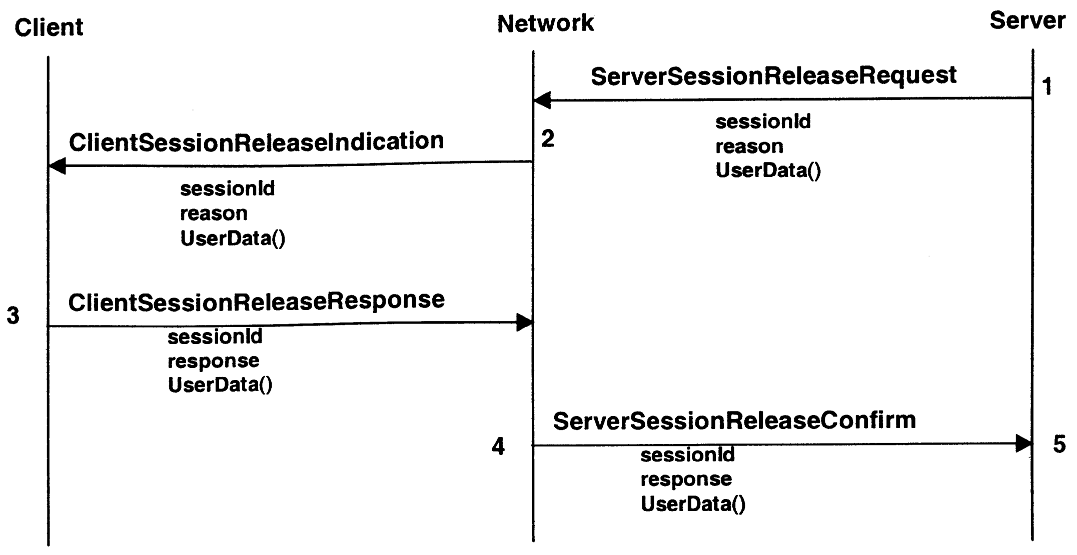


Figure 4-12 Scenario for Server initiated session release command sequence

4.9.4.1 Server Initiates Release Request

Step 1 (Server):

To start the procedure for releasing an existing session, a Server shall stop using all resources related to the session send ServerSessionReleaseRequest 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.

Step 2 (Network):

On receipt of ServerSessionReleaseRequest, 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 ClientSessionReleaseIndication 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.

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

Step 3 (Client):

On receipt of ClientSessionReleaseIndication, 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 ClientSessionReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. At this point, the Client shall consider the session to be terminated.

If the Client determines that the sessionId is invalid, flow shall shift to the "Client Rejects ClientSessionReleaseIndication" scenario.

Step 4 (Network):

On receipt of ClientSessionReleaseResponse, the Network shall release all resources assigned to the session and send ServerSessionReleaseConfirm to the Server. The values of the sessionId field shall be identical to the values received in ClientSessionReleaseResponse. At this point, the Network shall consider the session to be terminated.

Step 5 (Server):

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

4.9.4.2 Network Rejects Server Release Request

Step 2 (Network):

If the Network determines that the value of the sessionId field received in ServerSessionReleaseRequest is invalid or that the Server is not the owner of the session, the Network shall send ServerSessionReleaseConfirm to the Server. The value of the sessionId field shall be identical to the value received in ServerSessionReleaseRequest, and the value of the response 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 ServerSessionReleaseConfirm, the Network may take other actions such as initiating an audit with the Server.

Step 3 (Server):

On receipt of ServerSessionReleaseConfirm, 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.

4.9.4.3 Client Rejects Client Release Indication

Step 3 (Client):

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

Step 4 (Network):

On receipt of ClientSessionReleaseResponse which indicates that the session is invalid, the Network shall release resources assigned to the session. The Network shall send ServerSessionReleaseConfirm to the Server. The value of the sessionId field shall be identical to the value received in ClientSessionReleaseResponse since the Network previously validated it on receipt of ServerSessionReleaseRequest. The value of the reason field shall indicate that a procedure error has occurred with the Client. At this point, the Network shall consider the release procedure as complete and release all resources associated with the session. After sending ServerSessionReleaseConfirm, the Network may initiate an audit with the Client.

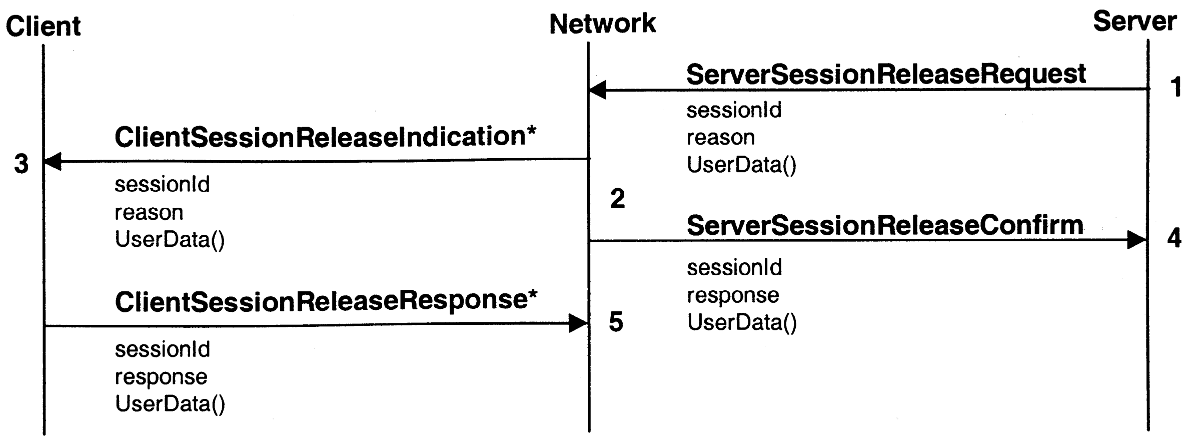
Step 5 (Server):

On receipt of ServerSessionReleaseConfirm, the Server shall consider the release procedure as completed and shall release all resources assigned to the session.

4.9.5 Server Continuous Feed Session Release Command Sequence

After a continuous feed session has been established, it may be terminated using the Server Release sequence. A Session Release command sequence for a continuous feed session may be initiated only by the Server.

Figure 4-13 describes the sequence of events that occur during a Server initiated release of a continuous feed session. When the continuous feed session is torn down, any sessions which are connected to the continuous feed session shall first be torn down by the Network.



* - Indicates that the Network repeats this procedure for each session connected to the CFS.

Figure 4-13 Scenario for Server initiated CFS Release command sequence

4.9.5.1 Server Initiates Continuous Feed Session Release Request

Step 1 (Server):

To start the procedure for releasing a Continuous Feed Session, a Server shall send ServerSessionReleaseRequest 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.

Step 2 (Network):

On receipt of ServerSessionReleaseRequest, 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 ClientSessionReleaseIndication for each Session 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 be set to RsnClSessionRelease to indicate that the session is being released because the session to which the session is connected has been released by the Server. The Network shall not wait for the ClientSessionReleaseResponse message to be received from each Clients before confirming the CFS release request to the server. After sending all of the ClientSessionReleaseIndication messages, the Network shall release all resources assigned to the continuous feed session and send ServerSessionReleaseConfirm to the Server. The value of the sessionId shall be the sessionId of the continuous feed session. At this point, the Network shall consider the continuous feed session to be terminated.

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

Step 3 (Client):

On receipt of ClientSessionReleaseIndication, 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 ClientSessionReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. At this point, the Client shall consider the session to be terminated.

If the Client determines that the sessionId is invalid, flow shall shift to the "Client Rejects ClientSessionReleaseIndication" scenario.

Step 4 (Server):

On receipt of ServerSessionReleaseConfirm, the Server shall release all resources assigned to the session. At this point, the Server shall consider the continuous feed session to be terminated. The Server shall also consider all sessions which were connected to the continuous feed session to be terminated and shall release all resources allocated for those sessions.

Step 5 (Network): 1

On receipt of each ClientSessionReleaseResponse, 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. If the Network does not receive a ClientSessionRelease Response message for one or more of the Client sessions before tMsg expires, it shall release the resources for the session and consider the session to be terminated. The Network may initiate an audit with the Client to determine its status.

4.9.5.2 Network Rejects Server Release Request

Step 2 (Network):

If the Network determines that the value of the sessionId field received in ServerSessionReleaseRequest is invalid or that the Server is not the owner of the session, the Network shall send ServerSessionReleaseConfirm to the Server. The value of the sessionId field shall be identical to the value received in ServerSessionReleaseRequest, 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 ServerSessionReleaseConfirm, the Network may take other actions such as initiating an audit with the Server.

Step 3 (Server):

On receipt of ServerSessionReleaseConfirm the Server shall terminate the session release procedure. The Server may take other actions such as initiating an audit with the Network.

4.9.5.3 Client Rejects Client Release Indication

Step 3 (Client):

If the Client determines that the value of the sessionId field received in ClientSessionReleaseIndication is invalid, the Client shall send ClientSessionReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received in ClientSessionReleaseIndication, and the value of the response field shall be set to RspClNoSession. At this point, the Client shall consider the release procedure terminated. After sending ClientSessionReleaseResponse, the Client may take other actions such as initiating an audit with the Network.

Step 4 (Network):

On receipt of ClientSessionReleaseResponse which indicates that the session is invalid, the Network shall release all resources assigned to the session. At this point, the Network may consider the release procedure as complete. The Network may initiate an audit with the Client.

4.9.6 Server Status Command Sequence

Figure 4-14 below illustrates the procedure used by the Server for issuing different status request messages. Refer to Table 4-61 for different types of status messages.

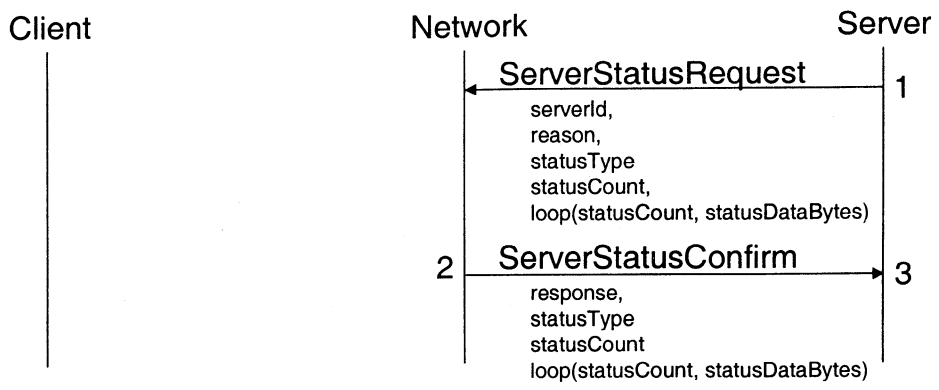


Figure 4-14 Scenario for Server initiated Network Status Command Sequence

Step 1 (Server)

The Server initiates a session status message by issuing a ServerStatusRequest message. The statusType field indicates the type of status which is being requested. The statusCount and statusDataBytes field shall contain any additional information required to complete the status request.

Step 2 (Network)

The Network receives the ServerStatusRequest message. If the Network is able to provide the requested status, it generates a ServerStatusConfirm. If it must first request a status from the Client, the Network issues a ClientStatusIndication message and waits for the ClientStatusResponse before issuing the ServerStatusConfirm to the Server.

Step 3 (Server)

The Server receives the ServerStatusConfirm message with the information it requested. For interpretation of the statusByte fields, which depend on the value of statusType, refer to Table 4-61.

4.9.7 Server Session Forward Command Sequence

Figure 4-15 describes the Session Forwarding scenario:

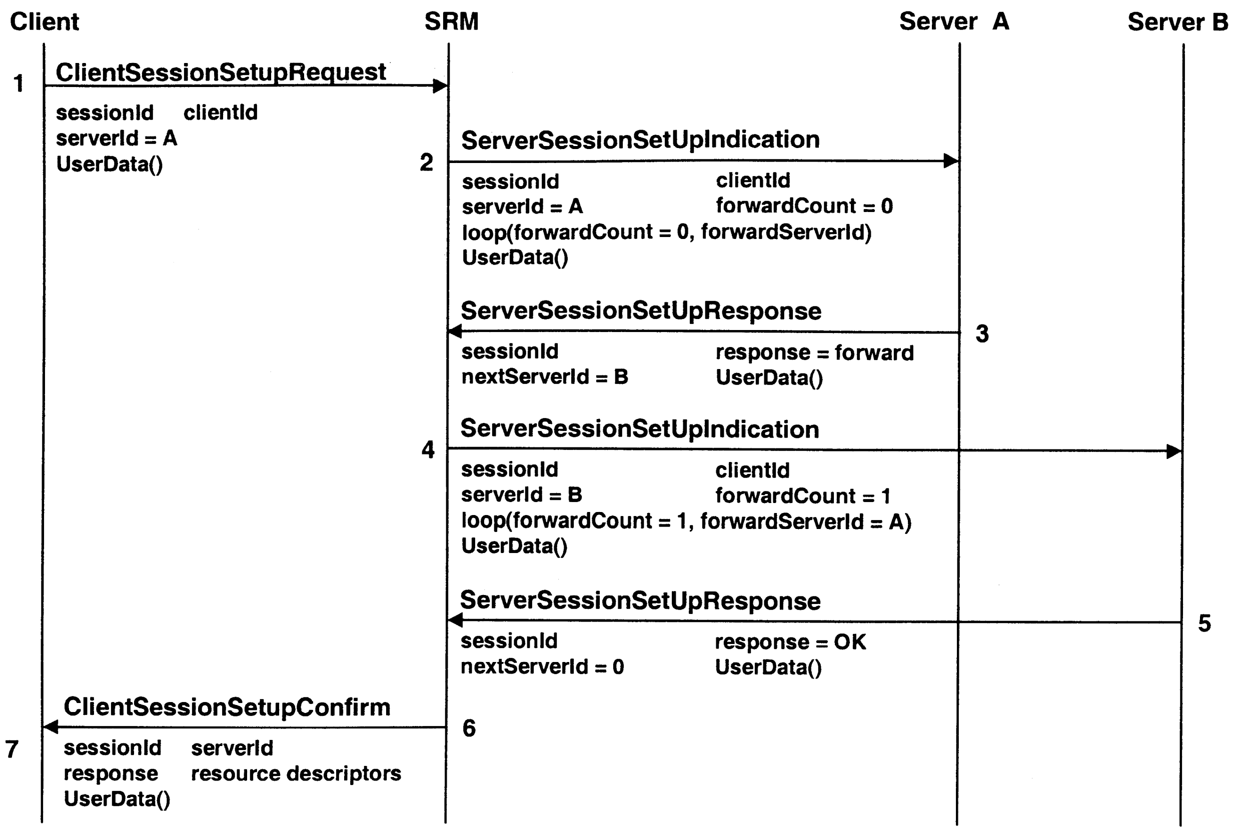

Client
SRM
Server A
Server B
1
ClientSessionSetupRequest
sessionId clientId
serverId = A
UserData()
2
ServerSessionSetUpIndication
sessionId clientId
serverId = A forwardCount = 0
loop(forwardCount = 0, forwardServerId)
UserData()
ServerSessionSetUpResponse
3
sessionId response = forward
nextServerId = B UserData()
4
ServerSessionSetUpIndication
sessionId clientId
serverId = B forwardCount = 1
loop(forwardCount = 1, forwardServerId = A)
UserData()
ServerSessionSetUpResponse
5
sessionId response = OK
nextServerId = 0 UserData()
ClientSessionSetupConfirm
6
7
sessionId serverId
response resource descriptors
UserData()

Figure 4-15 Scenario for Session Forward Command Sequence

4.9.7.1 Client Initiates Session Set-Up

Step 1 (Client):

The Client sends the standard ClientSessionSetupRequest to the SRM as described in the Client initiated Session Set-Up Command Sequence. The serverId field is set to Server A.

Step 2(Network):

The SRM receives the ClientSessionSetupRequest message from the Client, performs the necessary actions and notifies Server A of the incoming session by sending a ServerSessionSetUpIndication message as described in the Client initiated Session Set-Up Command Sequence. Since this is the initial request, the forward count is set to 0 and no forwardServerIds are sent.

Step 3(Server A):

Upon receipt of the ServerSessionSetUpIndication from the SRM, Server A may determine that it is necessary to forward this session. If the forward count received in the message does not exceed the maxForwardCount established by the Network in U-N Configuration or other means, the Server may proceed with the forward. The Server shall establish no session context, make no connections, allocate no resources, or start any applications on behalf of this session. The Server shall make the decision to forward based on its state (overload, time of day, etc.), information contained within the ServerSessionSetUpIndication (clientId, UserData(), etc.), or other policy implemented at the Server.

To forward this session the Server sends a ServerSessionSetUpResponse to the SRM with the response field set to rspForward. The nextServerId shall be set to the userId of Server B to indicate that this is the Server to which the session should be forwarded.

Step 4(Network):

The SRM detects the session forward based on the response code of *rspForward*. If the forward can not be completed by the SRM (Server B is invalid, maxForwardCount exceeded, etc.), flow shall shift to “Network Rejects Forward” scenario. If the requested forward is accepted, the SRM shall send a ServerSessionSetUpIndication to Server B with the serverId set to Server B, the forwardCount is incremented to 1 to indicate the number of times that the session has been forwarded and the forwardServerId list shall contain the ID of Server A.

Step 5 (Server B):

Upon receipt of the ServerSessionSetUpIndication by Server B, sends the ServerSessionResponse to the SRM as described in the Client initiated Session Set-Up Command Sequence.

Step 6 (Network):

Upon receipt of the ServerSessionResponse by the SRM, the Network processes the set-up as described in the Client initiated Session Set-Up Command Sequence.

Step 7 (Client):

Upon receipt of the ClientSessionSetUpConfirm, the Client shall processes the set-up as described in the Client initiated Session Set-Up Command Sequence.

4.9.7.2 Network Rejects Forward

Step 5 (Network):

If the forward requested in the ServerSessionSetupResponse can not be processed, the SRM shall terminate the session by sending a ClientSessionSetupConfirm with the response field set to rspNeForwardFailed to indicate the failure. The Network shall also terminate the session at the Server by issuing a release. At this point, the Network shall consider the session terminated.

4.9.8 Server Session Transfer Command Sequence

A Server may request that a session be transferred to another session gateway. Figure 4-16 describes the Session Transfer scenario. Note: the User-to-User clause of this part of ISO/IEC 13818 does not presently support the session transfer scenario.

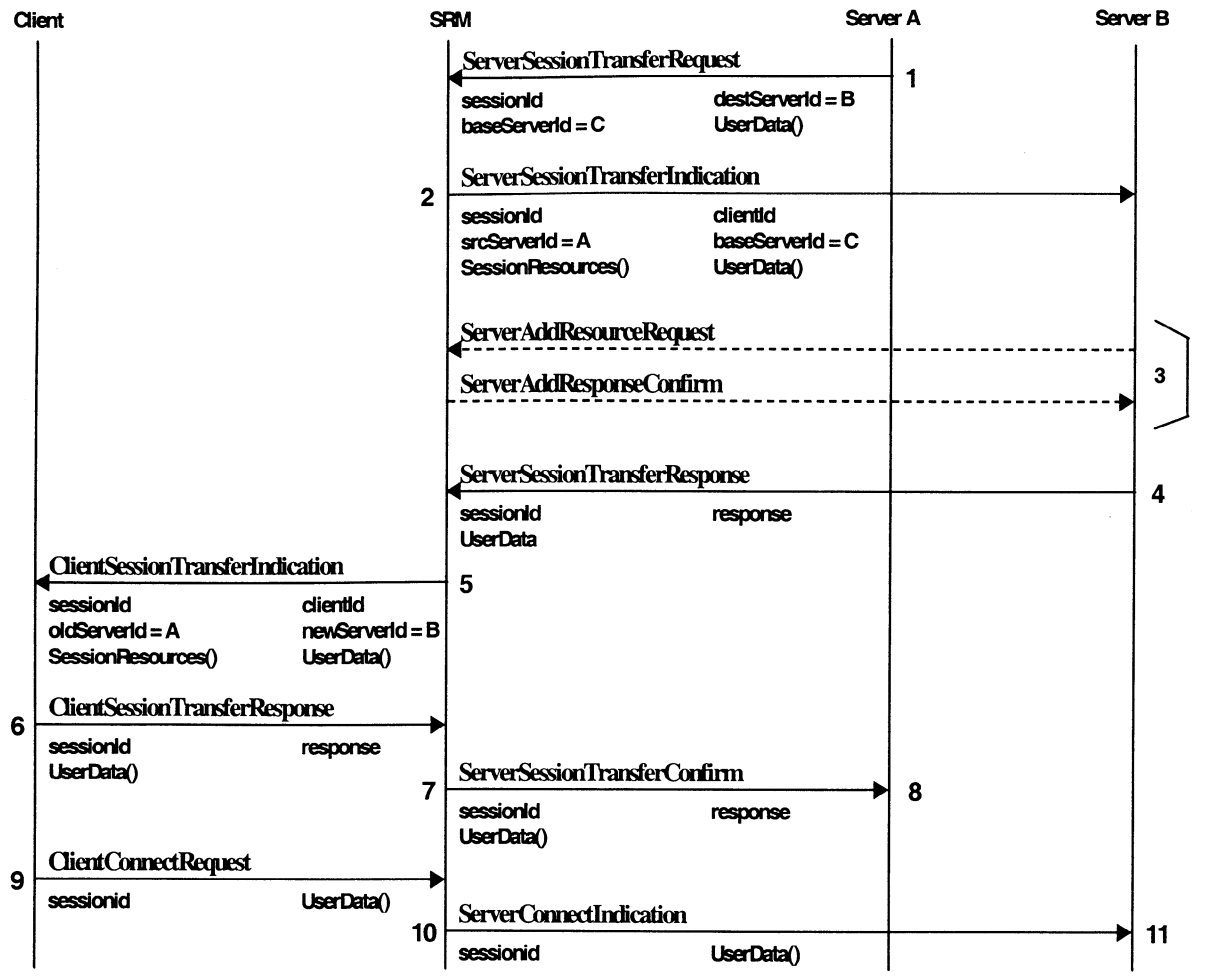


Figure 4-16 Scenario for Session Transfer Command Sequence

4.9.8.1 Server A Initiates Session Transfer

Step 1 (Server A):

This command sequence is initiated when Server A decides it is necessary to transfer control of this session to Server B. Server A sends a ServerSessionTransferRequest message to the SRM to request the transfer. The sessionId field shall indicate the session that is being transferred, the destServerId field shall indicate the Server that the session is being transferred to, the baseServerId field shall indicate the base serverId (this is the Server that the session shall be transferred to when Server B has finished with the session.)

Step 2 (Network):

Upon receipt of the ServerSessionTransferRequest. if the transfer can not be completed by the SRM (i.e. Server B is invalid), flow shall shift to "Network Rejects Transfer" scenario. If the Network can accept the transfer, the SRM sends a ServerSessionTransferIndication message to Server B to indicate the transfer request. The sessionId field shall contain the sessionId that is being transferred, the clientId field shall identify the Client which the session is active on, the srcServerId field shall identify the Server that the session is active on, the baseServerId field shall identify the Server to which the session belongs. The SessionResources() structure contains any resources which are allocated to the session.
Step 3 (Server B):

Upon receipt of the ServerSessionTransferRequest, Server B shall validate the transfer parameters. If Server B rejects the transfer request, flow will shift to "Server B Rejects the Transfer Request" scenario. If Server B accepts the transfer it then determines the resources required to support the session. Server B shall use the ServerAddResourceRequest message to request the addition of resources. If the resources can not be allocated, the flow shifts to "Server B Unable to Allocate Resources for Transfer" scenario.

Step 4 (Server B):

After all necessary resources for the session have been allocated, Server B sends a ServerSessionTransferResponse to the SRM. The sessionId field shall identify the session that is being transferred and the response field shall be set to rspOK to indicate that Server B has successfully transferred the session.
Step 5 (Network):

Upon receipt of the ServerSessionTransferResponse, the SRM sends a ClientSessionTransferIndication to the Client. The sessionId shall identify the session that is being transferred, the clientId identifies the Client, the oldServerId field indicates the Server which the session is being transferred from, the newServerId field indicates the Server which the session is being transferred to, the SessionResources structure shall contain the client view of the session after the session is transferred.
Step 6 (Client):

Upon receipt of the ClientSessionTransferIndication, if the Client can accept the transfer, it shall release any resources no longer indicated as belonging to the session and begin using any new resources for the session. If the Client can not accept the transfer, flow shall shift to "Client Rejects Transfer" scenario.

Step 7 (Network):

Upon receiving the ClientSessionTransferResponse, the Network shall release all resources which are allocated to Server A. The SRM then sends a ServerSessionTransferConfirm message to Server A to confirm that the session has been transferred.
Step 8 (Server A):

Server A receives the ServerSessionTransferConfirm. At this point, Server A is no longer involved in the session and should release all resources allocated for this session.

Step 9 (Client):

The client shall also send the ClientConnectRequest message to the Network.

Step 10 (Network):

Upon receiving the ClientConnectRequest, the Network shall send a ServerConnectIndication message to Server B to confirm that the session has been transferred.
Step 11 (Server B):

Server B receives the ServerConnectIndication message. This indicates that the transfer has been accepted by the Network and the Client.

4.9.8.2 Network Rejects Transfer Request

Step 2 (Network):

If the transfer requested in the ServerSessionTransferRequest can not be processed, the SRM shall send the ServerSessionTransferConfirm with the response field set to rspNeTransferFailed to indicate that the transfer was rejected.

4.9.8.3 Server B Rejects the Transfer Request

Step 3 (Server B):

If Server B can not accept this transfer (client not accepted, overload, etc.), Server B rejects the transfer by sending the ServerSessionTransferResponse with the response code set to "rspSeTransferReject".

Step 4 (Network):

Upon receipt of the ServerSessionTransferResponse from Server B, the Network shall send a ServerSessionTransferConfirm to Server A with the response code set to "rspSeTransferReject".

Step 5 (Server A):

Upon receipt of the ServerSessionTransferConfirm, the Server shall consider the transfer to have failed and shall resume processing of the session.

4.9.8.4 Server B Unable to Allocate Resources for Transfer

Step 3 (Server B):

If Server B is unable to get the resources to accept this transfer, it shall reject the transfer by sending a ServerSessionTransferResponse with the response code set to "rspSeTransferNoResource". This response indicates to Server A that it may need to delete its resources before it can successfully transfer the session.

Step 4 (Network):

Upon the receipt of the ServerSessionTransferResponse from Server B, the Network shall send a ServerSessionTransferConfirm to Server A with the response code set to "rspSeTransferNoResources".

Step 5 (Server A):

Upon receipt of the ServerSessionTransferConfirm, Server A shall consider the transfer failed. The Server may release its resources for the session and attempt to transfer the session again.

4.9.8.5 Client Rejects Transfer

Step 6 (Client):

If for any reason the Client can not accept the transfer, the Client will respond with a ClientSessionTransferResponse with the response field set to "rspClTransferReject".

Step 7 (Network):

Upon receipt of the ClientSessionTransferResponse message, the SRM will clear the session on Server B using the Client Session Clear command sequence and send a ServerSessionTransferConfirm to Server A with the response code set to "rspSeTransferNoResource".

Step 8 (Server A):

Upon receipt of the ServerSessionTransferConfirm message, Server A shall consider the transfer failed.

4.9.9 Transferred Session Release

There are two cases for releasing a session that has been transferred.

- SRM is selecting sessionIds
- Server is selecting sessionIds

4.9.9.1 SRM is Selecting sessionIds

The Session Release procedures for a transferred session are the same as a non-transferred session. The SRM owns all of the sessionIds and guarantees uniqueness.

4.9.9.2 Server is Selecting sessionId

If the Server is selecting the sessionId, the sessionId contains the deviceId of the original server (Server A). Server A must be informed when the session terminates so that it knows that the sessionId is no longer in use. When a transferred session is released, (either by the Client or Server) a ServerSessionReleaseIndication will be sent to both the original server and the current Server. Figure 4-17 describes the Client initiated session release and Figure 4-18 describes the Server initiated session release. No message fields are shown because they are the same as described in the Session Release command sequences.

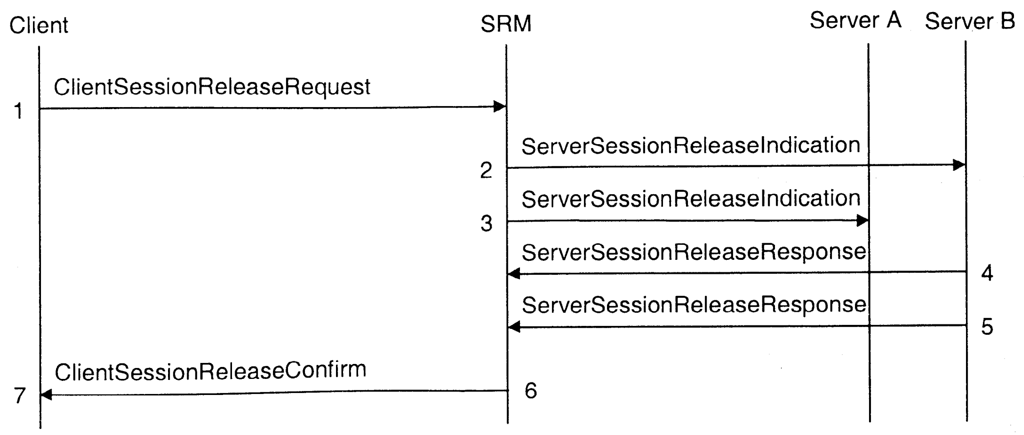

Client
SRM
Server A
Server B
1
ClientSessionReleaseRequest
2
ServerSessionReleaseIndication
3
ServerSessionReleaseIndication
ServerSessionReleaseResponse
4
ServerSessionReleaseResponse
5
7
ClientSessionReleaseConfirm
6

Figure 4-17 Scenario for Client initiated Transferred Session Release Command Sequence

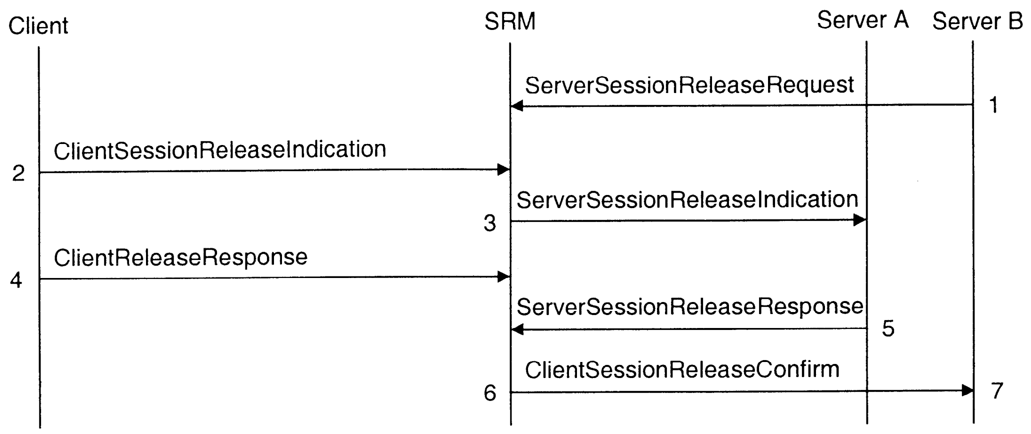

Client
SRM
Server A
Server B
ServerSessionReleaseRequest
1
2
ClientSessionReleaseIndication
3
ServerSessionReleaseIndication
4
ClientReleaseResponse
ServerSessionReleaseResponse
5
6
ClientSessionReleaseConfirm
7

Figure 4-18 Scenario for Server initiated Transferred Session Release Command Sequence

4.10 Network Initiated Command Sequences

The following Network initiated command sequences are defined in this subclause:

- Session release command sequence.
- Continuous feed session release command sequence.
- Client status request command sequence.
- Server status request command sequence.

4.10.1 Network Initiated Session Release Command Sequence

Figure 4-19 illustrates the normal procedure for session release initiated by the Network.

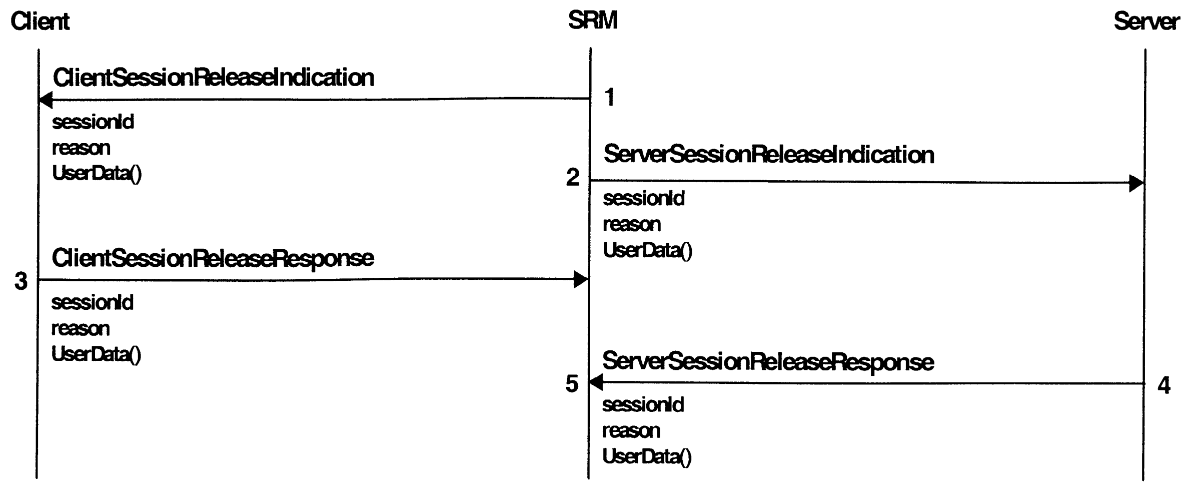


Figure 4-19 Scenario for Network initiated session release command sequence

4.10.1.1 Network Initiates Session Release

Step 1 (Network):

To start the procedure for releasing an existing session, the Network shall send ClientSessionReleaseIndication to the Client and send ServerSessionReleaseIndication 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.

Step 2 (Client):

On receipt of ClientSessionReleaseIndication, 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 ClientSessionReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. At this point, the Client shall consider the session to be terminated and may release any resources allocated to the session.

Step 3 (Server):

On receipt of ServerSessionReleaseIndication, 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 ServerSessionReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. At this point, the Server shall consider the session to be terminated and may release any resources allocated to the session.

Step 3 (Network):

On receipt of ClientSessionReleaseResponse, the Network shall release all Client interface resources assigned to the session.

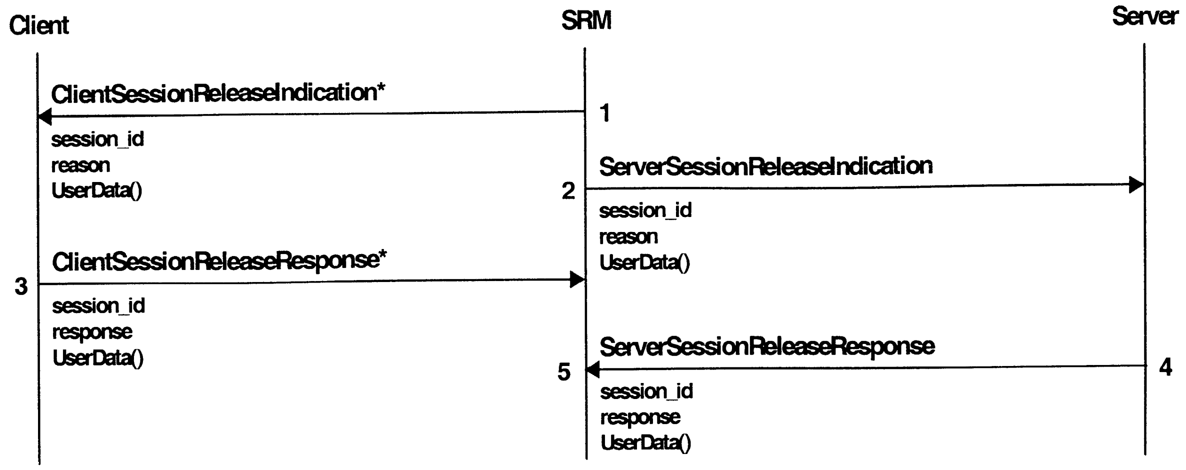
On receipt of ServerSessionReleaseResponse, the Network shall releases all Server interface resources assigned to the session.

After both the ClientSessionReleaseResponse and ServerSessionReleaseResponse have been received, the Network shall consider the session to be terminated.

4.10.2 Network Initiated Continuous Feed Session Release Command Sequence

The Network may initiate a continuous feed session release sequence using the Server Disconnect Indication message and the Client Disconnect Indication message.

Figure 4-20 describes the sequence of events that occur during a Network initiated continuous feed session release sequence.



* Indicates that this message sequence is repeated for each session that is connected to the continuous feed session.

Figure 4-20 Scenario for Network initiated CFS release command sequence

4.10.2.1 Network Initiates Continuous Feed Session Release

Step 1 (Network):

To start the procedure for releasing a continuous feed session, the Network shall send ClientSessionReleaseIndication to each Client which is connected to the continuous feed session and send ServerSessionReleaseIndication 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. The value of the reason field shall indicate why the Network is releasing the session.

Step 2 (Client):

On receipt of ClientSessionReleaseIndication, 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 ClientSessionReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. At this point, the Client shall consider the session to be terminated.

Step 3 (Server):

On receipt of ServerSessionReleaseIndication, 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 ServerSessionReleaseResponse to the Network. The value of the sessionId field shall be identical to the value received from the Network. At this point, the Server shall consider the session to be terminated and any sessions connected to the continuous feed session to be terminated.

Step 3 (Network):

On receipt of ClientSessionReleaseResponse, 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 ServerSessionReleaseResponse, the Network shall releases all Server interface resources assigned to the session. The Network shall consider the continuous feed session to be terminated.

4.10.3 Network Initiated Client Status Command Sequence

Figure 4-21 illustrates the procedure used by the Network for requesting the status of a Client. The particular type of status requested is determined by the statusType field. Possible status types are defined in Table 4-61.

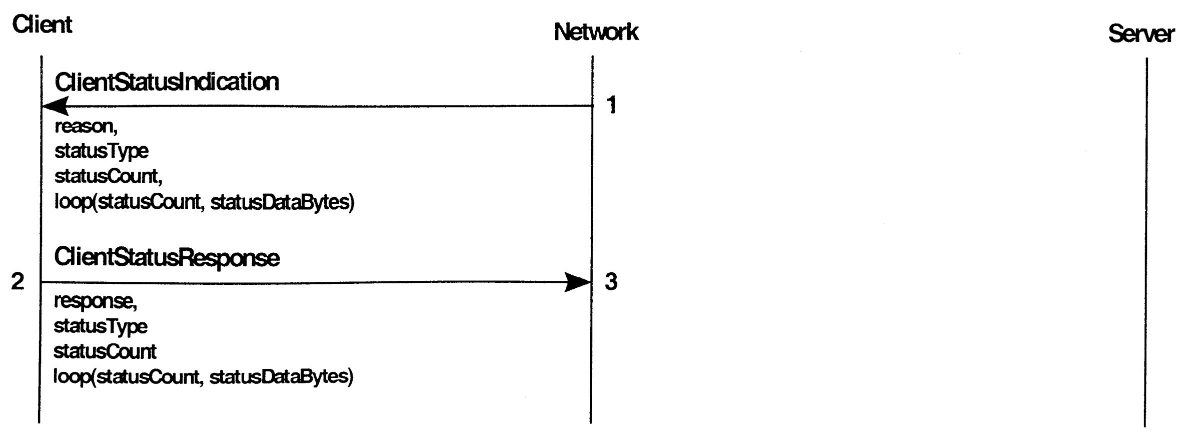


Figure 4-21 Scenario for Network Initiated Client Status command sequence

4.10.3.1 Network Initiates Client Status command sequence

Step 1 (Network):

The network initiates a session audit by issuing a ClientStatusIndication message. The Network sets statusType field to the desired status type. The reason field shall indicate the reason that the status is being requested. The statusCount and statusDataBytes fields shall be set to include any additional data which is required to complete the status request.

Step 2 (Client):

The Client receives the ClientStatusIndication message and sends a ClientStatusResponse to the Network. The response field shall indicate if the Client accepted the status request. If the Client accepts the request, the statusType field shall be set to the statusType received from the Network. The statusCount and statusDataBytes fields shall contain the status data for the requested status type. If the Client rejects the request, the statusCount shall be set to 0 and no statusDataBytes shall be sent.

Step 3 (Network):

The Network receives the ClientStatusResponse message which terminates the sequence.

4.10.4 Network Initiated Server Status Command Sequence

Figure 4-22 illustrates the procedure used by the Network for requesting a status from a Server. The particular type of status requested is determined by the statusType field. Possible status types are defined in Table 4-61.

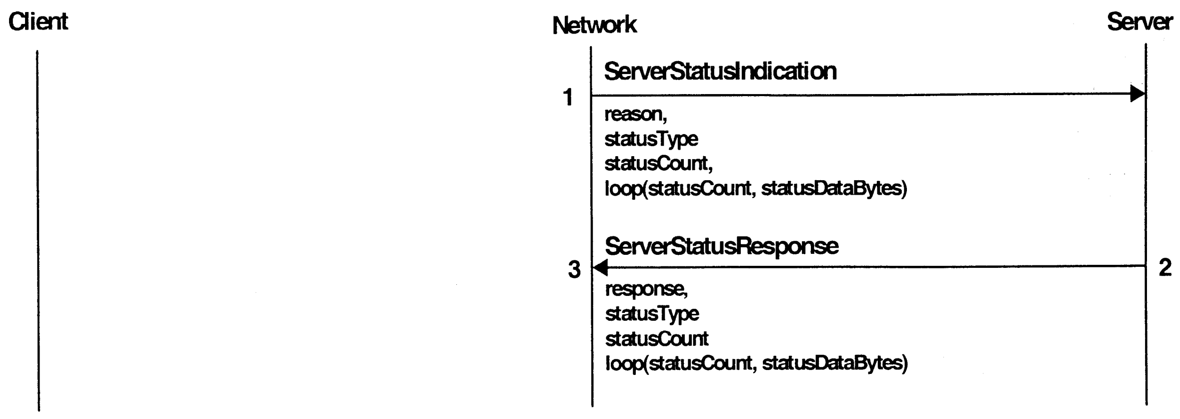


Figure 4-22 Scenario for Network Initiated Server Status command sequence

4.10.4.1 Network Initiates Server Status command sequence

Step 1 (Network):

The network initiates a session audit by issuing a ServerStatusIndication message. The Network sets statusType field to the desired status type. The reason field shall indicate the reason that the status is being requested. The statusCount and statusDataBytes fields shall be set to include any additional data which is required to complete the status request.

Step 2 (Server):

The Server receives the ServerStatusIndication message and sends a ServerStatusResponse to the Network. The response field shall indicate if the Server accepted the status request. If the Server accepts the request, the statusType field shall be set to the statusType received from the Network. The statusCount and statusDataBytes fields shall contain the status data for the requested status type. If the Server rejects the request, the statusCount shall be set to 0 and no statusDataBytes shall be sent.

Step 3 (Network):

The Network receives the ServerStatusResponse message which terminates the sequence.

4.11 Reset Procedures

The Reset procedure may be triggered from the Client, Server, or Network and shall be used for system recovery in instances where the state of the sessions are unknown. With this procedure it is possible to initiate a reset for one or more sessions simultaneously. In order to convey the nature of the reset condition, the Reset messages indicate the type of Reset as well as the particular reason for the condition. The inclusion of the userId in a reset message indicates that all sessions associated with that User are to be reset. The reason for the reset shall be indicated by reason data field.

Some example cases where the use of the Reset procedure is indicated are:

- Signaling anomalies detected by the DSM-CC signaling system. One such manifestation might be due to misalignment detected as a result of the session status procedures.
- Memory mutilation detected by the management system, e.g., losing of the association information between a sessionId and session resources.
- Start-up and restart of a Client, Server or SRM with the DSM-CC signaling system. The invocation of (re)start condition should reflect a major fault recovery (start-up) process. It should only be used in extraordinary situations by a maintenance activation process.

4.11.1 Client Initiated Reset Command Sequence

Figure 4-23 illustrates the procedure used by a Client for Reset.

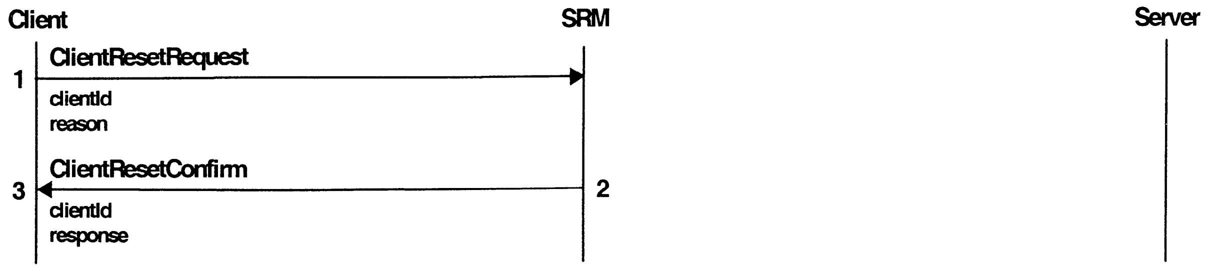


Figure 4-23 Scenario for Client initiated Reset message sequence

4.11.1.1 Client Initiates Reset command sequence

Step 1 (Client):

Following one of the conditions noted above, the Client attempts to re-synchronize the interface by transmitting a ClientResetRequest message to the Network. The message will contain the clientId parameter set to the value of the Client. All resources will be placed in the "idle" state and timer tMsg is set upon sending the ClientResetRequest.

No further messages will be sent while timer tMsg is running. With the exception of the ClientResetConfirm all messages received will be ignored.

On expiration of timer tMsg, the ClientResetRequest message will be re-transmitted. If no response is received after the second expired period the Reset procedure shall be terminated.

Step 2 (Network):

On receipt of the ClientResetRequest message the network shall clear all sessions for that client with "active" sessionIds. All timers shall be reset and all resources shall be released. Upon successful execution, an ClientResetConfirm will be sent.

If appropriate, the Network will attempt to re-synchronize the server by initiating the Reset procedure on the interface towards the server.

Step 3 (Client):

On receipt of the ClientResetConfirm message timer **tMsg** shall be stopped.

4.11.2 Server Initiated Reset Command Sequence

Figure 4-24 illustrates the procedure used by a Server for Reset.

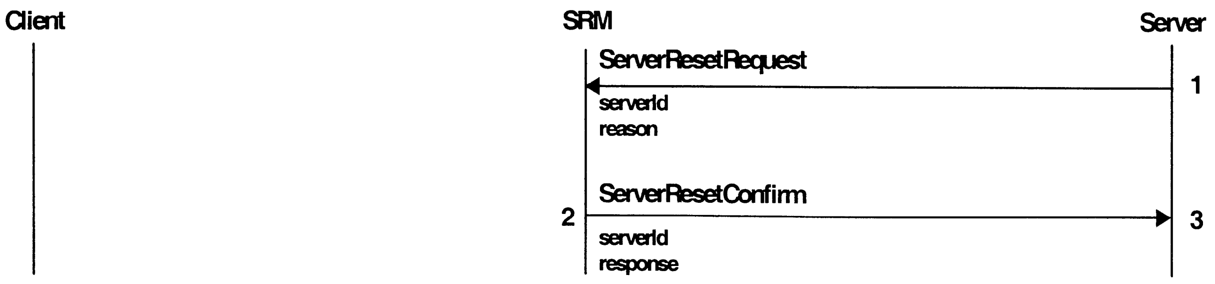


Figure 4-24 Scenario for a Server Initiated Reset message sequence

4.11.2.1 Server Initiates Reset command sequence

Step 1 (Server):

Following one of the conditions noted above, the Server attempts to re-synchronize the interface by transmitting a ServerResetRequest message to the Network. The message will contain the serverId parameter set to the value of the Server. All resources will be placed in the "idle" state and timer tMsg is set upon sending the ServerResetRequest.

No further messages will be sent while timer tMsg is running. With the exception of the ServerResetConfirm all messages received will be ignored.

On expiration of timer tMsg, the ServerResetRequest message will be re-transmitted. If no response is received after the second expired period the Reset procedure shall be terminated.

Step 2 (Network):

On receipt of the ServerResetRequest message the network shall clear all sessions for that Server with "active" sessionIds. All timers shall be reset and all resources shall be released. Upon successful execution, an ServerResetConfirm will be sent.

If appropriate, the Network will attempt to re-synchronize the clients by initiating the Reset procedure on the interface towards the clients.

Step 3 (Server):

On receipt of the ServerResetConfirm message timer tMsg shall be stopped.

4.11.3 Network Initiated Reset Command Sequence

Figure 4-25 illustrates the procedure used by the Network for Reset; for simplicity only the interaction with the server is shown. For a Network Initiated Reset to the Client, the ClientResetIndication and ClientResetResponse messages are used.

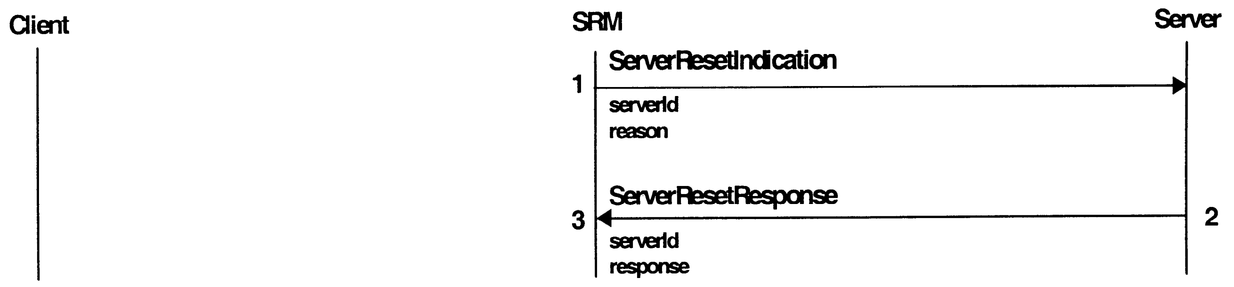


Figure 4-25 Scenario for Network Initiated Reset sequence message

4.11.3.1 Network Initiates Reset command sequence

Step 1 (Network):

Following one of the conditions noted above, the Network attempts to re-synchronize the interface by transmitting a ServerResetIndication message to the Server. The message will contain the clientId parameter set to the value of the Server. All resources will be placed in the "idle" state and timer tMsg is set upon sending the ServerResetResponse.

No further messages will be sent while timer tMsg is running. With the exception of the ServerResetResponse all messages received will be ignored.

On expiration of timer tMsg, the ServerResetIndication message will be re-transmitted. If no response is received after the second expired period the Reset procedure shall be terminated.

Step 2 (Server):

On receipt of the ServerResetIndication message the Server shall clear all sessions for that SRM with "active" sessionIds. All timers shall be reset and all resources shall be released. Upon successful execution, a ServerResetResponse message will be sent.

If appropriate, the Network will attempt to re-synchronize the clients by initiating the Reset procedure on the interface towards the clients.

Step 3 (Server):

On receipt of the ServerResetConfirm message timer tMsg shall be stopped.

5. User-to-User Interfaces

5.1 Introduction

This clause of ISO/IEC 13818-6 specifies a generic set of multimedia interfaces, called User-to-User interfaces. Each interface defines a set of operations that can be invoked on a service object. The operations offer function calls in a choice of programming languages, and Client/Service Remote Procedure Calls (RPC) in a choice of network protocol profiles. With a DSM-CC User-to-User Library, both application portability and network interoperability can be achieved. The User-to-User interfaces represent modular, basic building blocks that can be used to enable a range of capabilities from minimal, low-cost 'Consumer' Clients, that navigate and request multimedia data, to high-powered 'Producer' Clients, that configure and load the multimedia Service Domain.

5.1.1 Contents

This clause is organized as follows:

- The User-to-User System Environment. From a User-to-User perspective, this subclause explains system entities, objects, interfaces, object references and Clients, in the context of the DSM-CC system environment.
- Overview of the Interface Definition Language (IDL). Although CORBA specifies the complete IDL, this subclause covers the basic IDL vocabulary needed for an understanding of DSM-CC User-to-User.
- Common Definitions. This subclause specifies common types and constants that are used by the DSM-CC User-to-User interfaces.
- Application Portability Interfaces (API). The intended audience for this subclause is the application software programmer. This subclause is organized as two partitions. The first is the minimum compliant Core interface set. The second is the Extended interface set. Each interface is further partitioned to have a consumer Client subset.
- Service Inter-operability Interfaces (SII). The intended audience for this subclause is the network software programmer. This subclause defines structures and encodings needed for interoperability between Client and Service entities over a network. It is also organized as two partitions, Core and Extended.
- Application Boot Process. This subclause integrates the Application Portability Interfaces, User-to-Network Session protocol, Download protocol, and Service Inter-operability Interfaces into a working system model. It presents phases of application startup, and specifies the pre-conditions for each phase.

5.1.2 Intended Usage

The DSM-CC User-to-User interfaces enable a wide-range of multimedia applications to run in heterogeneous networked environments. For example, applications that use DSM-CC as a foundation include:

- Movies On Demand
- Movie Listing
- TV Program Guide
- Tele-Shopping
- Video-Conferencing
- Near Movies On Demand
- News on Demand
- Karaoke On Demand
- Games
- Tele-Medicine
- Distance Learning

and others

DSM-CC has developed relationships with other industry standards. The standards that either use or are compatible with DSM-CC User-to-User include:

- DAVIC. Digital Audio Visual Council
- DVB. Digital Video Broadcasting
- MHEG. Multimedia Hypermedia Experts Group
- ITU-T Study Group 8, T.120 and T.130 Video Conferencing
- CORBA. Common Object Request Broker
- TINA-C. Telecommunications Information Network Architecture
- UNO. Universal Networked Objects
- ONC. Open Networked Computing
- DCE, Distributed Computing Environment
- Languages C, C++, SQL

and others

The above uses and environments have inspired the development of the DSM-CC International Standard. As a result, the protocols offer a generic, unified and extensible set of functionalities. Many innovations are included which offer performance, size, scalability and security advantages. Here are some of the key characteristics of DSM-CC User-to-User:

- DSM-CC User-to-User specifies consistent, unified interfaces for commonly-used multimedia object types, with methods for creation, navigation, access and control.
- The User-to-User APIs enable applications to be designed that can be used in both interactive and broadcast carousel environments.
- The User-to-User interfaces leverage the power of inheritance, enabling developers to build on simple, generic libraries while adding application-specific functionalities.
- User-to-User defines programming interfaces that present function calls that are common to all Clients(given a language selection), regardless of hardware and operating system. To enable interoperability over the network, User-to-User specifies protocol profiles that establish common on-the-wire encodings and protocols between Clients and Services.
- Certain User-to-User operations enable 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 interfaces enable Client application interfaces to consist of simple stubs, that may be downloaded by the DSM-CC download mechanism. The stub is a surrogate for the remote service. It presents the interface of the service, but just marshals a message which is sent to the service object. The minimal Client need not be a name Service, perform authentication or authorization, or choose which instance of Service or asset to connect to. These functions will typically be performed by service brokers and asset brokers on the Server side of the network.
- A minimal consumer Client set of interfaces supports the requirements of a TV set-top device, where there is typically limited memory and no disk.
- While the interfaces are simplified to accomplish the minimal Client, a range of privileges (Access Roles) and extended interfaces support Clients with more capabilities and resources, such as peer Clients or authoring tool Clients.
- The Stream objects offer control of MPEG-2 and other continuous media streams. The normal play time concept is introduced, to insure accurate, contiguous and predictable timestamps for synchronized audio, video and stream event data. The stream operations enable consistent behavior, accurate to a frame, regardless of the network latency between Client and Server.
- Directory objects enable producer Clients to establish and load logical path hierarchies with multimedia objects and Services, and enable consumer Clients to navigate these paths, and access these objects and Services.
- Composite objects enable the opening of several child objects as a result of resolving a single name. Composite objects also simplify the versioning relationships between objects by enabling the association of compatible versions to a Composite name.
- Security applies to the entry into a Service Domain (Directory hierarchy) and to invocation of operations. Security features include:

- ⇒ Authentication of the principal end user on a per request basis.
- ⇒ Authorization of the principal end user to perform the requested operation based on Access Role.
- ⇒ Password or encryption access challenge/response control is supported on a per request basis(including entry into Service Domains)
- ⇒ All secure flag to trigger secure transmission for all messages to/from an object.

- The Remote Procedure call configuration interface allows for either synchronous or asynchronous (synchronous deferred) operation. Synchronous deferred operation supports pipelining of requests, where multiple parallel requests can be outstanding at a given time.
- Sorts and filters can be performed with a common syntax for both Directory and Database browsing. Query results can be pre-fetched using a windowing mechanism.

5.2 The User-to-User System Environment

The DSM-CC System Environment can be visualized as a distributed network of Users of varying capabilities. The Users may be connected by a DSM-CC network as defined in clause 4, User-to-Network Session Messages, or by private network (without the User-to-Network protocol).

The User entities are listed under physical and logical categories as follows:

5.2.1 U-U System Hardware User Entities

- **Hardware Servers**. These platforms are configured with MPEG-2 storage and delivery capability. Such Servers include a heterogeneous variety of storage and processor systems. The Server has operating system software that supports the operation of Services and applications, and isolates these from the underlying hardware devices and network protocols.
- **Hardware Clients** These devices run applications and provide the underlying capability of decoding and displaying MPEG-2 streams. Hardware Clients include a heterogeneous variety of set-top 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.
- **Other Platforms**. A heterogeneous variety of hardware/OS/Network platforms may act as Clients or Servers, or both, depending upon the application, independent of MPEG-2 streaming capability.

5.2.2 U-U System Logical Entities

- **Service Object Implementation**. A Service Object Implementation is a logical entity in the system that supports the syntax and semantics of an interface, meaning it can receive requests on that interface. The interface declares operations, and may inherit other interfaces. The Service Object Implementation is also referred to as a Service.
- **Client**. An application program or process becomes a Client when it initiates a request (invokes operations) to an Object Implementation. It is also referred to as a Client Application.
- **Application**. An application is a named entity that supports a functional theme, represented by a Client Application and one or more Service Object Implementations. One of the Service Object Implementations is designated the Primary Service for the application, and assumes the application's logical path name.
- **Object Reference**. An object reference is a local handle through which a Client Application makes requests to a remote Object Implementation.
- **Client-side DSM-CC Library Stubs**. These functional elements mediate between the Client Application and an RPC protocol, providing a simplified function call interface to the application, while encoding data for RPC requests, and decoding data for RPC replies.
- **Server-side DSM-CC Library Stubs**. These functional elements mediate between an RPC protocol and the Service Object Implementation, providing a programming interface to the application, while decoding data from RPC requests, and encoding data for RPC replies. Server-side Library Stubs are also known as Skeletons.

- **Server-side Service Gateway**. This Service Object Implementation acts as the entry point into a domain of Services, and is the root of its name space.
- **Client-Side Local Object**. This Object Implementation offers local functionality or simplified function call interface to more complex lower-layer network protocols. Local objects are also known as Pseudo Objects.

The system logical entities are replaceable functional elements that have standard interfaces between them. This standard defines the interfaces between these functional elements, as illustrated below.

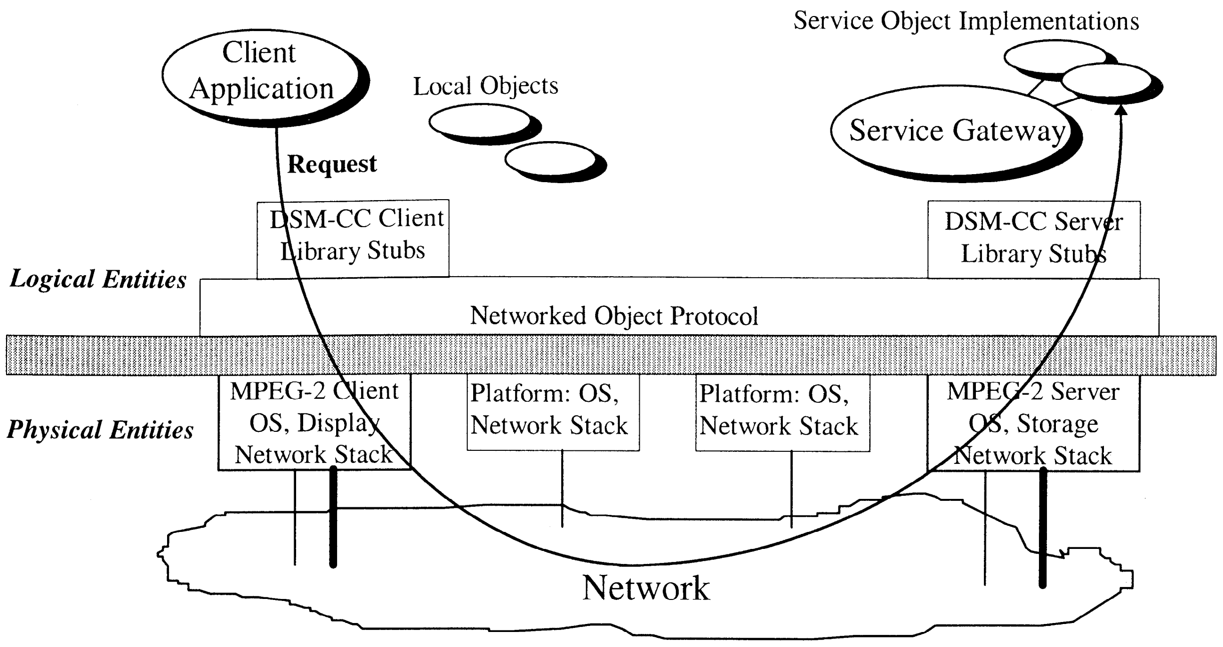

Service Object Implementations
Client Application
Local Objects
Service Gateway
Request
DSM-CC Client Library Stubs
DSM-CC Server Library Stubs
Logical Entities
Networked Object Protocol
MPEG-2 Client OS, Display Network Stack
Platform: OS, Network Stack
Platform: OS, Network Stack
MPEG-2 Server OS, Storage Network Stack
Physical Entities
Network

Figure 5-1 User-to-User System Environment

The hardware and software entities are logical components; they may exist on a single machine that supports all of 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. There may be one or more applications indicated by the graph. There is one top entry Service for each application, represented by the name of the application.

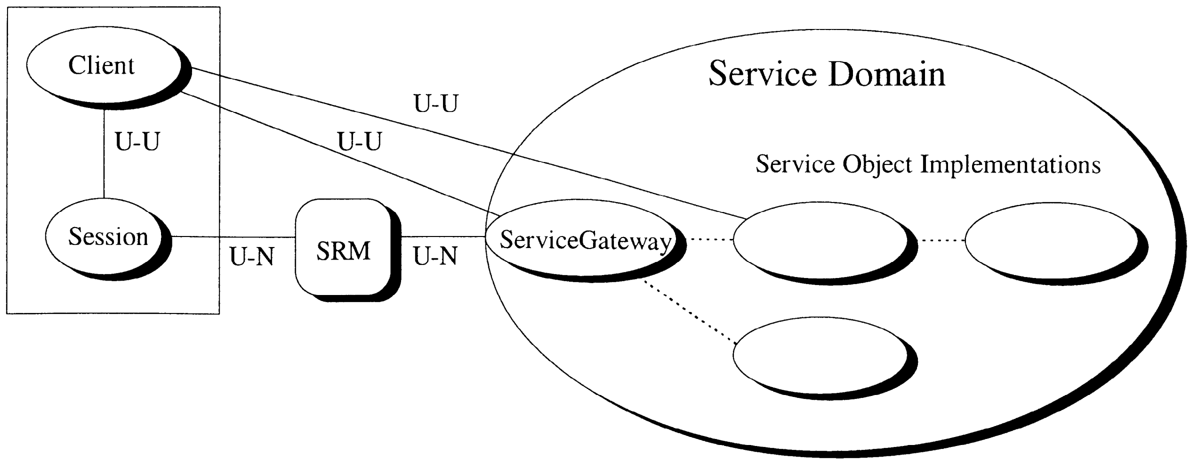


Figure 5-2 Logical Entity Relationships

The DSM-CC User-to-User Interfaces are designed as generic interface types that support access to multimedia Services and multimedia data. They are well-known basic building blocks for constructing more powerful object interfaces. DSM-CC supports single and multiple-inheritance, enabling new object interface definitions to include and build on the DSM-CC User-to-User interfaces.

In DSM-CC, the Name Service for objects is called a Directory. The Directory is a listing of names and corresponding objects. The operations of the Directory are given Access Roles, thus enabling certain Clients to create and populate a Directory hierarchy, while restricting other Clients to browsing and opening (resolving) a name to achieve the ability to communicate with an Object Implementation. A DSM-CC Directory Service can bind any kind of object, including application Services, Files, MPEG-2 Streams, Directories, and others. A Service may support (inherit) the Directory interface, thereby offering a directory hierarchy of objects. The hierarchy extends from a top node through a graph of sub-Directories, each serving to scope its own name space. All path names are therefore unique. A given object may be represented by more than one path name.

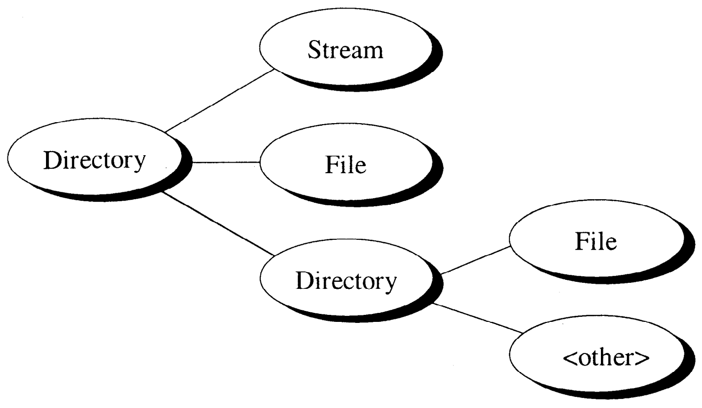


Figure 5-3 Multimedia Object Directory Hierarchy

5.2.3 Application and Service Interfaces

The Application Portability Interface is the interface between an application and the greater Client Operating System, which includes the DSM-CC Library, the processor operating system and the communications transport stack. The goal of the DSM-CC User-to-User Library is to provide Client applications a portable means of accessing Service objects (e.g. Directory, Stream and File). For most of the DSM-CC interfaces, operations translate directly 1-1 to Remote Procedure Calls, using a predictable, well-defined message assembly and data encoding rules. However, in some cases the interfaces supplied to the application by the DSM-CC Library system do not have a 1-1 mapping with the remote operations supported by Services. In other cases, the Object Implementation is completely local (i.e., there is no Remote Procedure Call).

In all cases, the DSM-CC Library must provide layered functionality, such as RPC message assembly, network lower layer protocols, or local Object Implementation. Therefore, the standard defines two interfaces for a Service available to layers of the Client; the interface seen by the application and the interface supported by the Server, as shown in Figure 5-4.

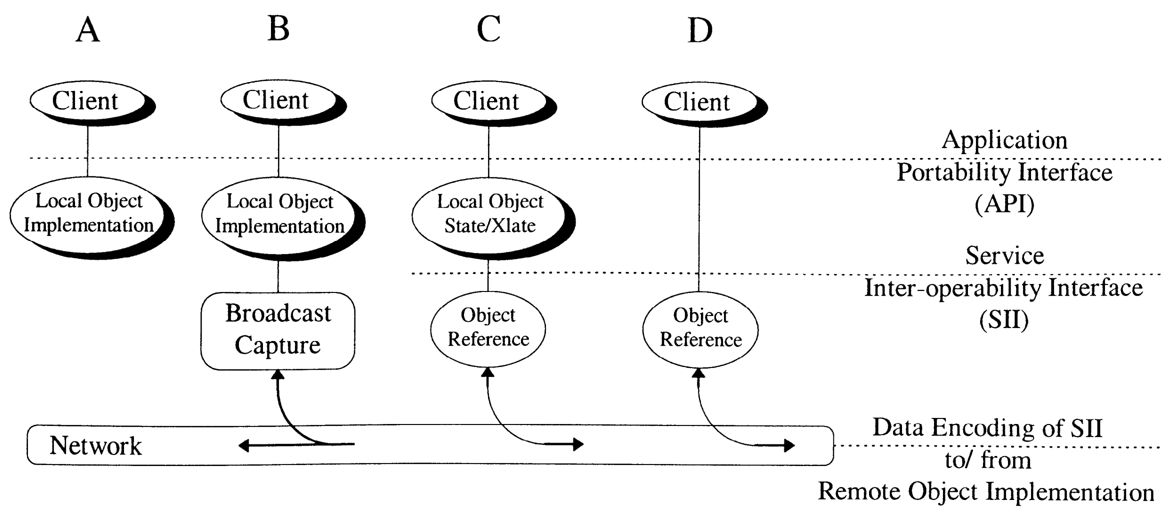

A
B
C
D
Client
Client
Client
Client
Local Object Implementation
Local Object Implementation
Local Object State/Xlate
Broadcast Capture
Object Reference
Object Reference
Network
Application Portability Interface (API)
Service Inter-operability Interface (SII)
Data Encoding of SII to/ from Remote Object Implementation

Figure 5-4 Models for Application and Service Interfaces

Figure 5-4 also illustrates four models for DSM-CC object access:

- Model A: The Client interfaces with a Local (Pseudo) Object. The Client makes requests across the Application Portability interface to invoke an operation that does not result in any network message sequence. These interfaces are specified under the heading **Application Portability (Local Library) Syntax.**
- Model B: The Client interfaces with a Local Object Implementation, which was received through a Network Broadcast. DSM-CC defines an Object Carousel for the broadcast delivery of objects.
- Model C: The Client interfaces with a Local Object implementation, which translates to a lower layer network message set, called the Service Inter-operability Interface. The operations are invoked on an object reference, which is local structure that acts as a proxy for the remote Service Object Implementation. The messages of the lower message set are 1-for-1 syntactically identical to the operations that are implemented at the remote Service. The local object interface specified under the heading **Application Portability (Local Library) Syntax**. The remote interface is specified under the heading **Service Inter-operability Syntax**.
- Model D: The Client interfaces directly through the Service Inter-operability Interface to the remote Service Object Implementation. In this case the Application Portability and Service Inter-operability Interfaces are one and the same, in addition to being 1-for-1 identical to the operations that are implemented at the remote Service. The layer between the SII and the Data Encoding is an RPC Stub that shall add a header with DSM-CC required parameters in the request and reply. The API presents the object reference as a pointer to a type with unknown structure, while the RPC Stub with SII interface defines the object reference structure for holding connection-related information. These interfaces are specified under the heading **Application Portability / Service Inter-operability Syntax.**

5.2.4 Categorization of Client Library Interface Sets

DSM-CC Library sets are defined in order to facilitate inter-operation. These are grouped as Consumer and Producer Clients of increasing capabilities. Core interfaces are defined as a foundation for minimal Clients. Extended interfaces are defined to facilitate portability and inter-operability.

5.2.4.1 Consumer Client

The Consumer Client will typically support a limited set of DSM-CC operations. Through download, it will add support for any additional operations needed to run an application. It may also store a basic set of operations 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 privileges (Access Role) given the Client End-User. The lowest Access Role that can be granted is that of READER. A READER cannot write to or update an object at the Server, it can only, navigate, open access to, and request delivery of objects and their data. The Consumer Client is defined as a READER with limited capabilities for saving application state, i.e., the ability to write files. Home set-top devices are usually classified as Consumer Clients.

5.2.4.2 Producer Client

A Producer Client, on the other hand, is granted OWNER privileges. In addition to the READER privileges, 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.

5.2.4.3 Client Library Profiles

The following list establishes an initial list of DSM-CC Library Profiles.

Core Consumer. This is the minimum-compliant Client. It supports READER operations for attaching/detaching to/from a Service Domain, navigating Directory hierarchies, resolving names to object references, controlling MPEG-2 audio-video Streams, and reading and writing Files.

Core Producer. This Client supports all operations of the DSM-CC User-to-User Core interfaces.

Extended Consumer. This Client supports READER operations of an extended DSM-CC Library, plus the ability to write to a Database.

Extended Producer. This Client supports the complete DSM-CC Library.

5.2.5 Core Interfaces

The Core interfaces represent the most fundamental DSM-CC User-to-User Interfaces. These interfaces serve as a basis for Core Client configurations.

- **Base** Interface.

⇒ This interface provides the commonly used operations close() and destroy().

- **Access** Interface.

⇒ This interface provides commonly used attributes for size, history (version and date), lock status and permissions.

- **Stream** Interface.

⇒ This interface enables a Client to interactively control MPEG continuous media streams. It inherits **Base** and **Access** interfaces.

- **File** Interface.

⇒ This interface enables a Client to read and write opaque data (File contents) of an object. It inherits **Base** and **Access** interfaces.

- **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. It additionally defines Access Roles for each of the inherited name Service operations. It inherits the **Access** interface.

- **BindingIterator** Interface.

⇒ This interface is defined by CORBA and is used for iteration through Directory lists.

- **Session** Interface.

⇒ This interface enables a Client to attach to a Service Domain.

- **ServiceGateway** Interface.

⇒ This interface inherits **Directory** and **Session** Interfaces.

- **First** Interface.

⇒ This interface provides function calls for an application to obtain root ServiceGateway and first Service. It is a Pseudo Object interface.

5.2.5.1 Core Client Application Portability Library

level: / interface:	Core Consumer API	Core Producer API

Base	close()	close()
		destroy()

Access	_get_Lock()
	_set_Lock()
		_get_Size()
		_get_Hist()
		_set_Hist()
		_get_Lock()
		_set_Lock()
		_get_Perms()
		_set_Perms()

Stream	Base Consumer operations
	Access Consumer operations
	resume()
	pause()
	status()
	reset()
	play()
	jump()
	_get_Info()
	_set_Info()*
		Base operations
		Access operations
		resume()
		pause()
		status()
		reset()
		play()
		jump()
		_get_Info()
		_set_Info()

File	Base Consumer operations
	Access Consumer operations
	read()
	_get_Content()
	_set_Content()
	write()
		Base operations
		Access operations
		read()
		_get_Content()
		_set_Content()
		_get_ContentSize()
		write()

BindingIterator	next_one()
	next_n()
	destroy()
		next_one()
		next_n()
		destroy()

Directory	Access Consumer operations
	list()
	resolve()
	open()
	close()
	get()
		Access operations
		list()
		resolve()
		open()
		close()
		get()
		bind()
		bind_context()
		rebind()
		rebind_context()
		unbind()
		new_context()
		bind_new_context()
		destroy()
		put()

Session	attach()
	detach()
		attach()
		detach()

ServiceGateway	Directory Consumer operations Session operations	Directory operations Session operations

level: interface:	Core Consumer API	Core Producer API

First	root()
	service()
		root()
		service()

* Although Stream _set_Info() is generated from IDL, its use by Consumer Client is not authorized, i.e., shall be blocked by Access control.

5.2.5.2 Core Client Service Inter-operability Library

level: interface:	Core Consumer SII	Core Producer SII

Base	close()	close()
		destroy()

Access	_get_Lock()
	_set_Lock()
		_get_Size()
		_get_Hist()
		_set_Hist()
		_get_Lock()
		_set_Lock()
		_get_Perms()
		_set_Perms()

Stream	Base Consumer operations
	Access Consumer operations
	resume()
	pause()
	status()
	reset()
	play()
	jump()
	_get_Info()
	_set_Info()*
		Base operations
		Access operations
		resume()
		pause()
		status()
		reset()
		play()
		jump()
		_get_Info()
		_set_Info()

File	Base Consumer operations
	Access Consumer operations
	read()
	_get_Content()
	_set_Content()
	write()
		Base operations
		Access operations
		read()
		_get_Content()
		_set_Content()
		_get_ContentSize()
		write()

BindingIterator	next_one()
	next_n()
	destroy()
		next_one()
		next_n()
		destroy()

Directory	Access Consumer operations
	list()
	resolve()
	open()
	close()
	get()
		Access operations
		list()
		resolve()
		open()
		close()
		get()
		bind()
		bind_context()
		rebind()
		rebind_context()
		unbind()
		new_context()
		bind_new_context()
		destroy()
		put()

* Although Stream _set_Info() is generated from IDL, its use by Consumer Client is not authorized, i.e., shall be blocked by Access control.

5.2.6 Extended Interfaces

The Extended Interfaces establish public, standard operations for commonly used functionality. These interfaces may be used individually, i.e. only the interface Stubs required by an application need be present in the Client.

- **Event** Interface.

⇒ This interface provides operations by which a Client can subscribe/unsubscribe to events, to be sent as Stream Event Descriptors in the MPEG stream.

- **Download** Interface.

⇒ This interface provides a function call interface for the state machine and message encodings in clause 7, U-N Download. It is used to cause the transfer of Client interface stubs and Client application from a Download Service.

- **Composite** Interface.

⇒ With this interface, versioned objects can be associated as a set, and opened with a single Directory open() invocation.

- **View** Interface.

⇒ The interface enables a Client to sort and filter objects by their attributes using standard SQL statements.

- **State** Interface.

⇒ This interface enables a Client to suspend and resume application state.

- **Interfaces** Interface.

⇒ The interface provides methods for publishing and verifying new interfaces, in order to insure consistent and complete interface definitions in a DSM-CC environment.

- **Security** Interface.

⇒ The interface provides a method to associate the passing of authentication parameters with **Session attach()**, **Directory open( )**, **Directory resolve()**, **Directory get()**, or other operations requiring authorization. It is a Pseudo Object interface.

- **Config** Interface.

⇒ This interface provides an API by which RPC can be configured for either synchronous or asynchronous operation. It is a Pseudo Object interface.

- **LifeCycle** Interface.

⇒ This interface is used by implementations for creation of objects, to insure unique object references in a DSM-CC environment. It is a Pseudo Object interface.

- **Kind** Interface.

⇒ This interface provides operations to determine which interfaces are supported by an object. It is a Pseudo Object interface.

5.2.6.1 Extended Client Application Portability Library

level: interface:	Extended Consumer API	Extended Producer API
Core interfaces (all)	Core Consumer API operations	Core Producer API operations

Download	info()
	alloc()
	start()
	cancel()
		info()
		alloc()
		start()
		cancel()

Event	subscribe()
	unsubscribe()
	notify()
	_get_EventList()
	_set_EventList()*
		subscribe()
		unsubscribe()
		notify()
		_get_EventList()
		_set_EventList()

Composite	list_subs()	list_subs()
		bind_subs()
		unbind_subs()

View	query()
	read()
	execute()
	_get_Style()
		query()
		read()
		execute()
		_get_Style()

State	suspend()
	resume()
		suspend()
		resume()

Interfaces		define()
		check()
		show()
		undefine()

Security	authenticate()	authenticate()

Config	wait()
	inquire()
	_get_DeferredSync()
	_set_DeferredSync()
	_get_ActiveRequests()
		wait()
		inquire()
		_get_DeferredSync()
		_set_DeferredSync()
		_get_ActiveRequests()

LifeCycle		create()

Kind		is_a()
		has_a()

* Although Event _set_EventList() is generated from IDL, its use by Consumer Client is not authorized, i.e., shall be blocked by Access control.

5.2.6.2 Extended Client Service-interoperability Library

level: interface:	Extended Consumer SII	Extended Producer SII
Core interfaces (all)	Core Consumer SII operations	Core Producer SII operations

SessionUU	attach()
	 detach()
		attach()
		 detach()

ServiceGatewayUU	ServiceGateway AccessRoles** Directory Consumer operations	ServiceGateway AccessRoles** Directory operations

SessionSI	attach()
	 detach()
		attach()
		 detach()

ServiceGatewaySI	ServiceGateway AccessRoles** Directory Consumer operations SessionSI operations	ServiceGateway AccessRoles** Directory operations SessionSI operations

DownloadSI	info()
	 proceed()
	 cancel()
		info()
		 proceed()
		 cancel()
		 install()
		 deinstall()

Event	subscribe()
	 unsubscribe()
	 _get_EventList()
	 _set_EventList()*
		subscribe()
		 unsubscribe()
		 _get_EventList()
		 _set_EventList()

Composite	list_subs()	list_subs()
		 bind_subs()
		 unbind_subs()

View	query()
	 read()
	 execute()
	 _get_Style()
		query()
		 read()
		 execute()
		 _get_Style()

State	suspend()
	 resume()
		suspend()
		 resume()

Interfaces		define()
		 check()
		 show()
		 undefine()

* Although Event _set_EventList() is generated from IDL, its use by Consumer Client is not authorized, i.e., shall be blocked by Access control.

** The ServiceGateway redefines AccessRoles for Producer operations inherited from Directory.

5.3 Overview of the Interface Definition Language(IDL)

The DSM-CC User-to-User interfaces are defined in Object Management Group (OMG) Interface Definition Language (IDL). The complete IDL syntax, grammar, language mappings and related topics are contained in "The Common Object Request Broker: Architecture and Specification," Revision 2.0, July 1995, from the Object Management Group. This specification is commonly referred to as CORBA 2.0. The OMG IDL is also standardized in ISO/IEC 14750.

IDL is a language for specifying interfaces. It is not a programming language, and therefore does not provide any procedural syntax. It is object-oriented, supporting the concepts of encapsulation and inheritance. It defines basic types such as char, short, long, and constructed types such as sequence (which is used to specify variable-length arrays). The scoping is much like ANSI C++. In IDL, a module contains interfaces, which in turn contain operations. (in ANSI C++, a namespace contains classes, which in turn contain methods).

```
module module_name {
    types
    ...
    interface interface_name {
        types
        attributes
        operations
        ...
    };
    other_interfaces
    ...
};
```

Figure 5-5 IDL File Format

Interfaces can be inherited with the syntax:

```
module module_name {
    interface interface_name :
        inherited_interface_name,
        other_inherited interface_name{
    ...
    };
};
```

Figure 5-6 Inheriting Interfaces with IDL

The inheritance feature of IDL enables new interfaces to be rapidly developed from basic 'building block' interfaces. All operations of the inherited interface are automatically included in the new interface. For example, the DSM-CC ServiceGateway interface inherits Directory and Session interfaces with the following IDL:

```
module DSM {
        interface ServiceGateway : Directory, Session {
        };
};
```

IDL is used to define interfaces in a generic way, independent of language and network protocol. CORBA 2.0 provides rules which map the IDL to C, C++ and Smalltalk languages. Given an IDL specification, the rules of the language mapping shall yield the declarations for constants, type definitions for all data types and function call prototypes for all operations. For example, the type struct in IDL is identical in syntax to the C language `struct`. OMG may define other language mappings in the future, which shall be considered normative as well.

In addition to the language mapping, rules for data encoding must be defined for each of the common IDL types. A data encoding is a flat sequence of octets that can be transferred between Client and object implementation using an inter-process control or network transport mechanism. Where the language mapping offers structure with accessible members, the data encoding is a contiguous, unparsed buffer. CORBA 2.0 provides a data encoding called Common Data Representation (CDR). DSM-CC Informative Annex C provides a summary of IDL to CDR, and in addition provides IDL to External Data Representation (XDR), the data encoding used by Open Networked Computing (ONC).

CORBA 2.0 also specifies a structure called an Interoperable Object Reference (IOR), which at a minimum contains a an object type id, a Protocol Profile Identifier, and an encapsulation of connection information. The Protocol Profile Identifier indicates the RPC, the data encoding and the structure of the encapsulation. The information in the IOR is used for all operation invocations on the Client/Service connection. DSM-CC has registered a range of Protocol Profile Identifiers with OMG, and defines several Protocol Profiles(identifier and associated encapsulation) for use in DSM-CC environments.

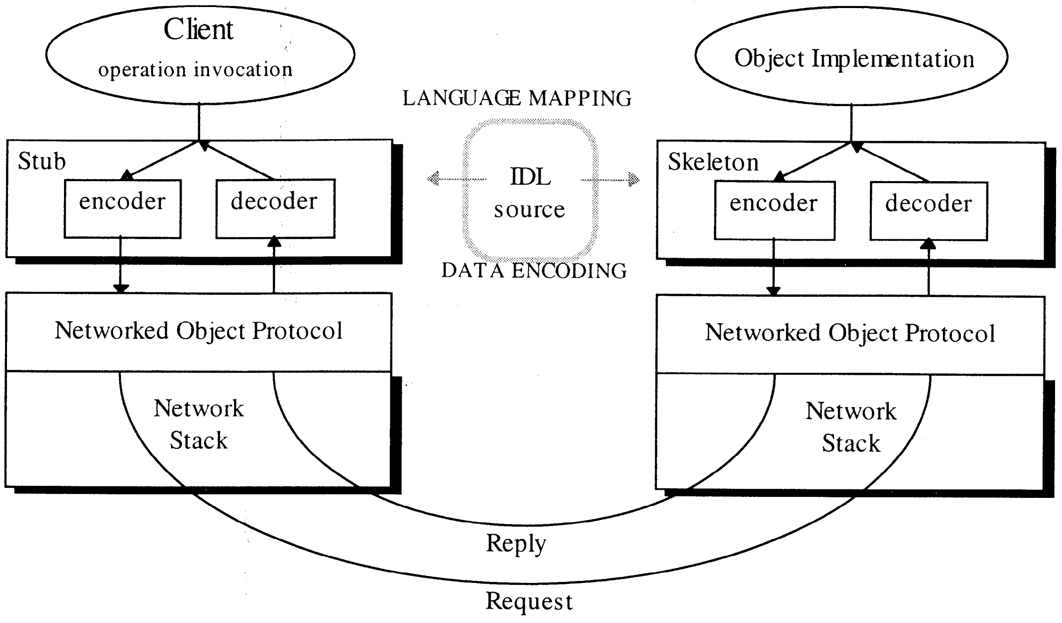


Figure 5-7 IDL to Entity Relationship

An IDL compiler is used to generate the Client Stub and a corresponding Server Stub, also known as the Skeleton. The upper interface of the Stub and Skeleton, as shown in Figure 5-7, presents the language mapping interface to the programmer. The lower interface presents the data encoding to the Networked Object Protocol. In the case of a full CORBA system, this is the Object Request Broker (ORB). In other systems, it may simply be an RPC request/reply mechanism.

5.3.1 Operations

In IDL, operation parameters are specified to be either input (in), output (out), or both input and output (inout). In an underlying RPC implementation, input parameters are placed in the RPC request, and output parameters are returned in the RPC reply.

For example, the DSM-CC Event subscribe() operation is defined as follows:

```
module DSM {
        exception INV_EVENT_NAME ExceptUser;
        interface Event {
                void subscribe (in string aEventName, out u_short eventId)
                        raises (INV_EVENT_NAME);
        }:
};
```

The exception INV_EVENT_NAME is declared using the exception declaration. The exception declaration is like a C language `struct` declaration. In addition to having an exception structure defined by the DSM Common macro ExceptUser, it will have a string id, e.g., "ex_DSM_INV_EVENT_NAME". All operations can return any of the standard CORBA System exceptions. The raises declaration identifies any additional user exceptions the operation can return.

The subscribe() operation is in the Event interface. It returns void. It has an string input argument that identifies an Event, and an unsigned short output argument that is a token associated with the Event. If the input argument is not valid, The Event subscribe() operation will return the INV_EVENT_NAME exception. Note: the Event interface is fully defined later in this clause.

5.3.2 Attributes

Within an interface, the attribute declaration can be used to define an identifier of a certain data type with a pair of implicit operations, one to set the value of the attribute, and the other to retrieve the value of the attribute. The readonly attribute declaration can be used to define an attribute that can only be retrieved by a Client.

For example, the Event interface declares an EventList attribute as an array of strings, as follows:

```
module DSM {
        exception INV_EVENT_NAME ExceptUser;
        interface Event {
                typedef sequence<string> EventList_T;
                        attribute Event_List_T EventList;
        }:
};
```

5.3.3 Language Mapping

At the programing level, the C language mapping for the above Event subscribe() IDL is:

```
void DSM_Event_subscribe
     (DSM_Event object, CORBA_Environment * ev,
           CORBA_string aEventName, DSM_u_short * eventId)
```

Two arguments are inserted in the C mapping. The first is an input argument pointing to the object that will receive the request. This pointer represents a structure that normally contains addressing or local context information. The second is an output argument that will hold the exception, if there is one.

The EventList attribute produces two operations in the C mapping:

```
DSM_Event_EventList_T DSM_Event__get_EventList
           (DSM_Event object, CORBA_Environment * ev);

void DSM_Event__set_EventList
           (DSM_Event object, CORBA_Environment * ev
                 SM_Event_EventList_T * EventList);
```

A C++ language mapping produces a name space DSM and within it, an Event class. In the Event class will be the methods for the operations and attributes defined in the IDL.

Note: In cases where the mapping of basic types in DSM-CC disagrees with that of OMG, the OMG mapping shall take precedent.

5.3.4 Encoding

Over the network, the above Event subscribe() becomes two messages, the request and reply of the Remote Procedure Call (RPC). A Protocol Profile (identified by a profileId in the IOR) is determined at the time the Client first obtains the IOR. This Protocol Profile establishes the data encoding to be used. For example, the UNO RPC uses Common Data Representation (CDR) encoding. For each IDL type, the encoding specifies the octet sequence to be carried over the network. DSM-CC Protocol Profiles are defined later in the Service Inter-operability subclause.

The RPC protocol defines message headers that provide a common transaction id for the request/reply pair, an operation id, etc. UNO defines the RPC and data encoding on the Inter-operability interface for a UNO RPC stack. ONC defines the RPC and data encoding on the inter-operability interface for an ONC RPC stack. Appendix G defines the UNO messages in IDL. Annex C defines UNO and ONC RPC on-the-wire message header mappings. The default for DSM-CC, unless otherwise specified by the Protocol Profile, is UNO/CDR.

The in arguments of the operation are placed in the request message body by the Client, in the order that they were specified in the IDL. In the case of Event subscribe(), this consists of one argument, a string. While at the programming interface, the string consisted of a pointer to a char(an array of char, null-terminated), a CDR encoding shall place a 4 byte length followed by the char array (null-terminated) in the byte stream.

The out arguments of the operation are placed in the reply message body by the object implementation responding to the request, again in the order that they were specified in the IDL. In this case, the reply message body consists of 2 bytes containing the eventId.

5.3.5 Typographical Conventions

In general, the DSM-CC User-to-User IDL syntax is placed within borders, to separate it from ordinary text.

The type styles shown below are used in this clause to distinguish programming statements from ordinary text and to distinguish among OMG IDL specifications, DSM-CC IDL specifications and language mappings. However, these conventions are not used in tables or subclause 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	DSM-CC User-to-User IDL, within a text border.
Times New Roman (bold)	DSM-CC User-to-User language and syntax elements in sentence text.
`Courier`	C language language and syntax elements.

5.3.6 Syntactical Conventions

The Core and Extended interfaces are scoped in module **DSM**. At the **DSM** level of scoping, Common DSM-CC data type definitions will be shared by Application and Service Inter-operability interfaces.

A Service Inter-operability Interface shall have the suffix **SI**, if it differs syntactically from the corresponding Application Portability Interface. The U-N Session Inter-operability Interface shall have the suffix **UU**.

#ifdef DSM_GENERAL shall enclose Producer Client operations. IDL Compilers can therefore generate efficient Consumer Client Stubs as the default case (**DSM_GENERAL** not defined). With **DSM_GENERAL** defined, IDL Compilers shall generate complete interface Stubs

#ifdef DSM_PSEUDO shall enclose local operations where they need to be separated from client/service operations.

#ifdef DSM_CONSUMER enables compilation of producer-only operations (without consumer operations).

The terms **TRUE** and **FALSE** shall indicate boolean 1 and 0, respectively.

5.4 Common Definitions

5.4.1 Basic Types

The following IDL specifies basic types. The basic type definitions of the language mapping may require modification to satisfy the requirements of the host processor and Operating System. For example, one system may define a long as a 32 bit quantity, whereas another may define it as a 64 bit quantity. OMG IDL defines the following ranges for integer data types:

short	$-2^{15} .. 2^{15}$
unsigned short	$0 .. 2^{16}$
long	$-2^{31} .. 2^{31}$
unsigned long	$0 .. 2^{32}$
longlong	$-2^{63} .. 2^{63}$
unsigned longlong	$0 .. 2^{64}$

```
module DSM {
    // machine-independent basic types
    //
    typedef short  s_short;                      // 16 bit signed integer
    typedef unsigned short u_short;              // 16 bit unsigned integer
    typedef long s_long;                   // 32 bit signed integer
    typedef unsigned long u_long;          // 32 bit unsigned integer
    //
    // note: this longlong is present to satisfy present OMG compilers
    // that do not support CORBA 2.0 extensions
    // If the IDL compiler cannot handle unsigned long long, treat it as an array
    // of 2 unsigned long where big-endian has MSB in octet 0 and LSB in octet 7
    // and little-endian has MSB in octet 7 and LSB in octet 0
    //
    // typedef s_long  s_longlong[2];            // 64 bit signed integer
    // typedef u_long  u_longlong[2];            // 64 bit unsigned integer
    //
    // else, if the IDL compiler supports CORBA 2.0 extensions
    // use the following two statements
    //
    typedef long long s_longlong;                        // 64 bit signed integer
    typedef unsigned long long u_longlong;       // 64 bit unsigned integer
};
```

5.4.2 Entity Identification

These types are used to hold information describing logical entities of the system.

The object reference(ObjRef) is defined as type CORBA Object. The Client Application has an opaque notion of object references, and is insulated from the representation of them (i.e., cannot determine any information in their structure). The Client merely receives the object reference in response to a resolve or open operation, and then invokes other operations using the object reference. It has no need to interpret or parse the object reference. The Client Library Stub shall define the Object structure and can use it to hold network addressing information or local state.

An **Opaque** type is defined for holding a sequence of octets. Opaque values are used read and write File contents, to hold suspended application state information (UserContext), to hold the identification of the End-User (Principal), and for other values where the structure of the data does not need to be statically specified.

Version is used for identification of Object Implementation variations, and for compatibility verification.

ServiceLocation is used to contain parameters needed to attach to a Service Domain. **ServiceDomain** is in NSAP address format. For an interactive Session, it is the globally unique Server network address. For a Broadcast Carousel, it is the unique identifier of the Carousel. **pathName** provides the logical path in the network system namespace. **initialContext** provides application state enabling the resumption of an application at a given point.

```
module DSM {
    // entity identification
    //
    // ObjRef is received by the Client in the reply to a resolve
    // Following that, operations can be invoked against it
    //
    typedef Object ObjRef;                  // implemented as void * for C mapping
    typedef sequence<ObjRef> ObjRefs;
    typedef sequence<octet> opaque;
    typedef opaque UserContext;             // context for application state
    typedef opaque Principal;               // system-wide identification of end user
    struct Version {char aMajor; char aMinor;};
    typedef sequence<octet, 20> ServiceDomain;   // ServerId NSAP
    struct ServiceLocation {
                ServiceDomain aServiceDomain;
                CosNaming::Name pathName;        // path name to resolve
                UserContext  initialContext;     // starting user context
    };
};
```

5.4.3 Interface Identification

DSM-CC identifies interfaces by two means, enumeration and string values. The IFKind is an unsigned long used to identify an interface. For the situation where the interface inherits other interfaces, the IntfCode is defined. The IntfCode identifies the numeric and string value of the most derived interface, and lists the IFKinds for the inherited interfaces. The IntfCode provides a repository identifier to uniquely scope the source of the interface definition. For object identification in CORBA mechanisms, string ids are used in the form "<Module>::<Interface>". For instantiable objects, 3-character string alias values are reserved, in order to reduce the size of Interoperable Object References.

```
module DSM {
    // interface identification
    //
    typedef u_long IFKind;
    typedef sequence<IFKind>  IFKindList;
    struct IntfCode {
        u_long anIFKind;// standard DSM-CC IFkind enumeration
        string kind;                  // DSM-CC format, "<Module>::<Interface>" or alias
        string repositoryId;          // id of repository where defined
        IFKindList includes;          // IFKinds of  inherited  interfaces
    };
    // The following are Service Tags registered with OMG for identifying interfaces
    // The reserved range for ISO/IEC WG11 in OMG is 0x49534F00 - 0x49534F7F
    // DSM interface IFKinds
    //
    const u_long  ik_null = 1230196480; // = 0x49534F00
    const u_long  ik_Base = 1230196481; // = 0x49534F01
    const u_long  ik_Access = 1230196482; // = 0x49534F02
    const u_long  ik_Stream = 1230196483; // = 0x49534F03
    const u_long  ik_File = 1230196484; // = 0x49534F04
    const u_long  ik_BindingIterator = 1230196485; // = 0x49534F05
    const u_long  ik_NamingContext = 1230196486; // = 0x49534F06
    const u_long  ik_Directory = 1230196487; // = 0x49534F07
    const u_long  ik_Session = 1230196488; // = 0x49534F08
    const u_long  ik_ServiceGateway = 1230196489; // = 0x49534F09
    const u_long  ik_First = 1230196490; // = 0x49534F0A
    const u_long  ik_Download = 1230196491; // = 0x49534F0B
    const u_long  ik_Event = 1230196492; // = 0x49534F0C
    const u_long  ik_Composite = 1230196493; // = 0x49534F0D
    const u_long  ik_View = 1230196494; // = 0x49534F0E
    const u_long  ik_State = 1230196495; // = 0x49534F0F
    const u_long  ik_Interfaces = 1230196496; // = 0x49534F10
    const u_long  ik_Security = 1230196497; // = 0x49534F11
    const u_long  ik_Config = 1230196498; // = 0x49534F12
    const u_long  ik_Lifecycle = 1230196499; // = 0x49534F13
    const u_long  ik_Kind = 1230196500; // = 0x49534F14
    const u_long  ik_SessionUU = 1230196501; // = 0x49534F15
    const u_long  ik_SessionSI = 1230196502; // = 0x49534F16
    const u_long  ik_DownloadSI = 1230196503; // = 0x49534F17
    //
    // aliases for string names
    //
    const string alias_Directory = “dir”;
    const string alias_BindingIterator = “bit”;
    const string alias_Stream = “str”;
    const string alias_File = “fil”;
    const string alias_Session = “ses”;
    const string alias_ServiceGateway = “srg”;
    const string alias_Interfaces = “inf”;
    const string alias_View = “viw”;
    const string alias_Download = “dnl”;
    const string alias_Event = “evt”;
};
```

5.4.4 Access Roles for Operations

An Access Role shall be associated with every DSM-CC operation, and with get and set operations individually on each attribute. An End-User Client (e.g., human) of a Service Domain shall also be given one or more Access Roles. In order for an operation request to be accepted, the Access Role of the End-User Client must satisfy the Access Role requirements of the operation. DSM-CC defines the following Access Roles:

- READER. A READER has read-only access to an object.
- WRITER. A WRITER can write to or update the state of an object. A WRITER also has READER privileges.
- BROKER. A BROKER can authorize access by a Client to an object. A BROKER also has WRITER privileges.
- OWNER. An OWNER can create and destroy objects. An OWNER also has BROKER privileges.
- MANAGER. A MANAGER has full system administration authority. A MANAGER also has OWNER privileges.

If not specified, the Access Role (ACR) for interface operations and attribute get operations shall default to READER. If not specified, the Access Role (ACR) for attribute put (set) operations shall default to WRITER.

A Service Domain may define additional Access Roles for further identification of an End-User Client's capabilities.

```
Module DSM {
    // Access Roles for operations
    //
    typedef char AccessRole;
    const AccessRole READER = 'R';
    const AccessRole WRITER = 'W';
    const AccessRole BROKER = 'B';
    const AccessRole OWNER = 'O';
    const AccessRole MANAGER = 'M';
};
```

5.4.4.1 Syntax for Access Control

The following rules enable the implementation of access control for DSM-CC:

Each operation 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 Control Role definition:

```
module DSM {
    interface Stream {
        const char resume_ACR = READER;
    };
};
```

Each attribute shall have separate Access Control Role 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>;

For example, the Stream Info attribute has the following Access Control Role definitions:

```
module DSM {
    interface Stream {
        const AccessRole Info_get_ACR = READER;
        const AccessRole Info_put_ACR = OWNER;
    };
};
```

5.4.5 Exceptions

An exception is an indication that an operation request was not performed successfully. The DSM-CC U-U operations are considered to be atomic. If an exception is received, all output parameters of the operation are invalid. 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 exception condition has occurred during the performance of a request.

The standard OMG System exceptions are used by DSM-CC. These exception identifiers may be returned as a result of any operation invocation, regardless of the interface specification. Standard System exceptions are not listed in raises expressions. The OMG IDL reference standard has a complete description of how exceptions are handled, plus code examples of C mappings. For example, the CORBA COMM_FAILURE exception has the string id "CORBA::COMM_FAILURE".

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.

The following are the standard CORBA System exceptions:

```
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 are the DSM-CC User-to-User exceptions:

1. The **ALREADY_BOUND** exception indicates a Name is already in use. The Client should pick another Name. It is identical in syntax to the CORBA CosNaming AlreadyBound exception.
2. The **BAD_COMPAT_INFO** exception is returned in response to a request to initiate a Download sequence. Certain operations will always carry a DownloadInfoRequest, which provides a CompatibilityDescriptor and Download negotiation parameters to the Service. The CompatibilityDescriptor describes the Client's current hardware and software configuration. **BAD_COMPAT_INFO** is an indication of an unrecognized CompatibilityDescriptor. This feature is described in clause 6 of this part of ISO/IEC 13818.
3. The **BAD_MODULE_ID** exception indicates a module identifier value does not exist in the present configuration.
4. The **BAD_MODULE_INFO** exception indicates an incorrect Download install configuration.
5. The **BAD_SCALE** exception indicates an incorrect Scale value.
6. The **BAD_START** exception indicates the Stream start time does not exist.
7. The **BAD_STOP** exception indicates the Stream Stop value does not exist.
8. The **BLOCK_SIZE** exception indicates the requested block size is unacceptable to the Server.
9. The **BUF_SIZE** exception indicates the requested buffer size is unacceptable to the Server.
10. The **CANNOT_PROCEED** exception indicates a Directory did not have permission to resolve a node in a logical path. It is identical in syntax to the CORBA CosNaming CannotProceed exception.
11. The **ILLEGAL_SYNTAX** exception indicates an input string parameter contained incorrect syntax for IDL or SQL, or other expected language.

12. The **INV_CURSOR** exception indicates a cursor is out of range of the query results.
13. The **INV_EVENT_ID** exception indicates an Event token is not recognized.
14. The **INV_EVENT_NAME** exception indicates the Event name does not exist for this Service.
15. The **INV_KIND** exception indicates an interface kind is not recognized.
16. The **INV_NAME** exception indicates a path name is incorrectly formatted. It is identical in syntax to the CORBA CosNaming InvalidName exception.
17. The **INV_OFFSET** exception indicates the offset is outside the range of File contents.
18. The **INV_SIZE** exception indicates the size exceeds server or network limits.
19. The **MPEG_DELIVERY** exception indicates the Server was unable to deliver a multimedia object over an MPEG stream.
20. The **NO_AUTH** exception indicates the End-User has not provided the correct authentication in the request.
21. The **NO_QUERY** exception indicates a query has not yet been performed.
22. The **NO_REF_TYPE** exception indicates the referenced object or data type does not exist in this context.
23. The **NO_RESUME** exception indicates that previous application saved state cannot be recovered.
24. The **NO_SUSPEND** exception indicates the object cannot save application state.
25. The **NOT_DEFINED** exception indicates a prerequisite data type or interface has not been defined.
26. The **NOT_FOUND** exception indicates a logical path name does not exist, that an intermediate node is not a Directory, or that it could not resolve the object. It is identical in syntax to the CORBA CosNaming NotFound exception.
27. The **OPEN_LIMIT** exception indicates the number of active object references is the maximum allowed.
28. The **PREV_DEFINED** exception indicates a data type or interface has already been defined.
29. The **READ_LOCKED** exception indicates reads are temporarily prevented for an object.
30. The **TIMEOUT** exception indicates the maximum time for a transaction has been exceeded.
31. The **UNK_USER** exception indicates the Principal End-User is unknown to the Service Domain.
32. The **WRITE_LOCKED** exception indicates writes are temporarily prevented for an object.
33. The **SERVICE_XFR** exception indicates a resolve operation was unsuccessful, and provides an alternate Service Domain location where the requested Service can be resolved.

```
module DSM {
    // User-to-User common exceptions
    //
    #define ExceptUser {u_long minor; u_long completed;}
    //
    exception ALREADY_BOUND {};     // new Name conflicts with existing Name
    exception BAD_COMPAT_INFO ExceptUser; // incorrect CompatibilityDescriptor
    exception BAD_MODULE_ID ExceptUser;        // Module Id out of range
    exception BAD_MODULE_INFO ExceptUser;   // Incorrect CompatibilityDescriptor
    exception BAD_SCALE ExceptUser;   //invalid scale
    exception BAD_START ExceptUser;   //stream does not contain this NPT
    exception BAD_STOP ExceptUser;    //invalid StopNPT, can never be reached
    exception BLOCK_SIZE ExceptUser; // block size out of range
    exception BUF_SIZE ExceptUser;    // buffer size out of range
    exception CANNOT_PROCEED {       // unable to resolve node in path
        CosNaming::NamingContext cxt;
        CosNaming::Name rest_of_name;
    };
    exception ILLEGAL_SYNTAX {string aMessage;}; // unrecognized syntax
    exception INV_CURSOR ExceptUser;             //cursor out of bounds
    exception INV_EVENT_ID ExceptUser;           // invalid event id
    exception INV_EVENT_NAME ExceptUser;   // invalid event name
    exception INV_KIND {string aMessage;}; //type previously defined
    exception INV_NAME { };                // incorrect name format
    exception INV_OFFSET ExceptUser; // offset exceeds file size -1 for read
    exception INV_SIZE ExceptUser;       //size exceeds network limits
    exception MPEG_DELIVERY ExceptUser;      //error delivering MPEG stream
    exception NO_AUTH{           //not allowed to open without authentication
        u_long minor;
        u_long completed;
        opaque authData;
    };
    exception NO_QUERY ExceptUser;  // a query has not been performed yet
    exception NO_REF_TYPE  {string aMessage;};          //missing type definition
    exception NO_SUSPEND  ExceptUser;          // unable to suspend state
    exception NO_RESUME ExceptUser; // unable to resume a previous session
    exception NOT_DEFINED {string aMessage;}; //type previously defined
    enum NotFoundReason { missing_node, not_context, not_object };
    exception NOT_FOUND {                // node in path not found
        NotFoundReason why;
        CosNaming::Name rest_of_name;
    };
    exception OPEN_LIMIT ExceptUser; // too many resources are open
    exception PREV_DEFINED {string aMessage;}; // type previously defined
    exception READ_LOCKED ExceptUser;          // reads prevented
    exception TIMEOUT ExceptUser;       // transaction timed out
    exception UNK_USER ExceptUser;    // Principal unrecognized
    exception WRITE_LOCKED ExceptUser;          // writes prevented
    exception SERVICE_XFR {ServiceLocation transfer;};  // resolve failed, alternate location given
};
```

5.4.6 Stream and Event Synchronization

Application time and scale values are defined to enable monitoring of continuous media streams. **AppNPT** indicates absolute time value, and **Scale** indicates rate and direction(forward or reverse). These are used in the DSM-CC **Stream** and **Event** interfaces. For a further description, see the Stream interface subclause.

```
module DSM {
    // stream and event synchronization
    //
    struct AppNPT {s_long aSeconds; u_long aMicroSeconds;};        // Normal Play Time
    struct Scale {s_short aNumerator; u_short aDenominator;};      //+FF,-Rewind, Rate
};
```

5.5 Application Portability Interfaces(API)

DSM-CC specifies interfaces as either abstract or instantiable. An abstract interface is never used to define a complete object. Instead it provides operations that are designed be inherited. An instantiable interface defines the interface of the complete, realizable Object Implementation. The instantiable interface can inherit one or more abstract or instantiable interfaces.

The Service Domain must implement all Core interfaces completely in order to be DSM-CC compliant. Extended interfaces are optional. If the Server implements an interface, it must implement all of the declarations, type definitions, attributes, access types, operations and exceptions of the interface. The minimal Client (Core Consumer) need only implement the specified Core READER plus File write() operations. A Client can choose to support selected operation groups by Access Role. For example, a Client could elect to support only the READER group of an interface, meaning only those operations in the interface with READER Access Role.

5.5.1 Core Interfaces

The Core DSM-CC User interfaces are the minimum set that must be supported by a DSM-CC compliant ServiceGateway Domain. They include the abstract interfaces Base, Access, NamingContext, and the instantiable interfaces Stream, File, Directory, Session and ServiceGateway.

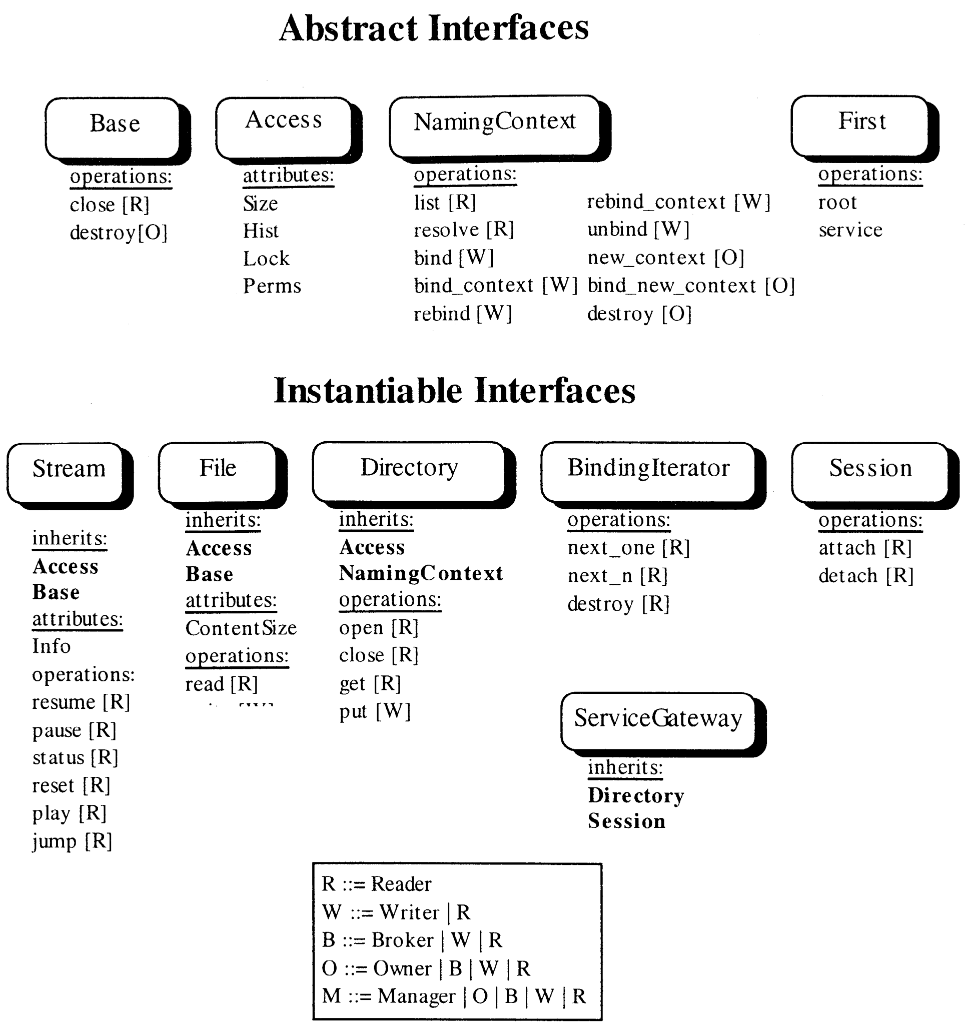


Figure 5-8 DSM Core Abstract and Instantiable Interfaces

5.5.1.1 Base

The Base interface provides common operations for deletion of DSM-CC object references and objects.

A Client (READER) may have obtained an object reference for an object, invoked some operations against it, and now has no more use for that object. The **Base close()** operation indicates the requesting Client no longer needs transient Client/Service connection-related resources associated with the object and shall not make any further requests.

When a Client (OWNER) wishes to delete the persistent data associated with an object, it may invoke the **Base destroy()** to destroy it, thus enabling the Object Implementation and related Services to free all state and storage resources associated with it. Following **destroy()**, the object shall cease to exist for all Clients.

5.5.1.1.1 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)

5.5.1.1.2 DSM Base close

DSM Base close	Close a reference to an object. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
  interface Base {
       const AccessRole close_ACR = READER;
       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 shall close whenever reasonable. If **OPEN_LIMIT** has been received when attempting to open an object reference, the Client shall need to close one or more other active references in order to free resources, before retrying the open.

Close means delete the object reference and therefore the Client's ability to communicate with the object.

If a parent Composite object is closed, its Child objects are closed as a result of the operation (See Composite interface description in the Extended Interfaces subclause).

Privileges Required:
READER

5.5.1.1.3 DSM Base destroy

DSM Base destroy	Destroy an object instance.

Application Portability / Service Inter-operability Syntax

```
module DSM {
  interface Base  {
#ifdef DSM_GENERAL
       const AccessRole destroy_ACR = OWNER;
       void destroy  ();
#endif
  };
};
```

Semantics

Base destroy() is used by the Client to delete a persistent Object Implementation. After this occurs, its reference will no longer be valid, and storage resources used for it will be freed.

Privileges Required:
OWNER

5.5.1.2 Access

The Access interface provides common description and access control attributes. These include size, version, date, lock and permissions attributes.

Principal End-User Clients are authorized to invoke operations through the following mechanisms:

- Capability Verification: A verification that the End-User has an Access Role (Capability) that matches or exceeds the Access Control Role of the operation.
    1. An Access Control Role is defined for each operation. e.g., OWNER, MANAGER, BROKER, WRITER or READER.
    2. The Principal End-User is the object OWNER, or a member of an authorized READER, WRITER, BROKER or MANAGER group in the Object Implementation's Service Domain.
- Authentication: Resolving an object or invoking selected operations may 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 Primary Services is performed by the Service Gateway or Directory of Services.

Authorization for access to an application's objects is performed by the Service or Directory in which the objects are bound..

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.

5.5.1.2.1 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 Domain boundaries, e.g., within the ServiceGateway, within a Directory, etc. The Client PrincipalId shall 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 shall identify groups that can access an object or invoke its operations. An object shall limit access to the Client who has the specified group or individual identification. Each method of that object shall 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 be carried in the ServiceContextList of the repeated open, in order to be granted access.

The OWNER may set an **allSecure** flag in an object's **Perms** attribute, indicating that all messages to/from the object must be secure, e.g., encrypted or scrambled. From resolve time through close, the Service shall effect secure transmission through appropriate lower network layer messaging.

5.5.1.2.2 Access Definitions

Application Portability / Service Inter-operability Syntax

```
 module DSM {
     struct DateTime {            // tm from ANSI C std. See Kernighan & Ritchie, 2nd edition, p. 255
         s_long tm_sec;           // seconds after midnight, 0-61
         s_long tm_min;           // minutes after the hour, 0-59
         s_long tm_hour;          // hours since midnight, 0-23
         s_long tm_mday;          // day of the month, 1-31
         s_long tm_mon;           // months since January, 0-11
         s_long tm_year;          // years since 1900
         s_long tm_wday;          // days since Sunday, 0-6
         s_long tm_yday;          // days since Jan 1, 0-365
         s_long tm_isdst;};       // Daylight Savings Time flag

     interface Access {
 #ifdef DSM_GENERAL
         // size
         const AccessRole Size_get_ACR = READER;
         readonly attribute u_longlong Size;          // size of all attributes in octets;
 #endif
         // history
         struct Hist_T {
                 Version  aVersion;          // object version
                 DateTime  aDateTime;};  // time created or last updated, GMT
 #ifdef DSM_GENERAL
         const AccessRole Hist_get_ACR = READER;
         const AccessRole Hist_put_ACR = BROKER;
         attribute Hist_T Hist;                  // version and time of persistent object
 #endif
         // lock status
         struct Lock_T {boolean readLock; boolean writeLock;};
         const AccessRole Lock_get_ACR = READER;
         const AccessRole Lock_put_ACR = WRITER;
         attribute Lock_T Lock;

         // permissions
         struct Perms_T {
                 // the next 4 are binary masks of binary flags signifying
                 // groups that can access the object
                 u_short managerPerm;
                 u_short brokerPerm;
                 u_longlong writerPerm;
                 u_longlong readerPerm;
                 opaque owner;                   //owner identifier = Principal
                 string aPassword;          //PIN
                 opaque authData;           //system-specific
                 // instruct lower layers to implement a secure connection for this object
                 boolean allSecure;};            // all methods parameters encrypted
 #ifdef DSM_GENERAL
         const AccessRole Perms_get_ACR = OWNER;
         const AccessRole Perms_put_ACR = OWNER;
         attribute Perms_T Perms;
 #endif
     };
 };
```

5.5.1.3 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 datatypes 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 indicates the stream absolute position relative to the beginning of the stream. Application NPT (AppNPT) is used for the application request interface and is specified in seconds and microseconds. MPEG Transport NPT is used to carry Normal Play Time in DSM-CC MPEG Stream descriptors. Transport NPT is specified in MPEG PTS 33 bit format. **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 one of the resolve operations. It returns an object reference for the stream, to be used with subsequent stream commands. Streams open in **O** mode, representative of Pause with AppNPT value 0,0. In case an internal error occurs (e.g.: a disk failure) the stream goes back to **O** mode. More than one stream can be opened at a time.

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	Application NPT is 0,0. The Server is not transporting the stream.
Pause	The Server is not transporting the media stream
Search Transport	The Server is searching for start NPT. When at start NPT, it will transport the media stream.
Transport	The Server is transporting the media stream and will pause at end of stream.
Transport Pause	The Server is transporting the media stream and will pause at stop NPT.
Search Transport Pause	The Server is searching for start NPT. When at start NPT, it will transport the media stream until stop NPT.
Pause Search Transport	The Server is transporting the media stream. It will transport the media stream until stop NPT, then will search to start NPT and transport the media stream.
End of Stream	Server NPT is at the maximum NPT of the stream. The Server is not transporting the stream.
Pre Search Transport	This is an exceptional status. The Server is stopping the sending of the media stream. Once it has stopped, it transitions to Search Transport
Pre Search Transport Pause	This is an exceptional status. The Server is stopping the sending of the media stream. Once it has stopped, it transitions to Search Transport Pause

Stream status() is used to inquire as to the current **AppNPT**, Scale and Mode of an open stream.

5.5.1.3.1 Stream Definitions, Exceptions

```
module DSM {
    interface Stream : Base, Access {
        // stream modes
        typedef u_long Mode;
        const Mode OPEN_M = 0;
        const Mode PAUSE_M = 1;
        const Mode TRANSPORT_M = 2;
        const Mode TRANSPORT_PAUSE_M = 3;
        const Mode SEARCH_TRANSPORT_M = 4;
        const Mode SEARCH_TRANSPORT_PAUSE_M = 5;
        const Mode PAUSE_SEARCH_TRANSPORT_M = 6;
        const Mode END_OF_STREAM_M = 7;
        const Mode PRE_SEARCH_TRANSPORT_M = 8;
        const Mode PRE_SEARCH_TRANSPORT_PAUSE_M = 9;
        struct Stat {
                AppNPT rPosition;
                Scale rScale;
                Mode aMode;};
        struct Info_T {
                string<255> aDescription;
                AppNPT duration;
                boolean audio;
                boolean video;
                boolean data;};
        const AccessRole Info_get_ACR = READER;
        const AccessRole Info_put_ACR = OWNER;
        attribute Info_T  Info;
    };
};
```

Note: **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.

5.5.1.3.2 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 time codes. Application NPT is defined as two values representing seconds and microseconds.

To understand DSM-CC'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 **AppNPT** start time 30.
2. At 10 seconds Stream pause is called. At this point **AppNPT** will be 40.
3. At 16 seconds Stream resume is called requesting that the stream continue playing at **AppNPT** = 80 at ten times normal speed.

4. At 26 seconds Base close is called. At this point **AppNPT** will be 180.

The coordination of NPT between the Server and the Client is independent of the API usage of **AppNPT**. There are two possible methods for maintaining NPT in the Client. The first method is to use NPT descriptors as described in the Normal Play Time subclause. The other method is to explicitly query the Server for **AppNPT** information. Due to latency considerations the second method may be less accurate.

5.5.1.3.2.1 Application NPT Values

The following restrictions apply to **AppNPT** values:

- The beginning of the Stream corresponds to 0 **aSeconds**, 0 **aMicroSeconds**.
- **AppNPT** values ascend from the beginning of stream to end of stream.
- **AppNPT** pause now and resume now value is indicated by negative infinity (0x80000000) **aSeconds**. These represent the Server view of what "now" means, i.e. when the Server receives a pause or resume **AppNPT** specified as 0x80000000 aSeconds, it will pause or resume at the earliest possible time. This is the only negative value allowed.
- **AppNPT** is the addition of **aSeconds** and **aMicroSeconds**.
- **aMicroSeconds** shall be less than 1000000.

5.5.1.3.3 Summary of Stream Primitives

Inherited from **Base**:

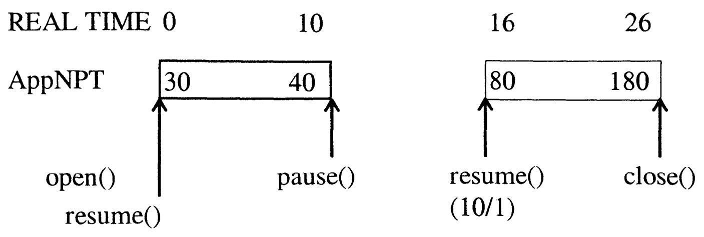
close, destroy

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 Stream resume	Start sending stream at NPT position within stream. (READER)
DSM Stream status	Obtain status of a stream. (READER)
DSM Stream reset	Reset the stream state machine. (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)



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 **AppNPT** 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 **AppNPT** remains at a point 40 seconds from the start of the stream. 6 seconds later, the viewer initiates fast forward with **AppNPT** 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.

5.5.1.3.4 Stream State Machine

The stream interface provides operations to control the delivery of a media stream, through pause, resume, etc. commands. The Server is represented as a stream object that provides this interface and can transport MPEG over the network. Upon receipt of a command, it will modify its state machine, and perform the indicated function. Since each command is atomic, if any of the supplied parameters is incorrect the whole command is ignored.

The stream temporal status is comprised of current NPT, start NPT, stop NPT, mode and viewing rate (scale).

Note that the DSM-CC Library provides the Client stream status with the **status()** function. The status value which the remote stream object returns allows the set-top 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(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(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.)

Since the Client can cascade methods, the sequence **resume(rStart)** plus **pause(rStop)** essentially equates to **play(rStart, rStop)**. Also the sequence **pause(rStop)** plus **resume(rStart)** essentially equates to **jump(rStop, rStart)**. However, an exception to this general rule exists, in the case a **play(rStart, rStop)** is received from the Server while the state machine is in either **T**, **TP** or **PST** mode: this exception aims to accomodate the requirement of having **play(rStart, rStop)** actually meaning play from **rStart** to **rStop**. There is no corresponding exception for the **jump()** command. Moreover it is worth noting that as all methods are atomic, the sequence of a correct **resume()** plus an incorrect **pause()** would affect the state machine according to the **resume()**, while the single **play()** would not. Similarily, the sequence of a correct **pause()** plus an incorrect **resume()** would affect the state machine according to the **pause()**, while the single **jump()** would not.

5.5.1.3.4.1 State Machine

The interface controls a state machine. The state machine, shown below with the main transitions, comprises the ten states of a) Open b) Pause c) Transport d) Transport Pause e) Search Transport f) Search Transport Pause g) Pause Search Transport h) End of Stream i) Pre Search Transport and l) Pre Search Transport Pause.

The open causes the state to transition to the Open state. Open state is equivalent to Pause state at **AppNPT** 0.0. The default values are a) rStart=0x00000000.0x00000000, b) rStop=0xEFFFFFFF.0xFFFFFFFF, and c) Scale=1,1.

If the state machine receives a **pause(rStop)** while a stop NPT is still set (that is: has not been reached yet), the **pause(rStop)** replaces the previous stop NPT. If the state machine receives **resume(rStart)** while a start NPT is still set (that is: has not been reached yet), the **resume(rStart)** replaces the previous start NPT. If the state machine receives a **resume(rStart)** while in **T** mode, or a **resume(rStart**=now) while in **TP** or **PST** modes, it stops sending the media stream, sets the start NPT, and clears the stop NPT. If the state machine receives a **play(rStart, rStop)** while in **T**, **TP** or **PST** mode, it stops sending the media stream and sets the start and stop NPTs.

The figure below shows the stream profile which corresponds to each state.

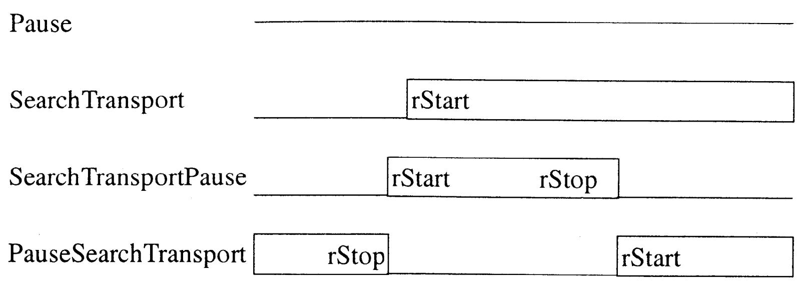


Figure 5-9 rStart and rStop State

O	Open	There is one phase. Application NPT is 0,0. The Server is not transporting the stream.
P	Pause	There is one phase. The Server is not transporting the media stream
ST	Search Transport	There are two phases with one time value to describe the transition. The Server is searching for start NPT. When at start NPT, it will transport the media stream. Since there is no rStop, the Server will continue to advance the stream and pause at End of Stream.
T	Transport	The Server is transporting the media stream. and will pause at End of Stream.
TP	Transport Pause	The Server is transporting the media stream and will pause at stop NPT.
STP	Search Transport Pause	There are three phases with two time values to describe the transitions. The Server is searching for start NPT. When at start NPT, it will transport the media stream until stop NPT. The common sequence which causes the state transition is either **play(rStart, rStop)** or **resume(rStart)** plus **pause(rStop)**.
PST	Pause Search Transport	There are three phases with two time values to describe the transitions. The Server is transporting the media stream. It will transport the media stream until stop NPT, then will search to start NPT and transport the media stream. The common sequence which causes the state transition is either **jump(rStop, rStart)** or **pause(rStop)** plus **resume(rStart)**.
EOS	End of Stream	Server NPT is at the maximum NPT of the stream. The Server is not transporting the stream.

PreST	Pre Search Transport	This is an exceptional status. The Server is stopping sending the media stream. Once it has stopped, it transitions to Search Transport
PreSTP	Pre Search Transport Pause	This is an exceptional status. The Server is stopping sending the media stream. Once it has stopped, it transitions to Search Transport Pause

The semantics of **Stream reset()** are to return to the Open state, which is a variation of the Pause state. The EndofStream is also a variation of the Pause state, where Application NPT is equal to the maximum NPT of the stream.

For Open and End of Stream states, the normal state transitions for Pause apply.

End of Stream Stream Mode descriptor shall be sent to the Client in the MPEG stream.

5.5.1.3.4.2 Basic State Machine

The interface below realizes the Basic State Machine with just the **Stream resume()** and **pause()** methods. For readibility reasons, the figure does not include the **O** nor **EOS** modes, as they behave exactly as the **P** mode, nor the **reset()** method, as it is trivial and causes the state machine to return to the **O** mode. It also does not include the **PreST** mode, **PreSTP** mode, the **resume(rStart)** method occurring in **T** mode, nor the **resume(rStart=now)** method occurring in **TP** or **PST** modes, as they are better explained in the subsequent paragraph. The interface does not include **play(rStart, rStop)** because the function is almost identical to the sequence **resume(rStart)** plus **pause(rStop)**: the exception to this general rule is highlighted in the subsequent paragraph. The interface does not include **jump(rStop, rStart)** because the function is identical to sequence **pause(rStop)** plus **resume(rStart)**.

The Basic State Machine is shown below.

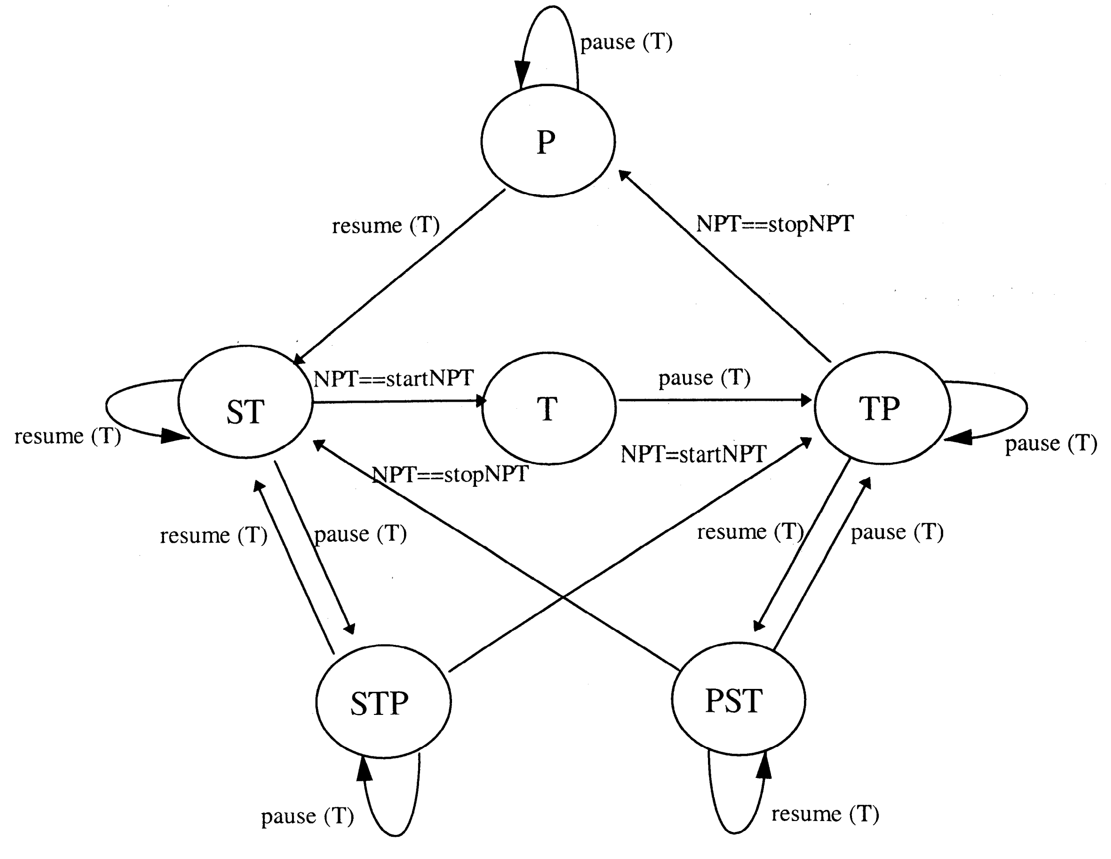


Figure 5-10 State Machine Transitions

The table below shows the transitions which the **resume()** and **pause()** method cause.

Previous State	resume()	pause()
ST	ST	STP
T	PreST	TP
TP	PST[1]	TP
P	ST	P
STP	ST	STP
PST	PST[1]	TP
PreST	PreST	PreSTP
PreSTP	PreST	PreSTP

Note 1: if the **rStart** parameter in **resume()** equals 'now', then the transition is to **PreST**.

The table below shows the transitions which the **startNPT** condition and the **stopNPT** condition cause.

Previous State	AppNPT = startNPT	AppNPT = stopNPT
O		
ST	T	
T		
TP		P
P		
STP	TP	
PST		ST

If the state was SearchTransport, the machine transitions to Transport when the stream position reaches the startNPT. If the state was SearchTransportPause, the machine transitions to TransportPause when the stream position reaches the startNPT. If the state was TransportPause, the machine transitions to Pause when the stream position reaches the stopNPT. If the state was PauseSearchTransport, the machine transitions to SearchTransport when the stream position reaches the stopNPT.

Transition due to startNPT or stopNPT cannot occur in PreSearchTransport or PreSearchTransportPause, as they are just defined as temporary modes entered by the State Machine while stopping sending the stream.

5.5.1.3.4.3 Complete state machine

There is an exceptional case where **play(rStart, rStop)** is not equivalent to the sequence **resume(rStart)** plus **pause(rStop)**: this happens when a **play(rStart, rStop)** is received while the Server is sending the media stream. The sequence **resume(rStart)** plus **pause(rStop)** would in essence cancel the previously set stop NPT and start NPT, and only set the stop NPT to **rStop**. This would cause the Server to continue playing up to **rStop**.

The above behavior does not correspond to the definition of the **play()** operation, thus a different transition needs to be defined. As the expected behavior is to cease sending the media stream, jump to **rStart** and send the media stream up to **rStop**, a new exceptional mode is defined to accomodate a transient condition. Upon receiving **play(rStart, rStop)** while in **T**, **TP** or **PST** mode, the state machine transitions to **PreSTP** and sets the start and stop NPTs: as soon as the Server ceases sending the stream, the state machine transitions to **STP**. This transition is expected to occur almost immediately.

If the state machine receives a **pause(rStop)** before leaving the **PreSTP** mode, it updates the stop NPT and remains in the **PreSTP** mode. If the state machine receives a **resume(rStart)** before leaving the **PreSTP** mode, it updates the start NPT, cancels the stop NPT and transitions to **PreST**. Then, as soon as the Server ceases sending the stream, the state machine transitions to **ST**. This transition is expected to occour almost immediately.

If the state machine receives a **resume(rStart)** before leaving the **PreST** mode, it updates the start NPT and remains in the **PreST** mode. If the state machine receives a **pause(rStop)** before leaving the **PreST** mode, it sets the stop NPT and transitions to **PreSTP** mode.

The **PreST** mode is also entered when a **resume(rStart)** is received while in **T** mode, or a **resume(rStart**=now) is received while in **TP** or **PST** mode, as the expected behavior of the Server is to cease sending the stream at the earliest possible time, and then resume from **rStart**.

5.5.1.3.5 DSM Stream pause

DSM Stream pause	Stop sending stream when AppNPT position is reached.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access {
        const AccessRole pause_ACR = READER;
        void pause (in AppNPT rStop)
                raises (MPEG_DELIVERY, BAD_STOP);
    };
};
```

Semantics

A Client calls **Stream pause()**to cause the video Server to stop sending the stream when it reaches **AppNPT rStop**.

If a **Stream pause()** is invoked while the state machine is in **T** mode, the state machine will immediately transition to **TP** mode.

If a **Stream pause()** is invoked while the state machine is in **TP** mode, the state machine will remain in **TP** mode, but the **rStop** value will be updated.

If a **Stream pause()** is invoked while the state machine is in **STP** or **ST** mode, the state machine will immediately transition to **STP** mode.

If a **Stream pause()** is invoked while the state machine is in **PST** mode (i.e., a resume is queued at **rStart**), the state machine will transition to **TP** mode, indicating MPEG Stream delivery will cease when **rStop** is reached, and no resume will occur.

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 **AppNPT** 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 negative infinity or an **rStop** greater than the current **AppNPT** will indicate pause immediately if there are no other commands in the stream state machine queue. An **rStop** which exceeds the Stream duration will result in a **BAD_STOP** exception.

Privileges Required
READER

Parameters

type/variable	direction	description
AppNPT rStop	input	AppNPT position at which the pause will occur.

5.5.1.3.6 DSM Stream resume

DSM Stream resume	Start sending stream at AppNPT position.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access {
        const AccessRole resume_ACR = READER;
        void resume (in AppNPT rStart, in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_SCALE);
    };
};
```

Semantics

A Client calls **Stream resume()** to cause the video Server to resume sending the stream at **rStart** at a rate and direction as specified by **rScale**.

If a **Stream resume()** is invoked while the state machine is in **ST**, **P**, or **STP** mode, the state machine will immediately transition to **ST** mode. The Server will then commence sending the stream from the position **rStart** (with scale **rScale**) at the earliest possible time. When the Server begins to send the MPEG stream the state machine will transition to **T** mode.

If a **Stream resume()** is invoked while the state machine is in **T** mode, or a **Stream resume(rStart**=now) is invoked while the state machine is in **TP** or **PST** mode, the state machine shall immediately transition to **PreST** mode. At the earliest possible time the Server will stop sending the stream and transition to **ST**; then, at the earliest possible time, the Server will commence sending the stream from the position **rStart** (with scale **rScale**) and transition to **T** mode.

If a **Stream resume(rStart**≠now) is invoked while the state machine is in **PST** mode, the state machine will update **rStart** and its associated **rScale**, and remain in **PST** mode. If a **Stream resume(rStart**≠now) is invoked while the state machine is in **TP** mode, the state machine will transition to **PST** mode and set **rStart** and its associated **rScale**. Both of these indicate that when the state machine will reach **rStop**, the state machine will transition to **ST** and then **T** mode as described above, and MPEG stream delivery will commence from the new **rStart** (with scale **rScale**).

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, with the exception of negative infinity which indicates resume now. An **rStart** which equals or exceeds the Stream duration will result in a **BAD_START** exception.

An **rScale** of 0/0 is indeterminate and will result in a BAD_SCALE exception. An **rScale** of 0/n (where n is not 0) indicates that the NPT is not changing.

Privileges Required
READER

Parameters

type/variable	direction	description
AppNPT rStart	input	AppNPT 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.

5.5.1.3.7 DSM Stream status

DSM Stream status	Obtain status of a stream.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access {
        const AccessRole status_ACR = READER;
        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 **AppNPT** 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.

Privileges Required
READER

Parameters

type/variable	direction	description
Stat rAppStatus	input	Expected stream status, AppNPT, Scale and Mode
Stat rActStatus	output	Actual stream status, AppNPT, Scale and Mode.

5.5.1.3.8 DSM Stream reset

DSM Stream reset	Reset the stream state machine. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access{
        const AccessRole reset_ACR = READER;
        void reset ();
    };
};
```

Semantics

Stream reset() is used to reset the Stream state machine to the **O** state (Pause, rStart = 0 0, rstop = 0xEFFF 0xFFFF).

Privileges Required
READER

5.5.1.3.9 DSM Stream jump

DSM Stream jump	When stream reaches stop AppNPT, resume at start AppNPT.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access{
        const AccessRole jump_ACR = READER;
        void  jump (
                    in AppNPT rStart,
                    in AppNPT rStop,
                    in Scale rScale)
                    raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE);
    };
};
```

Semantics

Stream jump(rStart, rStop) behaves exactly as if **Stream pause(rStop)** then **Stream resume(rStart)** have been called in quick succession. However **Stream jump()** is a single method, thus if an incorrect **rStart** or **rScale** parameter is given, the **rStop** is ignored too and the state machine is not affected at all. There are no other exceptions to the equivalence between **Stream jump(rStart, rStop)** and the sequence **Stream pause(rStop)** plus **Stream resume(rStart)**.

Privileges Required
READER

Parameters

type/variable	direction	description
AppNPT rStart	input	AppNPT position to resume from as a result of the jump.
AppNPT rStop	input	AppNPT position at which the jump will occur.
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.

5.5.1.3.10 DSM Stream play

DSM Stream play	Play stream from start AppNPT until stop AppNPT.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access{
        const AccessRole play_ACR = READER;
        void play (
                in AppNPT rStart,
                in AppNPT rStop,
                in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE);
    };
};
```

Semantics

Stream play(rStart, rStop) behaves almost exactly as if **Stream resume(rStart)** then **Stream pause(rStop)** have been called in quick succession. However **Stream play()** is a single method, thus if an incorrect **rStop** parameter is given, the **rStart** and the **rScale** are ignored too and the state machine is not affected at all. In addition, if **Stream play(rStart, rStop)** is invoked while the state machine is in **TP** or **PST** mode, it behaves differently from the sequence **Stream resume(rStart)** plus **Stream pause(rStop)**, as described already.

Privileges Required
READER

Parameters

type/variable	direction	description
AppNPT rStart	input	AppNPT position at which to resume play.
AppNPT rStop	input	AppNPT 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.

5.5.1.4 File

This subclause describes the interface for two operations that read and write files. When combined with the other Core DSM interfaces such as 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 can be realized. The IDL permits the Server Object Implementation to map to any of a variety of heterogeneous object and file systems.

5.5.1.4.1 File Definitions, Exceptions

```
module DSM {
    interface File : Base, Access {
        const AccessRole Content_get_ACR = READER;
        const AccessRole Content_put_ACR = WRITER;
        attribute opaque Content;              // file content
#ifdef DSM_GENERAL
        const AccessRole ContentSize_get_ACR = READER;
        readonly attribute u_longlong ContentSize;  // file content size in octets
#endif
    };
};
```

The ContentSize is implicitly updated at any time that the Contents of the File are updated. Contents of the File can be updated with File write(), File _set_Content() to the File object, or **Directory_put** (of the Content attribute) to the parent Directory object. A **Directory_get** (of the Content attribute) at the parent Directory object is equivalent to opening and reading an entire file. **Directory_get()** and **Directory_put()** are described later in this clause.

5.5.1.4.2 Summary of File Primitives

Inherited from **Base**:

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)

5.5.1.4.3 DSM File read

DSM File read	Random access read from a file. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface File : Base, Access {
        const AccessRole read_ACR = READER;
        void read (
                in u_longlong aOffset,
                in u_long aSize,
                in boolean aReliable,
                out opaque rData)
                raises (INV_OFFSET, INV_SIZE, READ_LOCKED);
    };
};
```

Semantics

File read() provides random access to opened files, using a File reference obtained from a previous resolve operation. **aOffset** is the absolute offset into the File Content attribute (from 0). Because **aOffset** and **aSize** are explicit parameters, seeks can be accomplished assuming the application maintains the current byte position in the file.

aSize specifies the amount of buffer the Client has allocated for the read. It can exceed the File (**ContentSize** - **aOffset**), in order to specify a read to End-of-File. **INV_OFFSET** shall be returned if **aOffset** is greater than File (**ContentSize** - 1). **INV_SIZE** shall be returned for **aSize** that exceeds the capacity of Server, Network Resources or Client as derived from Compatibility Information.

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 pre-fetch 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 to re-fetch lost or erroneous data if **aReliable** is TRUE. If **aReliable** is FALSE, the operation will not be retried in the event of time-out 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 re-fetched.

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.
opaque rData	output	Pointer to data returned by the File read.

5.5.1.4.4 DSM File write

DSM File write	Random access write to a file. (WRITER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface File : Base, Access {
        const AccessRole write_ACR = WRITER;
        void  write (
                    in u_longlong aOffset,
                    in u_long aSize,
                    in opaque 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. **aOffset** is the absolute offset into the File Content attribute (from 0). WRITER privileges are required. The **File write()** uses the File reference obtained from a previous resolve operation. 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. Appends may be performed by using the **ContentSize** attribute of the File as **aOffset**. **INV_OFFSET** shall be returned if **aOffset** is greater than **ContentSize**. **INV_SIZE** shall be returned for **aSize** that exceeds the capacity of Server, Network Resources or Client as derived from Compatibility Information.

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.
opaque rData	input	Pointer to data to be written.

5.5.1.5 Directory

The Directory interface provides a general name space for binding names to Services or data. A ServiceGateway 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 readily support DSM-CC, 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 **Directory close()** operation.

For completeness, the NamingContext operations are presented below; however, the CORBA Naming specification is the reference definition of operations that a directory must support. In some cases, DSM-CC defines equivalent syntax that supports DSM-CC extensions that are compatible with CosNaming. For example, the macro DSM_GENERAL is used to enable compilation of the Core Consumer Client.

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.

When an object is bound to a Directory using **Directory bind()** or **Directory bind_context()**, an objects **id**, **kind** and **BindingType** become known to the Directory. The **BindingType** is either a object or a name context, depending upon which of the two functions was called. In DSM-CC, the **id** is expected to be the simple string name of the object, and the **kind** is the interface type of the object, in the form "<module>::<interface>" or specified interface alias. The application can see the **id**, **kind** and **BindingType** as a result of the **Directory list()** operation.

A **Directory resolve()** or **Directory open(..DEPTH..)** with multiple **NameComponents** results in a cascade of single resolves, when following the path depth-wise. Each node in the path before the last one shall be a Directory and will propagate the resolve to the rest of the path. The last node in the path may or may not be a Directory.

5.5.1.5.1 Directory Definitions, Exceptions

Interface names are constructed from NameComponents. The NameComponent has two parameters, an id and a kind. The id is an arbitrary string selected by the application. In the DSM-CC environment, the kind is required to be a string constructed of module name and interface, in the form "<module>::<interface>", or a DSM-CC interface alias. For example, DSM-CC Directory uses "DSM::Directory" or the alias "dir" for the NameComponent kind value.

The CORBA Naming Service is used as a basis for directory functions. CosNaming definitions are shown below:

```
module CosNaming{
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
    // note: BindingType equates to the CORBA enum definition
    // while allowing extension for DSM-CC implementations
    typedef unsigned long BindingType;
    const BindingType nobject = 0;
    const BindingType ncontext = 1;
    struct Binding {
        Name binding_name;
        BindingType binding_type;
    };
    typedef sequence <Binding> BindingList;
};
```

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). The structure for these values is determined by their type. 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 NameComponents specified in a single **Directory open()** is specified by a **aPathType** and **rPathSpec**.

The **aPathType** 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 **rPathSpec** 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 TRUE, the **Step** name is to be resolved, otherwise the name is simply used to traverse further along the path. If the operation using **rPathSpec** returns object references, these will be contained in a separate resolved references output parameter. If the operation using **rPathSpec** returns data values, these will be contained in a separate resolved values output parameter. The definition of these types is as follows:

```
module DSM {
    // two types of path traversal, depth and breadth match traditional methods
    typedef char PathType;          // DEPTH or BREADTH
    const PathType DEPTH = 'D';
    const PathType BREADTH = 'B';
    struct Step {
        CosNaming::NameComponent name;
        boolean process;
    };
    typedef sequence<Step> PathSpec;
    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-CC environment or defined explicitly by a DSM-CC implementation.

```
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 {};
    };
};
```

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 InvalidName, AlreadyBound, and NotEmpty 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.

DSM::NOT_FOUND is equivalent to CosNaming::NamingContext NotFound. DSM::CANNOT_PROCEED is equivalent to CosNaming::NamingContext CannotProceed. DSM::INV_NAME is equivalent to CosNaming::NamingContext InvalidName. These exceptions are used in multiple DSM-CC interfaces, and are defined in the Common Definitions subclause.

5.5.1.5.2 Summary of Directory Primitives

<u>Inherited from **Access**:</u>

attributes: **Size, Hist, Lock, Perms**

<u>Inherited from **NamingContext**:</u>

DSM Directory list	Return a list of all the bindings to object references in the context. (READER)
DSM Directory resolve	Given a name, return an object reference for a Service Object Implementation instance. (READER)
DSM Directory bind	Bind an object reference to a name. (WRITER)
DSM Directory bind_context	Bind a naming context to a name. (WRITER)
DSM Directory rebind	Bind an object reference to a name, overwriting any previous binding. (WRITER)
DSM Directory rebind_context	Bind a context to a name, overwriting any previous binding. (WRITER)
DSM Directory unbind	Remove a binding for a name. (WRITER)
DSM Directory new_context	Create a new naming context. (OWNER)
DSM Directory bind_new_context	Create a new naming context and bind it to the given name. (OWNER)
DSM Directory destroy	Destroy the naming context. (OWNER)

<u>Defined in **Directory**:</u>

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 attribute values bound to a PathSpec. (READER)
DSM Directory put	Bind attribute values to a PathSpec, overwriting any previous bindings. (WRITER)

Directory defines AccessRoles of the inherited NamingContext operations as follows:

```
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;
    };
};
```

5.5.1.5.3 DSM Directory list

DSM Directory list	Return a list of all the bindings to object references in the context. (READER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
        void list (in unsigned long how_many,
                   out BindingList bl, out BindingIterator bi);
    };
};
```

Semantics

The list operation returns a list of bindings in the Directory. 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 **BindingIterator 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 **BindingIterator destroy()** operation discards any Server-side storage associated with the iterator and makes the iterator no longer valid to access.

Privileges Required:
READER

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.

5.5.1.5.4 DSM Directory resolve

DSM Directory resolve	Given a name, return an object reference for a Service Object Implementation instance. (READER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
        Object resolve (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
    };
};
```

Semantics

The **Directory resolve**() operation returns an object reference that is an Object Implementation instance of the name binding. If no name is bound, then the **NotFound** exception is raised.

Privileges Required:
READER

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.

5.5.1.5.5 DSM Directory bind

DSM Directory bind	Bind an object reference to a name. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void bind (in Name n, in Object obj)
            raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
#endif
    };
};
```

Semantics

The **Directory 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.

5.5.1.5.6 DSM Directory bind_context

DSM Directory bind_context	Bind a naming context to a name. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void bind_context (in Name n, in NamingContext nc)
            raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
#endif
    };
};
```

Semantics

The **Directory 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 **Directory 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.

5.5.1.5.7 rebind

rebind	Bind an object reference to a name, overwriting any previous binding. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void rebind (in Name n, in Object obj)
            raises (NotFound, CannotProceed, InvalidName);
#endif
    };
};
```

Semantics

The **Directory rebind()** operation associates an object reference with a name in a directory. Unlike the **Directory 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.

5.5.1.5.8 DSM Directory rebind_context

DSM Directory rebind_context	Bind a naming context to a name. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void rebind_context (in Name n, in NamingContext nc)
            raises (NotFound, CannotProceed, InvalidName);
#endif
    };
};
```

Semantics

The **Directory rebind_context()** operation associates a naming context with a name. Unlike the **Directory 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.

5.5.1.5.9 DSM Directory unbind

DSM Directory unbind	Remove a binding for a name. (WRITER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void unbind (in Name n)
            raises (NotFound, CannotProceed, InvalidName);
#endif
    };
};
```

Semantics

The **Directory 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.

5.5.1.5.10 DSM Directory new_context

DSM Directory new_context	Create a new naming context. (OWNER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        NamingContext new_context();
#endif
    };
};
```

Semantics

The **Directory new_context()** operation returns a newly-created NamingContext.

Privileges Required:
OWNER
Parameters

type/variable	direction	description
NamingContext	output	The newly-created naming context.

5.5.1.5.11 DSM Directory bind_new_context

DSM Directory bind_new_context	Create a new naming context and bind it to the given name. (OWNER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
   interface NamingContext {
#ifdef DSM_GENERAL
        NamingContext bind_new_context(in Name n)
                raises (AlreadyBound, NotFound, CannotProceed, InvalidName);
#endif
   };
};
```

Semantics

The **Directory 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.

5.5.1.5.12 DSM Directory destroy

DSM Directory destroy	Destroy the naming context. (OWNER)

Application Portability / Service Inter-operability Syntax

note: Directory inherits from CosNaming::NamingContext.

```
module CosNaming {
    interface NamingContext {
#ifdef DSM_GENERAL
        void destroy ()
            raises (NotEmpty);
#endif
    };
};
```

Semantics

The **Directory destroy()** operation destroys the Directory object. It does not destroy the objects that were bound to it.

Privileges Required:
OWNER

5.5.1.5.13 DSM Directory open

DSM Directory open	Find the objects associated with the names in the given path (READER).

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        const AccessRole open_ACR = READER;
        void open(
                in PathType aPathType,
                in PathSpec rPathSpec,
                out ObjRefs resolvedRefs)
                raises(OPEN_LIMIT, NO_AUTH, UNK_USER, SERVICE_XFR,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
    };
};
```

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 sequence of object references that corresponds to the **Steps** of the input **rPathSpec** that have process flag set. Therefore, note that the length of **rPathSpec** does not necessarily correspond to the length of the returned object references.

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.

Directory open() can be used to open multiple objects at one time. **Directory open()** can also traverse the name graph in either breath-first or depth-first fashion. **Directory resolve()**, on the other hand, opens a single object at the end of a depth-first path.

Directory open() and **Directory resolve()** are similar, but not equivalent. **Directory resolve()** shall return a single object reference which is at the end of a sequence of **NameComponents**. **Directory open()** can be used to obtain more than one object reference, either by depth-first path traversal or breadth-first path traversal. **Directory open()** can be used to resolve a single object, but there is more overhead in the message. **Directory open()** is useful where the Client needs to resolve a number of objects at one time (typically at a startup or transition points). The single message can improve response time, especially in systems where there is high latency on the request channel between the Client and the Service Domain. Regardless of whether an object is a file or a directory, a Client can use either **Directory resolve()** or **Directory open()** to obtain the object reference. **Directory resolve()** is equivalent to a **Directory Open()** in which **PathType** = DEPTH and the **PathSpec** has all process flags set to FALSE, except that last **Step** in the path, and the last **Step** represents a single object. See also Composite Application Portability and Service Inter-operability interfaces subclauses, for a variation of depth-first traversal.

Directory open() in which **PathType** = BREADTH indicates a an explicit resolve of multiple objects at the target Directory (flat, not hierarchical).

Privileges Required:
READER

Parameters

type/variable	direction	description
PathType aPathType	input	Defines the traversal form of rPathSpec.
PathSpec rPathSpec	input	A sequence of **Steps**, each representing a node in a directory hierarchy.
ObjRefs rPathRefs	output	The object references resolved as a result of this operation.

5.5.1.5.14 DSM Directory close

DSM Directory close	Close a reference to a Directory. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        const AccessRole close_ACR = READER;
        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.

If a Directory object is closed, object references that were resolved using that Directory remain valid.

Privileges Required:
READER

5.5.1.5.15 DSM Directory get

DSM Directory get	Return the attribute values bound to a PathSpec. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
        const AccessRole get_ACR = READER;
        void get(
                in PathType aPathType,
                in PathSpec rPathSpec,
                out PathValues rPathValues)
                (NO_AUTH, UNK_USER, SERVICE_XFR,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
    };
};
```

Semantics

The **Directory get()** operation provides a path traversal that returns one or more attribute values of a Directory entry, without returning any object references. **aPathType** specifies DEPTH or BREADTH. If **aPathType** is DEPTH, the **Steps** of **rPathSpec** are <Directory>, <Directory>, ... , <Object>, <Attribute>, indicating a linear path traversal and single attribute access. If **aPathType** is BREADTH, the **Steps** of rPathSpec shall be <Object>, <Attribute>, ...,<Attribute>. indicating multiple attribute access.

This operation looks up each path node sequentially, but not atomically (other directory operations may occur between the lookups of nodes). If the a particular resolve fails, then the entire operation raises the appropriate exception.

Directory get() and **Directory put()** operations are used to retrieve and set the values of an object's attributes, without resolving an object reference. The attribute values are stored at the Directory Service.

The data structures for the **rPathValues** of **Directory get()** and **Directory put()** are determined by their type. The TypeCode of the CORBA_any of **rPathValues** shall identify the structure of the attribute value.

The attribute NameComponent kind may be NULL, because rPathValues identifies the kind. The attribute NameComponent id shall be the identifier of the attribute, as defined in the IDL. For example, The id for the attribute DSM::File::Content shall be "Content".

Privileges Required:
READER

Parameters

type/variable	direction	description
PathType aPathType	input	Defines the traversal form rPathSpec.
PathSpec rPathSpec	input	A sequence of **Steps**, representing nodes in a directory hierarchy and attributes of a Directory entry.
PathValues rPathValues	output	The attribute values returned as a result of this operation.

5.5.1.5.16 DSM Directory put

DSM Directory put	Bind attribute values to a PathSpec, overwriting any previous bindings. (WRITER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Directory : Access, CosNaming::NamingContext {
#ifdef DSM_GENERAL
        const AccessRole put_ACR = WRITER;
        void put(
                in PathType aPathType,
                in PathSpec rPathSpec,
                in PathValues rPathValues)
                raises(UNK_USER, NO_AUTH, SERVICE_XFR,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
#endif
    };
};
```

Semantics

The **Directory put()** operation provides a path traversal that sets one or more attribute values of a Directory entry, without returning any object references. **aPathType** specifies DEPTH or BREADTH. If **aPathType** is DEPTH, the **Steps** of **rPathSpec** are <Directory>, <Directory>, ... , <Object>, <Attribute>, indicating a linear path traversal and single attribute access. If **aPathType** is BREADTH, the **Steps** of rPathSpec shall be <Object>, <Attribute>, ...,<Attribute>. indicating multiple attribute access.

This operation looks up each path node sequentially, but not atomically (other directory operations may occur between the lookups of nodes). If the a particular resolve fails, then the entire operation raises the appropriate exception.

Directory get() and **Directory put()** operations are used to retrieve and set the values of an object's attributes, without resolving an object reference. The attribute values are stored at the Directory Service.

The data structures for the **rPathValues** of **Directory get()** and **Directory put()** are determined by their type. The TypeCode of the CORBA_any of **rPathValues** shall identify the structure of the attribute value.

The attribute NameComponent kind may be NULL, because rPathValues identifies the kind. The attribute NameComponent id shall be the identifier of the attribute, as defined in the IDL. For example, The id for the attribute DSM::File::Content shall be "Content".

Privileges Required:
WRITER

Parameters

type/field	direction	description
PathType aPathType	input	Defines the traversal form rPathSpec.
PathSpec rPathSpec	input	A sequence of **Steps**, representing nodes in a directory hierarchy and attributes of a Directory entry.
PathValues rPathValues	input	The attribute values to be bound as a result of this operation.

5.5.1.6 Session

The Session interface enables a Client to establish a Session with a ServiceGateway Domain of Services, using the operation **Session attach()**. While in the Session, the Client can navigate the Domain of Services, open new Services, and perform any operations the Service interfaces support. When finished, the Client invokes **Session detach()** to close, i.e., perform **Base close()** against, all object references of the Session.

5.5.1.6.1 Service Transfer

Session attach() and **Session detach()** can be used to perform Service Transfer. Most applications will have some nesting of navigation and will require a suspension of one level of nest when proceeding to another. For example, a top level navigator can be located in one Service Domain and a movie browser in another. When going from the top navigator to the movie browser, the Client can suspend the navigator and attach to the movie browser. Then, when done with the movie browser, the Client can pop back to the top navigator by resuming it at the previous state. This is very common in applications and natural for human behavior.

Two models are available for Service Transfer:

1. Basic application level Service Transfer

   In this method, the parameters required in the next Session attach() (i.e., for the destinationServer) are sent to the Client from the Server at the application level. The Client uses this information to do one of the following:

   - Release the Session with the sourceServer and establish a new session with the destinationServer
   - Maintain the Service with the sourceServer and establish a new session with the destinationServer

   In the method when the Session with the sourceServer is released the Client cannot resume the context at a later time.

2. Enhanced application level Service Transfer through Session detach(), State suspend() and State resume().

   In this method, the parameters required in the next DSM Session attach are sent to the Client from the Server at the application level. The Client uses the State suspend() and State resume() operations to perform one of the following:

- Release the Session with the sourceServer (using **detach()** with **aSuspend** true)
- Suspend a service with forced release of its connection resources (using **suspend()** with **aRelease** true)
- Suspend a service without forced release of its connection resources; i.e., can be reassigned by the SessionGateway to another service within a time-out period (using **suspend()** with **aRelease** false)
- Maintain the Service with the sourceServer

In this method for the first three items above, the Client can resume the full context at a later time. Message flows for both the Basic and Enhanced Service Transfer are given in Annex L.

- Release the Session with the sourceServer
- Maintain minimum resources with the sourceServer
- Maintain the Service with the sourceServer

In the method for the first two options above, the Client can resume the full context at a later time.

Message flows for both the Basic and Enhanced Service Transfer are given in Annex L.

5.5.1.6.2 Summary of Session Primitives

Defined in Session:

DSM Session attach	Attach to a ServiceGateway Domain of Services. (READER)
DSM Session detach	Detach from a ServiceGateway Domain of Services. (READER)

5.5.1.6.3 DSM Session attach

DSM Session attach	Attach to a ServiceGateway domain of Services. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Session {
        const AccessRole attach_ACR = READER;
        void  attach (
                in ServiceDomain aServiceDomain,// optional ServerId
                in CosNaming::Name pathName,    // optional path name to resolve
                in UserContext  savedContext,   // previous application user context
                out ObjRefs resolvedRefs)       // objects resolved
                raises (OPEN_LIMIT, NO_AUTH, UNK_USER, SERVICE_XFR,
                        BAD_COMPAT_INFO, NO_RESUME,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
    };
};
```

Semantics
The Client invokes **Session attach()** to establish a Session context with a ServiceGateway. This operation is invoked on the local Session object. The identification of ServiceGateway may be provided by either **aServiceDomain** or **pathName**, or both. **aServiceDomain** is in NSAP address format. For an interactive Session, it is the globally unique Server network address. For a Broadcast Carousel, it is the unique identifier of the Carousel. **pathName** provides the logical path to a ServiceGateway and optionally a first Service.

aServiceDomain may be given with a 0 length **pathName**, and **pathName** may be given with a 0 length **aServiceDomain**. In the former, the **pathName** shall be a non-conflicting path to the Service Domain in the system environment namespace. If both are given, **aServiceDomain** uniquely identifies the server associated with the ServiceGateway, and **pathName** provides the logical name of the ServiceGateway followed by the path to the first Service.

A previous Session can be resumed by providing **savedContext** from that Session. If this is not available, **savedContext** shall be 0 length.

Depending on the **pathName** input parameter, **Session attach()** shall return object references for a ServiceGateway and optionally for a first Service. If the first Service is a Composite object, it shall return object references for the Composite parent and child objects.

Privileges Required:
READER

Parameters

type/variable	direction	description
ServiceDomain aServiceDomain	input	The Serverld entry point to a Service Domain, identifying a ServiceGateway.
CosNaming::Name pathName	input	The logical path to the first Service.
UserContext savedContext	input	Application state from a previously suspended application.
ObjRefs resolvedRefs	output	The object references resolved. ServiceGateway and optionally first Service objects.

5.5.1.6.4 DSM Session detach

DSM Session detach	Detach from a ServiceGateway domain of Services. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Session {
        const AccessRole detach_ACR = READER;
        void  detach (
                in boolean aSuspend,
                out UserContext savedContext)      // suspended user context
                raises (NO_SUSPEND);
    };
};
```

Semantics

A Client may invoke **Session detach()** to disconnect from a ServiceGateway and all objects of a Session. The operation is invoked on a remote ServiceGateway object reference. If **aSuspend** is TRUE, application state shall be returned as **savedContext**. If **UserContext** is not available, or if **aSuspend** is FALSE, **savedContext** shall be 0 length.The Client may later invoke **Session attach()** with this **savedContext** in order to resume the application from the point of suspension.

Privileges Required:
READER

Parameters

type/variable	direction	description
boolean aSuspend	input	If true, suspend application state and return savedContext.
UserContext savedContext	output	The suspended user context.

5.5.1.7 ServiceGateway

ServiceGateway inherits the Directory and Session interfaces. **Directory bind()**, **bind_context()**, **rebind()**, **rebind_context()** and **unbind()** require MANAGER privileges to be invoked on the ServiceGateway

5.5.1.7.1 Summary of ServiceGateway Primitives

Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Inherited from **Directory and NamingContext**:

operations: **open, close, get, put, list, resolve, bind, bind_context, rebind, rebind_context, unbind, new_context, destroy**

Inherited from **Session**:
operations: **attach, detach**

Application Portability Syntax

```
module DSM {
    interface ServiceGateway : Directory, Session {
        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;
    };
};
```

5.5.1.8 First

The First interface enables an application Client to obtain its first objects.

5.5.1.8.1 Summary of First Primitives

Defined in **First**:

DSM First root	Obtain the ServiceGateway object.
DSM First service	Obtain the Primary Service object.

5.5.1.8.2 DSM First root

DSM First root	Obtain the ServiceGateway object.

Application Portability (Local Library) Syntax

```
module DSM {
    interface First {
        Object  root ();
    };
};
```

Semantics

A Client may invoke **First root()** to obtain the object reference of the ServiceGateway for the current Session.

Privileges Required:
NONE

Parameters		
Object	output	Current ServiceGateway object.

5.5.1.8.3 DSM First service

DSM First service	Obtain the Primary Service object.

Application Portability (Local Library) Syntax

```
module DSM {
    interface First {
        Object  service ();
    };
};
```

Semantics

A Client may invoke **First service()** to obtain the object reference of the current Session's Primary Service. Following **Session attach()**, this shall be the object reference of the Service specified in the **pathName** of the **Session attach()**.

Privileges Required:
NONE

Parameters		
Object	output	Current Primary Service object.

5.5.2 Extended Interfaces

The DSM-CC Extended interfaces are optional. Each of these interfaces may be implemented at the discretion of the Service Provider.

The **Event** interface can be used to subscribe to receiving Stream Event Descriptors that are synchronized with audio and video in the MPEG stream.

Download provides a function call interface for the Download Protocol of clause 7.

The **Composite** interface is useful in high latency networks or for resolving multiple objects that have version interdependencies.

The **View** interface can be used for sorting/filtering Directory information or accessing a database.

With the **State** interface, an object declares that it supports functionality to suspend and resume application state.

In a CORBA Server environment, there is an Interface Repository where new interfaces, operations, and associated parameters may be discovered. In a DSM-CC system, the **Interfaces** operations are used to verify system integrity of all shared interfaces and types.

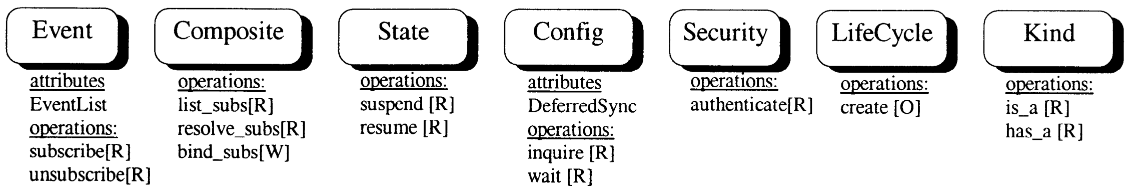
The **Security** interface may be used in secure system environments, where passwords or encrypt keys need to be exchanged.

The **Config** interface is useful if some applications require synchronous operations, while others can make asynchronous requests.

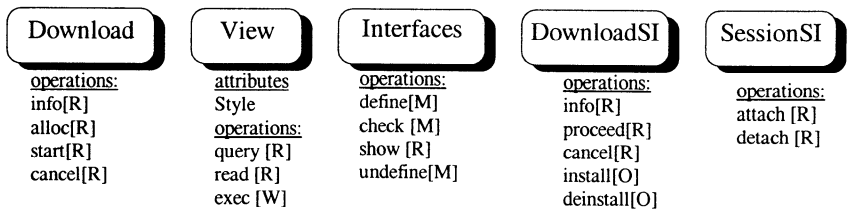
LifeCycle create() is a convenience function which assures uniqueness of Interoperable Object References.

The **Kind** interface enables Clients to dynamically determine which interfaces an object supports.

Abstract interfaces



Instantiable interfaces



R ::= Reader
W ::= Writer | R
B ::= Broker | W | R
O ::= Owner | B | W | R
M ::= Manager | O | B | W | R

5.5.2.1 Download

5.5.2.1.1 Download Definitions, Exceptions

```
module DSM {
    struct ModuleInfo {
        u_short moduleId;
        octet moduleVersion;
        u_long moduleSize;
        sequence<octet, 255>  moduleInfoBytes;
    };
    typedef sequence<ModuleInfo, 65535> ModuleInfoList;
    typedef sequence<u_short> ModuleList;
};
```

5.5.2.1.2 Summary of Download Primitives

Download info	Obtain information about prerequisite download modules. (READER)
Download alloc	Allocate memory buffers for a download. (READER).
Download start	Start and transfer the modules to the Client. (READER).
Download cancel	Cancel a download in progress. (READER).

5.5.2.1.3 DSM Download info

DSM Download info	Obtain information about prerequisite download modules. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Download {
        const AccessRole info_ACR = READER;
        void info (out  ModuleInfoList  rModulesInfo);
    };
};
```

Semantics

Download info() enables a Client to obtain information about modules that must be downloaded in order for an application to proceed. The fields of ModuleInfo are defined in clause 7.

Privileges Required
READER

Parameters

ModuleInfoList rModulesInfo	output	ModuleId, version, size and other information for each module of the Download image.

5.5.2.1.4 DSM Download alloc

DSM Download alloc	Allocate memory buffers for a download. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Download {
        const AccessRole alloc_ACR = READER;
        void  alloc (in u_short aModuleId,
                    in  Object rWriteBuffer,
                    out Object rReadBuffer)
                    raises (BAD_MODULE_ID);
    };
};
```

Semantics

With **Download alloc()**, the application negotiates buffer management for a module to be downloaded. It can choose to allocate memory or otherwise assign the buffer location for each module, or it can defer to the lower transport layer to choose and manage the buffer location. If the convention is for the application to assign the buffer space for the module, it will place a non-null pointer in **rWriteBuffer**. In this case the **rReadBuffer** reply will be null. The data will be placed at **rWriteBuffer**. If the application defers to the lower layer transport to assign the buffer location, it places null in **rWriteBuffer** and the lower layer transport will return a non-null reference to the **rReadBuffer** buffer location. In the second case, the data will be placed in **rReadBuffer**. The configuration step is performed individually on each module, following a **Download info()** invocation, and prior to **Download start().**

Privileges Required

READER

Parameters

u_short aModuleId	input	Identifier of a module.
Object writeBuffer	input	Whether to write data (TRUE) or address (FALSE) to rReadBuffer.
Object readBuffer	output	Buffer or location of buffer.

5.5.2.1.5 DSM Download start

DSM Download start	Start and transfer the modules to the Client. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Download {
        const AccessRole start_ACR = READER;
        void  start (in ModuleList aModuleList)
                    raises(BAD_MODULE_ID, MPEG_DELIVERY, TIMEOUT);
    };
};
```

Semantics

With **Download start()**, the Client requests that the transfer of modules begin. When it completes, all modules will be downloaded. A module sequence can be selected. From the application interface, aModuleList indicates only those modules that are to be placed in the Client's allocated memory.

Privileges Required
READER

Parameters

ModuleList aModuleList	input	A sequence of u_short listing modules by their id.

5.5.2.1.6 DSM Download cancel

DSM Download cancel	Cancel a download in progress. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Download {
        const AccessRole cancel_ACR = READER;
        void  cancel (in ModuleList aModuleList )
                raises (BAD_MODULE_ID, TIMEOUT) ;
    };
};
```

Semantics

The Client calls **Download cancel()** in order to cancel a download in progress. aModuleList identifies the ids of the modules to cancel. After this operation is invoked, the transport layer will not write to Client's allocated memory any modules that are on aModuleList. The transport layer will issue cancel to the Server when the download has progressed to the point where all remaining module ids to be downloaded are on aModuleList.

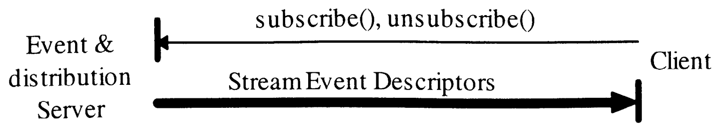
Privileges Required
READER

Parameters

ModuleList aModuleList	input	A sequence of u_short listing modules by their id.

5.5.2.2 Event

The event interface in the Common Object Services specification of the Object Management Group is the foundation on which the DSM-CC 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 set-top device, which receives the media stream. The stream object distributes the event data through a transport mechanism that differs from that of the object invocation interface. For example, the Object Implementation of the Event interface could deliver events through the MPEG-2 stream.



The Client discovers the existence of an event by accessing the **EventList** attribute. The Client invokes **Event subscribe()** to request that the event be sent. The Client provides an **eventName** from the EventList, which is a simple string. The stream object returns a **eventId** which uniquely identifies the event. There is only one eventId associated with a stream event name.

Note that the event includes both the **eventId** and the **rAppTime** to which the event relates. The inclusion of the time value enables the Client to schedule the reaction to the event to correlate with the presentation of a media stream.

Where the transport for event distribution is MPEG-2, the Stream Event structure shall be sent in the MPEG-2 stream encapsulated in the Stream Event descriptor, and placed in a DSM-CC_section as specified in clause 9, Transport.. This descriptor is defined in clause 6, Stream Descriptors. The stream object places the event data into the media stream near the companion media data. Unsolicited Stream Event Descriptors are allowed in the MPEG-2 stream, but shall be discarded by the DSM-CC Library.

5.5.2.2.1 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_ID = 0;
        typedef sequence<char, 255> eventName;
        typedef  sequence<eventName, 65535> EventList_T;
        const AccessRole EventList_get_ACR = READER;
        const AccessRole EventList_put_ACR = OWNER;
        attribute EventList_T EventList;
        // the following struct is sent in the MPEG stream
        //
        struct StreamEvent {
                u_short eventId;
                AppNPT rAppTime;
                sequence<octet> rPrivateData;};
    };
};
```

The constant declaration captures the convention that if the eventId field of the descriptor data is the value zero, it is understood that the event data which follows is invalid. If the event trigger time is maximum negative value, the semantics are to immediately respond to the event.

5.5.2.2.2 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)
DSM Event notify	Obtain Event data from a Stream Event descriptor. (READER)

5.5.2.2.3 DSM Event subscribe

DSM Event subscribe	Subscribe to receive an event over an MPEG stream. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Event {
        const AccessRole subscribe_ACR = READER;
        void subscribe(
                    in string aEventName,
                    out u_short eventId)
                    raises(INV_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 Service returns an eventId to associate with the event name. The scope of the eventId is at least the media stream. The Client, in other words, should not find multiple eventIds related to the same event in the 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 eventId	output	The eventId which the Service assigns, and which the Client should associate, with the event.

5.5.2.2.4 DSM Event unsubscribe

DSM Event unsubscribe	Indicate desire to no longer receive an event. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Event {
        const AccessRole unsubscribe_ACR = READER;
        void unsubscribe(
                    in u_short eventId)
                    raises(INV_EVENT_ID);
    };
};
```

Semantics

The Client invokes **Event unsubscribe()** to instruct the Service to not generate the event. The Client provides the eventId to describe the subscription to which the operation refers. The eventId with respect to the Client becomes stale. The Service can assign the eventId to other subscription requests. The exception relates to the situation where the Client provides a bogus eventId, for example a eventId which was valid but is now stale.

Privileges Required
READER

Parameters

type/variable	direction	description
u_short eventId	input	The eventId which identifies the previous subscription.

5.5.2.2.5 DSM Event notify

DSM Event notify	Obtain Event data from a Stream Event descriptor. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
  interface Event {
#ifdef DSM_PSEUDO
        void notify (
                out StreamEvent rStreamEvent);
#endif
    };
};
```

Semantics

Event notify() enables the Application Portability Interface applications to receive the data of the transmitted Stream Events Descriptors. This functionality is necessary for portable applications that rely on the information carried in the Stream Event Descriptors, such as the event NPT and event private data.

Note that the **Event notify()** operation is not related to a function in the Client-Server interface because it handles the reception of the Stream Event Descriptors at the Client to the Client's application.

When the Client is using the synchronous interface, it would typically set up a thread that calls **Event_notify()**. When the Event is received, **Event_notify()** would return with the Descriptor data. When the Client is using the synchronous deferred interface, it would call **Event_notify()** to obtain a DSM_RequestHandle. This RequestHandle is then used to inquire or wait for the arrival of the Stream Event Descriptor.

Privileges Required:
READER

Parameters

type/variable	direction	description
StreamEvent rStreamEvent	output	StreamEvent Structure constructed from Stream Event descriptor that was sent in an MPEG-2 stream.

5.5.2.3 Composite

The Composite interface provides the ability to associate a set of child objects related by a common parent. The child objects are marked as either required or optional. A required child object must be resolved in order for the application to start. An optional child may be resolved at any time during the running of the application. The child objects are associated as a set of compatible versions. A **Composite list_subs()** shall list all the child objects by name, version and whether they are required or optional.

Directory open() with **aPathType** = DEPTH must be used to resolve a Composite object. The Composite object shall be recognized by the BindingType **DSM::cobject** in the **Directory list()** reply. At the Application Portability Interface, **Directory open()** returns the parent Composite. A child object may be resolved with a subsequent **Directory open()** or **Directory resolve()**. If a parent Composite object is closed, its Child objects are closed as a result of the operation.

Composite bind_subs() and **Composite unbind_subs()** operations are provided for creating and destroying the Composite association.

For Composite bindings at a Directory, the BindingType shall be **DSM::cobject**.

5.5.2.3.1 Summary of Composite Primitives

Composite list_subs	For each ChildRef, list NameComponent, Version and whether required to be resolved upon opening. (READER)
Composite bind_subs	Bind sub-objects to a Composite object. (WRITER)
Composite unbind_subs	Unbind all sub-objects of a Composite object. (WRITER)

5.5.2.3.2 DSM Composite list_subs

DSM Composite list_subs	For each ChildRef, list NameComponent, Version and whether required to be resolved by resolveSubs. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    typedef u_long BindingType; // extends CosNaming BindingType
    const BindingType cobject = 2;
    interface Composite {
        struct ChildInfo {
            CosNaming::NameComponent n;
            Version rVersion;
            boolean required;
        };
        typedef sequence<ChildInfo> ChildInfos;
        //
        const AccessRole list_subs_ACR = READER;
        void list_subs (
             in CosNaming::Name name,
             out ChildInfos infos)
            raises (NOT_FOUND, INV_NAME);
    };
};
```

Semantics

A consumer Client invokes **Composite list_subs()** to list information concerning the sub-objects of a composite object. Information returned includes NameComponent, Version, and whether the sub-object is required or optional. An application developer can use Composite to combine several well-known Service types, thus making maximum use of existing U-U Library facilities.

Privileges Required
READER

Parameters

type/field	direction	description
CosNaming::Name name	input	The path name to the composite parent phantom.
ChildInfos rChildInfos	output	A sequence of sub-object info structures, each containing NameComponent, Version, and whether the sub-object is required or optional.

5.5.2.3.3 DSM Composite bind_subs

DSM Composite bind_subs	Bind sub-objects to a Composite object. (WRITER)

Application Portability / Service Inter-operability Syntax

```
module DSM
    interface Composite {
#ifdef DSM_GENERAL
        struct ChildBinding{
            CosNaming::NameComponent n;
            Version rVersion;
            boolean required;
            ObjRef obj;
        };
        typedef sequence<ChildBinding>  ChildBindings;
        const AccessRole bind_subs_ACR = WRITER;
        void bind_subs (
             in CosNaming::Name name,
             in ChildBindings rChildBindings)
            raises (NotFound, INV_NAME, AlreadyBound);
#endif
    };
};
```

Semantics

A producer Client uses **Composite bind_subs()** to bind a set of required and optional objects and their names to a composite binder. The composite set consists of compatible versions of objects.

Privileges Required
WRITER

Parameters

type/field	direction	description
CosNaming::Name name	input	The path name to the composite parent phantom.
ChildBindings rChildBindings	input	A sequence of sub-object references, each containing NameComponent, object reference, and whether the sub-object is required or optional.

5.5.2.3.4 DSM Composite unbind_subs

DSM Composite unbind_subs	Unbind all sub-objects of a Composite object. (WRITER)

Application Portability / Service Inter-operability Syntax

```
module DSM
    interface Composite {
#ifdef DSM_GENERAL
        const AccessRole unbind_subs_ACR = WRITER;
        void unbind_subs (in CosNaming::Name name)
            raises (NotFound, INV_NAME);
#endif
    };
};
```

Semantics

A producer Client uses **Composite unbind_subs()** to unbind all sub-objects from a parent composite object.

Privileges Required
WRITER

Parameters

type/field	direction	description
CosNaming::Name name	input	The path name to the composite parent phantom.

5.5.2.4 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 Style 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 query()** 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 Style for this case is either **SQL89, SQL92, or SQL3. View query()** 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 Styles, the SQL language syntax is used as the basic query form.

View read() is provided which will return a number of rows. A **View read()** from the calling application can result in a **View read()** RPC call by the DSM-CC Library which will pre-fetch results in anticipation of the cursor location in the next **View read()**.

Using the application portability interface, **Directory open()** can be pipelined with **View query()** and **View read()** 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.

5.5.2.4.1 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 Style 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 browse-able 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 View Style 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 View Style:

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

5.5.2.4.2 Database View

A View object may represent an actual database at the server. The View Styles 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.

5.5.2.4.3 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 query()** with a SQL statement.
2. The DSM-CC Library issues **Directory open()**, **View query()** and **View read()** 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 read()** 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 read()** in its local buffer, as described above.
2. The client can obtain a row by invoking **View read()** with cursor value that points into the matching result set at the Server. The DSM-CC Library will invoke the remote interface **View read()** as needed or to pre-fetch a window of rows in anticipation of further reads.
3. Finally the client issues **Base close()** is used to close the View and the query.

5.5.2.4.4 Definition: View Style Attribute

```
module DSM
    interface View{
        // View Style 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 this part of ISO/IEC 13818
        // 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';
        const AccessRole Style_get_ACR = READER;
        readonly attribute char Style;
    };
};
```

5.5.2.4.5 View Definitions: Statement, Result

```
module DSM {
    interface View {
        typedef string SQLStatement;
        //
        typedef u_short FieldCode;
        struct FieldDescribe {
                string fieldName;              // name of the field
                FieldCode aType;               // type of the field
                opaque typeParameters;         // parameters related to the given type
        };
        typedef sequence<FieldDescribe> ResultDescribe;
        //
        // FieldCodes for standard SQL types
        const FieldCode VTC_CHAR =                   1;
        const FieldCode VTC_SMALLINT =               2;
        const FieldCode VTC_INTEGER =                3;
        const FieldCode VTC_FLOAT =                  4;
        const FieldCode VTC_SMALLFLOAT =             5;
        const FieldCode VTC_DECIMAL =                6;
        const FieldCode VTC_REAL =                   7;
        const FieldCode VTC_DOUBLEPRECISION =        8;
        const FieldCode VTC_CORBA_TYPECODE =         9;
        //
        // these structs are placed into the opaque typeParameters field of
        // the FieldDescribe struct depending on the value of the FieldCode
        struct InfoChar {
                u_short length;
                boolean nullTerminated;
        };
        struct InfoFloat {
                u_short precision;
        };
        struct InfoDecimal {
                u_short precision;
                u_short scale;
        };
        //
        // type definitions for the returned data
        typedef opaque Field;
        typedef sequence<Field> Row;
        typedef sequence<Row> Result;
    };
};
```

5.5.2.4.6 Summary of View Primitives

Defined in View:	
DSM View query	Execute a SQL select statement, fetch an initial set of result objects. (READER)
DSM View read	Read additional result rows in the context of a View query. (READER)

DSM View execute	Execute a SQL write statement. (WRITER)

5.5.2.4.7 DSM View query

DSM View query	Execute a SQL select statement. Fetch an initial set of result objects.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface View {
        const AccessRole query_ACR = READER;
        void query(
                in SQLStatement aSQLStatement,
                in u_short maxRows,
                out ResultDescribe describe,
                out Result aResult,
                out View iterator)
                raises (ILLEGAL_SYNTAX);
    };
};
```

Semantics

View query() sends the SQL statement specified by **aSQLStatement** to the View object for execution. **maxRows** defines the maximum number of rows to return. **describe** contains a description of the fields in the data returned in **aResult**.

The iterator output is a View object reference. A **View query()** can be sent to this iterator to perform a further reduction of the current result set by means of a SQL statement. It also enables views to be created and manipulated without closing or destroying the initial View.

Privileges Required
READER

Parameters

type/variable	direction	description
SQLStatement aSQLStatement	input	Standard SQL statement, subject to restrictions of the View Style.
u_short maxRows	input	Maximum number of rows to return.
ResultDescribe describe	output	Description of return fields.
Result aResult	output	Results buffer of rows and fields.
View iterator	output	New View iterator for reading additional rows and further reducing the result. These are accomplished by sending a **View read** or **query** to the iterator.

5.5.2.4.8 DSM View read

DSM View read	Obtain the attributes of a single row or object.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface View {
        const AccessRole read_ACR = READER;
        void read(
                in u_short aCursor,
                in u_short maxRows,
                out Result aResult)
                raises(NO_QUERY, INV_CURSOR);
    };
};
```

Semantics

View read() is used to read additional rows after making a query. **aCursor** identifies an object in the list of objects, either in the name context, or in the select full result.

read() can be called only for the interator object returned from a query. If **read()** is called for a View object without first making a query, **NO_QUERY** exception is raised.

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.
u_short maxRows	input	Maximum number of rows to return.
Result aResult	output	The Object's attribute values or row field values.

5.5.2.4.9 DSM View execute

DSM View execute	Execute a SQL write statement.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface View {
        const AccessRole execute_ACR = WRITER;
        void execute(
                    in SQLStatement aSQLStatement)
                    raises(ILLEGAL_SYNTAX);
    };
};
```

Semantics

View execute() sends the SQL statement specified by **aSQLStatement** to the View object for execution of SQL inserts, deletes, and updates.

Use of **View execute()** is not permitted for the **NON_DB** View Style. For NON_DB Views, Directory commands such as **Directory bind()** and **unbind()** should be used instead.

Privileges Required:
WRITER

Parameters

type/variable	direction	description
SQLStatement aSQLStatement	input	Standard SQL statement, subject to restrictions of the View Style.

5.5.2.5 State

The State interfaces enables an application to suspend state, and later resume where it left off.

5.5.2.5.1 Summary of State Primitives

DSM State suspend	Suspend application state for a Service. (READER)
DSM State resume	Resume a Service from previous application state. (READER)

5.5.2.5.2 DSM State suspend

DSM State suspend	Suspend application state for resumption at a later time. (READER)

Service Inter-operability IDL Syntax

```
module DSM {
    interface State {
        const AccessRole suspend_ACR = READER;
        void suspend(
                in boolean  aRelease,
                out UserContext  savedContext)
                raises (NO_SUSPEND);
    };
};
```

Semantics

State suspend() is used to request that a Service Object Implementation instance preserve its Client specific state or return it to the Client. If this object is a parent Composite with open child objects, **State suspend()** cascades to the child objects (how the suspend cascades to other open objects is outside the scope of DSM-CC). By setting the **aRelease** flag to 1, the Client indicates it is not planning to resume this application in the immediate future.

Privileges Required:
READER

Parameters

type/variable	direction	description
boolean aRelease	input	A hint to the underlying transport. If FALSE, an indication the Client intends to resume soon. If TRUE, an indication the Client intends to resume much later.
UserContext savedContext	output	Opaque state information.

5.5.2.5.3 DSM State resume

DSM State resume	Resume from a ServiceGateway domain of Services. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface State{
        const AccessRole resume_ACR = READER;
        void  resume (
                    in UserContext  savedContext,
                    out ObjRefs  restoredRefs)
                    raises (NO_RESUME);
    };
};
```

Semantics

State resume() is used to request an object instance to restore a previous state using a **savedContext** sequence of octets. The structure of **restoredRefs** is the same as that for the **resolvedRefs** in the **Session attach()**.

Privileges Required:
READER

Parameters

type/variable	direction	description
UserContext savedContext	input	savedContext from a previous **State suspend()** or **Session detach()**.
ObjRefs restoredRefs	output	object references enabling the application to continue where it left off.

5.5.2.6 Interfaces

In a Service Domain where new interfaces need to be made public, an object can define its interfaces by use of the **Interfaces define()** operation. In addition to new interfaces, it can declare whether it inherits well-known interfaces such as DSM-CC Directory, Stream and File, or whether it inherits other interfaces known to the Interfaces object. More importantly, through **Interfaces define()**, an interface can be verified as being consistent with the complete and coherent system-wide interface set.

The DSM-CC application space is constructed as a name space graph starting at the ServiceGateway. The nodes represent objects of the various types specified by DSM-CC 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 browse-able 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()**.

An Interface Repository is recommended to insure that the object interfaces that exist in the Server are coherent and non-conflicting. DSM-CC interfaces provide a well-known base set of interfaces. However, as applications are developed, there will be new interfaces that build on the DSM-CC primitives. As these are installed in the system, it is desirable that they be verified for correctness and that they do not conflict with the existing interfaces in the system. The repository must assure all interfaces defined with it represent a valid collection of IDL definitions. For example, all inherited interfaces exist, there are no duplicate operation names or other name collisions, all parameters have known types, etc. This validation is performed by **Interfaces check()**.

5.5.2.6.1 TCKind Constants

```
module DSM {
    typedef u_long TCKind;
    //
    // TCKinds from CORBA 2.0
    const TCKind tk_null = 0;
    const TCKind tk_void = 1;
    const TCKind tk_short = 2;   // = DSM_s_short
    const TCKind tk_long = 3;    // = DSM_s_long
    const TCKind tk_ushort = 4;  //  = DSM_u_short
    const TCKind tk_ulong = 5;   //  = DSM_u_long
    const TCKind tk_float = 6;
    const TCKind tk_double = 7;
    const TCKind tk_boolean = 8;
    const TCKind tk_char = 9;
    const TCKind tk_octet = 10;
    const TCKind tk_any = 11;
    const TCKind tk_TypeCode = 12;
    const TCKind tk_Principal = 13;
    const TCKind tk_objref = 14;
    const TCKind tk_struct = 15;
    const TCKind tk_union = 16;
    const TCKind tk_enum = 17;
    const TCKind tk_string = 18;
    const TCKind tk_sequence = 19;
    const TCKind tk_array = 20;
    const TCKind tk_alias = 21;
    const TCKind tk_except = 22;
    const TCKind tk_longlong = 23;
    const TCKind tk_ulonglong = 24;
    // 25 - 0x00FFFFFF  and 0x80000000 - 0xFFFFFFFF OMG reserved
    //
    // Component Tags registered with OMG
    // The reserved range for ISO/IEC WG11 in OMG is 0x49534F00 - 0x495351FF
    //
    // TCKinds specific to DSM-CC.
    // 0x495341F00 - 0x495341F7F ISO/IEC and OMG reserved
    // DSM-CC constructed TCKinds
    const TCKind tk_IntfCode = 1230196608; // 0x49534F80 identifies supported interfaces
    const TCKind tk_Access_Hist = 1230196609; //  0x49534F81 has Version and DateTime
    const TCKind tk_Version = 1230196610; //  0x49534F82 has major and minor
    const TCKind tk_DateTime = 1230196611; //  0x49534F83 has fields
    const TCKind tk_Access_Lock = 1230196612; //  0x49534F84 has readLock and writeLock
    const TCKind tk_Access_Perms = 1230196613; //  0x49534F85 has fields
    const TCKind tk_Stream_Info = 1230196614; //  0x49534F86 has description, duration, flags
    const TCKind tk_Event_EventList = 1230196615; //  0x49534F87 is a sequence of string
    const TCKind tk_Config_ActiveRequests = 1230196616; //  0x49534F88 is a seq of RequestHandle
    // 0x49534F89 - 0x49534FFF ISO/IEC and OMG reserved
};
```

5.5.2.6.2 Exception TCKind Constants

```
module DSM {
    // CosNaming exceptions
    const TCKind tk_NotFound = 1230196736; // 0x49535000
    const TCKind tk_CannotProceed = 1230196737; // 0x49535001
    const TCKind tk_InvalidName = 1230196738; // 0x49535002
    const TCKind tk_AlreadyBound = 1230196739; // 0x49535003
    const TCKind tk_NotEmpty = 1230196740; // 0x49535004
    //0x49535005 - 0x49535030 ISO/IEC Reserved
    //
    // DSM-CC common exceptions
    const TCKind tk_ALREADY_BOUND = 1230196784; // 0x49535030
    const TCKind tk_BAD_COMPAT_INFO = 1230196785; // 0x49535031
    const TCKind tk_BAD_MODULE_ID = 1230196786; // 0x49535032
    const TCKind tk_BAD_MODULE_INFO = 1230196787; // 0x49535033
    const TCKind tk_BAD_SCALE = 1230196788; // 0x49535034
    const TCKind tk_BAD_START = 1230196789; // 0x49535035
    const TCKind tk_BAD_STOP = 1230196790; // 0x49535036
    const TCKind tk_BLOCK_SIZE = 1230196791; // 0x49535037
    const TCKind tk_CANNOT_PROCEED = 1230196792; // 0x49535038
    const TCKind tk_ILLEGAL_SYNTAX = 1230196793; // 0x49535039
    const TCKind tk_INV_CURSOR = 1230196794; // 0x4953503A
    const TCKind tk_INV_EVENT_ID = 1230196795; // 0x4953503B
    const TCKind tk_INV_EVENT_NAME = 1230196796; // 0x4953503C
    const TCKind tk_INV_KIND = 1230196797; // 0x4953503D
    const TCKind tk_INV_NAME = 1230196798; // 0x4953503E
    const TCKind tk_INV_OFFSET = 1230196799; // 0x4953503F
    const TCKind tk_INV_SIZE = 1230196800; // 0x49535040
    const TCKind tk_MPEG_DELIVERY = 1230196801; // 0x49535041
    const TCKind tk_NO_AUTH = 1230196802; // 0x49535042
    const TCKind tk_NO_QUERY = 1230196803; // 0x49535043
    const TCKind tk_NO_REF_TYPE = 1230196804; // 0x49535044
    const TCKind tk_NO_SUSPEND = 1230196805; // 0x49535045
    const TCKind tk_NO_RESUME = 1230196806; // 0x49535046
    const TCKind tk_NOT_DEFINED = 1230196807; // 0x49535047
    const TCKind tk_NOT_FOUND = 1230196808; // 0x49535048
    const TCKind tk_OPEN_LIMIT = 1230196809; // 0x49535049
    const TCKind tk_PREV_DEFINED = 1230196810; // 0x4953504A
    const TCKind tk_READ_LOCKED = 1230196811; // 0x4953504B
    const TCKind tk_TIMEOUT = 1230196812; // 0x4953504C
    const TCKind tk_UNK_USER = 1230196813; // 0x4953504D
    const TCKind tk_WRITE_LOCKED = 1230196814; // 0x4953504E
    const TCKind tk_SERVICE_XFR = 1230196815; // 0x4953504F
    // 0x49535050 - 0x495350FF ISO/IEC reserved
};
```

5.5.2.6.3 Interfaces Definitions

```
module DSM {
    // A TypeCodeBuf holds a Corba 2.0 TypeCode
    typedef opaque TypeCodeBuf;
    typedef sequence<TypeCodeBuf>  TypeCodeList;
    //
    interface Interfaces {
        typedef opaque ReferenceData;
        typedef string InterfaceDef;
    };
};
```

5.5.2.6.4 Summary of Interfaces Primitives

DSM Interfaces show	Show an interface definition, IntfCode and TypeCodes. (READER)
DSM Interfaces define	Define an object interface to the System (MANAGER)
DSM Interfaces check	Verify the coherence of an interface with the repository. (MANAGER)
DSM Interfaces undefine	Remove an object interface definition from the System. (MANAGER)

5.5.2.6.5 DSM Interfaces show

DSM Interfaces show	Show an interface definition, IntfCode and TypeCodes. (READER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Interfaces {
#ifdef DSM_GENERAL
        const AccessRole show_ACR = READER;
        void show (
            in string aStrKind,       // DSM-CC format, "Module::Interface Type"
            in IFKind anIFKind,   // IFKind
            out string rIDL,                    // IDL used in previous define
            out IntfCode rIntf,       // DSM-CC interface code with included IFKinds
            out TypeCodeList  rTypes)       // TypeCodes required at this level of interface
        raises (NotFound, INV_NAME);
#endif
    };
};
```

Semantics

A READER Client can call **Interfaces show()** to retrieve the IDL, IntfCode, TypeCodes of an interface or type. The semantics of the output parameters **IFKind** and **TypeCodeList** are the same as those of the **Interfaces define()** operation.

If **anIFKind** is non-zero, then it will be used to lookup the interface parameters. Otherwise, **aStrKind** will be used to lookup the interface information. The Interfaces show() function enables the following sequence: A Client, after invoking **Directory list()**, discovers a new interface kind from the **NameComponent kind** of one of the Bindings in the **Directory list()** reply. The Client then resolves an object that supports DSM::Interfaces for this system, and invokes **Interfaces show()** with **aStrKind** from the **NameComponent kind**. The Client then iterates the **Interfaces show()** with **anIFKind** from each of the IntfCode included **IFKinds**, to obtain all type information for the interface inheritance hierarchy. If the Client supports these types, or if it can dynamically interpret them, it can then communicate with the new interface.

IDL, IntfCodes and TypeCodes must be available for inter-operability reasons. The IDL is used to generate language mappings. IntfCodes are used by **Kind_is_a()** and other operations that desire the efficiency of u_long vs. string format. TypeCodes are used by exception handlers, **Directory_get()**, **Directory_put()**, **View_Read()**, etc., to identify the values in the **any** structure. These IntfCodes and TypeCodes are necessary at application compile time, if structures are to be pre-compiled as opposed to dynamically interpreted. A compiler may invoke **Interfaces define()** or **Interfaces show()** in the process of generating the language mapping.

aStrKind can be formatted as follows:

- It can refer to an interface, in which case it will be of the form "<Module>::<Interface>".
- It can refer to an interface by it's DSM-CC reserved 3-character string alias.
- It can also refer to common types at the Repository level, in which case it will be of the form "<Type>".
- It can also refer to common types at the Module level, in which case it will be of the form "<Module::Type>".
- It can furthermore refer to types within an interface, in which case it will be in the form "<Module::Interface Type>".

In the above, "<Module>" refers to the module symbolic name in the IDL. "<Interface>" refers to the interface symbolic name in the IDL. "<Type>" refers to the Type definition symbolic name in the IDL.

Privileges Required:
READER

Parameters

type/variable	direction	description
string aKind	input	The DSM-CC NameComponent kind in the form "Module::Interface", or "Module::Type", or "Module::Interface Type".
string rIDL	output	The IDL defined for the interface or type.
IntfCode rIntf	output	The defined IntfCode for this interface.
TypeCodeList rTypes	output	DSM-CC TypeCodes used by this interface, or TypeCodes representing common type definitions.

5.5.2.6.6 DSM Interfaces define

DSM Interfaces define	Define an object interface to the System (MANAGER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
   interface Interfaces {
#ifdef DSM_GENERAL
         const AccessRole define_ACR = MANAGER;
         void define (
             in ReferenceData id,   // unique Identifier
             in InterfaceDef rIDL,  // IDL definition
             out IntfCode rIntf,       // DSM-CC interface code with included IFKinds
             out TypeCodeList rTypes)     // TypeCodes required at this level of interface
             raises (PREV_DEFINED, ILLEGAL_SYNTAX, NO_REF_TYPE);
#endif
     };
};
```

Semantics

Interfaces define() is used by a MANAGER to declare object interfaces and type definitions to the system environment. The interface definition specifies the exported interface of the object, i.e., exported methods and attributes, plus data type definitions. The interface definition may 'include' other interfaces to order to enable new interfaces to extend the functionality of existing interfaces. The object type is specified in OMG IDL. This interface definition will replace any previous definition.

The interface is verified for completeness and coherence with the other interfaces that have previously been defined, i.e., success from this operation means that the set of system interfaces including this one will have a definitions for all referenced types and will be without name conflicts.

A maximum of one interface can be defined per **Interfaces_define()** operation. **IntfCode** is returned which provides an **IFKind** enumeration unique for this system environment. The **IFKind** is the input to the operation **Kind has_a()**.The **IntfCode** is the output of the operation **Kind is_a()**. **IntfCode** also contains the kind string to be used in **NameComponent kind**, the id of the Interface Repository and the **IFKind** of each of the included interfaces. A sequence of **TypeCode** is returned which identifies the component types used by this interface. If no interface is defined, i.e., component types are defined outside of interfaces, an **IFKind** of ik_null and the defined **TypeCode**s 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 READER.

Following the **Interfaces define()**, the object type may be used in **Lifecycle 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 manager of the interface.
InterfaceDef rIDL	input	The IDL for an interface or common types definition.
IntfCode rIntf	output	The defined IntfCode for this interface.
TypeCodeList rTypes	output	DSM-CC TypeCodes used by this interface, or TypeCodes representing common type definitions.

5.5.2.6.7 DSM Interfaces check

DSM Interfaces check	Verify the coherence of an interface with the repository. (MANAGER)

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Interfaces {
#ifdef DSM_GENERAL
        const AccessRole check_ACR = MANAGER;
        void check (in IFKind  anIFKind)
            raises (NO_REF_TYPE);
#endif
    };
};
```

Semantics

An MANAGER may use **Interfaces check()** to verify the repository coherency. **anIFKind** represents the interface to be checked. NO_REF_TYPE indicates a missing type definition.

Privileges Required:
MANAGER

Parameters

type/variable	direction	description
IFKind anIFKind	output	Identification of the interface to be checked in the context of the system interfaces repository.

5.5.2.6.8 DSM Interfaces undefine

DSM Interfaces undefine	Remove an object interface definition from the System. (MANAGER)

Application Portability / Service Inter-operability Syntax

```
  module DSM {
     interface Interfaces {
  #ifdef DSM_GENERAL
           const AccessRole undefine_ACR = MANAGER;
           void undefine (in ReferenceData id, out IFKindList usedBy);
  #endif
     };
  };
```

Semantics

An MANAGER may use **Interfaces undefine()** to remove the definition of an interface and its associated type definitions from the interfaces repository. If another interface includes this interface, a system incoherence will result, indicated by non-zero **includedIn** output. When this has been fixed, an **interfaces check()** should be performed on the **includedIn** interfaces to again verify system coherency.

Privileges Required:
MANAGER

Parameters

type/variable	direction	description
ReferenceData id	input	Immutable identification information, chosen by the manager of the interface.
IFKindList includedIn	output	Identification of the interfaces that have included this interface.

5.5.2.7 Security

5.5.2.7.1 DSM Security authenticate

DSM Security authenticate	Request authentication with password or decryption key. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    struct AuthRequest {
        string aPassword;
        opaque authData;
    };
    typedef AuthRequest AuthRequest_T;
    interface Security {
        const AccessRole authenticate_ACR = READER;
        void authenticate (in AuthRequest_T authInfo);
    };
};
```

Semantics

The purpose of **Security authenticate()** is to enable the Client to provide authentication information for the purpose of obtaining access to (i.e., resolving) an object. The authenticate must be given with **a** resolve operation if the object has either a non-Null **aPassword** or **authData** with length greater than 0 in its **Perms** attribute. The input argument **authInfo** is included in the **ServiceContextList** of the immediately succeeding resolve operation. If a **Directory open()**, **Directory get()**, or **Session attach()** operation is received without a corresponding authenticate, and authentication is required as described above, a **NO_AUTH** exception will be given. **NO_AUTH** can carry an encrypt key challenge. 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 carry an **authData** challenge which must be successfully processed by the Client. The Client must then send a **Security authenticate()** with proper password or processed encrypt data followed by the repeated resolve operation (a new RPC transaction).

This standard does not specify an encryption algorithm. It does enable the following sequence: a) the Service passes **authData** encrypt challenge to the Client, b) the Client processes the received **authData** via the encryption algorithm, c) the Client returns transformed **authData** response to the Service, and d) the Service verifies the challenge/response via the encryption algorithm.

The atomic operation of the authenticate with the following operation from the Client is implicit by the fact that the input structure **AuthRequest_T** is placed in the **ServiceContextList** of the associated operation.

AuthRequest_T can be sent in the **ServiceContextList** of the **ClientSessionSetupRequest**. Also, in any UNO or DSM-CC ONC RPC request, it can be in the **ServiceContextList** of the RPC message request header, as described in Annex C. Thus, authentication can occur for access to any object and with any RPC request, in addition to **Session attach()**. In DSM-CC, an owner of an application object can require authentication by setting the **Perms** attribute of the object.

Privileges Required:
READER

Parameters

type/variable	direction	description
Password aPassword	input	A character string password for authentication to open an object.
AuthRequest_T authInfo	input	A sequence of bytes used by an encryption or other authentication mechanism.

5.5.2.8 Config

The Config interface enables an application to choose synchronous or synchronous deferred invocation behavior. In synchronous mode, the request blocks until the output parameters are valid. In synchronous deferred mode, the operation returns immediately. A RequestHandle can be tested to determine when the ouput parameters are valid.

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-CC 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.

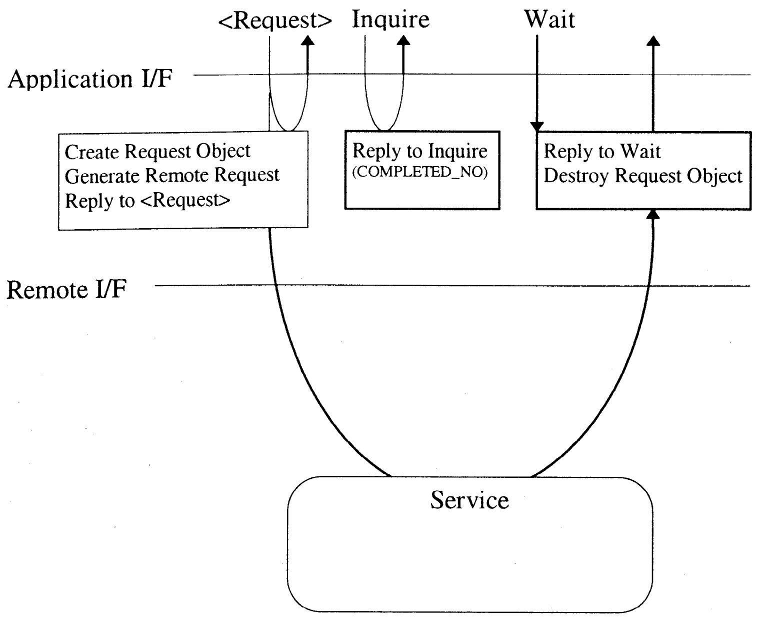


Figure 5-11 Application and Service I/O

DSM-CC 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-CC primitive, at which time the DSM-CC 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.

The Request Object is destroyed in the DSM-CC 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-CC Library determines that the parent Service is closed, the corresponding request object is destroyed.

5.5.2.8.1 Config Definitions

```
module DSM {
    // RequestHandle is 0 if Config:: DeferredSync is FALSE (synchronous RPC)
    typedef u_long RequestHandle;
    interface Config {
        // if TRUE RPC mode is deferred synchronous
        attribute boolean DeferredSync;
        typedef sequence<RequestHandle, 1024> RequestList;
        readonly attribute RequestList ActiveRequests;
    };
};
```

5.5.2.8.2 Summary of Config Primitives

DSM Config inquire	Inquire whether an operation's output parameters are valid.
DSM Config wait	Wait for an operation's output parameters to be valid.

5.5.2.8.3 DSM Config inquire

DSM Config inquire	Inquire whether an operation's output parameters are valid.

Application Portability (Local Library) Syntax

```
module DSM {
    interface Config {
        void inquire (in RequestHandle aRequest);   // inquire as to status
    };
};
```

Semantics

Config inquire() enables the Client to check the status of the transaction. If **Config 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. A successful **Config 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.

Privileges Required:
NONE

Parameters

type/variable	direction	description
RequestHandle aRequest	input	A handle for an outstanding request.

5.5.2.8.4 DSM Config wait

DSM Config wait	Wait for an operation's output parameters to be valid.

Application Portability (Local Library) Syntax

```
module DSM {
    interface Config {
        void wait (in RequestHandle aRequest);        // like CORBA get_response
    };
};
```

Semantics
Config wait() enables a Client to block until an operation's output parameters are valid. This is similar to the CORBA get_response operation.

Privileges Required:
NONE

Parameters

type/variable	direction	description
RequestHandle aRequest	input	A handle for an outstanding request.

5.5.2.9 LifeCycle

The **LifeCycle create()** operation is used to create a unique Interoperable Object Reference that is usable for Client-Server communications.

LifeCycle create() does not create the object itself. There are too many initialization variables for different kinds of objects to define a universal operator that actually creates the entire object for all language variations. There are many language dependencies (consider C++ constructors using inheritance vs. C). **LifeCycle Create()** does, however, generate an IOR that is unique and usable for later operations such as **Directory bind()**. The IOR remains unique and is valid across Server boots, with the exception that, if the Server address changes, the address in the IOR must be updated. It is up to the Server implementation to maintain the persistency of the associated object.

5.5.2.9.1 DSM LifeCycle create

DSM LifeCycle create	Create an object reference of a specified kind. (OWNER)

Application Portability (Local Library) Syntax

```
module DSM {
   interface LifeCycle {
#ifdef DSM_GENERAL
        const AccessRole create_ACR = OWNER;
        void create (
                in string aKind,
                in Version rVersion,
                out IOP::IOR rObjRef);
#endif
   };
};
```

Semantics

LifeCycle create() is used to create an IOR which can be associated with an instance of an object type previously defined by **Interfaces define()**. An IOR with unique object_key and valid type_id is generated. The host/port address of the caller is set in the host and port fields. The address can be modified if the object resides at a different address then the caller. The type_id is given the string format "Module::Interface" in the same format as the DSM-CC

NameComponent.kind. Once created, additional tagged protocol profiles can be added at any time. The application must not alter the initial opaque data placed in the object_key, but can append to the object_key in order to extend the object identification.

LifeCycle create() is typically called as an Application Portability Interface, but is not precluded from being called as a Service Inter-operability Interface. This object reference can be used to bind the resulting IOR to a Directory, e.g. using **Directory bind()**.

Privileges Required:
OWNER

Parameters

type/variable	direction	description
string aTypeId	input	A DSM-CC kind, as obtained by Interfaces define(), or constructed in the form "<module>::<interface>"
Version rVersion	input	Major and minor version of the new object.
ObjRef rObjRef	output	An Interoperable Object Reference with a valid type_id, version, host/port address of the caller, and unique objectkey

5.5.2.10 Kind

Kind provides local operations to determine the interfaces an object supports. If an object includes other interfaces, the **Kind has_a()** operation can be invoked on it to determine whether a specific interface is included. The **Kind is_a()** operation can be invoked to list the interfaces the object supports. An application would typically include the DSM-CC interfaces in various object types. For example, the application could define an interface type for several different formats of files, including the DSM::File interface in each one, and use **kind** to tell them apart.

5.5.2.10.1 Summary of Kind Primitives

DSM Kind has_a	Determine whether an object supports an interface. (READER)
DSM Kind is_a	Show all interfaces an object supports. (READER)

5.5.2.10.2 DSM_Kind_has_a

DSM Kind has_a	Determine whether an object supports an interface. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Kind {
#ifdef DSM_GENERAL
        const AccessRole has_a_ACR = READER;
        void has_a (
                in IFKind anIFKind,
                out boolean aVerdict);
#endif
    };
};
```

Semantics

Kind has_a() enables a Client to test whether an object includes(inherits) a specified interface. The interface is identified by anIFKind. If aVerdict is TRUE, the object includes that interface. If aVerdict is FALSE, it does not. has_a() is an application interface resulting in a local invocation only.

Privileges Required:
READER

Parameters

type/variable	direction	description
IFKind anIFKind	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 anIFKind.

5.5.2.10.3 DSM_Kind_is_a

DSM Kind is_a	Show all interfaces an object supports. (READER)

Application Portability (Local Library) Syntax

```
module DSM {
    interface Kind {
#ifdef DSM_GENERAL
        const AccessRole is_a_ACR = READER;
        void is_a (
                out IntfCode whatItIs);
#endif
    };
};
```

Semantics

Kind is_a() returns the Interface Code for an object. The Interface Code contains the IFKind and string kind of the most derived interface, the repositoryId of the defining InterfaceRepository, and the sequence of inherited IFKinds. **Kind is_a()** is an application interface resulting in a local invocation only.

Privileges Required:
READER

Parameters

type/variable	direction	description
IntfCode whatItIs	output	Identifies the TCKind of object, and the TCKinds of its inherited interfaces.

5.5.3 C Language Mappings

The application portability interfaces provide a true API with a language mapping. It can be generated using an OMG or DSM-CC IDL compiler from the Client-Service IDL. If ANSI C is the language of choice, the mapping shown in this subclause will be used. Applications can link with this interface as a function call library which in turn will invoke the corresponding remote procedure calls.

The application portability interfaces define a library of functions calls that can be used by Client applications to invoke the DSM-CC Client-Service interface and local DSM-CC Library functionality.

Applications that use these functions will be portable between Clients that contain the corresponding DSM-CC remote access Library.

5.5.3.1 Scoped Identifiers

A global identifier is derived from the IDL global name by converting occurrences of "::" to "_" and eliminating any leading underscores.

In some cases, very long identifiers are generated that exceed the capabilities of C compilers. To eliminate excessively long identifiers, the IDL compiler may generate a #define macro to substitute a unique, short sequence of characters in the place of a long sequence of characters. In particular, the following macro is used to shorten several CosNaming identifiers:

```
#define CosNaming COS
```

5.5.3.2 C Mapping for Operations

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 the operation succeeded the ev structure shall have the major exception type of CORBA_NO_EXCEPTION, otherwise it shall have one of the major exception types CORBA_SYSTEM_EXCEPTION or CORBA_USER_EXCEPTION, and shall contain either a CORBA system exception structure, or one of the possible DSM exception structures, as defined by the **raises** statement.
DSM_RequestHandle	output	Synchronous Deferred completion status.

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-CC:

5.5.3.2.1 C Mapping for Basic Data Types

The basic IDL data types used in DSM map as follows:

IDL	DSM shorthand	C
short	**s_short**	`CORBA_short (16 bit)`
unsigned short	**u_short**	`CORBA_unsigned_short (16 bit)`
long	**s_long**	`CORBA_long (32 bit)`
unsigned long	**u_long**	`CORBA_unsigned_long (32 bit)`
long long	**s_longlong**	`CORBA_longlong (64 bit)`
unsigned long long	**u_longlong**	`CORBA_unsigned_longlong (64 bit)`

The implementation is responsible for providing the typedefs for CORBA_short, CORBA_long, etc., consistent with the IDL requirements for these types.

5.5.3.2.2 Constants

Constant identifiers are #defined in the C mapping:

5.5.3.2.3 Struct Types

A struct in IDL maps directly to the equivalent C struct.

5.5.3.2.4 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 at application level to:

```
typedef struct {
      CORBA_unsigned_long _maximum;
      CORBA_unsigned_long _length;
      CORBA_octet * _buffer;
} rSeq;
```

It is encoded on the wire by CDR as

```
struct {
      CORBA_unsigned_long _length;
      CORBA_octet * _buffer;
};
```

The CORBA data type opaque is a sequence<octet>.

5.5.3.2.5 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.

5.5.3.2.6 Any

The CORBA typedef any maps as follows in C:

```
typedef struct any {
      CORBA_TypeCode _type;
      void * _value;}
      CORBA_any;
```

The TypeCode format is defined in the UNO CORBA 2.0 Inter-operability Specification. The first field is always an unsigned long that contains the tcKind. tcKinds are enumerated in the Interfaces interface.
Complex TypeCodes can contain a parameter list or an encapsulated CDR structure following the tcKind.

5.5.3.2.7 ev

ev is the environment structure. It is defined in the specific language mappings of the CORBA IDL. For example, it is the first parameter in the reply of the C language function mapping. In DSM-CC, **ev** is somewhat opaque in that it is specified as a structure with at a least **_ex** and **_major** members, and possibly additional implementation-specific members. **_ex** contains a string identifier plus the specific exception structure related to an invocation. **_major** identifies the exception type. _major is one of CORBA_NO_EXCEPTION, CORBA_SYSTEM_EXCEPTION or CORBA_USER_EXCEPTION.

```
enum CORBA_ExceptionType {
      CORBA_NO_EXCEPTION,
      CORBA_SYSTEM_EXCEPTION,
      CORBA_USER_EXCEPTION
    };

typedef struct CORBA_Environment {
      CORBA_ExceptionType  _major;
      CORBA_Exception *  _ex;
      ... // implementation-specific members
      } CORBA_Environment;
```

The CORBA_Exception is a structure that contains a string id in the form "<module>::<interface>::<exception>" or "<module>::<exception>". Note: the CDR Data Encoding for CORBA_exception is a string id followed by the exception body (the CORBA System exception structure or a User exception structure defined by the raises statement of the operation). CORBA provides the functions CORBA_exception_id() for accessing the string id, CORBA_exception_value() for accessing the exception structure, and CORBA_exception_free() for freeing memory associated with the exception. The member names are not defined, and it is recommended that the above pseudo-functions be used to access CORBA_Exception member values.

5.5.3.2.8 Object

Object in the C mapping is declared as type void *. This definition permits a flexibility of Client implementation.

5.5.3.3 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 subclause. Refer to that subclause for the descriptions of the interfaces or their operations.

5.5.3.3.1 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 all output parameters are valid.

5.5.3.3.1.1 Base

```
void DSM_Base_close (
      DSM_Base object,
      CORBA_Environment * ev)

void DSM_Base_destroy (
      DSM_Base object,
      CORBA_Environment * ev)
```

5.5.3.3.1.2 Access

```
DSM_u_longlong DSM_Access__get_Size (
      DSM_Access object,
      CORBA_Environment * ev)

DSM_Access_Hist_T DSM_Access__get_Hist (
      DSM_Access object,
      CORBA_Environment * ev)

void DSM_Access__set_Hist (
      DSM_Access object,
      CORBA_Environment * ev)
      DSM_Access_Hist_T val)

DSM_Access_Lock_T DSM_Access__get_Lock (
      DSM_Access object,
      CORBA_Environment * ev)

void DSM_Access__set_Lock (
      DSM_Access object,
      CORBA_Environment * ev)
      DSM_Access_Lock_T val)

DSM_Access_Perms_T DSM_Access__get_Perms (
      DSM_Access object,
      CORBA_Environment * ev)

void DSM_Access__set_Perms (
      DSM_Access object,
      CORBA_Environment * ev)
      DSM_Access_Perms_T val)
```

5.5.3.3.1.3 Stream

```
void DSM_Stream_pause (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_AppNPT * rStop)

void DSM_Stream_resume (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_AppNPT * rStart,
      DSM_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_AppNPT * rStart,
      DSM_AppNPT * rStop,
      DSM_Scale * rScale)

void DSM_Stream_play (
      DSM_Stream object,
      CORBA_Environment * ev,
      DSM_AppNPT * rStart,
      DSM_AppNPT * rStop,
      DSM_Scale * rScale)

DSM_Stream_Info_T DSM_Stream__get_Info (
      DSM_Stream object,
      CORBA_Environment * ev)

void DSM_Stream__set_Info (
      DSM_Stream object,
      CORBA_Environment * ev)
      DSM_Stream_Info_T val)
```

5.5.3.3.1.4 File

```
void DSM_File_read (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_u_longlong * aOffset,
      DSM_u_long aSize,
      CORBA_boolean * aReliable,
      DSM_opaque * rData)

void DSM_File_write (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_u_longlong * aOffset,
      DSM_u_long aSize,
      DSM_opaque * rData)

DSM_u_longlong DSM_File__get_ContentSize (
      DSM_File object,
      CORBA_Environment * ev)

DSM_opaque DSM_File__get_Content (
      DSM_File object,
      CORBA_Environment * ev);

void DSM_File__set_Content (
      DSM_File object,
      CORBA_Environment * ev,
      DSM_opaque * Content);
```

5.5.3.3.1.5 Directory

```
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_ObjRefs * resolvedRefs)
```

```
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)

CORBA_boolean CosNaming_BindingIterator_next_one (
      CosNaming_BindingIterator object,
      CORBA_Environment * ev,
      CosNaming_Binding * b)

CORBA_boolean CosNaming_BindingIterator_next_n (
      CosNaming_BindingIterator object,
      CORBA_Environment * ev,
      CORBA_unsigned_long how_many,
      CosNaming_BindingList * bl)

void CosNaming_BindingIterator_destroy (
      CosNaming_BindingIterator object,
      CORBA_Environment * ev)
```

5.5.3.3.1.6 Session

```
void DSM_Session_attach (
      DSM_Session object,
      CORBA_Environment * ev,
      CORBA_sequence_octet * aServiceDomain,
      CosNaming_Name * pathName,
      DSM_UserContext * savedContext,
      DSM_ObjRefs * resolvedRefs)

void DSM_Session_detach (
      DSM_Session object,
      CORBA_Environment * ev,
      CORBA_boolean aSuspend,
      DSM_UserContext * savedContext)
```

5.5.3.3.1.7 First

```
CORBA_Object DSM_First_root (
      DSM_First object,
      CORBA_Environment * ev)

CORBA_Object DSM_First_service (
      DSM_First object,
      CORBA_Environment * ev)
```

5.5.3.3.1.8 Event

```
void DSM_Event_subscribe (
      DSM_Event object,
      CORBA_Environment * ev,
      CORBA_string aEventName,
      DSM_u_short * eventId)

void DSM_Event_unsubscribe (
      DSM_Event object,
      CORBA_Environment * ev,
      DSM_u_short eventId)

void DSM_Event_notify (
      DSM_Event object,
      CORBA_Environment * ev,
      DSM_Event_StreamEvent * rStreamEvent)

DSM_Event_EventList_T DSM_Event__get_EventList (
      DSM_Event object,
      CORBA_Environment * ev);

void DSM_Event__set_EventList (
      DSM_Event object,
      CORBA_Environment * ev,
      DSM_Event_EventList_T * EventList);
```

5.5.3.3.1.9 Download

```
void DSM_Download_info (
      DSM_Download object,
      CORBA_Environment * ev,
      DSM_Download_ModuleInfoList * rModulesInfo)

void DSM_Download_alloc (
      DSM_Download object,
      CORBA_Environment * ev,
      DSM_u_short aModuleId,
      CORBA_Object rWriteBuffer,
      CORBA_Object * rReadBuffer)

void DSM_Download_start (
      DSM_Download object,
      CORBA_Environment * ev,
      DSM_Download_ModuleList * aModuleList)

void DSM_Download_cancel (
      DSM_Download object,
      CORBA_Environment * ev,
      DSM_Download_ModuleList * aModuleList)
```

5.5.3.3.1.10 Composite

```
void DSM_Composite_bind_subs (
      DSM_Composite object,
      CORBA_Environment * ev,
      CosNaming_Name * name,
      DSM_Composite_ChildRefs * rChildRefs)
```

```
void DSM_Composite_unbind_subs (
      DSM_Composite object,
      CORBA_Environment * ev,
      CosNaming_Name * name)

void DSM_Composite_list_subs (
      DSM_Composite object,
      CORBA_Environment * ev,
      CosNaming_Name * name,
      DSM_Composite_ChildInfos * infos)
```

5.5.3.3.1.11 View

```
void DSM_View_query (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_View_SQLStatement aSQLStatement,
      DSM_u_short maxRows,
      DSM_View_ViewDescribe * describe,
      DSM_View_Result * aResult,
      DSM_View * iterator)

void DSM_View_read (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_u_short aCursor,
      DSM_u_short maxRows,
      DSM_View_Result * aResult)

void DSM_View_execute (
      DSM_View object,
      CORBA_Environment * ev,
      DSM_View_SQLStatement aSQLStatement)

CORBA_char DSM_View__get_Style (
      DSM_View object,
      CORBA_Environment * ev)
```

5.5.3.3.1.12 State

```
void DSM_State_suspend (
      DSM_State object,
      CORBA_Environment * ev,
      CORBA_boolean aRelease,
      DSM_UserContext * savedContext)

void DSM_State_resume (
      DSM_State object,
      CORBA_Environment * ev,
      DSM_UserContext * savedContext,
      DSM_ObjRefs * restoredRefs)
```

5.5.3.3.1.13 Interfaces

```
void DSM_Interfaces_show (
      DSM_Interfaces object,
      CORBA_Environment * ev,
      CORBA_string aStrKind,
      DSM_IFKind anIFKind,
      CORBA_string * rIDL,
      DSM_IntfCode * rIntf,
      DSM_TypeCodeList * rTypes)

void DSM_Interfaces_define (
      DSM_Interfaces object,
      CORBA_Environment * ev,
      DSM_Interfaces_ReferenceData * id,
      DSM_Interfaces_InterfaceDef rIDL,
      DSM_IntfCode * rIntf,
      DSM_TypeCodeList * rTypes)

void DSM_Interfaces_check (
      DSM_Interfaces object,
      CORBA_Environment * ev,
      DSM_IFKind anIFKind)

void DSM_Interfaces_undefine (
      DSM_Interfaces object,
      CORBA_Environment * ev,
      DSM_Interfaces_ReferenceData * id,
      DSM_IFKindList * usedBy)
```

5.5.3.3.1.14 Security

```
void DSM_Security_authenticate (
      DSM_Security object,
      CORBA_Environment * ev,
      DSM_AuthRequest_T * authInfo)
```

5.5.3.3.1.15 LifeCycle

```
void DSM_LifeCycle_create (
      DSM_LifeCycle object,
      CORBA_Environment * ev,
      CORBA_string aKind,
      DSM_Version * rVersion,
      IOP_IOR * rObjRef)
```

5.5.3.3.1.16 Kind

```
void DSM_Kind_has_a (
      DSM_Kind object,
      CORBA_Environment * ev,
      DSM_IFKind anIFKind,
      CORBA_boolean * aVerdict)

void DSM_Kind_is_a (
      DSM_Kind object,
      CORBA_Environment * ev,
      DSM_IntfCode * whatItIs)
```

5.5.3.3.2 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 DSM-CC interfaces:

5.5.3.3.2.1 Config

These functions are used to configure the DSM-CC 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)

CORBA_boolean DSM_Config__get_DeferredSync (
      DSM_Config object,
      CORBA_Environment * ev);

 void DSM_Config__set_DeferredSync (
      DSM_Config object,
      CORBA_Environment * ev,
      CORBA_boolean DeferredSync)

DSM_Config_RequestList DSM_Config__get_ActiveRequests (
      DSM_Config object,
      CORBA_Environment * ev)
```

5.5.3.3.2.2 How to Convert Synchronous to Synchronous Deferred

In the following example, we show the C mapping differences between Synchronous and Synchronous Deferred:

Synchronous:

```
void DSM_Base_close (
      DSM_Base object,
      CORBA_Environment * ev)
```

Synchronous Deferred:

```
DSM_RequestHandle DSM_Base_close (
      DSM_Base object,
      CORBA_Environment * ev)
```

Any Synchronous API that returns void can be converted to Synchronous Deferred by defining it to return DSM_RequestHandle instead of void, as shown above.

5.6 Service Interoperability Interfaces(SII)

This subclause describes the interfaces that are use by the DSM-CC Library Stubs to invoke RPCs and U-N Session message sequences over the network.

5.6.1 ConnBinder and Resource to Connection Association

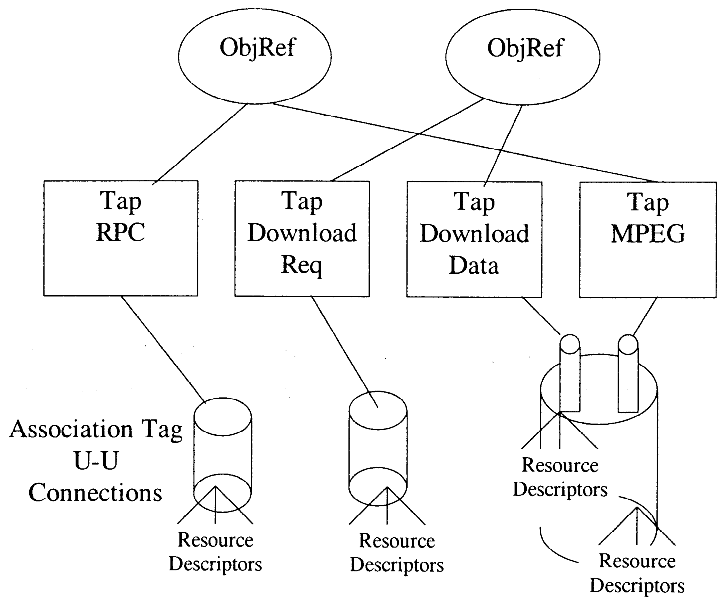
As a result of Broadcasts, Session Establishment and RPC resolve operations, a Client shall receive connection-related information for potentially multiple communication paths to objects. For each object, channels can be setup that have various purposes.

For example, communication with an Stream object (using object access Model D) may require RPC and MPEG channels. Communication with an Object Carousel Stream (using object access model B) may require elementary stream identification.

The U-U Protocol defines the **Tap**, which establishes the link from a User-to-User object reference to a lower layer communication channel. It further defines **ConnBinder** as a sequence of all the Taps used for communication with a given object.

The **Tap** has the following information:

1. An **id**. This identifies the Tap to the Client. It may be reused.
2. A **use**. This is a indication as to the type of the connection.
3. An **association tag**. This tag identifies a set of U-N Network ResourceDescriptors which have the same association tag value.
4. A **selector**. This is an opaque value which is only non-zero when the Server performs internal multiplexing of a network-level connection. If this is the case the selector is used to select a particular application-level association from those defined by the multiplexing scheme. The selector format is Tap use specific. An example of a tap selector might be a PID when the network connection is a single program transport stream.



IDL Syntax: ConnBinder and Tap

```
module DSM {
    struct Tap {
        u_short id;             // identifier
        u_short use;            // the use for the Tap
        u_short assocTag;       // the group identifier for network resource descriptors
        sequence<octet, 255> selector;  // upper protocol selection info
    };
    typedef sequence<Tap, 255> ConnBinder;  // typically have request and data channels

    typedef u_short TapUse;
    const TapUse UNKNOWN_USE = 0;
    const TapUse MPEG_TS_UP_USE = 1;                  // MPEG transport upstream from Client
    const TapUse MPEG_TS_DOWN_USE = 2;                         // MPEG transport downstream to
Client
    const TapUse MPEG_ES_UP_USE = 3;                  // MPEG elementary upstream from Client
    const TapUse MPEG_ES_DOWN_USE = 4;                         // MPEG elementary downstream to
Client
    const TapUse DOWNLOAD_CTRL_USE = 5;               // control request/response
    const TapUse DOWNLOAD_CTRL_UP_USE = 6;  // control request from Client
    const TapUse DOWNLOAD_CTRL_DOWN_USE = 7; // control response to Client
    const TapUse DOWNLOAD_DATA_USE = 8;               // data request/response
    const TapUse DOWNLOAD_DATA_UP_USE = 9;  // data response upstream from Client
    const TapUse DOWNLOAD_DATA_DOWN_USE = 10; // data block downstream to Client
    const TapUse STR_NPT_USE = 11;                             // NPT Descriptors
    const TapUse STR_STATUS_AND_EVENT_USE = 12; // Stream Mode and Event Descriptors
    const TapUse STR_EVENT_USE = 13;                  // Stream Event Descriptor
    const TapUse STR_STATUS_USE =14;                  // Stream Mode Descriptor
    const TapUse RPC_USE = 15;                        // RPC bi-directional
    const TapUse IP_USE = 16;                         // IP bi-directional
    const TapUse SDB_CTRL_USE = 17;            // control channel for Switched Digital Broadcast
    const TapUse T120_TAP1 = 18;               // reserved for use and definition by T.120
    const TapUse T120_TAP2 = 19;               // reserved for use and definition by T.120
    const TapUse T120_TAP3 = 20;               // reserved for use and definition by T.120
    const TapUse T120_TAP4 = 21;               // reserved for use and definition by T.120
    const TapUse BIOP_DELIVERY_PARA_USE = 22; // Module delivery parameters
    const TapUse BIOP_OBJECT_USE = 23;   // BIOP objects in Modules
    const TapUse BIOP_ES_USE = 24;                    // Elementary Stream
    const TapUse BIOP_PROGRAM_USE = 25;               // Program
    const TapUse BIOP_DNL_CTRL_USE = 26;              // Download control messages
};
```

The ConnBinder shall be carried in each DSM-CC IOR.

For U-N Sessions, it is not assumed that the IORs can be decoded immediatly following Session Establishment (U-N Download might deliver the decoder). Therefore, a separate ConnBinder for Download Taps is explicitly placed in the **SessionUU attach()** output. The Server in its negotiations with the network shall determine the association tag and resource group assignments for each end-to-end connection. During the Session, the network shall inform the Client of changes to the association tags and related network resources through U-N Session add resource messages. Meanwhile, in the resolve reply, the Server shall return the association tag, use and selector for each Tap in the ConnBinder of the object reference.

Multiple Taps may share the same association tag, enabling one communication path to be used for more than one purpose.

5.6.1.1 Selector

A 0 length Tap selector shall indicate no selector information is present. Otherwise, the first 16 bits of the selector data shall be a type identifier to specify the remaining structure of the selector.

```
module DSM {
    typedef u_short SelectorType;
    // SelectorType 0  is ISO/IEC reserved
    const SelectorType MESSAGE = 1;
    struct MessageSelector {
        u_long transactionId;
        u_long timeout;
    };
};
```

SelectorType MESSAGE identifies a **MessageSelector.** It is used by the Object Carousel. It contains a transactionId field and a timeout field. The value of the transactionId field shall be set to the transactionId of the DownloadInfoIndication message that contains the module delivery parameters. The timeout field shall indicate the timeout period to be used to time out the acquisition of the DownloadInfoIndication message. The units of the timeout field are microseconds. Refer to the clause 11, U-U Object Carousel, for further information.

5.6.2 Remote Procedure Call

The User-to-User primitives require the use of remote procedure calls to invoke operations over the network. The preferred and default RPC is Universal Networked Objects (UNO). The preferred and default data representation is Common Data Representation (CDR), also defined in the same specification. The required object reference format is the Interoperable Object Reference (IOR). These protocols and representations are defined by OMG specification "CORBA 2.0 Inter-operability: Universal Networked Objects.". UNO does not require the use of TCP.

An alternative RPC may be chosen and negotiated by exchange of the Download InfoRequest CompatibilityDescriptor and the IOR's ProfileId. Following this negotiation, the RPC request and reply message headers shall be those of the chosen RPC, and the data representation shall conform to the chosen encoding.

In a DSM-CC network, the RPC request header shall contain the following parameters:

- Requesting Principal (type opaque). The Requesting Principal is the identifier of the human or process that is controlling the requesting Client Application.
- Object Key(type opaque). The object key identifies a client/service connection at the server Object.
- Service Context (type IOP::ServiceContextList). The ServiceContextList carries optional client/service information.

In a DSM-CC network, the RPC reply header shall (at a minimum)contain the following parameters:

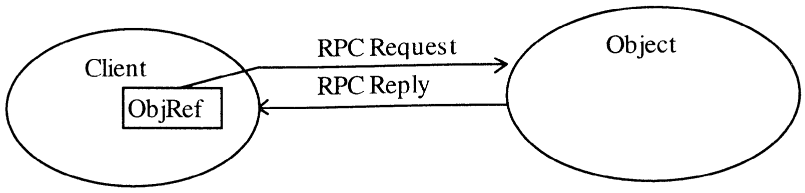
- Reply Status (type enumeration). The Reply Status, as defined by GIOP, is an enumeration:

```
module GIOP {
    enum ReplyStatusType {
        NO_EXCEPTION,
        USER_EXCEPTION,
        SYSTEM_EXCEPTION,
        LOCATION_FORWARD};
};
```

- Service Context (type IOP::ServiceContextList). The ServiceContextList carries optional client/service information. It is defined later in this subclause.

5.6.3 The Object Reference

A Service Object Implementation instance is uniquely identified within the full DSM-CC system environment by its object reference. At the API level the object reference is a handle, with no need by the application to access its contents. However, in the RPC Stub, it is structured, and at a minimum contains the IOP Interoperable Object Reference(IOR).



Session attach(), Directory resolve(),and **Directory open()** are the resolve operations of DSM-CC User-to-User. A resolve operation takes a name input and returns a reference to an object instance. This reference is called an object reference. It contains addressing information that uniquely identifies the object, e.g., by host, port, version and object_key. The object reference is used in construction of the Network and RPC request headers that are used in routing the RPC request to the object.

DSM-CC uses the Interoperable Object Reference(IOR) format defined by OMG for object references at the Client-Service Inter-operability Interface.

```
module IOP {
    typedef unsigned long ProfileId;
    const ProfileId TAG_INTERNET_IOP = 0;
    const ProfileId TAG_MULTIPLE_COMPONENTS = 1;
    struct TaggedProfile {
        ProfileId tag;
        sequence<octet>  profile_data;
    };
    struct IOR {
        string type_id;
        sequence<TaggedProfile> taggedProfileList;
    };
    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId  tag;
        sequence<octet> component_data;
    };
    typedef sequence <TaggedComponent> MultipleComponentProfile;
};
```

DSM-CC has registered a range of ProfileId values with OMG. DSM-CC defines several Profile Body structures, each with an assigned ProfileId. The ProfileBody is placed in the encapsulation profile_data of the IOR. The first octet of the encapsulation is a byte_order, where a value of 0 (FALSE) indicates big-endian, and 1 (TRUE) indicates little-endian. Following that, the CORBA 2.0 encapsulation contains the encoded (according to ProfileId) connection information. The encapsulation begins at an octet according to the alignment rules of the data encoding, i.e., there may be padding for proper alignment to the first data type in the encapsulation.

The following range of ProfileIds has been registered with OMG:

Profile Tags	0x49534F00 - 0x49534F0F	(the first 3 octets spell "ISO")

5.6.3.1 Min Protocol Profile

The DSM-CC Min Protocol Profile is a CORBA 2.0 single profile. It uses CDR encoding and UNO RPC. It has the minimum required ConnBinder. The ConnBinder must contain the RPC_USE Tap. The selector field of this Tap further identifies the connection, according to the network stack in use.

```
module DSM {
const IOP::ProfileId TAG_MIN = 1230196480; // 0x49534F00
    struct MinProfileBody {
        ConnBinder cbind;
    };
};
```

5.6.3.2 Child Protocol Profile

The DSM-CC Child Protocol Profile is a CORBA 2.0 single profile. It uses CDR encoding and UNO RPC. It is typically used for Composite child object references. It has the minimum required ConnBinder, plus a string for identifying the object.

```
module DSM {
const IOP::ProfileId TAG_CHILD = 1230196481; // 0x49534F01
    struct ChildProfileBody {
        ConnBinder cbind;
        string nameId;
    };
};
```

5.6.3.3 Options Protocol Profile

The DSM-CC Options Profile is a CORBA 2.0 multiple component profile. It uses CDR encoding and UNO RPC. It enables a variable list of information to be included in the IOR. The Component Tag range of 0x49534F40 - 0x49534F7F is reserved with OMG, and used by DSM-CC for IOR connection-related information. DSM-CC TaggedComponents are encoded as a ComponentId followed by length, then followed by the encoding of the type identified. DSM-CC defines the following TaggedComponents for use in this profile.

```
module DSM {
    const IOP::ProfileId TAG_OPTIONS = 1230196482; // 0x49534F02

    const IOP::ComponentId TAG_ConnBinder = 1230196544; // 0x49534F40
    //
    const IOP::ComponentId TAG_IIOPAddr= 1230196545; // 0x49534F41
    typedef IIOP::ProfileBody IIOPAddrComponent;
    //
    const IOP::ComponentId TAG_Addr = 1230196546; // 0x49534F42
    struct AddrComponent {string host; u_short port;};
    //
    const IOP::ComponentId TAG_NameId = 1230196547; // 0x49534F43
    typedef string NameIdComponent;
    //
    const IOP::ComponentId TAG_IntfCode = 1230196548; // 0x49534F44
    //
    const IOP::ComponentId TAG_ObjectKey = 1230196549; // 0x49534F45
    typedef sequence<octet> ObjectKeyComponent;
    //
    const IOP::ComponentId TAG_ServiceLocation = 1230196550; // 0x49534F46
};
```

TAG_ConnBinder identifies a **ConnBinder** structure. It is mandatory for all DSM-CC IORs. The ConnBinder is described in the previous subclause.

TAG_IIOPAddr identifies a **IIOP::ProfileBody**. It is used to hold iiop_version, host, port and object_key, as defined by the CORBA IIOP ProfileBody. iiop_version describers the version of IIOP that the agent at the specified address is prepared to communicate with. The host is the Internet host to which messages may be sent. It may be a fully quallified domain name or Internet standard "dotted decimal" form (e.g., "192.231.79.52"). The port is the TCP/IP or UDP/IP port number at the specified host where the target agent is listening for requests. The object_key is opaque value supplied by the agent producing the IOR. It uniquely identifies the object instance.

```
module IIOP {
    struct Version{
        char major; char minor;
    };
    struct ProfileBody {
        Version iiop_version;
        string host;
        unsigned short port;
        sequence<octet> object_key;
    };
};
```

TAG_Addr identifies an **AddrComponent**. It provides IP host address and port. The semantics of host and port are the same as described above for **TAG_IIOPAddr**.

TAG_NameId identifies a **NameIdComponent**. It is used to hold the bound name identifier of the object, such as for a child of a Composite object.

TAG_IntfCode identifies an **IntfCodeComponent**. It describes inherited interfaces.

TAG_ObjectKey identifies an **ObjectKeyComponent**. It contains the unique identification of an object instance in the context of a Server Object Implementation.

TAG_ServiceLocation identifies a **ServiceLocation**. It may be used to convey **aServiceDomain**, **pathName** and **savedContext** for a subsequent **Session attach()**.

5.6.3.4 Lite Protocol Profiles

The Lite Profiles have the same IDL syntax as their counterpart DSM-CC CDR profiles. The encoding is CDR-Lite. CDR-Lite differs from CDR in two ways:

- It uses the maximum value on sequences and strings to reduce the size of the encoded length value. An IDL-specified maximum of 255 results in an encoded octet to hold the sequence or string length. An IDL-specified maximum of 65,535 results in encoded unsigned short to hold the sequence or string length.
- It removes the CDR requirement for byte alignment in order to achieve compact packing of data.

A Multiple Component Profile called **LiteComponentProfile** is used for TAG_LITE_OPTIONS, in order to reduce the size of the profile.

```
module DSM {
    struct LiteComponent {
        IOP::ComponentId tag;
        sequence<octet, 255> component_data;
    };
    typedef sequence<LiteComponent, 255> LiteComponentProfile;

    const IOP::ProfileId TAG_LITE_MIN = 1230196483; //  0x49534F03
    const IOP::ProfileId TAG_LITE_CHILD = 1230196484; //  0x49534F04
    const IOP::ProfileId TAG_LITE_OPTIONS = 1230196485; //  0x49534F05
};
```

5.6.3.5 BIOP Protocol Profile

The BIOP profile identifies the default data encoding method used within DSM-CC U-U Object Carousels. The default data encoding is CDR-Lite.

```
module DSM {
    const IOP::ProfileId TAG_BIOP = 1230196486; //  0x49534F06
};

module BIOP {
    const IOP::ComponentId  TAG_ObjectLocation =  1230196560; //  0x49534F50

    struct ObjectLocation {
        unsigned long              carouselId;
        unsigned short             moduleId;
        DSM::Version               version;
        sequence <octet, 255>      objectKey;
    };
};
```

BIOP::TAG_ObjectLocation identifies a **BIOP::ObjectLocation** component. It uniquely locates the object within the Broadcast network. Refer to clause 11, U-U Object Carousel, for further information.

5.6.3.6 ONC Protocol Profile

The DSM-CC ONC Profile is a CORBA 2.0 multiple component profile. It uses XDR encoding and ONC RPC. It enables a variable list of information to be included in the IOR. The TaggedComponents of the Options Profile may be included. In addition, DSM-CC defines the following TaggedComponents for use in this profile.

```
module DSM {
    const IOP::ProfileId TAG_ONC = 1230196487; //  0x49534F07
};

module DSM_ONC {
    const IOP::ComponentId TAG_Intf  = 1230196568; //  0x49534F58

    struct IntfComponent {
        unsigned long aProgram;
        unsigned long aVersion;
    };
};
```

DSM_ONC::TAG_Intf identifies the **DSM_ONC::IntfComponent**. This structure contains ONC Program and ONC Version. Refer to Informative Annex C, ONC RPC XDR Mappings, for further information.

5.6.4 ServiceContextList

The UNO RPC Request and Reply Message Headers carry a **ServiceContextList** parameter that is useful in carrying certain information related to the RPC but not necessarily information that is exposed to the application level. Examples of this are authentication, compatibility information, and End-User preferences. DSM-CC uses this encoding to carry similar information in the U-N Session Establishment messages **uuData** field. The ServiceContextList is defined by CORBA 2.0, shown below:

```
module IOP {
    typedef unsigned long ServiceID;
    struct ServiceContext {
        ServiceID  context_id;
        sequence<octet> context_data;
    };
    typedef sequence <ServiceContext> ServiceContextList;
    const ServiceID TransactionService =  0;
};
```

The context_data in the ServiceContext is a UNO encapsulation, meaning that it always begin with an octet that has a boolean value for byte order. FALSE (0) byte_order indicates big-endian, TRUE (1) byte_order indicates little-endian for the ensuing encapsulation.

If there is no ServiceContext, the **ServiceContextList** shall consist of a DSM::u_long with value = 0.

Note: The CORBA 2.0 ServiceId should not be confused with the DSM-CC ServerId. The ServiceId identifies some application contextual information, whereas the DSM-CC ServerId is a Server identifier as specified by clause 4 of this part of ISO/IEC 13818.

5.6.4.1 ServiceContext

The ServiceContext is identified by a ServiceContext identifier. Within DSM-CC, ServiceContext identifiers are expected to be allocated as ISO reserved and private, where identifiers 0-255 are ISO reserved and 256-2^{32} are for private application use.

The ServiceContext identifier shall have the following format:

MSB Octet: a disposition which indicates how the receiving object should respond to a ServiceContext indication:

0: Act on the message and the ServiceContext
1: Return an error if the context_id is unknown
2: Ignore the ServiceContext if the context_id is unknown

The 24 bits of the 3 LSB octets are the actual id with reserved and private values as described above. The following are reserved values for DSM-CC:

value	structure	description
0		ISO/IEC reserved
1	DSM::CompatibilityDescriptor	Compatibility information format (ref clause 6) in request or reply. Used for operation-specific compatibility exchange.
2	DSM::Download InfoRequest	For negotiation large transfer flow-control in a request. An operation may imply a possible Download. Without changing the syntax of the operation, Download request and response can be carried in the ServiceContextList.
3	DSM::Download InfoResponse	For negotiation large transfer flow-control in a reply.
4	DSM::AuthRequest_T	A **Security authenticate**() request can cause insertion of authentication data in the ServiceContextList of the following operation.
5	DSM::ConnBinder	For carrying Download Taps information in a File read() or other reply to a large transfer request.
6	DSM::Version	To identify version selection in a resolve request, for use where a version other than latest version is required.

5.6.5 Core Interfaces

5.6.5.1 Base

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the Base API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Base {
        const AccessRole close_ACR = READER;
        void  close ();
#ifdef DSM_GENERAL
        const AccessRole destroy_ACR = OWNER;
        void destroy  ();
#endif
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

Base close() implies that network resources needed for the Client communication to that object can be deleted, if these resources are not used for communication between the Client and another object.

Base destroy() indicates that the object shall cease to exist for all Clients, and all resources related to it should be deleted.

5.6.5.2 Access

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the Access API for data type and attribute descriptions.

Application Portability / Service Inter-operability Syntax

```
 module DSM {
 struct DateTime {              // tm from ANSI C std. See Kernighan & Ritchie, 2nd edition, p. 255
         s_long tm_sec;         // seconds after midnight, 0-61
         s_long tm_min;         // minutes after the hour, 0-59
         s_long tm_hour;        // hours since midnight, 0-23
         s_long tm_mday;        // day of the month, 1-31
         s_long tm_mon;         // months since January, 0-11
         s_long tm_year;        // years since 1900
         s_long tm_wday;        // days since Sunday, 0-6
         s_long tm_yday;        // days since Jan 1, 0-365
         s_long tm_isdst;};     // Daylight Savings Time flag

     interface Access {
 #ifdef DSM_GENERAL
         // size
         const AccessRole Size_get_ACR = READER;
         readonly attribute u_longlong Size;          // size of all attributes in octets;
 #endif
         // history
         struct Hist_T {
                 Version  aVersion;          // object version
                 DateTime  aDateTime;};  // time created or last updated, GMT
 #ifdef DSM_GENERAL
         const AccessRole Hist_get_ACR = READER;
         const AccessRole Hist_put_ACR = BROKER;
         attribute Hist_T Hist;                  // version and time of persistent object
 #endif
         // lock status
         struct Lock_T {boolean readLock; boolean writeLock;};
         const AccessRole Lock_get_ACR = READER;
         const AccessRole Lock_put_ACR = WRITER;
         attribute Lock_T Lock;

         // permissions
         struct Perms_T {
                 // the next 4 are binary masks of binary flags signifying
                 // groups that can access the object
                 u_short managerPerm;
                 u_short brokerPerm;
                 u_longlong writerPerm;
                 u_longlong readerPerm;
                 opaque owner;                   //owner identifier = Principal
                 string aPassword;          //PIN
                 opaque authData;           //system-specific
                 // instruct lower layers to implement a secure connection for this object
                 boolean allSecure;};            // all methods parameters encrypted
 #ifdef DSM_GENERAL
         const AccessRole Perms_get_ACR = OWNER;
         const AccessRole Perms_put_ACR = OWNER;
         attribute Perms_T Perms;
 #endif
     };
 };
```

Service Inter-operability Semantics

The End-User's Principal Id shall be sent in the RPC message header of the request. The Requesting Principal is available as an index associating the End-User's AccessRoles. The DSM-CC Library may optionally send authentication data (**AuthRequest_T** containing encryption information or password) in the ServiceContextList of the request. At the Service, authorization to invoke shall be performed on a per operation basis, according to the Access Control Role of the operation, the AccessRole of the End-User, and authentication data. It is beyond the scope of DSM-CC to describe the authentication and authorization mechanisms.

TapUse

RPC_USE

5.6.5.3 Stream

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the Stream API for state machine and individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface Stream : Base, Access {
        typedef u_long Mode;
        const Mode OPEN_M = 0;
        const Mode PAUSE_M = 1;
        const Mode TRANSPORT_M = 2;
        const Mode TRANSPORT_PAUSE_M = 3;
        const Mode SEARCH_TRANSPORT_M = 4;
        const Mode SEARCH_TRANSPORT_PAUSE_M = 5;
        const Mode PAUSE_SEARCH_TRANSPORT_M = 6;
        const Mode END_OF_STREAM_M = 7;
        const Mode PRE_SEARCH_TRANSPORT_M = 8;
        const Mode PRE_SEARCH_TRANSPORT_PAUSE_M = 9;
        struct Stat {
                AppNPT rPosition;
                Scale rScale;
                Mode aMode;};
        struct Info_T {
                string<255> aDescription;
                AppNPT duration;
                boolean audio;
                boolean video;
                boolean data;};
        const AccessRole Info_get_ACR = READER;
        const AccessRole Info_put_ACR = OWNER;
        attribute Info_T  Info;
        const AccessRole pause_ACR = READER;
        void pause (in AppNPT rStop)
                raises (MPEG_DELIVERY, BAD_STOP);
        const AccessRole resume_ACR = READER;
        void resume (in AppNPT rStart, in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_SCALE);
        const AccessRole status_ACR = READER;
        void  status (in Stat rAppStatus, out Stat rActStatus)
                raises (MPEG_DELIVERY);
        const AccessRole reset_ACR = READER;
        void reset ();
        const AccessRole jump_ACR = READER;
        void   jump (
                in AppNPT rStart,
                in AppNPT rStop,
                in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE);
        const AccessRole play_ACR = READER;
        void  play (
                in AppNPT rStart,
                in AppNPT rStop,
                in Scale rScale)
                raises (MPEG_DELIVERY, BAD_START, BAD_STOP, BAD_SCALE);
    };
};
```

Possible TapUses

RPC_USE	operation requests and replies
MPEG_DOWN_TS_USE	MPEG-2 Transport Stream delivery
MPEG_DOWN_ES_USE	MPEG-2 Elementary Stream delivery
STR_NPT_USE	NPT Descriptors
STR_STATUS_AND_EVENT_USE	Stream Mode and Event Descriptors
STR_STATUS_USE	Stream Mode Descriptor

Service Inter-operability Semantics

The DSM-CC Library Client Stream stub may be modified to intercept MPEG-2 Stream descriptors that contain Normal Play Time and Stream state machine information. In this case, **Stream status()** may become a local library operation that returns parameters of higher accuracy.

5.6.5.3.1 Transport and Application Level NPT

There are two formats for NPT, transport level NPT and application level NPT.

The transport NPT on the MPEG transport, as described in clause 8 of this part of ISO/IEC 13818, is in PTS 33 bit format. The reason is that networks may want to deal with Audio, Video and DSM-CC descriptors using the same timestamp format. The conversion from PTS format to seconds and microseconds is easy and not compute-intensive. the conversion the other way is not. The Server will format 33 bit values (in 64 bit parameters) on the MPEG downstream for all NPT descriptors.

The application NPT is raised to the human level, i.e. seconds and microseconds, to satisfy requirements from MHEG and many application-level users. This is referred to as application NPT. This can be carried on the RPC request to the Server in this form (the current IDL). The Server can convert from application NPT to transport NPT bit format if needed.

Conversion from transport NPT to application NPT is included in clause 8 of this document.

5.6.5.3.2 Consistent Quantization Rules

The Stream Service shall always apply consistent quantization rules:

- Resume at current NPT (retrieved by **Stream status()**) after pause now will be no loss (will not drop nor duplicate stream data).
- Resume now after pause now will be no loss (will not drop nor duplicate stream data).
- Resume at other time than now means no guarantee of the frame relationship to a previous pause (now).

Regarding Stream Mode descriptors and NPT descriptors in the MPEG stream:

- Sending of NPT and Stream Mode descriptors is optional. However, sending of Stream Mode descriptor for End of Stream is mandatory. The Stream object must send an EOS Stream Mode Descriptor when the State Machine transitions to EOS.
- There is a general policy to include descriptors just before discontinuities.
- Use post_discontinuity indicator to define when effective (see subclause 8.4.1).
- Stream Mode descriptors may be sent in the MPEG transport stream upon occurrence of a stream state machine change.

5.6.5.4 File

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the File API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface File : Base, Access {
        const AccessRole Content_get_ACR = READER;
        const AccessRole Content_put_ACR = WRITER;
        attribute opaque Content;           // file content
#ifdef DSM_GENERAL
        const AccessRole ContentSize_get_ACR = READER;
        readonly attribute u_longlong ContentSize; // file content size in octets
#endif
        const AccessRole read_ACR = READER;
        void read (
                in u_longlong aOffset,
                in u_long aSize,
                in boolean aReliable,
                out opaque rData)
                raises (INV_OFFSET, INV_SIZE, READ_LOCKED);
        const AccessRole write_ACR = WRITER;
        void  write (
                in u_longlong aOffset,
                in u_long aSize,
                in opaque rData)
                raises (INV_OFFSET, INV_SIZE, WRITE_LOCKED);
    };
};
```

TapUse

RPC_USE

5.6.5.5 BindingIterator

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the Directory API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax: CosNaming Types

```
module CosNaming{
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
    // note: BindingType equates to the CORBA enum definition
    // while allowing extension for DSM-CC implementations
    typedef unsigned long BindingType;
    const BindingType nobject = 0;
    const BindingType ncontext = 1;
    struct Binding {
        Name binding_name;
        BindingType binding_type;
    };
    typedef sequence <Binding> BindingList;
};
```

Application Portability / Service Inter-operability Syntax: Operations

```
module CosNaming {
    interface BindingIterator {
        boolean next_one (out Binding b);
        boolean next_n (in unsigned long how_many,
                        out BindingList bl);
        void destroy ();
    };
};
```

TapUse

RPC_USE

5.6.5.6 NamingContext

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the Directory API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
  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 { };

          void list (in unsigned long how_many,
                   out BindingList bl, out BindingIterator bi);
          Object resolve (in Name n)
              raises (NotFound, CannotProceed, InvalidName);
  #ifdef DSM_GENERAL
          void bind (in Name n, in Object obj)
              raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
          void bind_context (in Name n, in NamingContext nc)
              raises (NotFound, CannotProceed, InvalidName, AlreadyBound);
          void rebind (in Name n, in Object obj)
              raises (NotFound, CannotProceed, InvalidName);
          void rebind_context (in Name n, in NamingContext nc)
              raises (NotFound, CannotProceed, InvalidName);
          void unbind (in Name n)
              raises (NotFound, CannotProceed, InvalidName);
          NamingContext new_context();
          NamingContext bind_new_context (in Name n)
              raises (NotFound, AlreadyBound, CannotProceed, InvalidName);
          void destroy ()
              raises (NotEmpty);
  #endif
      };
  };
```

TapUse

RPC_USE

5.6.5.7 Directory

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the Directory API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
 module DSM {
     interface Directory : Access, CosNaming::NamingContext {
         // Access Control Roles for inherited NamingContext operations
         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;
         void open(
                 in PathType aPathType,
                 in PathSpec rPathSpec,
                 out ObjRefs resolvedRefs)
         raises(OPEN_LIMIT, NO_AUTH, UNK_USER, SERVICE_XFR,
                 NOT_FOUND, CANNOT_PROCEED, INV_NAME);
         const AccessRole close_ACR = READER;
         void  close ();
         const AccessRole get_ACR = READER;
         void get(
                 in PathType aPathType,
                 in PathSpec rPathSpec,
                 out PathValues rPathValues)
         raises(NO_AUTH, UNK_USER, SERVICE_XFR,
                 NOT_FOUND, CANNOT_PROCEED, INV_NAME);
 #ifdef DSM_GENERAL
         const AccessRole put_ACR = WRITER;
         void put(
                 in PathType aPathType,
                 in PathSpec rPathSpec,
                 in PathValues rPathValues)
         raises(NO_AUTH, UNK_USER, SERVICE_XFR,
                 NOT_FOUND, CANNOT_PROCEED, INV_NAME);
 #endif
     };
 };
```

TapUse

RPC_USE

Service Inter-operability Semantics

AccessRoles are defined for all inherited CosNaming operations. The Directory Service therefore performs authorization of Clients to perform these operations.

A **ConnBinder** must be present in the object reference (IOR) of the **Directory resolve()** and **Directory open()** RPC replies.

The API level **Directory resolve()** or **Directory open()** may result in a local operation to return an object reference that was previously obtained through **SessionUU attach()** or **Directory open()** RPC of a Composite object.

5.6.6 Extended Interfaces

5.6.6.1 SessionUU

The SessionUU interface provides definitions for uuData of the DSM-CC U-N Session establishment and teardown messages. The in parameters become the uuData in the ClientSessionSetupRequest, and the out parameters become the uuData in the ClientSessionSetupConfirm.

pseudo-IDL

```
module DSM {
    interface SessionUU {
        const AccessRole attach_ACR = READER;
        void  attach (
                    in opaque downloadInfoReq,          // download info request
                    in CosNaming::Name pathName,        // path name to resolve
                    in UserContext savedContext,        // previous application user context
                    in Principal aPrincipal,            // identification of End User
                    in IOP::ServiceContextList inSC, // optional Service info
                    out opaque downloadInfoResp,        // download info response
                    out ConnBinder downloadTaps,        // download connection info
                    out ObjRefs resolvedRefs,                   // objects resolved
                    out IOP::ServiceContextList  outSC) // optional Service info
                    raises (NO_AUTH, BAD_COMPAT_INFO, UNK_USER,
                            SERVICE_XFR, NO_RESUME, OPEN_LIMIT,
                            NOT_FOUND, CANNOT_PROCEED, INV_NAME);
        const AccessRole detach_ACR = READER;
        void  detach (
            in boolean aSuspend,
            in Principal aPrincipal,            // identification of End User
            in IOP::ServiceContextList inSC,            // optional Service info
            out IOP::ServiceContextList outSC,          // optional Service info
            out UserContext savedContext) // suspended user context
            raises (NO_SUSPEND);
    };
};
```

Service Inter-operability Semantics

Both **SessionUU attach()** and **SessionUU detach()** are used by local library Session object operations **Session attach()** and **Session detach()**, respectively.

SessionUU attach() specifies the parameters to be included in the uuData fields of the User-Network Session Establishment messages. The input parameters are placed in uuData field of the U-N ClientSessionSetupRequest, and the output parameters are retrieved from uuData field of the U-N ClientSessionSetupResponse. **downloadInfoRequest** and **DownloadInfoResponse** contain the data encodings specified in clause 7. An identification of a previously suspended user context **savedContext** 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. A path specification **pathName** names the path to ServiceGatewayUU and possibly a first Service to be opened. **resolvedRefs** shall contain an object reference for the ServiceGatewayUU and resolved object reference(s) for the first Service. A default Service may be specified in the second **Step** of **pathName** as a NULL string **NameComponent.id**. This shall be an indication to the ServiceGatewayUU that it can choose the first Service for the Client. Each resolved IOR of the **SessionUU attach()** output must contain a **ConnBinder**.

The ServiceGatewayUU output from SessionUU attach() is presented to the Client as a ServiceGateway object reference, i.e., the Client only sees the ServiceGateway, while the remote Server implements ServiceGatewayUU.

SessionUU detach() specifies the parameters to be included in the uuData fileds of the User-Network Session Teardown messages.

In addition to Session Teardown, all object references are closed for that Session. If **aSuspend** is true, the ServiceGatewayUU shall inform Services which are maintaining user context for this End User to return user context state for a possible resumption of those Services. It is up to the application to determine which state is to be returned and maintained between Sessions. The Client may later invoke **attach()** with this **UserContext** in order to resume from the saved state.

aPrincipal specifies a specific user (i.e., human) or requesting principal in the OMG sense. Each system environment shall establish a format for recognizing the identity of subscribers and other End User Clients. This is used on all RPC messages, as well, for identifying the requester. It is also used in DSM-CC for identifying the owner of an object, and would commonly be used for obtaining a Client's permissions to access or perform operations on an object. **ServiceContextList** carries optional application-specific information passed between Client and Service.

5.6.6.1.1 Partial Path

On a ClientSessionSetupConfirm, a valid session can exist even though the entire **pathName** has not been resolved. The partial path must include at least the object reference of the first **Step** in the path, a ServiceGatewayUU. The length of the resolved references specify the portion of the path that was successfully resolved.

5.6.6.2 ServiceGatewayUU

ServiceGatewayUU inherits the Directory interface. **Directory bind()**, **bind_context()**, **rebind()**, **rebind_context()** and **unbind()** require MANAGER privileges to be invoked on the ServiceGatewayUU object.

The Client only sees the local ServiceGateway object at the Application Portability interface. This object translates to the Service Inter-operability ServiceGatewayUU interface, if the U-N Session protocol is used. The Server in this case would implement the ServiceGatewayUU interface.

5.6.6.2.1 Summary of ServiceGatewayUU Primitives

Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Inherited from **Directory and NamingContext**:

operations: **open, close, get, put, list, resolve, bind, bind_context, rebind, rebind_context, unbind, new_context, destroy**

Service Inter-operability Syntax

```
module DSM {
    interface ServiceGatewayUU : 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;
    };
};
```

TapUse

RPC_USE

5.6.6.3 SessionSI

The SessionSI interface may be used in systems where there is no DSM-CC User-to-Network signaling. It provides operations to establish a Session without management of network resources. The Session in this case is a context for active object references and their state, plus it may serve to identify the requesting Principal involved.

The Client only sees the local ServiceGateway object at the Application Portability interface. This object in turn calls the Service Inter-operability ServiceGatewaySI interface, if the RPC Session protocol is used. The Server in this case would implement the ServiceGatewaySI interface.

Service Inter-operability Syntax

```
module DSM {
    interface SessionSI {
        const AccessRole attach_ACR = READER;
        void  attach (
                in InfoRequest downloadInfoReq,   // download info request
                in CosNaming::Name pathName,      // path name to resolve
                in UserContext savedContext,      // previous application user context
                out InfoResponse downloadInfoResp,      // download info response
                out ObjRefs resolvedRefs)         // objects resolved
                raises (NO_AUTH, UNK_USER, OPEN_LIMIT, SERVICE_XFR,
                        BAD_COMPAT_INFO, NO_RESUME,
                        NOT_FOUND, CANNOT_PROCEED, INV_NAME);
        const AccessRole detach_ACR = READER;
        void  detach (
            in boolean aSuspend,
            out UserContext savedContext) // suspended user context
            raises (NO_SUSPEND);
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

SessionSI attach() is invoked against a local SessionSI object, which acts as a logical path root to remote Services. The first node in the pathName is that of a ServiceGateway. The SessionSI object shall resolve the ServiceGateway, and if the path extends beyond the ServiceGateway, shall propagate the resolve.

A Client invokes **SessionSI attach()** to open a new Session over the network using an RPC. **downloadInfoRequest** and **DownloadInfoResponse** contain Download negotiation parameters as specified in the DownloadSI interface. An identification of a previously suspended user context **savedContext** 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. A path specification **pathName** names the path to ServiceGateway and possibly a first Service to be opened. A default Service may be specified in the second **Step** of **pathName** as a NULL string **NameComponent.id**. A default Service **NameComponent.id** is an indication to the ServiceGateway that it can choose the first Service for the Client. **resolvedRefs** shall contain references for the resolved object references.

A Client invokes **SessionSI detach()** to close all object references for a Session using an RPC. If **aSuspend** is true, the ServiceGateway will inform Services which are maintaining user context for this End User to return user context state for a possible resumption of those Services. It is up to the application to determine which state is to be returned and maintained between Sessions. The Client may later invoke **attach()** with this **UserContext** in order to resume from the saved state.

5.6.6.4 ServiceGatewaySI

ServiceGatewaySI inherits the Directory and SessionSI interfaces. **Directory bind()**, **bind_context()**, **rebind()**, **rebind_context()** and **unbind()** require MANAGER privileges to be invoked on the ServiceGatewaySI object.

5.6.6.4.1 Summary of ServiceGatewaySI Primitives

Inherited from **Access**:
attributes: **Size, Hist, Lock, Perms**

Inherited from **Directory and NamingContext**:

operations: **open, close, get, put, list, resolve, bind, bind_context, rebind, rebind_context, unbind, new_context, destroy**

Inherited from **SessionSI**:
operations: **attach, detach**

Service Inter-operability Syntax

```
module DSM {
    interface ServiceGatewaySI : Directory, SessionSI {
        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;
    };
};
```

TapUse

RPC_USE

5.6.6.5 DownloadSI

When an RPC stack is present in the Client, the DownloadSI interface can be used to perform lower layer U-N Download requests. This interface implements model 2 of U-N Download in clause 7 (where ControlUp, DataUp and ControlDown are carried on an RPC bi-directional channel). All control messages including DataRequest are exchanged over this interface. The DownloadDataBlocks are sent over a high-bandwidth downstream channel, e.g., one which conveys an MPEG-2 stream. Thus, taps are returned for the RPC and for the data downstream channels. While the Download Application Portability Interface is simplified by not exposing network flow-control to the application, the DownloadSI Service Inter-operability Interface at the lower layer will perform the details of windowing and flow control as described in clause 7. The RPC interface provides a reliable Download plus invocation authorization on each message, where higher security is desired.

The pre-conditions for this interface are that a Client can send messages to an object reference for an application Service that supports this interface. The first operation is a **DownloadSI info()**. The reply to this RPC is a sequence of DownloadInfo structures that contain size, private opaque data, and description of each module that must be downloaded in order for the Client to run the application. Following this, a series of **DownloadSI proceed()** messages to be sent to control the Download transfer. The request message body of **DownloadSI proceed()** is the DownloadDataRequest as described in clause 7. A **DownloadSI cancel()** can be sent to cancel a download in progress.

The Client can recognize an object that supports the Download interface through a **Directory list()** operation at the parent directory where the object's name and object reference are bound. NameComponent kind in the **Directory list()** reply identifies whether the object is a Download Service. Following a resolve of an object reference, **Kind has_a()** may be invoked to test whether Download is supported as an inherited interface.

The U-N Download data encoding of clause 7 also supports broadcast download in various forms. This does not require RPC control as described in this subclause.

CDR-Lite encoding will convert the DownloadSI programming interface to exact data encoding specified in clauses 6 and 7.

Service Inter-operability Syntax: InfoRequest and InfoResponse

```
module DSM {
    struct SubDescriptor {
        octet subDescriptorType;
        octet subDescriptorLength;
        sequence<octet, 255> additionalInformation;
    };
    struct InterfaceDescriptor {
        octet descriptorType;
        octet descriptorLength;
        u_long specifier;
        u_short model;
        u_short version;
        sequence<SubDescriptor, 255> subDescriptorList;
    };
    struct CompatibilityDescriptor {
        u_short length;
        sequence<InterfaceDescriptor, 65535> interfaceDescriptorList;
    };
    struct InfoRequest {
        u_long bufferSize;
        u_short maximumBlockSize;
        CompatibilityDescriptor userCompatibilitiesBytes;
        sequence<octet, 65535> privateDataBytes;
    };
    struct ModuleInfo {
        u_short moduleId;
        octet moduleVersion;
        u_long moduleSize;
        sequence<octet, 255>  moduleInfoBytes;
    };
    typedef sequence<ModuleInfo, 65535> ModuleInfoList;
    //
    // The complete InfoResponse contains Server calculated
    // transport parameters plus application level information
    //
    struct InfoResponse {
        u_long downloadId;
        u_short blockSize;
        octet windowSize;
        octet ackPeriod;
        u_long tCDownloadWindow;
        u_long tCDownloadScenario;
        CompatibilityDescriptor userCompatibilitiesBytes;
        ModuleInfoList modulesInfo;
        sequence<octet, 65535> privateDataBytes;
    };
};
```

Service Inter-operability Syntax: Operations

```
module DSM {
    interface DownloadSI {
#ifdef DSM_CONSUMER
        // Download DataRequest is sent to start the Download or
        // to cause the Server to send the next group of Download DataBlocks
        struct DataRequest {
                u_short moduleId;
                u_short blockNumber;
                octet downloadReason;
        };
        // when the Server receives a Download DataRequest, it
        // will send a series of ackPeriod Download DataBlocks:
        struct CancelRequest {
                u_short moduleId;
                u_short blockNumber;
                octet downloadCancelReason;
                char privateDataLen;
                sequence<octet, 65535> privateDataBytes;
        };
        const AccessRole info_ACR = READER;
        void info (in InfoRequest reqNegotiation,
                    out InfoResponse respNegotiation)
                raises (BAD_COMPAT_INFO, BUF_SIZE, BLOCK_SIZE);
        const AccessRole proceed_ACR = READER;
        void proceed (in DataRequest ackNack)
                raises(BAD_MODULE_ID, TIMEOUT, MPEG_DELIVERY);
        const AccessRole cancel_ACR = READER;
        void  cancel (in CancelRequest cancelReq)
                raises (BAD_MODULE_ID, TIMEOUT) ;
#endif
#ifdef DSM_GENERAL
        struct ModuleInstallInfo {
                u_short aModuleId;
                // path to object containing Download data
                CosNaming::Name n;
        };
        typedef sequence<ModuleInstallInfo> ModuleInstallList;
          };
        const AccessRole install_ACR = OWNER;
        void install (in CompatibilityDescriptor compatInfo,
                    in ModuleInfoList modulesInfo,
                    in ModuleInstallList pathsInfo)
                raises (BAD_COMPAT_INFO, BAD_MODULE_INFO,
                            INV_NAME, NOT_FOUND);
        const AccessRole deinstall_ACR = OWNER;
        void deinstall (in CompatibilityDescriptor compatInfo)
                raises (BAD_COMPAT_INFO);
#endif
    };
};
```

TapUse

RPC_USE	operation requests
DOWNLOAD_DATA_DOWN_USE	Download DataBlocks

Service Inter-operability Semantics

DownloadSI info() is used to obtain download module information and negotiate flow-control parameters for a Download. **DownloadSI info()** enables a Client to obtain information about modules that must be downloaded in order for an application to proceed. It is also used to negotiate flow-control parameters based on buffer size, block size and network reliability. For efficiency reasons, it is recommended that the first InterfaceDescriptor in the CompatibilityDescriptor identify the dominant Client hardware specifier, model and version. The **InfoRequest** and **InfoResponse** fields are described in clause 7.

DownloadSI proceed() is used iteratively to transfer a series of DownloadDataBlocks to the Client. The DownloadDataBlocks are normally transfered over the channel identified by the DOWNLOAD_DATA_DOWN_USE Tap. The **DataRequest** fields are described in clause 7.

DownloadSI cancel() is used to cancel a Download in progress. The **CancelRequest** fields are described in clause 7.

Informative: A new interface type that inherits both File and Download is possible. In this case, the U-N download protocol described in clause 7 can be used for large transfers requested by **File read()**. To accomplish this, the **InfoRequest** is carried in the ServiceContextList of the associated operation. The **InfoResponse** and **ConnBinder** are carried in the ServiceContextList of the RPC reply. The data to be transferred is treated as a single module to be downloaded. The negotiation of flow-control parameters is the same as the U-N Download protocol. Since the Client is making the selection, there is no compatibility matching. This mechanism may also be used for other objects that have operations that potentially require large data transfers. The Download interface must be supported by the target object.

DownloadSI install() enables an owner to bind download configuration to the Server. The configuration consists of the compatibility information that describes the Client configuration, module information for the modules to be downloaded, and the path name to download module data for each module. The **CompatibilityDescriptor** fields are defined in clause 6.For efficiency purposes, it is recommended that the first InterfaceDescriptor in the CompatibilityDescriptor identify the dominant Client hardware specifier, model and version. The ModuleInfoList contains information for each module to be downloaded. ModuleInfo fields are defined in clause 7. The ModuleInstallList contains the pathNames to objects at the Server that contain the download module data.

DSM DownloadSI deinstall() enables an OWNER to unbind a Download configuration from the Server. The Client CompatibilityDescriptor identifies the download configuration to be removed.

5.6.6.6 Event

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface, except for **Event notify()**, which is a local operation. Refer to the Event API for individual operation descriptions. Note that **Event notify()** is implemented as a local library operation at the Client. To the IDL Compiler, defining DSM_SERVER will cause **Event notify()** to not be compiled for the Server Stub.

Application Portability / Service Inter-operability Syntax

```
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_ID = 0;
        typedef sequence<char, 255> eventName;
        typedef  sequence<eventName, 65535> EventList_T;
        const AccessRole EventList_get_ACR = READER;
        const AccessRole EventList_put_ACR = OWNER;
        attribute EventList_T EventList;
        // the following struct is sent in the MPEG stream
        //
        struct StreamEvent {
                  u_short eventId;
                  AppNPT rAppTime;
                  sequence<octet>  rPrivateData;
        };
        const AccessRole subscribe_ACR = READER;
        void subscribe(
                  in string aEventName,
                  out u_short eventId)
                  raises(INV_EVENT_NAME);
        const AccessRole unsubscribe_ACR = READER;
        void unsubscribe(
                  in u_short eventId)
                  raises(INV_EVENT_ID);
#ifdef DSM_PSEUDO
        void notify (out StreamEvent rStreamEvent);
#endif // DSM_PSEUDO
    };
};
```

Possible TapUses

RPC_USE	operation requests
STREAM_EVENT_USE	Stream Event Descriptors
STR_STATUS_AND_EVENT_USE	Stream Mode and Event Descriptors

Service Inter-operability Semantics

The local Event object shall monitor the MPEG-2 Stream for Stream Event descriptors that match the eventId for subscribed events. It shall hold event information for each eventId until such time an application level **Event notify**() is invoked or the object is closed.

5.6.6.7 Composite

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the Composite API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    typedef u_long BindingType; // extends CosNaming BindingType
    const BindingType cobject = 2;
    interface Composite {
        struct ChildInfo {
            CosNaming::NameComponent n;
            Version rVersion;
            boolean required;
        };
        typedef sequence<ChildInfo> ChildInfos;
        //
        const AccessRole list_subs_ACR = READER;
        void list_subs (
             in CosNaming::Name name,
             out ChildInfos infos)
            raises (NOT_FOUND, INV_NAME);
#ifdef DSM_GENERAL
        struct ChildBinding{
            CosNaming::NameComponent n;
            Version rVersion;
            boolean required;
            ObjRef obj;
        };
        typedef sequence<ChildBinding>  ChildBindings;
        const AccessRole bind_subs_ACR = WRITER;
        void bind_subs (
            in CosNaming::Name name,
            in ChildBindings rChildBindings)
            raises (NOT_FOUND, INV_NAME, ALREADY_BOUND);
        const AccessRole unbind_subs_ACR = WRITER;
        void unbind_subs (in CosNaming::Name name)
            raises (NOT_FOUND, INV_NAME);
#endif
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

Directory open() with **aPathType** = DEPTH must be used to resolve a Composite object. The ObjRefs returned are the sequence of object references whose corresponding **aPathSpec Step Process** flags areTRUE (the last of which being the Composite Parent object reference), followed by the Child object references.

While **Directory open()** at the Application Portability Interface returns the parent Composite to the Client Application, the corresponding **Directory open()** at the Service Interoperability Interface returns the object references of the Composite parent and the required child objects over the network. The child objects are associated as being compatible versions, such that when the **Directory open()** is invoked, the set of required compatible sub-objects are resolved.

Opening a required child is a local operation, because the object reference is present from the previous **Directory open()**. Opening an optional child causes an object reference for the child to be returned over the network.

With **Composite bind_subs()**, subset of child objects are labeled as required, i.e., they should be opened together by a **Directory open()** operation. Another subset are labeled optional, i.e., they may be opened individually at any time during the running of the application. The optional sub-objects are included with the composite set for version compatibility reasons.

Each child object reference shall have a NameId in its IOR to provide the child's symbolic name.

5.6.6.8 View

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the View API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax: View Styles

```
module DSM
    interface View{
        // View Style 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 this part of ISO/IEC 13818
        // 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';
        const AccessRole Style_get_ACR = READER;
        readonly attribute char Style;
    };
};
```

Application Portability / Service Inter-operability Syntax: Statement and Result

```
module DSM {
    interface View {
        typedef string SQLStatement;
        //
        typedef u_short FieldCode;
        struct FieldDescribe {
                string fieldName;               // name of the field
                FieldCode aType;                // type of the field
                opaque typeParameters;          // parameters related to the given type
        };
        typedef sequence<FieldDescribe> ResultDescribe;
        //
        // FieldCodes for standard SQL types
        const FieldCode VTC_CHAR =                  1;
        const FieldCode VTC_SMALLINT =              2;
        const FieldCode VTC_INTEGER =               3;
        const FieldCode VTC_FLOAT =                 4;
        const FieldCode VTC_SMALLFLOAT =            5;
        const FieldCode VTC_DECIMAL =               6;
        const FieldCode VTC_REAL =                  7;
        const FieldCode VTC_DOUBLEPRECISION =       8;
        const FieldCode VTC_CORBA_TYPECODE =        9;
        //
        // these structs are placed into the opaque typeParameters field of
        // the FieldDescribe struct depending on the value of the FieldCode
        struct InfoChar {
                u_short length;
                boolean nullTerminated;
        };
        struct InfoFloat {
                u_short precision;
        };
        struct InfoDecimal {
                u_short precision;
                u_short scale;
        };
        //
        // type definitions for the returned data
        typedef opaque Field;
        typedef sequence<Field> Row;
        typedef sequence<Row> Result;
    };
};
```

Application Portability / Service Inter-operability Syntax: Operations

```
module DSM {
    interface View {
        // query() executes an SQL query (select) in the server and
        // returns the first set of rows
        const AccessRole query_ACR = READER;
        void query(
                    in SQLStatement aSQLStatement,
                    in u_short maxRows,
                    out ResultDescribe describe,
                    out Result aResult,
                    out View iterator)
                    raises (ILLEGAL_SYNTAX);
        // read() returns additional rows from the query
        const AccessRole read_ACR = READER;
        void read(
                    in u_short aCursor,
                    in u_short maxRows,
                    out Result aResult)
                    raises(NO_QUERY, INV_CURSOR);
        // execute() executes an SQL statement in the server (for example "update")
        const AccessRole execute_ACR = WRITER;
        void execute(
                    in SQLStatement aSQLStatement)
                    raises(ILLEGAL_SYNTAX);
    };
};
```

TapUse

RPC_USE

5.6.6.9 State

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the State API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
    interface State {
        const AccessRole suspend_ACR = READER;
        void suspend(
                    in boolean        aRelease,
                    out UserContext   savedContext)
                    raises (NO_SUSPEND);
        const AccessRole resume_ACR = READER;
        void  resume (
                    in UserContext    savedContext,
                    out ObjRefs       restoredRefs)
                    raises (NO_RESUME);
    };
};
```

TapUse

RPC_USE

Service Inter-operability Semantics

In addition to suspending application state, **State suspend()** provides the **aRelease** flag for management of network resources. If **aRelease** is true then the object instance forces network resources to be released immediately. When **aRelease** is not set to true the object instance preserves the association tags but the resources might be reassigned or released after a local time-out. In either case, if this object is a parent Composite with open child objects, **State suspend()** cascades to the child objects.

A **ConnBinder** must be present in the object reference (IOR) of the **State resume()** RPC reply.

5.6.6.10 Interfaces

The Application Portability Interface maps 1-1 to the Service Inter-operability Interface. Refer to the Interfaces API for individual operation descriptions.

Application Portability / Service Inter-operability Syntax

```
module DSM {
#ifdef DSM_GENERAL
    // A TypeCodeBuf holds a Corba 2.0 TypeCode
    typedef opaque TypeCodeBuf;
    typedef sequence<TypeCodeBuf>  TypeCodeList;
    //
    interface Interfaces {
        typedef opaque ReferenceData;
        typedef string InterfaceDef;
        const AccessRole show_ACR = READER;
        void show (
                in string aStrKind,         // DSM-CC format, "Module::Interface Type"
                in IFKind anIFKind,         // IFKind
                out string rIDL,            // IDL used in previous define
                out IntfCode rIntf,         // DSM-CC interface code with included IFKinds
                out TypeCodeList  rTypes)// TypeCodes required at this level of interface
                raises (NOT_DEFINED, INV_KIND);
        const AccessRole define_ACR = MANAGER;
        void define (
                in ReferenceData id,        // unique Identifier
                in InterfaceDef  rIDL,      // IDL definition
                out IntfCode rIntf,         // DSM-CC interface code with included IFKinds
                out TypeCodeList  rTypes)// TypeCodes required at this level
                raises (PREV_DEFINED, ILLEGAL_SYNTAX, NO_REF_TYPE);
        const AccessRole check_ACR = MANAGER;
        void check (in IFKind  anIFKind)
                raises (NO_REF_TYPE);
        const AccessRole undefine_ACR = MANAGER;
        void undefine (in ReferenceData id, out IFKindList usedBy);
    };
#endif
};
```

TapUse

RPC_USE

5.7 Application Boot Process

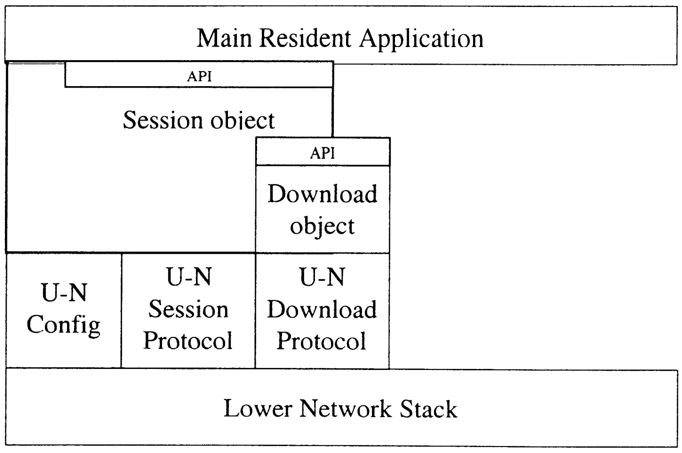
This subclause describes the process that establishes a Session, performs Download, and starts a Client application. The process consists of a series of steps, each with pre and post-conditions. It is possible that certain Client configurations will satisfy the post-conditions for a step through previously resident functional elements. Thus, if post-conditions for a given step are already satisfied, that step need not be performed.

The Application boot process described here is informative. The process conforms to the U-N Session messages defined in clause 4, the U-N Download messages defined in clause 7, and the User-to-User operations defined in this clause. The operations and their supporting IDL definitions are normative. The pre and post-conditions are normative.

The Application Boot Process encompasses the following lifecycle phases:

1. **Session attach()** is invoked by the Main Resident Application on a local Session object, in order to establish a Session, set up connections to initial remote Service objects, download the functional elements needed to run the application, and decode the parameters that will be used by application.
2. Following **Session attach()**, the Client application is launched.
3. The newly started application invokes **First service()** to obtain the object reference for its first remote Service. The application may also invoke **First root()** to obtain the object reference for the ServiceGateway. During the Session, it may resolve new object references, and invoke object operations on local Object Implementations and active object references.
4. When finished, **Session detach()** is invoked by the Client Application on the object reference of the ServiceGateway, in order to teardown the Session and disconnect from remote Service objects.

5.7.1 Session attach() Pre-conditions



The following pre-conditions must be satisfied in order to call **Session attach()** for the U-N Case.

- Lower Network Stack is present. It includes the Physical, Data link, Network and Transport layers of the OSI Model.
- Session input parameters are present.
    1. ClientId. ClientId is carried in the U-N Session protocol to globally identify the Client device. ClientId is a 20 octet value.
    2. Service location information. These include ServerId, optional path name, and optional user context. The 20 octet ServerId globally identifies the Server-side device that will negotiate the U-N Session messages. There is a 1-1 relationship between ServerId and ServiceGateway, i.e., there may only be one ServerId for a given ServiceGateway instance. The path name is the complete logical path from a ServiceGateway to a Service. The user context is an opaque value that holds application initial parameters
    3. CompatibilityDescriptor. The CompatibilityDescriptor contains configuration information to clearly distinguish the Client profile, e.g., system hardware descriptor, system software descriptor, and descriptor(s) identifying the network protocol(s) the Client supports. This descriptor can identify the

protocol to be used for Download. In the absence of such identification, the Download message set of clause 7 shall be used as the default.

4. Transaction timeout values. Timeouts can occur on the U-N Session and Download message sequences, resulting in a cancellation of the transaction. The Timeouts are specified in the U-N Session and U-N Download, clauses 4 and 7, of this part of ISO/IEC 13818.

- Other parameters are also optionally present: PrincipalId to identify the End-User to a Service Domain (there may be more than one PrincipalId for more than one Service Domain). Service Context information such as preferences. DSM-CC does not define how these are obtained. They may be included in the Server Binding. Additionally, a display name may be associated with the path name. The display name is the End-User's alias for an application or Service name. It is recommended that the display name be formated as a UNIcode string, to accommodate alphabets with more than 255 characters.
- U-N Session Protocol is present. The U-N Session protocol includes U-N Session state machine and the ability to send/receive U-N messages.
- Download Protocol is present. The Download protocol includes the encoded U-N Download message set. The Download message set is replaceable by other protocol, such as broadcast Download or RPC. A Download InfoRequest message including CompatibilityDescriptor is pre-encoded.
- Session Local Object Implementation is present. The Session object provides attach() and detach() operations to the upper programming layer, the Session state machine, and U-N Session encoded protocol to the lower layer. The Session object is replaceable by other Session objects that could interface with different lower layer protocols (such as broadcast or RPC).
- Download Local Object Implementation object is present. The Download object provides info(), alloc(), start() and cancel() operations to the upper programming layer, the Download state machine, and Download encoded protocol to the lower layer. The Download object may also interface to other lower layer protocols, such as broadcast Download or RPC-based Download.
- A Main Resident Application is running. This application shall invoke **Session attach()**, and then launch the newly loaded Client application.

5.7.2 Session attach() Procedure

For an interactive Session based on DSM-CC User-to-Network signaling, **Session attach()** performs the following steps:

1. Resolve path-specific parameters to be used in U-N Session Establishment, if necessary.
2. Use **SessionUU attach()** to populate the uuData of Session Establishment messages, and establish the U-N Session. The syntax is defined in the Service Inter-operability Interfaces subclause.
3. Perform a Download sequence to download DSM-CC Library stubs, application Client executable and other functional elements needed to run the application.
4. Decode Downloaded data to produce progamming level structs and parameters, and bind ObjectReferences with network resources.

5.7.2.1 Resolving Path-specific Parameters

Path-specific parameters may be known in advance by the calling application, or may be stored in a local binding list in the Session object. This subclause is informative and describes the use of the Server Binding List.

The Server Bindings can be used to hold various path-specific parameters required for U-N Session Establishment. The Session object maintains this state to enable resolution of a ServerId and SavedContext, given a PathName. At a minimum each Server Binding contains fields for ServerId, CosNaming::Name, and SavedContext. A default ServerId can exist with a NULL (length 0) PathName.

Server Binding List:

ServerId	PathName	SavedContext
20 octet	CosNaming::Name	opaque

A default Server Binding with a ServerId and NULL PathName is used if the caller of Session attach() does not have a name for a remote Service.

A display name, if present in the ServerBinding, can be used as an End-User alias for the Server Binding. In this case, the Session attach() may accept either the display name alias of the Service (in the NameComponent id) or the full PathName to the Service.

PrincipalId (End-User identification) and ServiceContextList parameters (such as user preferences), may also be present in the Server Binding, or may be global Client parameters common to many Service paths. The default values for these parameters are NULL (zero-length).

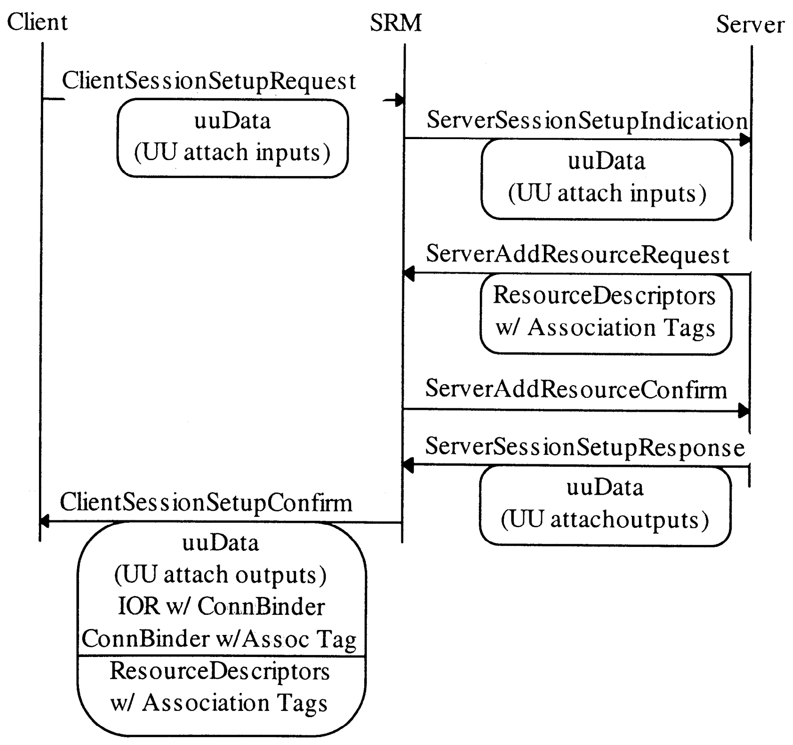
5.7.2.1.1 Post-condition

These values are known:

ServerId, pathName, savedContext, PrincipalId, ServiceContextList, DownloadInfoRequest

5.7.2.2 Establishing the U-N Session

The **uuData** for all U-N Session messages is encoded as big-endian CDR. The U-N Session Establishment sequence with **uuData** is shown below:


Client
SRM
Server
ClientSessionSetupRequest
uuData (UU attach inputs)
ServerSessionSetupIndication
uuData (UU attach inputs)
ServerAddResourceRequest
ResourceDescriptors w/ Association Tags
ServerAddResourceConfirm
ServerSessionSetupResponse
uuData (UU attachoutputs)
ClientSessionSetupConfirm
uuData (UU attach outputs) IOR w/ ConnBinder ConnBinder w/Assoc Tag
ResourceDescriptors w/ Association Tags

Note that _long in following CDR big-endian data encoding diagrams represents the count of elements in a sequence, according to CORBA terminology.

5.7.2.2.1 ClientSessionSetupRequest

The marshaled input parameters of the **SessionUU attach()** are the **uuData** of ClientSessionSetupRequest, as described below. The resulting **uuData** field of **ClientSessionSetupRequest** is as follows.

- A U-N Download InfoRequest structure.

- A CosNaming::Name identifying a path to the desired Service. The Name extends logically from the remote ServiceGateway. The terminal leaf shall contain a name identifier that identifies the desired initial Service. This is placed in the **pathName** parameter used in ClientSessionSetupRequest.
- A UserContext value, for resumption of previously suspended application state. This is placed in the input **savedContext** parameter.
- An End-User identification of the consumer, unique within the context of the ServiceGateway Service Domain. This is placed in the **aPrincipal** parameter.
- Optional ServiceContextList information, placed in the **inSC** parameter. The ServiceContextList defined later in this subclause.

The Network shall take the **uuData** of **ClientSessionSetupRequest** and place it in the **uuData** of **ServerSessionSetupIndication**, as part of the Session establishment sequence.

U-U Data Client Session Setup Request			opaque downloadInfoReq
	CosNaming::Name pathName sequence<>		u_long _length
		CosNaming:: NameComponent	string id string kind
		<other NameComponents>	
			UserContext savedContext
			Principal aPrincipal
			IOP::ServiceContextList inSC

The Server may use the Client CompatibilityDescriptor, Service Name, PrincipalId and ServiceContextList to determine parameters for negotiating session resources with the SRM.

5.7.2.2.2 ClientSessionSetupConfirm

Near the completion of the User-to-Network session establishment, the Server shall return **ServerSessionSetupConfirm** to the network, with **uuData** containing the marshaled output parameters of **SessionUU attach()**. This **uuData** is forwarded to the Client in the **ClientSessionSetupResponse**. The output parameters of **SessionUU attach()** are one of two choices:

1. If the U-N **response** field indicates the session is established:

- U-N Download InfoResponse, placed in the **downloadInfoResp** parameter.
- Connection information for Download data and control channels, placed in the **downloadTaps** parameter.
- Resolved Interoperable Object References(IOR) for the ServiceGatewayUU and optionally for the first Service, as were identified in **pathName**. These are placed in **resolvedRefs** parameter.
- Optional ServiceContextList information, placed in the **outSC** parameter.

U-U Data Client Session Setup Response U-N response field indicates a valid session		opaque downloadInfoResp
		ConnBinder downloadTaps
	ObjRefs resolvedRefs sequence<IOP::IOR>	u_long _length
		IOP::IOR string id u_long ProfileId opaque encapsulation
		<other IOP::IORs>
		IOP::ServiceContextList outSC

2. If the U-N **response** field indicates the session is NOT established, uuData shall contain:

- Exception. This shall contain exception type and value.

U-U Data Client Session Setup Response U-N response field indicates an invalid session	any exception	DSM Typecode	string kind string repositoryId u_long TCKind
			opaque value

5.7.2.2.3 Session Establishment Post-conditions

A table of tag/network resource associations contains the Resource Association Tags and Resource Descriptors that were returned in the **ClientSessionSetupResponse**.

The **uuData** is not decoded. **DownloadInfoResponse** and **downloadTaps** are present in CDR big-endian format.The encoding of other **uuData** parameters is not yet known.

The undecoded **resolvedRefs** contains an IOR for the ServiceGatewayUU, and may contain IORs of the first Service. If the first Service is a Composite, IORs are present for the parent and child references. If not, the first Service is referenced by a single IOR.

5.7.2.3 Download

The Download sequence returns all functional elements needed to initially run the application, including the Client Application itself, DSM-CC Library Stubs for objects the application will reference, and the RPC Protocol, if it is not already present.

As part of the Application Boot Process, the execution of U-N Download does not require the RPC to be present. Following U-N Download, the RPC must be present.

downloadTaps from **uuData** of **ClientSessionSetupResponse** provide the connection information for Download data channels. This ConnBinder shall be encoded as CDR, big-endian.

DSM::ConnBinder		u_long _length
	Tap	u_short id u_short use u_short assocTag sequence<octet, 255> selector
	<other Taps>	

The Download Protocol receives **DownloadInfoResponse** from **uuData** of **ClientSessionSetupResponse.** The Download Client state machine of clause 7 is DCActive.

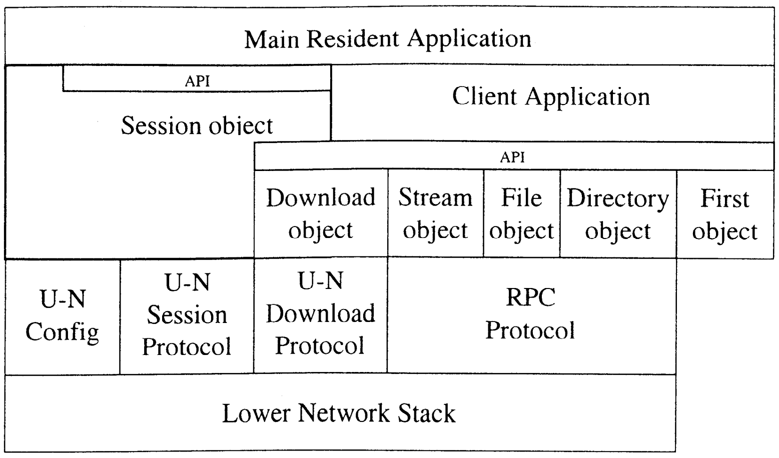
Download start() is invoked by the **Session attach()** procedure to send a series of DataRequest messages to retrieve all required DownloadDataBlocks, as described in clause 7, U-N Download.

The Download process retrieves byte sequences that must be decoded in order for Client Application to access object structures at a programming level. Following Download, the encoder/decoder must be present, either in the RPC or in the DSM-CC Library stubs.

The IORs (**resolvedRefs**) from **SessionUU attach()** are now decoded to determine to the Protocol ProfileId (and hence the encoding/decoding specification) for each Client/object binding. The IOR contains a string identifier for the object type, a length indicating a number of TaggedProfiles, and an array of TaggedProfiles (DSM-CC typically uses 1 TaggedProfile). The string identifier shall be either the NameComponent kind format ("<Module>::<Interface>") or an alias (see the Entity Identification subclause). The Tagged Profile contains a ProfileId and a CORBA 2.0 encapsulation. The profileId specifies the RPC, the data encoding, and the structure of the encapsulation. The top format of the IOR shall be be big-endian CDR. The encapsulation encoding and all other encodings for the object shall be determined by the ProfileId. The IOR encoding is shown below:

IOP::IOR	string id		u_long length char ... /0
	sequence<TaggedProfile>		u_long length
		TaggedProfile	u_long profileId
			encapsulation
		<other TaggedProfile>	

The Downloaded data encodings are now converted to programming level constructs. Download returns, **Session attach()** returns, and the Main Resident Application launches the new Client Application.



5.7.3 Session Tear-down

1. The marshaled input parameters **SessionUU detach()** specify the **uuData** field of **ClientReleaseRequest**. This user data is then forwarded by the network to the **Server** in **ServerReleaseRequest**. The input parameters in **SessionUU detach()** are:

   - A boolean suspend indication which indicates to the Server that application state should be preserved for later resumption. This is placed in the **aSuspend** parameter.
   - Optional ServiceContextList information, placed in the **inSC** parameter.

2. The output parameters of **ServiceGateway detachUU()** are marshaled together and placed in the **uuData** field of ServerReleaseConfirm. This user data is then forwarded by the network to the Client in **ClientReleaseConfirm**. The output parameters of **ServiceGateway detachUU()** are:

   - A UserContext value containing state for later resumption of this session, placed in the **savedContext** parameter. A value of zero will indicate an exception occurred in the process of suspending the session state, and that state from this session is no longer valid.
   - Optional ServiceContextList information, placed in the **outSC** parameter.

5.7.4 Session Transfer Implications

While the User-to-User Interface does not preclude the use of the User-to-Network Session Transfer (clause 4), a transfer to a new Service Domain implies that object references must be updated in order for the Client to communicate with the new Service. No mechanism is described in this part of ISO/IEC 13818 for notifying the Client of the new object references, nor for notifying that a transfer is about to take place. In addition, no provision has been made in this part of ISO/IEC 13818 for the Client to download an application from the new Service Domain.

In order to meet the requirement for transferring a Client between Service Domains, a User-to-User Service Transfer is defined in subclause 5.5.1.6.3, DSM Session attach.

Refer to Informative Annex L, Service Transfer Message Flows, for further information.

6. User Compatibility

In order to download data or software to a given user, certain information concerning that user may need to be transferred as part of the request to insure that appropriate data be delivered. Compatibility descriptors are used to transfer this information.

6.1 Compatibility Descriptors

Compatibility descriptors may be used to convey an inventory of available hardware or software on a Client to a Server. From this information, the Server may make decisions as to the appropriate data to download to the Client. Compatibility descriptors may also be used by the Server to inform a class of Clients which information to download.

The format of the compatibility descriptor enables organizations to define various subDescriptor's to describe the details of hardware and software modules. Possible subDescriptor's may include processor, memory, operating system, or network protocol stack.

Compatibility descriptors shall have the format shown in Table 6-1.

Table 6-1 Compatibility Descriptor Format

Syntax	Num. of Bytes
compatibilityDescriptor() {	
compatibilityDescriptorLength	2
descriptorCount	2
for (i=0; I < descriptorCount; i++) {	
descriptorType	1
descriptorLength	1
specifierType	1
specifierData	3
model	2
version	2
subDescriptorCount	1
for (j = 0; j < subDescriptorCount; j++) {	
subDescriptor()	
}	
}	
}	
subDescriptor() {	
subDescriptorType	1
subDescriptorLength	1
for (k=0; k<subDescriptorLength; k++) {	
additionalInformation	1
}	
}	

The **compatibilityDescriptorLength** is a two byte field that defines the total length of the descriptors that follow including the descriptorCount but not including the compatibilityDescriptorLength itself.

The **descriptorCount** indicates the number of descriptors which follow the descriptorCount field.

The **descriptorType** is a one byte field that is used to distinguish the type of the hardware or software that is being referenced by this descriptor. Allowable values for the descriptorType are shown in Table 6-2.

Table 6-2 descriptorType field values

descriptorType	Description
0x00	Pad descriptor.
0x01	System Hardware descriptor.
0x02	System Software descriptor.
0x03 - 0x3F	ISO/IEC 13818-6 reserved.
0x40 - 0xFF	User Defined.

The Pad descriptor may be used to provide alignment for any data which follows.

The System Hardware descriptor is used to identify the specifier, model, and version of the manufacturer of the user device.

The System Software descriptor is used to identify the specifier, model, and version of the manufacturer of the system software of the user device.

The **descriptorLength** field is a one byte field that is the total length of the descriptor, not including the descriptorType and descriptorLength fields.

The specifier is a globally unique identifier for an organization that is responsible for defining the semantics of the model and version fields, and any subDescriptors within the encapsulating descriptor. The specifier consists of the specifierType field and the specifierData field.

The **specifierType** is a one byte field that is used to distinguish the format of the specifierData field. The definition of specifierType values are shown in Table 6-3.

Table 6-3 specifierType field values

specifierType	Description
0x00	ISO/IEC 13818-6 reserved.
0x01	IEEE OUI.
0x02 - 0x7F	ISO/IEC 13818-6 reserved.
0x80 - 0xFF	User Defined

The **specifierData** field is a three byte field to uniquely identify an organization. The value assigned to this field is dependent on the specifierType field.

The **model** field is a two byte field whose semantics are specified by the organization identified by the specifier. The use of this field is intended to distinguish between various models defined by the organization. A model value of all 1's indicates that this descriptor applies to all models.

The **version** field is a two byte field whose semantics are specified by the organization identified by the specifier. The use of this field is intended to distinguish between different versions of a model defined by the organization. A version of all 1's indicates that this descriptor applies to all versions.

The **subDescriptorCount** is a one byte field set to the number of subDescriptors for the descriptor.

The subDescriptor contains additional descriptors whose semantics are specified by the organization identified by the specifier.

The **subDescriptorType** is a one byte field that determines the type of the subDescriptor. The semantics of this field are specified by the organization identified by the specifier.

The **subDescriptorLength** is a one byte field that is the total length of all additionalInformation fields included in the subDescriptor.

The **additionalInformation** fields allow the inclusion of arbitrary data. The syntax and semantics of any additional information are specified by the organization identified by the specifier.

6.1.1 IEEE OUI Specifier

When the specifierType of the IEEE Organization Unique Identifier (OUI) is used, the specifierData shall include a three byte IEEE OUI as described in IEEE-802.1990. ISO/IEC reserved specifierTypes shall also use a three byte identifier. The format of the specifierData is shown in Table 6-4.

Table 6-4 specifierData definition using IEEE OUI

Syntax	Num. of Bytes
specifierData() {	
org	**3**
}	

7. User-to-Network Download

7.1 Overview

The U-N Download protocol is a "lightweight" and fast protocol to download data or software to a Client. To reduce confusion between the entity performing the "server side" of the download and the common use of the term "Server" in this part of ISO/IEC 13818, the entity performing the "server side" of the download shall be called the "Download Server". The download protocol is "lightweight" to enable implementation on Clients with limited memory.

The download protocol supports the following scenarios:

1. Flow-controlled download. This scenario embodies the downloading of a complete set of data from one Download Server to one Client in a flow-controlled way. The Client controls the transfer of the data via a control channel to the Download Server.
2. Data carousel. This scenario embodies the cyclic transmission of data by the Download Server. Clients will typically only acquire a subset of the transmitted data, depending upon the application. In this scenario, the Download Server may serve multiple Clients simultaneously.
3. Non-flow-controlled download. This scenario embodies the downloading of a complete set of data in a non-flow-controlled way. The Download Server can use this scenario to download data to several Clients simultaneously because the transfer of data is not controlled by the Client. Instead, the transfer is based on mutual agreements about the transfer parameters.

The three download scenarios all share the same message set, although not all messages are used in each scenario. The message set is divided into two categories: control messages and data messages. The control messages are request-response messages similar to the DSM-CC U-N messages. The Download Server and Clients may use these messages to exchange information about the download process prior to the actual data transfer. The data messages are used to transport the download data and, for the flow-controlled download, to acknowledge the transmission of the data.

A complete flow-controlled or non-flow-controlled download operation embodies the transfer of 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 modules is to indicate groups of data that need to be loaded into contiguous memory, allowing the Client to fragment the allocated memory chunks by module size, rather than having to allocate a memory chunk of the image size. Blocks carry the actual downloaded data. Each block contains data from only one module.

For the flow-controlled download scenario, and optionally for the non-flow-controlled download scenario, the block size is negotiated to meet requirements for efficiency and effective error detection. In this negotiation, the Client sets the initial limits on the maximum block size. If possible, the Download Server will use that block size. However, the block size may be limited by Download Server constraints (for example, a download image may have been divided previously into blocks that satisfy the most restrictive Client to which the download image will be served) and Network constraints such as bit error rates and maximum transport packet size. The protocol provided will run with very small block sizes, which allows its use over highly error-laden networks as well as networks which only support very small network block sizes.

In the flow-controlled download scenario, the Client and Download Server negotiate a window size for a one way sliding window protocol. The sliding window protocol applies only to data messages and not control message exchanges. The complexity of the sliding window algorithm is restricted to the Download Server side. The size of the window can be negotiated by the Client and the Download Server. When the window is negotiated, the Download Server also selects the number of blocks the Client is expected to receive before sending an acknowledgment. The number of blocks is known as the ack period. The value chosen for the ack period shall be equal to or smaller than the window size. The ack period is employed as a simple way to limit the rate that the Client sends acknowledgments back to the Download Server, and therefore limits the network traffic and protocol stack processing. The window size is only relevant for the flow-controlled scenario.

The data carousel scenario embodies the cyclic transmission of data to Clients. The data transmitted within the data carousel is organized in "modules" which are divided into "blocks". All blocks of all modules within the data carousel are of the same size, except for the last block of each module which may be of a smaller size. Modules are a delineation of logically separate groups of data within the data carousel. The modules of a data carousel are described by control messages. These messages may describe all modules, or a subset of modules, transmitted in the data carousel. Based on the control messages, the Clients may acquire a subset of the modules from the network.

The different scenarios may be executed over either a reliable or unreliable network transport. In the case of a reliable transport, various fields may be unused in some messages.

7.1.1 Download Network Models

The Download scenarios may be implemented over many different network models. Each scenario places certain requirements on the underlying network model. These requirements are best defined by examining the message flows defined by the download protocol.

There are four message flows present in the download protocol: ControlDown, ControlUp, DataDown, DataUp. The ControlDown flow describes download control messages that are sent by the Download Server to the Client. The ControlUp flow describes download control messages that are sent by the Client to the Download Server. The DataDown flow describes download data messages that are sent by the Download Server to the Client. The DataUp flow describes download data messages that are sent by the Client to the Download Server.

For the flow-controlled scenario, communication must be possible from the Client to the Download Server, and from the Download Server to the Client. This scenario requires all four download message flows: ControlDown, ControlUp, DataDown, and DataUp.

For the data carousel scenario, communication must be possible from the Download Server to the Client. This scenario requires two flows: ControlDown and DataDown.

The non-flow-controlled scenario requires communication from the Download Server to the Client, and may utilize communication from the Client to the Download Server. This scenario requires the flows: ControlDown and DataDown, and optionally ControlUp.

For the download protocol to be used on a particular network model, the download message flows must be mapped onto one or more network connections. This mapping is network specific and outside the scope of this part of ISO/IEC 13818; however, some example network models and flow mappings are described in this clause to clarify the mapping process. In these examples, the lines represent network connections with a label indicating the flows mapped over that connection.

The first example network model has four connections. One connection from the Download Server to the Client carries the ControlDown flow, while another connection from the Download Server to the Client carries the DataDown flow. A connection from the Client to the Download Server carries the ControlUp flow, and another connection from the Client to the Download Server carries the DataUp flow.

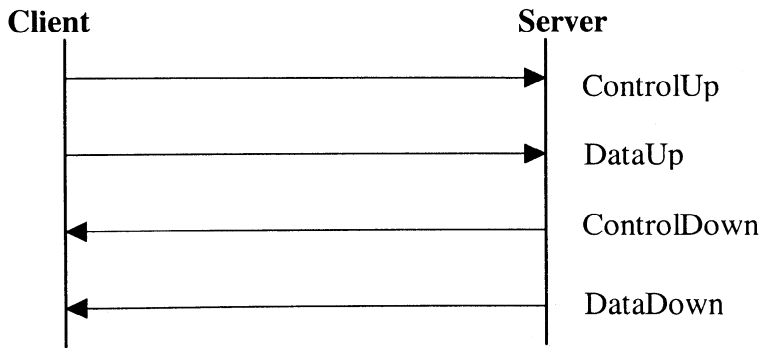


Figure 7-1 Network Model 1

The second example network model is similar to the first model, but the ControlUp and DataUp flows are mapped to a single connection from the Client to the Download Server. For example, this connection may be over a low-bandwidth control channel.

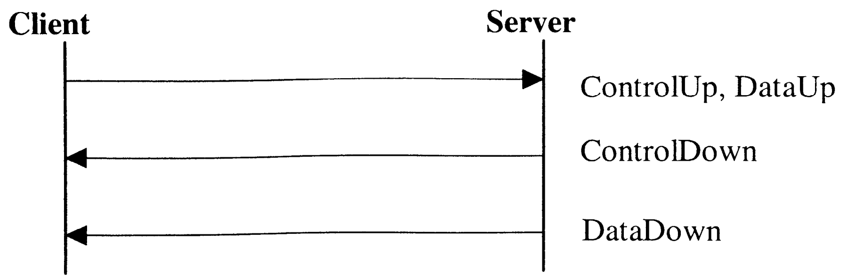


Figure 7-2 Network Model 2

The third example network model has a single connection in each direction. The connection from the Download Server to the Client carries the ControlDown and DataDown flows, while the connection from the Client to the Download Server carries the ControlUp and DataUp flows.

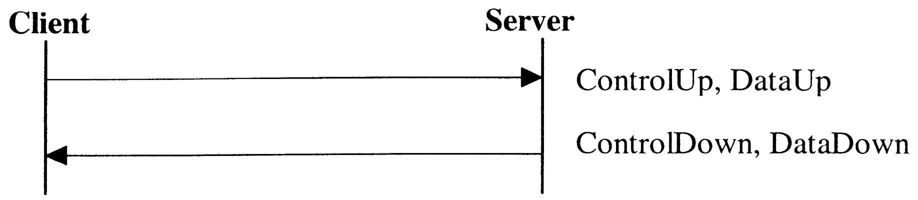


Figure 7-3 Network Model 3

The fourth example network model has a single, typically broadcast, connection from the Download Server to the Client. The connection carries the ControlDown and DataDown flows. Note that since the ControlUp and DataUp flows are not mapped to connections in this network model, the flow-controlled scenario can not be supported. Additionally, the non-flow-controlled scenario can only be supported by this network model if the optional ControlUp flow is not used.

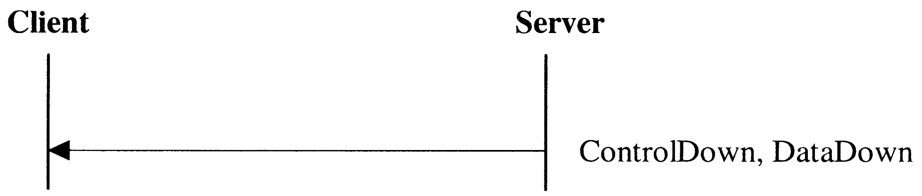


Figure 7-4 Network Model 4

7.1.2 Preconditions and Assumptions

The following preconditions and assumptions shall apply to the download scenarios:

1. The download scenario to be executed shall be known by convention or by some other means, such as data in U-N Configuration Messages.
2. The mapping of download message flows to the underlying network model shall be known by convention or by some other means, such as data in U-N Configuration Messages.
3. The connections to support the download messages flows in the underlying network model shall have been established.
4. For the flow-controlled download scenario and for the non-flow-controlled download scenario when the optional DownloadInfoRequest message is used, the transaction state machine parameters for the download control messages shall be known. These parameters are the time-out value for a control message (tMsg), the maximum number of allowed re-transmissions (retransBound), and also the hold timer for the expiration state (tHold). These values may be obtained through U-N Configuration Messages.
5. For the non-flow-controlled download and the data carousel scenarios, if MPEG-2 Transport Streams are used to deliver the data, then the data delivery rate, specified as a leak rate in the Transport Stream System Target Decoder (T-STD) as defined in ISO/IEC 13818-1 (MPEG-2 Systems), shall be known.

6. If Download is used in combination with U-N session messages, the Client shall know whether or not to place the DownloadInfoRequest message in the uuData field of the ClientSessionSetupRequest message. This may be known by convention or through U-N Configuration Parameters defined in clause 3.

7.2 Download Message Set

The download messages are divided into two categories: download control messages and download data messages. Control messages are typical request-response messages similar to other DSM-CC User-to-Network messages. The control messages are DownloadInfoRequest, DownloadInfoResponse, DownloadInfoIndication, DownloadCancel, and DownloadServerInitiate. Data messages are used for the actual data transfer of modules and the associated acknowledgments. The data messages are DownloadDataBlock and DownloadDataRequest.

7.2.1 Download Control Message Format

Download control messages utilize the DSM-CC message header as defined in clause 2.

Table 7-1 Format of Download Data Messages

Syntax	Num. of Bytes
downloadControlMessage() {	
dsmccMessageHeader()	
controlMessagePayload()	
}	

The dsmccMessageHeader is defined in clause 2.

The controlMessagePayload is defined by the definitions of the download control messages in subclause 7.3.

7.2.2 Download Data Message Format

Download data messages have a download-specific format. Table 7-2 defines the structure of the download data messages.

Table 7-2 Format of Download Data Messages

Syntax	Num. of Bytes
downloadDataMessage() {	
dsmccDownloadDataHeader()	
dataMessagePayload()	
}	

The dsmccDownloadDataHeader is defined in subclause 7.2.2.1.

The dataMessagePayload is one of the download data messages defined in subclause 7.3.

7.2.2.1 DSM-CC Download Data Header

The DSM-CC download data messages begin with the dsmccDownloadDataHeader. This header contains information about the type of message being passed, as well as any adaptation data which may be needed by the transport mechanism, such as conditional access information needed to decode the data. Table 7-3 defines the format of the DSM-CC download data header.

Note that this header uses a format which is compatible with the dsmccMessageHeader defined in clause 2, with the transactionId field replaced by a downloadId. This change is due a difference in the semantics of transactionId versus downloadId. A transactionId correlates a request-response pair of messages, while a downloadId associates an entire set of data messages.

Table 7-3 DSM-CC Download Data Header Format

Syntax	Num. of Bytes
dsmccDownloadDataHeader() {	
protocolDiscriminator	1
dsmccType	1
messageId	2
downloadId	4
reserved	1
adaptationLength	1
messageLength	2
for(adaptationLength>0) {	
dsmccAdaptationHeader()	
}	
}	

The **protocolDiscriminator** field is used to indicate that the message is a MPEG-2 DSM-CC message. The value of this field is defined in clause 2.

The **dsmccType** field is used to indicate the type of the DSM-CC message. Clause 2 defines the possible dsmccType values.

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

The **downloadId** field is used to associate the download data messages and the download control messages of a single instance of a download scenario.

The **reserved** field is reserved by ISO/IEC 13818-6 and shall be set to 0xFF.

The **adaptationLength** field indicates the total length in bytes of the dsmccAdaptationHeader. This length shall be a multiple of four bytes. Any messages that include a non-zero length dsmccAdaptationHeader shall append padding bytes to make the adaptationLength a multiple of four bytes.

The **messageLength** field is used to indicate the total length in bytes following this field. This length includes the dsmccAdaptationHeader indicated in the adaptationLength and the message payload indicated by the messageId field.

The **dsmccAdaptationHeader** is defined in clause 2.

7.3 Message Descriptions

The messageId values for the Download messages are specified as follows:

Table 7-4 DSM-CC Download messageId assignments

Message Name	messageId	Description
DownloadInfoRequest	0x1001	Client requests download parameters
DownloadInfoResponse, DownloadInfoIndication	0x1002	Download Server provides download parameters
DownloadDataBlock	0x1003	Download Server sends one download data block
DownloadDataRequest	0x1004	Client acknowledges downloaded data blocks
DownloadCancel	0x1005	Client or Download Server aborts the download scenario in progress
DownloadServerInitiate	0x1006	Download Server requests Client to initiate a download

In the following subclauses, the message bodies of the download messages are defined. The semantics of the fields of the messages are applicable for all download scenarios unless stated otherwise.

7.3.1 DownloadInfoRequest

The DownloadInfoRequest message shall be sent from the Client to inform the Download Server of the capabilities and limitations of the Client. The Download Server uses the information to select an appropriate download image for the Client. The algorithm for this selection is outside the scope of this part of ISO/IEC 13818. The DownloadInfoRequest message shall be used in the flow-controlled download scenario, and optionally in the non-flow-controlled download scenario. It is not used in the data carousel scenario.

Table 7-5 DownloadInfoRequest Message

Syntax	Num. of Bytes
DownloadInfoRequest() {	
dsmccMessageHeader()	
bufferSize	4
maximumBlockSize	2
compatibilityDescriptor()	
privateDataLength	2
for(i=0;i<privateDataLength;i++) {	
privateDataByte	1
}	
}	

The **bufferSize** field indicates the maximum number of bytes the Client can receive from the Download Server before requiring flow control (an acknowledgment). The Download Server selects a window size no larger than the number of blocks in the buffer size (windowSize <= bufferSize / blockSize). The value of bufferSize shall be equal to or larger than maximumBlockSize (bufferSize >= maximumBlockSize). A value of bufferSize equal to 0 means there is unlimited buffer size available, or equivalently the Client can absorb data at a rate greater than the maximum physical network can deliver.

The **maximumBlockSize** field indicates the maximum block size in number of bytes that the Client agrees to support. The Download Server shall select a blockSize which shall be no larger than this size. A value of 0 means that the Client places no restrictions on the maximum block size.

The **compatibilityDescriptor** structure as defined in clause 6. This information will be used by the Download Server to select the correct download image to send to the Client. The algorithm for this selection is outside the scope of this part of ISO/IEC 13818.

The **privateDataLength** field defines the length in bytes of the following privateDataByte fields.

The data in the **privateDataByte** field is carried from the Client to the Download Server transparently. For example, this field could be used to carry information to fully or partially specify what data to download, or information about which of multiple possible download connections the Client would prefer.

7.3.2 DownloadInfoResponse and DownloadInfoIndication

DownloadInfoResponse and DownloadInfoIndication refer to the same syntactic message definition, with slightly differing semantics. The DownloadInfoResponse message shall be used as a response to the DownloadInfoRequest message, while the DownloadInfoIndication shall be used for the data carousel scenario and non-flow-controlled scenario when no DownloadInfoRequest is sent. In both cases, this message shall be sent from the Download Server to Client to inform the Client of download parameters.

When a DownloadInfoIndication is sent by the Download Server, the transactionId field in the dsmccMessageHeader shall be used as a versioning mechanism. The Download Server shall set the transactionId field to an arbitrary value, and continue to use that value for each transmission of the DownloadInfoIndication, so long as the entire DownloadInfoIndication message remains unchanged. If any field of the DownloadInfoIndication message is modified, then transactionId shall be incremented, modulo the field size of transactionId. The downloadId correlates the DownloadInfoIndication messages with their corresponding download scenario in progress.

Table 7-6 DownloadInfoResponse and DownloadInfoIndication message

Syntax	Num. of Bytes
DownloadInfoResponse(), DownloadInfoIndication() {	
dsmccMessageHeader()	
downloadId	4
blockSize	2
windowSize	1
ackPeriod	1
tCDownloadWindow	4
tCDownloadScenario	4
compatibilityDescriptor()	
numberOfModules	2
for(i=0;i< numberOfModules;i++) {	
moduleId	2
moduleSize	4
moduleVersion	1
moduleInfoLength	1
for(i=0;i< moduleInfoLength;i++) {	
moduleInfoByte	1
}	
}	
privateDataLength	2
for(i=0;i< privateDataLength;i++) {	
privateDataByte	1
}	
}	

The **downloadId** field is the identifier of the download scenario in progress. The downloadId shall be uniquely defined within the Network for data carousel scenario and unique within the connection for the flow-controlled and non-flow-controlled scenarios. This identifier shall be used in all of the subsequent DownloadDataBlock, DownloadDataRequest, and DownloadCancel messages used by the download scenario in progress.

The **blockSize** field is the length in bytes of the data in every block carried in the DownloadDataBlock messages, except for the last block of each module which may be smaller than blockSize. In a DownloadInfoResponse, the value of this field shall be less than or equal to the maximumBlockSize sent in the associated DownloadInfoRequest message.

The **windowSize** is the number of blocks in the sliding window. A value of 0 means that the window is the size of the entire image and that no acknowledgments are to be sent by the Client. A window size of 0 may only be used in a downloadInfoResponse if the Client set the bufferSize to 0 in the DownloadInfoRequest message. This field is unused for non-flow-controlled download and data carousel scenarios and shall be set to 0 in these scenarios.

The **ackPeriod** is the number of blocks the Client would normally be required to receive before sending a positive acknowledgment. The ackPeriod does not limit when a negative acknowledgment can be sent. The Client shall send a positive acknowledgment after successfully storing the last block in the image. This field is unused for non-flow-controlled download and data carousel scenarios and shall be set to 0 in these scenarios.

The **tCDownloadWindow** field indicates the time out period in microseconds for each acknowledgment. This field is unused for non-flow-controlled download and data carousel scenarios and shall be set to 0 in these scenarios.

The **tCDownloadScenario** field indicates the time out period in microseconds for the entire download scenario in progress.

The **compatibilityDescriptor** structure is defined in clause 6. The Download Server may use this structure to indicate for which Clients the described modules are appropriate.

The **numberOfModules** field is the number of modules described in the loop following this field. For flow-controlled and non-flow controlled download scenarios, the loop describes all the modules that have to be downloaded by the Client. For the data carousel scenario, the loop describes a subset of all the modules associated with this data carousel, although it may describes all of them.

The **moduleId** field is an identifier for the module that is described by the moduleSize, moduleVersion, and moduleInfoByte fields. The moduleId is unique within the scope of the downloadId.

The **moduleSize** field is the length in bytes of the described module.

The **moduleVersion** field is the version of the described module.

The **moduleInfoLength** field defines the length in bytes of the moduleInfo field for the described module.

The **moduleInfoByte** information describes the module. In general, the information is implementation-specific. Typical module information may include module type (e.g., non-executable, driver, application) or entry point.

The **privateDataLength** field defines the length in bytes of the following privateDataByte fields.

The data in the **privateDataByte** field is carried from the Client to the Download Server transparently. This field may be used to carry information to enhance the information in the moduleInfo fields, or information about which of multiple possible download connections the Client should use.

7.3.3 DownloadDataBlock

The DownloadDataBlock message shall be sent from the Download Server to the Client. It contains a single data block of a module. The DownloadDataBlock message is used in all download scenarios.

Table 7-7 DownloadDataBlock

Syntax	Num. of Bytes
DownloadDataBlock() {	
dsmccDownloadDataHeader()	
moduleId	2
moduleVersion	1
reserved	1
blockNumber	2
for(i=0;i<N;i++) {	
blockDataByte	1
}	
}	

The **moduleId** field identifies to which module this block belongs.

The **moduleVersion** field identifies the version of the module to which this block belongs.

The **reserved** field is reserved by ISO/IEC 13818-6 and shall be set to 0xFF.

The **blockNumber** field identifies the position of the block within the module. Block number 0 shall be the first block of a module.

The **blockDataByte** conveys the data of the block.

7.3.4 DownloadDataRequest

For flow-controlled downloads, the DownloadDataRequest message shall be sent from the Client to the Download Server. It is used to control the flow of data from the Download Server to the Client, such as commence data transmission, to positively or negatively acknowledge data, or complete the download scenario. Sending of the DownloadDataRequest with any reason excluding rsnEnd indicates the Client shall be ready to accept more data. The DownloadDataRequest message is not used in the non-flow-controlled or the data carousel scenarios.

Table 7-8 DownloadDataRequest message

Syntax	Num. of Bytes
DownloadDataRequest() {	
dsmccDownloadDataHeader()	
moduleId	2
blockNumber	2
downloadReason	1
}	

The composite of the **moduleId** and **blockNumber** fields provide a sequence number for the ordered delivery of data blocks. The moduleId and blockNumber fields shall point to the next block to be received. In the flow-controlled scenario, the sequence of the blocks in the image shall be defined by the sequence which the modules are described in the DownloadInfoRequest message, together with the definition that the first block in a module shall be block number 0. In the non-flow-controlled and data carousel scenarios, the sequence of blocks may be arbitrary.

The **downloadReason** field indicates the reason for the response. Table 7-9 defines the allowed downloadReason codes.

Table 7-9 downloadReason assignments

downloadReason	**Value**	**Description**
	0x00	ISO/IEC 13818-6 reserved
rsnStart	0x01	Start request for download
rsnAckCont	0x02	Acknowledges reception of previous blocks and requests continuation of transmission from indicated block
rsnNakRetransBlock	0x03	Requests re-transmission of blocks starting from indicated block because blocks are missing
rsnNakRetransWindow	0x04	Requests re-transmission of blocks starting from indicated block because timer tCDownloadWindow expired
rsnEnd	0x05	End download because all blocks are successfully received
	0x06 - 0xEF	ISO/IEC 13818-6 reserved
	0xF0 - 0xFF	Private use

In the case of the beginning of an instance of a download scenario where no block have been received, the DownloadDataRequest message shall be sent with the downloadReason set to rsnStart, and with the moduleId and blockNumber set to point to the first block of the image. In the case of the successful completion of a download, the DownloadDataRequest message shall be sent with the downloadReason set to rsnEnd, and with the moduleId and blockNumber fields set to point to the first block of the image.

7.3.5 DownloadCancel

The DownloadCancel message shall be sent by either the Client or the Download Server to abort the download scenario in progress. The transmission of this message implies the termination of the complete download scenario. The DownloadCancel message may be used in all download scenarios.

Table 7-10 DownloadCancel message

Syntax	Num. of Bytes
DownloadCancel() {	
dsmccMessageHeader()	
downloadId	**4**
moduleId	**2**
blockNumber	**2**
downloadCancelReason	**1**
reserved	**1**
privateDataLength	**2**
for(i=0;i<privateDataLength;i++) {	
privateDataByte	**1**
}	
}	

The **downloadId** field is the identifier of the instance of the download scenario in progress. It shall be used this to associate the DownloadCancel message to a particular download scenario in progress or data carousel.

The **moduleId** and **blockNumber** fields indicate the last processed DownloadDataBlock message at the time of the cancel. If no data blocks have been processed, these fields shall be set to 0.

The **downloadCancelReason** field contains a reason code that explains the reason for the cancellation. Table 7-11 defines the possible downloadCancelReason codes, and which users are allowed to send these codes.

Table 7-11 downloadCancelReason assignments

downloadCancelReason	Value	Sender	Description
	0x00		ISO/IEC 13818-6 reserved
rsnScenarioTimeout	0x01	Server, Client	Timer tCDownloadScenario expired
rsnInsufMem	0x02	Server, Client	Insufficient memory
rsnAuthDenied	0x03	Server	Download authorization denied
rsnFatal	0x04	Server, Client	Fatal error
rsnInfoRequestError	0x05	Server	The Download Server cannot accommodate the requested maximum blockSize or bufferSize.
rsnCompatError	0x06	Server	The Download Server cannot determine an appropriate image from the compatibilityDescriptor provided by the Client.
rsnUnreliableNetwork	0x07	Server	The Client indicated protocol settings for a reliable download (maximumBlockSize and bufferSize set to 0 in DownloadInfoRequest) but Download Server has determined that the level of service provided by the network layer is unreliable.
rsnInvalidData	0x08	Client	The Client received data which did not match the description in the moduleInfoByte fields in the DownloadInfoResponse/ DownloadInfoIndication message.
rsnInvalidBlock	0x09	Client	The Client received a block with an invalid moduleId or blockNumber; i.e., the value of the moduleId and/or blockNumber are not defined in the DownloadInfoResponse/ DownloadInfoIndication message.
rsnInvalidVersion	0x0A	Client	The Client received a block with an unexpected value of the moduleVersion field; i.e. the value of the moduleVersion field is not equal to the value defined in the DownloadInfoResponse/ DownloadInfoIndication message.
rsnAbort	0x0B	Client	The Client aborts the download scenario in progress
rsnRetrans	0x0C	Server, Client	The sender has reached the maximum number of allowed re-transmissions.
rsnBadBlockSize	0x0D	Client	The Client can not support the selected value for blockSize.
rsnBadWindow	0x0E	Client	The Client can not support the selected value of windowSize.
rsnBadAckPeriod	0x0F	Client	The Client can not support the selected value for ackPeriod.
rsnBadWindowTimer	0x10	Client	The Client can not support the selected value for tCDownloadWindow.
rsnBadScenarioTimer	0x11	Client	The Client can not support the selected value for tCDownloadScenario.
rsnBadCapabilities	0x12	Client	The Client can not parse the compatibilityDescriptor.
rsnBadModuleTable	0x13	Client	The Client can not parse the module table.
	0x14 - 0xEF		ISO/IEC 13818-6 reserved
	0xF0 - 0xFF		Private use

The **reserved** field is reserved by ISO/IEC 13818-6 and shall be set to 0xFF.

The **privateDataLength** field defines the length in bytes of the following privateDataByte fields.

The data in the **privateDataByte** field is carried from the Client to the Download Server or from the Download Server to the Client transparently. For example, this message may provide more detailed, implementation-specific information on why the download is being canceled.

7.3.6 DownloadServerInitiate

The DownloadServerInitiate message shall be sent from the Download Server to the Client. In the flow-controlled download scenario, it is a request to the Client to initiate a download by sending the DownloadInfoRequest message. In the non-flow-controlled download scenario and the data carousel scenario, the DownloadServerInitiate message may be used to inform the Client about the connection on which the DownloadInfoIndication messages are located.

Table 7-12 DownloadServerInitiate message

Syntax	Num. of Bytes
DownloadServerInitiate() {	
dsmccMessageHeader()	
serverId	**20**
compatibilityDescriptor()	
privateDataLength	**2**
for(i=0;i<privateDataLength;i++) {	
privateDataByte	**1**
}	
}	

The **serverId** is the globally unique OSI NSAP address of the Download Server to which the Client sends a DownloadInfoRequest, if appropriate. Note the OSI NSAP format enables the use of many different types of lower level network addresses. Therefore, this field is used for the same purpose even when the Download protocol is used outside the context of a User-Network session.

The **compatibilityDescriptor** as defined in clause 6. The Download Server may use this structure to indicate for which Clients the message is appropriate. This field is used by the Client to determine whether subsequent actions are appropriate. This structure may also be used to inform Clients that they should listen for the DownloadInfoIndication message or send a DownloadInfoRequest.

The **privateDataLength** field defines the length in bytes of the following privateDataByte fields.

The data in the **privateDataByte** field is carried from the Download Server to the Client transparently. For example, this message could provide implementation-specific information on why the Download Server wishes the Client to initiate a download. Alternatively, this field may contain information about where the associated DownloadInfoIndication messages are located.

7.4 Message Sequence for Flow-Controlled Download Scenario

Figure 7-5 illustrates the message exchanges for the flow-controlled download scenario.

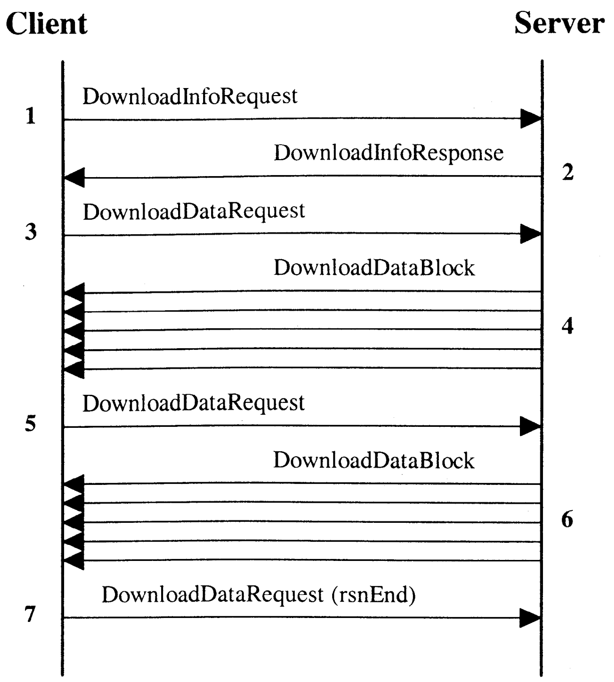


Figure 7-5 Message sequence for flow-controlled download scenario.

7.4.1 Getting Download Protocol Parameters

Step 1 (Client):

As the first step in the flow-controlled download scenario, the Client and Download Server exchange basic parameter information to be used during the download. The Client initiates this information exchange by sending the DownloadInfoRequest.

The Client provides a maximumBlockSize and the bufferSize available for receiving download data without requiring the Download Server to pause while waiting for an acknowledgment.

Step 2 (Server):

The Download Server sends a DownloadInfoResponse message that includes downloadId, blockSize, windowSize, ackPeriod, tCDownloadScenario, and the module table. The downloadId shall be used to correlate the subsequent DownloadDataBlock messages to this download. The Download Server shall select a blockSize and windowSize that meet the requirements:

1. blockSize <= maximumBlockSize
2. windowSize * blockSize <= bufferSize

The ackPeriod shall be less than or equal to the windowSize. An ackPeriod less than the windowSize allows the Client to send an acknowledgment before the Download Server stalls due to a full window. A larger ackPeriod reduces the acknowledgment traffic back to the Download Server. 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 acknowledgment through the network and the Client and Download Server protocol stacks. The transferRate is the expected average delivery rate (bytes/second) that the Download Server would provide if it did not have to wait for acknowledgments. The term (ackLatency * transferRate / blockSize) is the expected number of blocks that the Download Server could send during the period that an acknowledgment is delivered. The first suggestion says that an acknowledgment may be sent in advance of the window being filled such that the acknowledgment may be received before the Download Server stalls because the window shall be full. The second suggestion says that acknowledgments are not sent any more often than the period that it takes to deliver an acknowledgment. There are other reasons, such as burden on the Download Server and network to handle acknowledgments, 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 to interpret the moduleId and blockNumber fields in the DownloadDataBlock and DownloadDataRequest messages. In particular, this information is needed to know when the download is complete.

tCDownloadScenario shall be selected by the Download 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 if transmission errors are not accounted for. The Download Server shall use some conservative estimate of bit rate such that the download will not time-out needlessly during an otherwise successful download.

7.4.2 Starting Download

Step 3 (Client):

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 Download Server shall not start the download until the initial DownloadDataRequest with the downloadReason set to rsnStart is sent by the Client and received by the Download Server.

Step 4 (Server):

Upon receiving the DownloadDataRequest with the downloadReason set to rsnStart from the Client, the Download Server starts with the transmission of the first block of the image. The Download Server shall send the modules in the order in which the modules are described in the module table in the DownloadInfoResponse message. The Download Server shall send the data blocks within a module in order, with the first block having blockNumber set to 0. Therefore, the first block of the image shall be the first block of the first described module in the DownloadInfoResponse message, followed by the second block of the first described module in the DownloadInfoResponse message. The Download Server will stall the transmission of the blocks when it has filled the transmission window.

7.4.3 Acknowledgments

Step 5 (Client):

If the Client has successfully received and stored ackPeriod blocks, it transmits an acknowledgment to the Download Server. An acknowledgment shall be sent at least once per windowSize blocks. The acknowledgment is implemented by the DownloadDataRequest message with the downloadReason set to rsnAckCont and the moduleId and blockNumber fields set to point to the next block to be received. Note the importance of having stored the blocks before sending an acknowledgment, since the acknowledgment implies that the Client is ready to receive windowSize blocks beyond the acknowledged block. In other words an acknowledgment advances the window to windowSize blocks beyond the acknowledged block. Figure 7-6 shows a simple example of DownloadDataRequest messages used in a flow-controlled download with ackPeriod equal to 3. The example uses shorthand of ack(0,3) indicating a DownloadDataRequest message with downloadReason rsnAckCont, moduleId of 0, and blockNumber of 3.

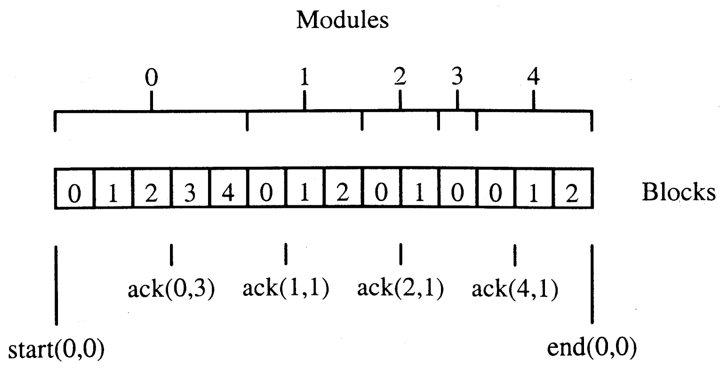


Figure 7-6 Example flow-controlled download

A negative acknowledgment shall be sent by the Client to the Download Server if a block is received that is not the next expected in sequence block of the image. The negative acknowledgment is implemented by the DownloadDataRequest message with the downloadReason set to rsnNakRetransBlock and the moduleId and blockNumber fields point to the next expected block of the image. This negative acknowledgment indicates that the block pointed to has not been received correctly and needs to be retransmitted.

A negative acknowledgment shall also be sent by the Client to the Download Server if the tCDownloadWindow timer expires as described in subclause 7.4.4. The negative acknowledgment is implemented by the DownloadDataRequest message with the downloadReason set to rsnNakRetransWindow and the moduleId and blockNumber fields point to the next expected block of the image.

The Download Server, upon receiving a DownloadDataRequest with downloadReason set to rsnNakRetransWindow or rsnNakRetransBlock, shall resume sending at the block that is pointed to by the moduleId and blockNumber in the DownloadDataRequest. A negative acknowledgment implicitly is an acknowledgment for the block just before the block pointed to in the negative acknowledgment.

Step 6 (Server):

Upon receiving the DownloadDataRequest message from the Client, the Download Server advances the window to windowSize blocks beyond either the explicitly acknowledged block from a positive acknowledgment or the implicitly acknowledged block from a negative acknowledgment. It then resumes the transmission of the blocks starting from the block that was pointed to by the DownloadDataRequest message. The Download Server will stall the transmission again when the updated transmission window is full.

Step 7 (Client):

When the last block of the last module has been successfully received, the Client shall immediately, regardless of the ackPeriod, send a DownloadDataRequest message with the downloadReason set to rsnEnd. The moduleId and blockNumber fields of this message shall point to the first block of the image.

7.4.4 Timers and Re-transmission

The Download protocol employs two classes of re-transmission schemes. All of the download control messages use the DSM-CC message header and are therefore able to take advantage of the standard timer/re-transmission schemes like all other U-N messages. The download data messages, however, require an additional timer/re-transmission scheme since the message flow is more than just a single request and response message pair.

The Client uses two timers for the download data messages; tCDownloadWindow and tCDownloadScenario. The Download Server uses one timer for the download data messages; tSDownloadScenario.

The tCDownloadScenario timer shall be used to time-out the entire download procedure. The Download Server shall provide the value of the timer in the DownloadInfoResponse message. This timer is useful for the Client to detect severe failures, such as the Download Server not responding during the download. When the Client sends the initial DownloadDataRequest message, the Client starts the tCDownloadScenario timer. If this timer expires, the Client shall abort the download, and send a DownloadCancel message with the downloadCancelReason set to rsnScenarioTimeout to the Download Server, and cancel the tCDownloadWindow timer. After the download has timed out, the Client shall discard any download messages from the Download Server that have the downloadId equal to the canceled download. The Client may retry the download by sending a DownloadInfoRequest message.

Upon receipt of the initial DownloadDataRequest message, the Download Server shall start the tSDownloadScenario timer. This timer shall be used to time-out the entire download procedure. Similar to the corresponding timer on the Client, if this timer expires the Download Server shall send a DownloadCancel with the downloadCancelReason set to rsnScenarioTimeout to the Client. The Download Server may then flush any state associated with this download and discard any further messages received with the downloadId equal to the canceled download.

The tCDownloadWindow timer shall be used to time-out acknowledgments. The Download Server shall provide the value for this timer in the DownloadInfoResponse message. This timer is useful for the Client to detect that an acknowledgment was not received by the Download Server. When the Client sends the DownloadDataRequest message with a downloadReason equal to rsnStart, rsnAckCont, rsnNakRetransBlock, or rsnNakRetransWindow, the Client starts or restarts the tCDownloadWindow timer. If this timer expires, then the Client shall send a DownloadDataRequest with downloadReason set to rsnNakRetransWindow to the Download Server, and also restart the tCDownloadWindow timer again. The timer can time-out repeatedly, and each time the DownloadDataRequest with downloadReason set to rsnNakRetransWindow would be resent. Eventually, however, this process will end because time will exceed the maximum value of the tCDownloadScenario timer. This timer shall be used to detect severe failures. After a number of If the Client ends the download by sending the DownloadDataRequest with downloadReason set to rsnEnd, the Client shall stop the tCDownloadWindow timer.

7.4.5 Abort

Either the Client or the Download Server can abort the download scenario in progress by sending a DownloadCancel message. Some potential reasons for an abort are:

1. Download Server responds with an invalid 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 either because the moduleId, blockNumber, or moduleVersion fields are not equal to their counterparts in the DownloadInfoResponse message, or because the received data does not match the description given by moduleSize and the data in the moduleInfoByte fields.
4. The end-user at the Client side decides to abort the download procedure.

7.4.6 Flow-Controlled Scenario over Reliable Transport

The download protocol may be implemented on top of either a reliable or unreliable network transport layer. In the scenario of a reliable network, certain optimizations may be made. For a definition of a reliable transport layer, see clause 1 and clause 9. These optimizations rely on the underlying reliable protocol providing acknowledgments and flow control. In this scenario, when the Client sends a DownloadInfoRequest message, the maximumBlockSize and bufferSize fields shall be set to 0. In turn, the Download Server shall set the windowSize and ackPeriod fields to 0 in the DownloadInfoResponse message. If these fields are set to 0, the Client shall not send DownloadDataRequest messages to the Download Server.

7.5 Message Sequence for Data Carousel Scenario

Figure 7-7 illustrates the message exchanges for the data carousel scenario.

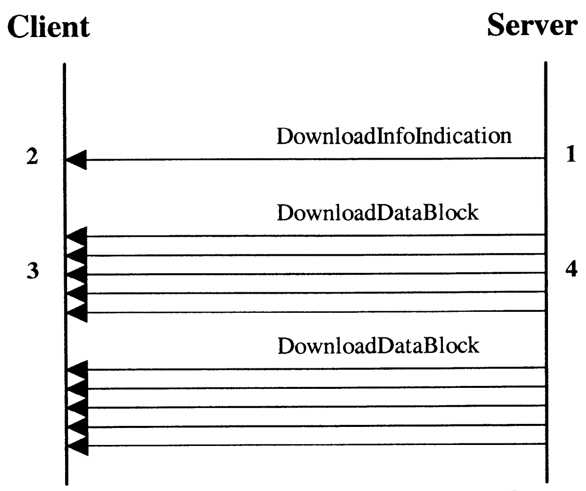


Figure 7-7 Message sequence for data carousel scenario.

7.5.1 Getting Data Carousel Parameters

Step 1 (Server):

The Download Server sends periodic DownloadInfoIndication messages on the ControlDown flow. In the DownloadInfoIndication message, the transactionId in the dsmccMessageHeader shall be used as a versioning mechanism. The Download Server shall set the transactionId field to an arbitrary value, and continue to use that value for each transmission of the DownloadInfoIndication, as long as the entire DownloadInfoIndication message remains unchanged. If any field of the DownloadInfoIndication message is modified, then transactionId shall be incremented, modulo the field size of transactionId. The downloadId correlates the DownloadInfoIndication messages with their corresponding download scenario in progress.

The data in the DownloadInfoIndication message may be used by the Client to determine which, if any, of the described modules are appropriate for the Client. The downloadId shall be used to associate the subsequent DownloadDataBlock messages with the data carousel. The Download Server shall select a value of downloadId that is unique within the scope of the Network. The tCDownloadScenario field shall be set to a value greater than or equal to the time for one full rotation of the carousel. The modules described in one DownloadInfoIndication message may be a subset of all the modules that are cyclically transmitted by the Download Server.

Step 2 (Client):

The Client receives basic information about the data carousel from the Download Server through a DownloadInfoIndication message on the ControlDown flow. Once the Client has acquired the DownloadInfoIndication message, it may discard future DownloadInfoIndication messages with the same transactionId. When the Server updates the content of the data carousel, it will send a new DownloadInfoIndication message with a new transactionId, which may invalidate previous DownloadInfoIndication messages.

The data in the DownloadInfoIndication message may be used by the Client to determine which, if any, of the described modules are appropriate for the Client. The algorithm by which this determination is made is outside the scope of this part of ISO/IEC 13818.

7.5.2 Starting Acquisition and Module Re-Assembly

Step 3 (Client):

Clients will typically retrieve only a subset of the described modules from the data carousel. Upon receiving the DownloadInfoIndication message, the Client typically would allocate memory for each of the desired module. To acquire a particular module, the Client shall look for the DownloadDataBlock messages on the DataDown flow.

The fixed blockSize and the module table in the DownloadInfoIndication message enable the Client to start receiving data out of sequence during a non-flow-controlled or data carousel download. The Client may even begin receiving data if it starts listening in the middle of a module. To do so, the Client may choose to track which blocks from the desired modules have been received using a scoreboard or a similar method.

7.5.2.1 Pseudo-Code Example of Module Re-assembly

Below is some simple pseudo code that demonstrates how a scoreboard may work. This code does not show all the detail of a full implementation and is not normative.

Fields from DownloadInfoIndication used in the pseudo code example:

```
        blockSize
        numberOfModules
        moduleId[ ]
        moduleSize[ ]
        moduleInfoByte[ ]
```

Variables:

M[m]	moduleId of module m (in array of size numberOfModules)
W[m]	Boolean for each module to indicate whether the Client needs to download this module
S[m]	size in bytes of module with moduleId m
SB[m]	size in blocks of module with moduleId m
A[m]	address in memory where block is to be stored
R[m][b]	Boolean for each block of image to indicate if received yet
SBI	total number of blocks in the image
RBI	number of received unique blocks

Initialization:

```
    SBI=RBI=0;
    for(m=0; m<numberOfModules; m++) {
        M[m] = moduleId[m];
        if(CLIENT_WANTS_MODULE(moduleInfoByte[m])) {
            W[m] = 1;
            S[m] = moduleSize[m];
            A[m] = ALLOCATE_MEMORY(S[m]);
            SB[m] = (S[m] + blockSize - 1) / blockSize;
            SBI += SB[m];
            for(j=0; j<SB[m]; j++)
                R[m][j] = 0;
        }
        else {
            W[m] = 0;
        }
    }
```

For each block received (moduleId, blockNumber):

```
    RECEIVE_BLOCK(moduleId,blockNumber);
    b = blockNumber;
    m = MODULEID_TO_INDEX(moduleId, M[ ]);
    if(W[m] &&!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 download scenario to track which blocks have been received to generated an acknowledgment.

7.5.3 Timers

Step 3 continued (Client):

The tCDownloadScenario shall be used in the data carousel scenario to time-out the acquisition of a module. The timer shall started when the Client first starts looking for the DownloadDataBlock messages of the module.

7.5.4 Module Coherency

Step 4 (Server):

The Download Server shall cyclically transmit the blocks of all modules in the data carousel on the DataDown flow. The Download Server may transmit the modules and blocks in arbitrary order.

Note that the downloadId field of the DownloadDataBlock messages is not intended to enable multiplexing. It is assumed that some other method exists for the Client to efficiently filter the blocks of the from the desired image from the connection. If MPEG-2 Transport Streams are used, then all the blocks of one data carousel shall be carried in Transport Stream packets with the same PID (see clause 9).

Each module has a version associated with it. This version shall be carried in the version fields of the module table in the DownloadInfoIndication message. The version provides Clients a means to detect module coherency problems. Such problems may happen when a module is updated while a Client has a download scenario in progress. If the Client detects that the version of a particular block does not correspond with the version of the module received in the DownloadInfoIndication message, then a coherency problem has been detected and the acquisition shall be aborted.

7.5.5 Data Delivery Rate

In the case that the DownloadDataBlock messages are not encapsulated in an MPEG-2 Transport Stream, the method of controlling rate is outside the scope of this part of ISO/IEC 13818.

In the case that the DownloadDataBlock messages are encapsulated in an MPEG-2 Transport Stream, the message delivery shall be regulated by the Transport Stream System Target Decoder (T-STD) model defined in subclause 2.4.2 of ISO/IEC 13818-1 (MPEG-2 Systems). The specification of the leak rate from the Transport buffer is outside the scope of this part of ISO/IEC 13818, but implementations may use the target values tabulated in Table 7-13.

Table 7-13 Download Leak Rates assignments

Leak Rate Class	Download T-STD Leak Rate (Rx_n)
A	0.10 Mbit/s
B	0.40 Mbit/s
C	0.80 Mbit/s
D	1.60 Mbit/s
E	3.20 Mbit/s
F	6.40 Mbit/s
G	9.60 Mbit/s
H	12.80 Mbit/s
I	25.60 Mbit/s

The communication of the leak rate class from the Download Server to the Client is outside the scope of this part of ISO/IEC 13818. A typical implementation may convey this parameter in the compatibilityDescriptor defined in clause 6.

7.6 Message Sequence for Non-Flow-Controlled Download Scenario

The message sequence for the non-flow-controlled download scenario is similar, but not identical, to the message sequence of the data carousel scenario. There are small differences between these scenarios because the non-flow-controlled download scenario is dedicated to downloading a complete image to the Client, where the data carousel scenario merely involves the cyclic transmission and reception of modules. Figure 7-8 illustrates the message exchanges for the non-flow-controlled download scenario.

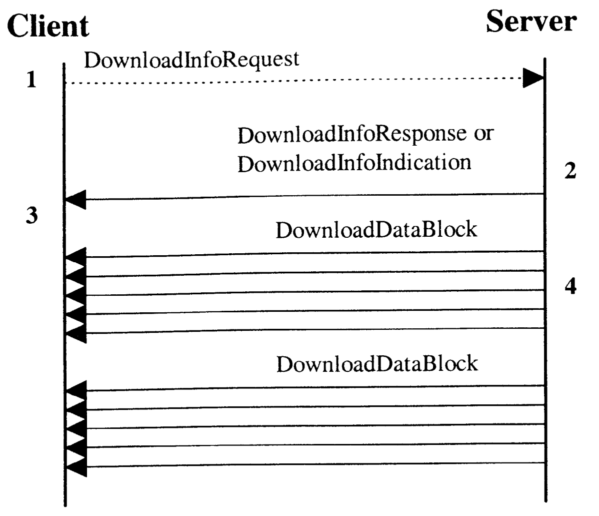


Figure 7-8 Message sequence for non-flow-controlled download scenario.

7.6.1 Getting Download Protocol Parameters

The non-flow-controlled scenario may start with the DownloadInfoRequest message sent from the Client to the Download Server on the ControlUp flow. In this case, the Download Server replies with the DownloadInfoResponse message to the Client on the ControlDown flow.

Alternatively, the Client need not issue the DownloadInfoRequest message. Instead, the non-flow-controlled scenario may start with the Client's receiving of a DownloadInfoIndication message on the ControlDown flow.

The DownloadInfoResponse/DownloadInfoIndication message contains valid downloadId, blockSize, tCDownloadScenario fields and describes all the modules of the image. The message may contain a compatibilityDescriptor, which the Client may use to determine if the described image is appropriate.

The Download Server cyclically transmits the blocks of all the modules of the image. The Download Server may transmit the modules and blocks in arbitrary order.

7.6.2 Image Assembly and Coherency

Upon receiving the DownloadInfoResponse/DownloadInfoIndication message, the Client typically would allocate memory for each of the modules in the image. Subsequently, it will look for DownloadDataBlock messages on the DataDown flow.

The fixed blockSize and the module table in the DownloadInfoResponse/DownloadInfoIndication message enable the Client to start receiving data out of sequence. The Client may even begin receiving data if it starts listening in the middle of a module. To do so, the Client may choose to track which blocks from the desired modules have been received using a scoreboard or a similar method. A pseudo-code example of such an algorithm is shown in subclause 7.5.2.1.

7.6.3 Timers

The tCDownloadScenario shall be used to time-out the acquisition of the image. The timer shall be started when the Client first starts looking for the DownloadDataBlock messages.

7.7 Protocol State Machines for flow-controlled download scenario

This subclause defines the protocol state machines for the Download Server and the Client for the flow-controlled download scenario. State machines are not specified for the non-flow-controlled download scenario and the data carousel scenario; however, part of the state machine for the flow-controlled download scenario shall be used when the DownloadInfoRequest message is used within the non-flow-controlled download scenario (see subclause 7.6).

7.7.1 State Variables common to Client and Download Server

7.7.1.1 Service Type: reliableService, unreliableService

These are predicate variables whose values are known a priori. Only one may be TRUE for any flow-controlled download.

7.7.1.2 Download configured bufferSize: bufferSize

This is the bufferSize as reported by the Client in the DownloadInfoRequest message.

7.7.1.3 Download configured maximumBlockSize: blockSize

This is the maximumBlockSize as reported by the Client in the DownloadInfoRequest message.

7.7.1.4 Download Identifier: Did

This is the downloadId allocated by the Server. For a Client this value is obtained from the downloadId field as reported by the Download Server in the DownloadInfoResponse message.

7.7.1.5 Download negotiated blockSize: Did.blockSize

This is the negotiated blockSize for this download session.

7.7.1.6 Download negotiated windowSize: Did.windowSize.

This is the windowSize as reported by the Download Server in the DownloadInfoResponse message.

7.7.1.7 Download negotiated Acknowledgment Period: Did.ackPeriod.

This is the ackPeriod value as reported by the Download Server in the DownloadInfoResponse message.

7.7.1.8 Download negotiated Window Timer: Did.tWindow

This is the window timer. Its value is reported by the Download Server in the DownloadInfoResponse message as tCDownloadWindow.

7.7.1.9 Download negotiated Scenario Timer: Did.tScenario

This is the scenario timer. Its value is reported by the Download Server in the DownloadInfoResponse message as tCDownloadScenario.

7.7.1.10 Download negotiated compatibilities: Did.compatibilities

This is the compatibilities structure for the Download image.

7.7.1.11 Download Number of Modules: Did.numModules

The number of modules in the module list as reported by the Download Server in the DownloadInfoResponse message.

7.7.1.12 Download Module Identifier: Did.moduleId

There is one Did.moduleId variable for each moduleId value in the module list as reported by the Download Server in the DownloadInfoResponse message.

7.7.1.13 Download Module Version: Did.moduleId.version

The moduleVersion for the corresponding moduleId in the module list as reported by the Download Server in the DownloadInfoResponse message.

7.7.1.14 Download Module Size: Did.moduleId.moduleSize

The size, in bytes, of the module for the corresponding moduleId in the modules list as reported by the Download Server in the DownloadInfoResponse message.

7.7.1.15 Download Expired downloadId Holding timer: Did.tHold

This is the holding timer used to retire downloadIds once they are moved to the DSExpire or DCExpired state. The purpose of this timer is to prevent premature reuse of downloadId values.

7.7.2 Client-only State Variables

7.7.2.1 Download Lower Receive Window Edge: Did.NmoduleId, Did.NblockNum.

These two variables work in conjunction to indicate the block to be received next. It is understood that all blocks before the indicated block have been received and loaded into memory.

7.7.2.2 Number received blocks: Did.Nblock.

The Client maintains this variable to count the number of received blocks after it has send a DownloadDataRequest message. The Did.Nblock variable is used to determine when the next DownloadDataRequest message has to be send.

7.7.2.3 Acknowledgment threshold: Did.AckThreshold.

This variable is the threshold value used by the Client to determine when a new block has to be send. The Download Server provides a hint for this variable in the form of Did.AckThreshold.

7.7.3 Server-only State Variables

7.7.3.1 Lower Transmit Window Edge: Did.LmoduleId, Did.LblockNum

These two variables work in conjunction to identify the block to be sent next by the Download Server.

7.7.3.2 Upper Transmit Window Edge: Did.UmoduleId, Did.UblockNum

For an unreliableService, these two variables work in conjunction to identify the upper credit window on a send. The Server may send up to this window edge without receiving an acknowledgment.

7.7.3.3 Data Sending Rate Timer: Did.tSend

This variable is used to model the periodic sending of blocks at some given rate. In actual implementations this functionality may be performed using another method.

7.7.4 Client Conditions

7.7.4.1 Invalid ServerId

The Client could not parse or understand the ServerId field.

7.7.4.2 Number of re-transmission exceeded

The Client has reached the maximum allowed number of re-transmissions for the DownloadInfoRequestMessage.

7.7.4.3 Unacceptable blockSize

For an unreliableService, blockSize has to be smaller than or equal to maximumBlockSize sent in the DownloadInfoRequest message. For a reliableService, blockSize shall be 0. Otherwise, blockSize is unacceptable.

7.7.4.4 Unacceptable WindowSize

For an unreliableService, windowSize shall be larger than or equal to the ackPeriod. For a reliableService, windowSize shall be 0. Otherwise, windowSize is unacceptable.

7.7.4.5 Unacceptable Acknowledgment Period

For an unreliableService, ackPeriod shall be smaller than windowSize. For a reliableService, the ackPeriod shall be 0. Otherwise, ackPeriod is unacceptable.

7.7.4.6 Unacceptable Window Timer

For a reliableService, the tCDownloadWindow field shall be 0. Otherwise, it is unacceptable.

7.7.4.7 Unacceptable Scenario Timer

For an unreliableService, the value of the tCDownloadScenario field shall be larger than or equal to tCDownloadWindow. Otherwise, it is unacceptable for an unreliableService.

7.7.4.8 Unacceptable Compatibilities

The compatibilities field could not be parsed by the Client or the Client did not match the description in the compatibilityDescriptor.

7.7.4.9 Unacceptable Module Table

The module table fields in the message are either improperly formatted or contain modules whose version, size or module information is unacceptable to the Client.

7.7.4.10 Acknowledgment period full

The acknowledgment period full condition becomes true when the Did.Nblock variable is equal to or exceeds the Did.AckThreshold variable: Did.Nblock >= Did.AckThreshold.

The value of the threshold is a local Client matter but the value shall not be larger than Did.windowSize. The Download Server provides a hint for the threshold by means of the ackPeriod field in the DownloadInfoResponse message. If this value is used the threshold shall be Did.ackPeriod.

7.7.4.11 Download complete

The last block of the last module of the image has been received and therefore the download is complete.

7.7.5 Download Server Conditions

7.7.5.1 Unacceptable maximumBlockSize

For a reliableService, the maximumBlockSize shall be 0. For an unreliableService, the maximumBlockSize shall not be 0. Otherwise, it is invalid.

7.7.5.2 Unacceptable bufferSize

For a reliableService, the bufferSize shall be 0. For an unreliableService, the buffersize shall not be 0. Otherwise, it is invalid.

7.7.5.3 Unacceptable Compatibilities

The compatibilities structure of the DownloadInfoRequest message could not be interpreted by the Server.

7.7.6 Client Procedures

7.7.6.1 Initial Setup of State Variables

Prior to sending the DowloadInfoRequest message to the Download Server:

- blockSize is set to the maximum allowed block size for the Client.
- bufferSize is set to the buffer size allowed by the Client before flow control.
- If this is a reliableService, then blockSize and bufferSize shall be set to 0.

Upon receipt of a valid DownloadInfoResponse message from the Download Server:

- Did is set to the downloadId field.
- Did.blockSize is set to the blockSize field.
- Did.ackPeriod is set to the ackPeriod field.
- Did.tWindow is set to the tCDownloadWindow field.
- Did.tScenario is set to the tCDownloadScenario field.
- Did.compatibilities is set to the Compatibilities field.
- The Module Directory is setup:
    - Did.numModules is set to the numberOfModules field.
    - A Did.moduleId is set to the moduleId field for each module in the Message.
    - A Did.moduleId.version is set to the moduleVersion field for each module in the Message.
    - A Did.moduleId.moduleSize is set to the moduleSize field for each module in the Message.
- The lower receive window edge is initialized:
    - Did.NmoduleId is set to the first module of the module list.
    - Did.NblockNum is set to 0.
- Did.Nblock is set to zero.
- Did.AckThreshold is set to a value less than or equal to windowSize (preferably Did.ackPeriod).

7.7.6.2 Sending DownloadDataRequest Messages

When sending a DownloadDataRequest message:

- the value of the downloadReason field is indicated as DwnDataReq:reason.
- the value of the moduleId field is Did.NmoduleId
- the value of the blockNumber field is Did.NblockNum

Did.tWindow is restarted once the message is sent.

7.7.6.3 Sending DownloadCancel Messages

When sending a DownloadCancel message the value of the downloadCancelReason field is indicated DwnCancel:reason.

7.7.6.4 Increment Lower Receive Window Edge

Incrementing the Lower Receive Window Edge is done as follows:

Did.NblockNum = Did.NblockNum + 1

```
if Did.NblockNum == number of Blocks in Did.NmoduleId then
        Did.NmoduleId = next module of module list
        Did.NblockNum = 0
```

where number of blocks in module = (Did.moduleId.moduleSize + (Did.blockSize - 1))/Did.blockSize.

When a DownloadDataBlock message is received which has moduleId and blockNumber fields that point to a block after the expected block (as indicated by Did.NmoduleId and Did.NblockNum), then this is an out of sequence block and a DownloadDataRequest message with reason of rsnNakRetransBlock shall be sent to the Download Server. If a DownloadDataBlock message is received which has moduleId and blockNumber fields that point to a block before the expected block then this is a duplicate message and it shall be discarded without error and without updating the lower receive window edge.

When the lower receive window edge is equal to the last block of the last module and this block is received then the download of the image is complete.

7.7.6.5 Increment block counter

For unreliableService, incrementing the received Block Counter is:

```
Did.Nblock = Did.Nblock + 1
```

7.7.6.6 Transition to DCExpire State

Whenever the state machine transitions to the DCExpire state the following actions shall automatically be performed in addition to any other actions that may be listed in the Actions section of the table:

- All timers shall be stopped.
- All unneeded memory shall be de-allocated.
- Did.tHold timer is set. The value is a local matter.

7.7.7 Download Server Procedures

7.7.7.1 Initial Setup of State Variables

The state variables shall be setup according to the following criteria:

- A unique downloadId value shall be allocated and assigned to Did.
- Did.blockSize is set from the calculated blockSize value.
- Did.bufSize is set from the received bufferSize value.
- Did.compatibilities is set from the calculated compatibilities value.
- Did.windowSize is set from the calculated value.
- Did.ackPeriod is set from the calculated value.
- Did.tScenario is set from tSDownloadScenario.
- Did.numModules is set from download image information.
- Each Did.moduleId is set from the download image information.
- Each Did.moduleId.version is set from the download image information.
- Each Did.moduleId.moduleSize is set from the download image information.
- Did.LmoduleId is set to the first module of the module list and Did.LblockNum is set to 0.
- Did.UmoduleId and Did.UblockNum are set in accordance with the procedures stated in this subclause.

After receiving the DownloadInfoRequest message the Server shall calculate blockSize, windowSize, ackPeriod, compatibilities, and tCDownloadScenario values based upon the received maximumBlockSize, bufferSize, and possibly compatibilities information. The actual methods used to calculate these values is a local matter and beyond the scope of this recommendation.

If this is a reliableService, windowSize, ackPeriod, tCDownloadWindow and tCDownloadScenario shall all be 0.

The blockSize shall be smaller than or equal to maximumBlockSize.

7.7.7.2 Increment Lower Transmit Window Edge

The lower transmit window edge is set upon receipt of a valid DownloadDataRequest message as follows:

```
    Did.LmoduleId = moduleId
    Did.LblockNum = blockNumber
```

To increment the lower transmit window edge:

```
    Did.LblockNum = Did.LblockNum + 1
    If Did.LblockNum == number of blocks in module Did.LmoduleId then
        Did.LmoduleId = next module of module list
        Did.LblockNum = 0
```

where number of blocks in module = (Did.moduleId.moduleSize + (Did.blockSize - 1))/Did.blockSize.

7.7.7.3 Set Upper Transmit Window Edge

The upper transmit window edge is set upon receipt of a valid DownloadDataRequest message by adding Did.windowSize to the newly received lower transmit window edge values and adjusting this to the appropriate moduleId and blockNum value accounting for the number of blocks in each module. If this edge exceeds the end of the image then Did.UmoduleId is set to the last module of the image Did.UblockNum is set to 0.

7.7.7.4 Sending DownloadDataBlock Messages

Each DownloadDataBlock message is constructed from the block at the current lower transmit window edge, i.e. Did.LmoduleId, Did.LblockNum.

If this is an unreliableService, up to the full credit window may be sent at a time (i.e. from lower transmit window edge to upper transmit window edge) depending upon local implementation decisions. The number of blocks sent is then added to the lower transmit window edge. The state variable Did.tSend is used to initiate and clock the sending of data blocks in this state machine. It should be noted that this is simply a formal abstraction of the actual process and implementations may perform this action using other methods as long as they comply to the behavior of this protocol.

7.7.7.5 Sending DownloadCancel Messages

When sending a DownloadCancel message the value of the reason field is indicated as DwnCancel:reason.

7.7.7.6 Transition to DSExpire State

Whenever the state machine transitions to the DSExpire state the following actions shall automatically be performed in addition to any other actions that may be listed in the Actions section of the table:

- All timers shall be stopped.
- All unneeded memory shall be de-allocated.
- Did.tHold timer is set. The value is a local matter.

7.7.8 State Machine SDL

State machines are described using the Specification and Description Language, in Normative Annex A.

7.8 Partial Protocol State Machines for non-flow-controlled download scenario

When the DownloadInfoRequest message is used within the non-flow-controlled download scenario, then the following state machines are mandatory for the Client and the Server. The Client state-machine shall be equal to the flow-controlled scenario, except that the DCActiveUnr and DCActiveRel states may and should be replaced by implementation specific states that acquire the blocks from the non-flow-controlled DataDown flow. The Server state-machine shall be equal to the state-machine for the flow-controlled case, except that the states WFStart and DSActive may and should be replaced by implementation specific states that transmit the blocks to the Clients.

If the DownloadInfoRequest message is not used for the non-flow-controlled download scenario, both the state machines for the Client and the Server may be implementation specific.

8. Stream Descriptors

Stream descriptors may be used to provide DSM-CC information that is correlated with a MPEG-2 Transport Stream or Program Stream. These descriptors are in the format of Program and Program Element Descriptors as defined in ISO/IEC 13818-1. Four DSM-CC stream descriptors are defined within the MPEG-2 Systems descriptor_tag table. Table 8-1 contains the descriptor_tag assignments and, for reference purposes, lists the assignment space for the MPEG-2 Systems defined values. For details of the MPEG-2 Systems defined values, see ISO/IEC 13818-1, Table 2-39, Program and program element descriptors. Note also that DSM-CC defines other descriptor tags in this space.

Table 8-1 MPEG-2 Systems descriptor_tag assignments for DSM-CC

descriptor_tag	TS	PS	DSMCC Section Type
0-18	n/a	n/a	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 defined
19	X	X	DSM-CC carousel_identifier_descriptor (see clause 11, Object Carousel)
20	X	X	DSM-CC association_tag_descriptor (see clause 11, Object Carousel)
21	X	X	DSM-CC deferred_association_tags_descriptor (see clause 11, Object Carousel)
22	X	X	ISO/IEC 13818-6 reserved
23	X	X	NPT Reference descriptor
24	X	X	NPT Endpoint descriptor
25	X	X	Stream Mode descriptor
26	X	X	Stream Event descriptor
27-63	n/a	n/a	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 defined
64-255	n/a	n/a	User Private per ITU-T Rec. H.222.0 | ISO/IEC 13818-1

8.1 Normal Play Time

Normal Play Time (NPT) is a continuous timeline over the duration of an event. An event is defined in ISO/IEC 13818-1 as a collection of elementary streams with a common time base, an associated start time, and an associated end time. A typical, but not normative example, is a single television show including the audio and the video.

The NPT refers to the real time of the event. For example, when an event is presented in reverse, the NPT counts down rather than up, and when an event is presented at 10 times the normal rate, the NPT progresses at 10 times the normal rate. In this way, NPT increases and decreases in a way similar to a counter on a video tape recorder. The NPT provides an absolute timeline to which references can be made for operations such as jumping to a particular point in the event.

The System Time Clock (STC) of a stream, recovered from the MPEG-2 Transport Stream Program Clock Reference (PCR), alone does not provide an adequate mechanism to determine the NPT of an event. One problem is that the STC always moves forward at the normal rate regardless of the presentation direction and speed of the event. A second problem is that the STC may be discontinuous over the course of the event, leading to possible situations where a given STC time occurs at more than one point in an event.

8.1.1 NPT Reference Descriptor

To enable the determination of the NPT time of an event, the NPT Reference Descriptor is defined. The format of the NPT Reference Descriptor is shown in Table 8-2.

Table 8-2 NPT Reference Descriptor

Syntax	Number of Bits	Mnemonic
NPTReferenceDescriptor() {		
descriptorTag	**8**	**uimsbf**
descriptorLength	**8**	**uimsbf**
postDiscontinuityIndicator	**1**	**bslbf**
contentId	**7**	**uimsbf**
reserved	**7**	**bslbf**
STC_Reference	**33**	**uimsbf**
reserved	**31**	**bslbf**
NPT_Reference	**33**	**tcimsbf**
scaleNumerator	**16**	**tcimsbf**
scaleDenominator	**16**	**tcimsbf**
}		

The **descriptorTag** field is an 8 bit field that identifies the type of stream descriptor. The value of the descriptorTag field for the NPT Reference Descriptor is 23.

The **descriptorLength** field is an 8 bit field specifying the number of bytes of the descriptor immediately following descriptorLength field.

The **postDiscontinuityIndicator** field is a 1 bit field. A value of 0 indicates that the NPT Reference Descriptor is valid upon reception. A value of 1 indicates that the NPT Reference Descriptor will become valid at the next system time base discontinuity as defined in ISO/IEC 13818-1.

The **contentId** field is a 7 bit field that identifies which of a nested set of content is being presented. The contentId field may be used to indicate a transition to a different NPT time base within an existing NPT time base. For example, this field may be changed when a commercial is presented within a television show, and then changed back when the television show is resumed.

The **STC_Reference** field is a 33 bit unsigned integer that indicates the STC value for which the NPT equals the value given in the NPT_Reference field. The value of the STC_Reference field is specified in units of the period of the system clock frequency, as defined in ISO/IEC 13818-1, divided by 300, yielding units of 90 kHz. The **STC_Reference** is derived from the system clock frequency as shown in Equation 8-1.

Equation 8-1

$$\text{STC\_Reference}_k = (\text{STC}_{\text{NPT}(k)} / 300) \% 2^{33}$$

where $\text{STC}_{\text{NPT}(k)}$ is the value of the System Time Clock when the NPT equals the value of the NPT_Reference.

The **NPT_Reference** is a signed 33 bit integer indicating the NPT value at the STC value given in the STC_Reference field.

The **scaleNumerator** is a signed 16 bit integer used with the **scaleDenominator**, a 16 bit unsigned integer, to define the rate of change of the NPT in relation to the STC. A value of 1 for scaleNumerator with a value of 1 for scaleDenominator indicates that the NPT is changing at a rate equivalent to the STC, yielding the standard presentation rate. A value of 0 for scaleNumerator with a non-zero value for scaleDenominator indicates that the NPT is not changing in relation to the STC, yielding a constant value of the NPT. A value of 0 for scaleNumerator with a value of 0 for scaleDenominator indicates that the scaleNumerator and scaleDenominator fields are not provided in the NPT Reference Descriptor. A non-zero value for scaleNumerator with a value of 0 for scaleDenominator shall not be used.

8.1.2 Reconstruction of NPT

A Client receiving the NPT Reference Descriptor is able to reconstruct the value of the NPT for any point in the segment of the stream where the relation indicated by the NPT Reference Descriptor is valid. The reconstructed value of NPT may be used by the Client as a jump point or as the basis of a display to a user.

To reconstruct the value of NPT, the scaleNumerator and scaleDenominator fields must be determined. If these fields are not provided in the NPT Reference Descriptor, they may be calculated utilizing two instances of the NPT Reference

Descriptor, i and j, using Equation 8-2 and Equation 8-3. The value of NPT, NPT_k, for a given value of the STC, STC_k, within the interval over which the NPT Reference Descriptor is valid, or the interval over which both descriptors are valid in the case the scaleNumerator and scaleDenominator are calculated, can then be determined using Equation 8-4.

Equation 8-2

$$\text{scaleNumerator} = \text{NPT\_Reference}_j - \text{NPT\_Reference}_i$$

Equation 8-3

$$\text{scaleDenominator} = \text{SCR\_Reference}_j - \text{SCR\_Reference}_i$$

Equation 8-4

$$NPT_k = (\, (\text{scaleNumerator} \times (\, (STC_k / 300) - \text{STC\_Reference})\,) / \text{scaleDenominator}) + \text{NPT\_Reference}$$

8.1.3 NPT Conversion to Seconds and Microseconds

The unit of the NPT is the period of the System Clock Frequency divided by 300. At times, it may be necessary to convert this value to a corresponding number of seconds and microseconds. This can be accomplished utilizing Equation 8-5 and Equation 8-6.

Equation 8-5

$$\text{NPT\_seconds} = \text{NPT} / (\text{System\_Clock\_Frequency} / 300)$$

Equation 8-6

$$\text{NPT\_microseconds} = (\, (\text{NPT} \times 1000000) / (\text{System\_Clock\_Frequency} / 300)\,) - (\text{NPT\_seconds} \times 1000000)$$

The formula for converting a value of NPT given in seconds and microseconds back to the NPT in the proper units is shown in Equation 8-7.

Equation 8-7

$$\text{NPT} = (\text{NPT\_seconds} \times (\text{System\_Clock\_Frequency} / 300)\,) + (\, (\text{NPT\_microseconds} \times (\text{System\_Clock\_Frequency} / 300)\,) / 1000000)$$

8.1.4 NPT Uncertainty

Situations are possible where the receiver may not have the information necessary to correctly reconstruct the NPT. One case is when a stream is joined and an NPT Reference Descriptor has not yet been received. At that point, a receiver can not be certain of the values for NPT_Reference, scaleNumerator, and scaleDenominator for the stream. In this case, if the values of NPT_Reference, scaleNumerator, and scaleDenominator are not available by other means, the receiver shall set NPT_Reference to 0, scaleNumerator to 1, and scaleDenominator to 1.

Another case of uncertainty is the period after an STC (PCR) discontinuity and before reception of an NPT Reference Descriptor providing the new value of STC_Reference. The stream may reduce the risk of this period by providing an NPT Reference Descriptor in advance of the discontinuity, with the values valid for after the discontinuity by using the postDiscontinuityIndicator field. If no advanced reception of a valid NPT Reference Descriptor occurred, the receiver may approximate the new value of STC_Reference by interpolating from before the discontinuity.

8.1.4.1 Frequency of NPT Reference Descriptor

As described in subclause 8.1.4, periods of uncertainty exist in the NPT value while the receiver waits for an NPT Reference Descriptor. To reduce these periods of uncertainty when NPT Reference Descriptors are carried within an MPEG-2 stream, at least one NPT Reference Descriptor shall occur in a period of one second.

8.1.5 NPT Endpoint Descriptor

The NPT Endpoint Descriptor contains information allowing the Client to maintain the NPT for a specific event. The format of the NPT Endpoint Descriptor is shown Table 8-3.

Table 8-3 NPT Endpoint Descriptor

Syntax	Number of Bits	Mnemonic
NPTEndpointDescriptor() {		
descriptorTag	**8**	**uimsbf**
descriptorLength	**8**	**uimsbf**
reserved	**15**	**bsblf**
startNPT	**33**	**uimsbf**
reserved	**31**	**bsblf**
stopNPT	**33**	**uimsbf**
}		

The **descriptorTag** field is an 8 bit field that identifies the type of stream descriptor. The value of the descriptorTag field for the NPT Endpoint Descriptor is 24.

The **descriptorLength** field is an 8 bit field specifying the number of bytes of the descriptor immediately following descriptorLength field.

The **startNPT** field is a 33 bit unsigned integer whose value is the value of the NPT at the beginning of the current event.

The **stopNPT** field is a 33 bit unsigned integer whose value is the value of the NPT at the end of the current event.

8.2 Stream Mode Descriptor

The Stream Mode Descriptor contains information about the mode of the stream state machine, allowing Clients to better synchronize their actions with stream state changes. The format of the Stream Mode Descriptor is shown in Table 8-4.

Table 8-4 Stream Mode Descriptor

Syntax	Number of Bits	Mnemonic
StreamModeDescriptor() {		
descriptorTag	**8**	**uimsbf**
descriptorLength	**8**	**uimsbf**
streamMode	**8**	**uimsbf**
reserved	**8**	**bsblf**
}		

The **descriptorTag** field is an 8 bit field that identifies the type of stream descriptor. The value of the descriptorTag field for the Stream Mode Descriptor is 25.

The **descriptorLength** field is an 8 bit field specifying the number of bytes of the descriptor immediately following descriptorLength field.

The **streamMode** field is an 8 bit field whose value indicates the current state of the stream state machine. The values for streamMode are shown in Table 8-5. The stream state machine states are defined in clause 5, U-U Interfaces.

Table 8-5 streamMode field values

streamMode	Description
0	Open
1	Pause
2	Transport
3	Transport Pause
4	Search Transport
5	Search Transport Pause
6	Pause Search Transport
7	End of Stream
8	Pre Search Transport
9	Pre Search Transport Pause
10 - 255	ISO/IEC 13818-6 reserved

8.3 Stream Event Descriptor

The Stream Event Descriptor contains information allowing the transmission of application-specific events as defined in clause 5, so that they may be synchronous with the stream. Note that the definition of event in this context is not the same as event as it relates to NPT. The format of the Stream Event Descriptor is shown in Table 8-6.

Table 8-6 Stream Event Descriptor

Syntax	Number of Bits	Mnemonic
StreamEventDescriptor() {		
descriptorTag	**8**	**uimsbf**
descriptorLength	**8**	**uimsbf**
eventId	**16**	**uimsbf**
reserved	**31**	**bsblf**
eventNPT	**33**	**uimsbf**
for (i=0; i < N; i++) {		
privateDataByte	**8**	**uimsbf**
}		
}		

The **descriptorTag** field is an 8 bit field that identifies the type of stream descriptor. The value of the descriptorTag field for the Stream Event Descriptor is 26.

The **descriptorLength** field is an 8 bit field specifying the number of bytes of the descriptor immediately following descriptorLength field.

The **eventId** field is an 8 bit field whose value is the type of the application specific event.

The **eventNPT** field is a 33 bit unsigned integer whose value is the value of the NPT when the event occurred, or the value of the NPT when the event will occur.

The **privateDataByte** fields allows inclusion of application specific data in the Stream Event Descriptor.

9. Transport

9.1 DSM-CC Requirements on Lower-Level Network Transport Protocol

9.1.1 U-N Message Categories

The DSM-CC U-N Message categories

1. U-N Configuration
2. U-N Session Messages
3. U-N Download
4. Switched Digital Broadcast Channel Change Protocol
5. U-N Pass-Thru Messages

are designed to be transport protocol (e.g., UDP/IP) independent. The transport mechanism includes the transport layer and all underlying layers. The table below summarizes the minimum level of service that the transport layer shall provide.

Table 9-1 U-N and Download Transport protocol requirements

Transport Function	Requirement
Reliability of Data	Error detection shall 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 Re-assembly	Transport is responsible for any required fragmentation and re-assembly. As such, the maximum U-N message size may be limited by the transport protocol chosen.
Delivery Order of Messages	Transport need not be responsible for in order delivery of messages.
Addressing	Transport shall be able to deliver a message to its intended recipient.

9.1.2 U-U Interface Categories

The DSM-CC U-U interface categories are:

1. U-U RPC Library
2. Session Objects
3. Download Objects
4. U-U Object Carousel
5. Application Local Objects
6. Stream Descriptors

U-U RPC Library requires the use of a remote procedure call (RPC) mechanism. Different RPC protocols have specific requirements on their transport layers; therefore, these requirements are beyond the scope of this part of ISO/IEC 13818.

Session Object structures may be carried within the uuData field in U-N Session Messages. U-U Object Carousel Object structures and Download Object structures may be carried in Download messages. Therefore, from a transport point of view, each of these are subject to U-N category requirements if carried within Download messages.

Stream Descriptors provide a mechanism for the U-U Interface to provide additional stream information about a MPEG program. Therefore, they are subject to ISO/IEC 13818-1 requirements with additional constraints described in this clause.

Application Local Objects are used by a Client application and do not have an external interface. Therefore, there are not covered in this clause.

9.2 Encapsulation within MPEG-2 Transport Streams

9.2.1 Role of MPEG-2 Transport Stream in the Protocol Stack

None of the DSM-CC messages or interfaces are required to be carried within an MPEG-2 Transport stream. However, if MPEG-2 Transport Streams are used to deliver DSM-CC protocols, then the encapsulation of these messages as defined in this section shall be used.

MPEG-2 Systems, ISO/IEC 13818-1, defines a private_section structure which DSM-CC uses to provide re-assembly of Transport Stream Packets into DSM-CC messages. This part of ISO/IEC 13818defines additional semantics on private_sections to support additional DSM-CC requirements. Called DSMCC_section, the structure is compatible with the private_section syntax so that compliant MPEG-2 Systems decoders may be used.

9.2.2 DSM-CC Sections

When DSM-CC U-N and Download messages are encapsulated in MPEG-2 Transport Streams, the DSMCC_section syntax shall be used. Other data payloads may also use this syntax. This structure inherits all of the Private_section syntax as defined in ISO/IEC 13818-1. Special semantics apply to the encoding of particular fields in the DSMCC_section header. The mapping of the DSMCC_section into MPEG-2 Transport Stream Packets and the maximum length of a DSMCC_section are governed by the semantics for Private_sections defined in ISO/IEC 13818-1.

In some implementations, it is desirable to use the CRC_32 available in Private_sections. Because some systems may have difficulty calculating a CRC_32, the DSMCC_section syntax defines an alternative to using CRC_32. To be consistent with ISO/IEC 13818-1, if the section_syntax_indicator is set to '1', then the CRC_32 shall be present and correct. In the case where the section_syntax_indicator is '0', the syntax of the section is the same as when the section_syntax_indicator is '1', except that the CRC_32 field is replaced with the checksum field. The resultant syntax is still compliant to ISO/IEC 13818-1, since the payload following the section_length field shall be treated as private data.

Since the section_syntax_indicator bit itself may be subject to a bit error, the private_indicator field shall be set to the complement of the section_syntax_indicator value. If the section_syntax_indicator is '0', then the private_indicator shall be verified to be '1', and if it is not, the section has suffered an error. Similarly, if the section_syntax_indicator is '1' then private_indicator shall be '0'.

When section_syntax_indicator is '0' (CRC is not used) and the checksum field has been set to 0, another form of error detection shall be provided at a different layer. This requirement is imposed to ensure the DSMCC_section maintains the minimal requirements this part of ISO/IEC 13818 imposes on its transport protocol (see Table 9-1 U-N and Download Transport protocol requirements).

For syntax and semantics related to the carriage of private_sections (and therefore DSMCC_sections) within the MPEG Transport Stream, see ISO/IEC 13818-1 section 2.4.4 Program specific information. This includes the setting of the payload_unit_start_indicator, the presence of the pointer_field in the Transport Stream packet payload, and the use of packet stuffing bytes.

Unless otherwise restricted, DSM-CC tables (i.e., one or more DSMCC_sections with the same table_id) may be contained in Transport Stream packets with the same value PID as other private_section formatted tables (e.g., in ISO/IEC 13818-1 stream_type 0x05), if table_id parsing is done.

Table 9-2 DSM-CC Section Format

Syntax	No. of bits	Mnemonic
DSMCC_section() {		
table_id	8	**uimsbf**
section_syntax_indicator	1	**bslbf**
private_indicator	1	**bslbf**
reserved	2	**bslbf**
dsmcc_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 == 0x3A) {		
LLCSNAP()		
}		
else if (table_id == 0x3B) {		
userNetworkMessage()		
}		
else if (table_id == 0x3C) {		
downloadDataMessage()		
}		
else if (table_id == 0x3D) {		
DSMCC_descriptor_list()		
}		
else if (table_id == 0x3E) {		
for (i=0;i<dsmcc_section_length-9;i++) {		
private_data_byte		
}		
}		
if(section_syntax_indicator == '0') {		
checksum	32	**uimsbf**
}		
else {		
CRC_32	32	**rpchof**
}		
}		

Note: For LLCSNAP, see subclause 9.2, Encapsulation within MPEG-2 Transport Streams.

9.2.2.1 Semantic definition of fields in DSMCC_section

table_id -- This is an 8 bit field which, in the case of a DSMCC_section, shall be set to identify the type of data in the DSMCC_section payload. This field defines particular encoding rules for the table_id_extension, version_number, section_number, and last_section_number fields.

Table 9-3 contains the MPEG-2 Systems table_id assignments for DSM-CC and, for reference purposes, lists the assignment space for the MPEG-2 Systems defined values. For details of the MPEG-2 Systems defined values, see ISO/IEC 13818-1, Table 2-26, table_id assignment values.

Table 9-3 DSM-CC table_id assignments

table_id	DSMCC Section Type
0x00 - 0x37	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 defined
0x38 - 0x39	ISO/IEC 13818-6 reserved
0x3A	DSM-CC Sections containing multi-protocol encapsulated data
0x3B	DSM-CC Sections containing U-N Messages, except Download Data Messages
0x3C	DSM-CC Sections containing Download Data Messages
0x3D	DSM-CC Sections containing Stream Descriptors
0x3E	DSM-CC Sections containing private data
0x3F	ISO/IEC 13818-6 reserved
0x40 - 0xFE	User private
0xFF	forbidden

Multi-protocol encapsulated data shall include U-U RPCs, if transported over MPEG-2 Transport Streams, and may include private multi-protocol encapsulated data.

section_syntax_indicator -- This is a 1 bit field. When set to '1' it indicates the presence of the CRC_32 field. When set to '0' it indicates the presence of the checksum field.

private_indicator -- This is a 1 bit flag. It shall be set to the complement value of section_syntax_indicator flag.

reserved -- This 2 bit field shall be set to '11'.

dsmcc_section_length -- This 12 bit field specifies the number of remaining bytes in the DSMCC_section immediately following this field up to the end of the DSMCC_section. The value in this field shall not exceed 4093 (indicating a section maximum data length of 4084 bytes, following the last_section_number field and up to the CRC_32/checksum field).

table_id_extension -- This is a 16-bit field. If the value of the table_id field equals 0x3B, this field conveys a copy of the least significant 2 Bytes of the transaction_id field in the dsmccMessageHeader of the conveyed U-N Message. If the value of the table_id field equals 0x3C, this field conveys a copy of the moduleId field of the conveyed DownloadDataBlock or DownloadDataRequest Message. If the value of the table_id field is not equal to either 0x3B or 0x3C, then the value and use of this field are defined by the user.

version_number -- This field is a 5-bit field. If the value of the table_id field equals 0x3A or 0x3B, this field shall be set to zero. If the value of the table_id field equals 0x3C and a DownloadDataBlock Message is conveyed, this field shall have the value of the least significant 5 bits of the moduleVersion field of the conveyed DownloadDataBlock Message. If the value of the table_id field equals 0x3C and a DownloadDataRequest Message is conveyed, this field shall be set to zero. If the value of the table_id field is not in the range of 0x3A to 0x3C, then the value and use of this field are defined by the user.

current_next_indicator -- This is a 1 bit flag. If the value of the table_id field has a value in the range of 0x3A to 0x3C, this bit shall be set to '1'. Otherwise, the value and use of this field are defined by the user.

section_number -- This field is a 8-bit field. If the value of the table_id field equals 0x3A or 0x3B, this field shall be set to zero. If the value of the table_id field equals 0x3C, this field shall have a value of the least significant 8 bits of the blockNumber field of the conveyed DownloadDataBlock or DownloadDataRequest Message. If the value of the table_id field is not in the range of 0x3A to 0x3C, then the value and use of this field are defined by the user.

last_section_number -- This field is a 8-bit field. This field shall be set to the maximum value that is encoded in the section_number field for the same table_id, table_id_extension and version_number field.

CRC_32 -- This field shall be set as defined in ISO/IEC 13818-1 Annex B. This field is only present when section_syntax_indicator is set to '1'.

checksum -- A 32 bit checksum calculated over the entire DSMCC_section. The checksum shall be calculated by treating the DSMCC_section as a sequence of 32-bit integers and performing one's complement addition (an Exclusive-Or or XOR operation) over all the integers, most significant byte first, then taking the one's complement of the result. For the purpose of computing the checksum, the value of the checksum field shall be considered 0. If the message length is not a multiple of four bytes, the message shall be considered to be appended with zeroed bytes for the purpose of checksum calculation only. If the computed result is 0, then the result shall be set to 0xFFFFFFFF (the alternative value for a one's complement representation of 0). In cases where a checksum is not desired, the value of this field shall be set to 0 to indicate the checksum has not been calculated. This feature is useful for networks where error detection is provided at a protocol layer lower than the MPEG-2 Transport Stream. This field is only present when section_syntax_indicator is set to '0'.

9.2.3 DSM-CC Stream Types

DSM-CC has defined distinct stream_type values so that simple parsing of the ISO/IEC 13818-1 Program Map Table (PMT) may be done to find the PIDs for each of the different DSM-CC section types (which, in turn, enables parsing by PIDs). Filtering may also be done on the table_id field within the DSMCC_section.

Table 9-4 contains the MPEG-2 Systems stream type assignments for DSM-CC and, for reference purposes, includes the assignment space for the MPEG-2 Systems defined values. For details of the MPEG-2 Systems defined values, see ISO/IEC 13818-1, Table 2-29, Stream type assignments.

Table 9-4 DSM-CC Stream Types

stream_type	Description
0x00-0x09	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 defined
0x0A	Multi-protocol Encapsulation
0x0B	DSM-CC U-N Messages
0x0C	DSM-CC Stream Descriptors
0x0D	DSM-CC Sections (any type, including private data)
0x0E - 0x7F	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 reserved
0x80 - 0xFF	User private

Since table_id parsing is optional, restrictions on content types shall be placed on stream_types 0x0A-0x0C as follows:

- Only DSMCC_sections with table_id 0x3A shall be contained within Transport Stream packets of stream_type 0x0A.
- Only DSMCC_sections with table_id 0x3B and 0x3C shall be contained within Transport Stream packets of stream_type 0x0B.
- Only DSMCC_sections with table_id 0x3D shall be contained within Transport Stream packets of stream_type 0x0C.

If table_id parsing is available, stream_type 0x0D may be used to map all DSM-CC sections to Transport Stream packets of the same PID.

9.2.4 DSM-CC Multi-protocol Encapsulation

The DSM-CC U-U service inter-operability interface (SSI) requires the use of a remote procedure call (RPC) mechanism. Different RPC protocols have specific requirements on their transport layers; therefore, these requirements are beyond the scope of this part of ISO/IEC 13818.

The SSI uses RPC protocols whose lower layer protocol stacks are outside the scope of this part of ISO/IEC 13818. However, if these messages are to be carried over an MPEG-2 Transport Stream, they shall be encapsulated according to ISO/IEC 8802-2 Logical Link Control (LLC) and ISO/IEC 8802-1a SubNetwork Attachment Point (SNAP) specifications. This encapsulation method may also be used for other payloads to provide a uniform method of data carriage over MPEG Transport Streams.

The LLC/SNAP structure allows for the encapsulation of a wide selection of OSI Layer 3 (network) protocols, including e.g. the Internet Protocol (IP). The selected Layer 3 protocol, in turn, may encapsulate a selected transport Layer 4 protocol such as TCP or UDP.

9.2.5 U-N Message Categories

DSM-CC U-N Messages -- U-N Configuration, U-N Session Messages, Download, Switched Digital Broadcast Channel Change Protocol, and U-N Pass-Thru -- are defined in clauses 3, 4, 7, 10, and 12, respectively. If encapsulated into a Transport Stream, U-N Messages shall be conveyed directly into the payload of a DSMCC_section. A single DSMCC_section shall contain data from no more than one U-N message.

When DownloadDataBlock messages are carried in MPEG-2 Transport Streams, only DownloadDataBlock messages with the same value downloadId shall be contained in Transport Stream packets with the same value PID.

9.2.6 U-U Service Inter-operability Interface using Remote Procedure Call

User-User SSI is defined in clause 5. When these message are to be carried in an MPEG-2 Transport Stream, the DSM-CC Multi-protocol Encapsulation (subclause 9.2.4) shall be required. The use of this encapsulation method for the delivery of other user-defined messages is optional. For example, if TCP/IP is encapsulated in this manner to carry U-U RPC messages, the same method may 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 contained in Transport Stream packets with the same value PID.

9.2.7 DSM-CC Stream Descriptors

In the cases where DSM-CC Stream Descriptors (e.g., Normal Play Time time stamp, StreamMode and StreamEvent descriptors) are carried in an MPEG-2 Transport Stream, they shall be carried in a DSMCC_descriptor_list() (see Table 9-5) within a DSMCC_section. Multiple descriptors may be carried in the same descriptor list.

Table 9-5 DSM-CC Descriptor List

Syntax	No. of bits	Mnemonic
DSMCC_descriptor_list() {		
for(i=0;i<N;i++) {		
stream_descriptor()		
}		
}		

9.2.7.1 Semantic definition of fields in DSM-CC Descriptor List

stream_descriptor() -- Defined in clause 8, Stream Descriptors.

9.3 Encapsulation within MPEG-2 Program Streams

9.3.1 DSM-CC Stream Descriptors

In the cases where DSM-CC Stream Descriptors are carried in an MPEG-2 Program Stream, they shall be carried in a DSMCC_program_stream_descriptor_list() (see Table 9-6) as a PES packet as defined in ISO/IEC 13818-1. Multiple descriptors may be carried in the same descriptor list.

Table 9-6 DSM-CC Program Stream Descriptor List

Syntax	No. of bits	Mnemonic
DSMCC_program_stream_descriptor_list() {	24	**bslbf**
packet_start_code_prefix	8	**uimsbf**
stream_id	16	**uimsbf**
packet_length	8	**uimsbf**
for(i=0;i<N;i++) {		
dsmcc_discriminator		
stream_descriptor()		
}		
}		

9.3.1.1 Semantic definition of fields in DSM-CC_program_stream_Descriptor List

packet_start_code_prefix -- This field shall be set to 0x000001, per ISO/IEC 13818-1 (PES Packet semantics).

stream_id -- This is a 8-bit field specifying the bit stream identification that takes the value '1111 0010' for the DSM-CC bit stream. Note: the same stream_id value as ISO/IEC 13818-1 Annex A is used.

packet_length -- A 16-bit field specifying the number of bytes in the DSMCC_program_stream_descriptor_list following the last byte of the field. The value of 0 shall not be used.

dsmcc_discriminator -- This is an 8-bit unsigned integer that takes the value 0x80 for the DSMCC_program_stream_descriptor_list. Note: this is equivalent to ISO/IEC 13818-1 Annex A command_id field and is used for forward compatibility.

stream_descriptor() -- Defined in clause 8, Stream Descriptors.

9.3.2 U-N Messages and U-U SSI

The carriage of DSM-CC Messages, other than Stream Descriptors, within Program Streams is not defined by this part of ISO/IEC 13818.

10. U-N Switched Digital Broadcast -- Channel Change Protocol

10.1 Overview

The Switched Digital Broadcast (SDB) Channel Change Protocol (CCP) is used between a Client and an inter-working unit (IWU) to enable the Client to remotely switch from channel to channel in a broadcast environment. This is useful in systems where the interface from the Client to the network provides only limited bandwidth such that Clients are not able to receive all broadcast programs simultaneously. Annex H provides an example of such a service.

10.1.1 Preconditions and Assumptions

The SDB CCP is assumed to be part of a protocol stack. The CCP messages are designed to be carried on top of various protocols (e.g., IP, ATM). Constraints on specific lower level protocols are given in clause 9, "Transport".

Before a Client may use the SDB Channel Change Protocol, a communication path shall have been established between the Client and the SDB Server, which provides the SDB service. DSM-CC U-N Session Messages may be used to establish a SDB Service Session (and connections, as resources of this service) between a Client and the SDB Server. However, other means may be used to establish this communication path (e.g., via provisioning).

In Annex H, an example of the life cycle for a SDB service is given, including a U-N Session Setup scenario.

10.1.2 General Message Format

The SDB CCP messages use a request/response mechanism. Request messages are generated when the Client initiates a message sequence. The SDB Server responds to a Request message with a Confirm message. Messages which are sent asynchronously to the Client from the SDB Server are Indication messages. The Client responds to an Indication message with a Response message.

The SDB CCP messages have a common message format, and share the same header as other U-N Messages. Table 10-1 defines the general format of DSM-CC SDB Channel Change Protocol messages.

Table 10-1 General Format of DSM-CC SDB CCP Messages

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

The **DSMCCMessageHeader** is defined in the clause 2 of this part of ISO/IEC 13818.

The **MessagePayload** is constructed from resource descriptors and data fields and differs in structure depending on the function of the particular message. Subclause 10.2 defines the DSM-CC SDB CCP Messages.

10.2 Switched Digital Broadcast Channel Change Protocol Messages

This subclause defines the SDB Channel Change Protocol messages. Each message is identified by a specific messageId which is encoded to indicate the class and direction of the message. Table 10-2 defines the encoding of the messageId fields used in the SDB CCP messages.

Table 10-2 DSM-CC SDB Channel Change Protocol messageIds

messageId	Message Name	Description
0x0000	Reserved	ISO/IEC 13818-6 Reserved.
0x0001	SDBProgramSelectRequest	Sent from a User to the SDB Server to request that a broadcast program be provided.
0x0002	SDBProgramSelectConfirm	Sent from the SDB Server to a User in response to the SDBProgramSelectRequest.
0x0003	SDBProgramSelectIndication	Sent from the SDB Server to a User to indicate that a new broadcast program will be provided.
0x0004	SDBProgramSelectResponse	Sent from a User to the SDB Server in response to the SDBProgramSelectIndication message.
0x0005 - 0x7FFF	Reserved	ISO/IEC 13818-6 Reserved.
0x8000 - 0xFFFF	User Defined	User Defined SDB message.

10.2.1 Use of Private Data in SDB CCP messages

SDB CCP messages allow private data to be used. Table 10-3 defines the format of PrivateData(), which is transported in SDB messages.

Table 10-3 DSM-CC SDB Private Data Format

Syntax	Num. of Bytes
PrivateData(){	
privateDataLength	2
for(i=0;i<privateDataLength;i++) {	
privateDataByte	1
}	
}	

The **privateDataLength** field shall indicate the total number of privateDataBytes.

The **privateDataByte** fields contain the private data. The format and usage of this data is outside of the scope of this part of ISO/IEC 13818.

10.2.2 Use of BroadcastProgramId in SDB CCP messages

In SDBProgramSelect messages a broadcastProgramId field is used to specify a single broadcast program at the interface between Client and SDB Server. Table 10-4 specifies the allowed values for the broadcastProgramId field.

Table 10-4 DSM-CC SDB broadcastProgramIds

broadcastProgramId	Broadcast Program Name	Description
0x00000000	NoProgram	Identifies that a broadcast program has not been selected or indicated.
0x00000001 - 0x7FFFFFFF	Broadcast Program Numbers	Uniquely identifies a single broadcast program.
0x80000000 - 0xFFFFFFFF	User Defined	User Defined special purpose SDB broadcastProgramIds.

10.2.3 SDB CCP message definitions

All of the SDB CCP messages contain a sessionId field. In the case, that the U-N session was set up dynamically by using the UN Session Setup, this field shall be encoded with the same value as it was negotiated during the SessionSetup procedure. If the session was set up statically via provisioning, then the encoding of this field has to be mutually agreed upon between Client and SDB Server.

10.2.3.1 SDBProgramSelectRequest message definition

This message is sent from a Client to the SDB Server to request that a selected broadcast program shall be established. The SDB Server will respond with a SDBProgramSelectConfirm message. Table 10-5 defines the DSM-CC SDBProgramSelectRequest message.

Table 10-5 DSM-CC SDBProgramSelectRequest Message

Syntax	Num. of Bytes
SDBProgramSelectRequest(){	
sessionId	**10**
reserved	**2**
broadcastProgramId	**4**
PrivateData()	
}	

The **sessionId** is used to identify a session throughout its life cycle.

The **reserved** field is reserved ISO/IEC 13818-6 and shall be set to 0xFFFF.

The **broadcastProgramId** field shall be set by the Client and shall contain a value which identifies the new broadcast program which shall be provided now to the Client.

The PrivateData() structure is defined in Table 10-3. The definition of the content of this field is outside the scope of this part of ISO/IEC 13818.

10.2.3.2 SDBProgramSelectConfirm message definition

This message is sent from the SDB Server to a Client in response to a SDBProgramSelectRequest message. Table 10-6 defines the DSM-CC SDBProgramSelectConfirm message.

Table 10-6 DSM-CC SDBProgramSelectConfirm Message

Syntax	Num. of Bytes
SDBProgramSelectConfirm(){	
sessionId	**10**
response	**2**
broadcastProgramId	**4**
PrivateData()	
}	

The **sessionId** is used to identify a session throughout its life cycle.

The **response** field shall be set by the SDB Server to a value which indicates the SDB Server's response to the SDBProgramSelectRequest message. The valid values for the encoding of this field are given in Table 10-10 DSM-CC SDB Response Codes.

The **broadcastProgramId** field shall be set to a value indicating the broadcast program that is provided to the Client.

The PrivateData() structure is defined in Table 10-3. The definition of the content of this field is outside the scope of this part of ISO/IEC 13818. However, it is expected that the PrivateData() structure contains connection information necessary for the Client to receive the broadcast program.

10.2.3.3 SDBProgramSelectIndication message definition

This message is sent from the SDB Server to a Client to indicate that a new broadcast program is provided. The Client shall respond with a SDBProgramSelectResponse message. Table 10-7 defines the DSM-CC SDBProgramSelectIndication message.

Table 10-7 DSM-CC SDBProgramSelectIndication Message

Syntax	Num. of Bytes
SDBProgramSelectIndication(){	
sessionId	**10**
reason	**2**
broadcastProgramId	**4**
PrivateData()	
}	

The **sessionId** is used to identify a session throughout its life cycle.

The **reason** field shall be set by the SDB Server to a value which indicates why the SDB Server has selected the broadcast. The valid values for the encoding of this field are given in Table 10-9 DSM-CC SDB Reason Codes.

The **broadcastProgramId** field shall be set to a value specifying the broadcast program which will be provided by the SDB Server.

The PrivateData() structure is defined in Table 10-3. The definition of the content of this field is outside the scope of this part of ISO/IEC 13818. However, it is expected that the PrivateData() structure contains connection information necessary for the Client to receive the broadcast program.

10.2.3.4 SDBProgramSelectResponse message definition

This message is sent from the Client to the SDB Server in response to a SDBProgramSelectIndication message. Table 10-8 defines the SDBProgramSelectResponse message.

Table 10-8 DSM-CC SDBProgramSelectResponse Message

Syntax	Num. of Bytes
SDBProgramSelectResponse(){	
sessionId	**10**
response	**2**
PrivateData()	
}	

The **sessionId** is used to identify a session throughout its life cycle.

The **response** field shall be set by the Client to a value which indicates the Client's response to the SDBProgramSelectRequest message. The valid values for the encoding of this field are given in Table 10-10 DSM-CC SDB Response Codes.

The PrivateData() structure is defined in Table 10-3. The definition of the content of this field is outside the scope of this part of ISO/IEC 13818.

10.3 SDB Channel Change Protocol Command Scenarios

10.3.1 Client Initiated Program Select Command Sequence

Figure 10-1 illustrates the procedure for program selection initiated by the Client.

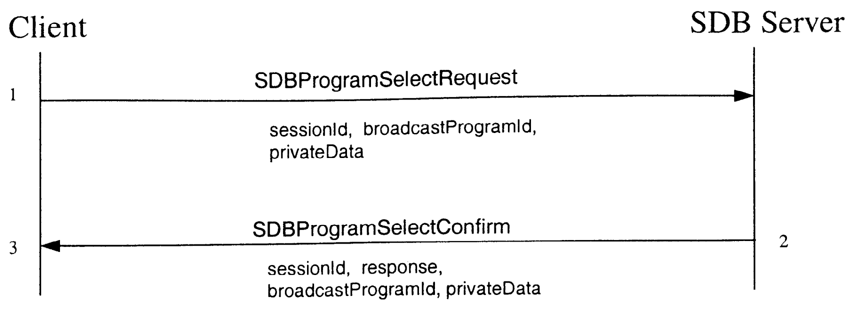

Client
SDB Server
1
SDBProgramSelectRequest
sessionId, broadcastProgramId, privateData
SDBProgramSelectConfirm
3
2
sessionId, response, broadcastProgramId, privateData

Figure 10-1 Scenario for Client Initiated Program Select Sequence

Step 1 Client:

To select a broadcast program to be provided, the Client shall send a SDBProgramSelectRequest to the SDB Server and start timer tMsg. The broadcastProgramId shall be set to a value which identifies the broadcast program the Client wants to receive.

If timer tMsg expires before a SDBProgramSelectConfirm is received, then the Client shall assume the program select request failed. The Client may repeat the SDBProgramSelectRequest message or it may initiate an audit with the Network.

Step 2 SDB Server:

On receipt of a SDBProgramSelectRequest, the SDB Server shall verify that the sessionId field refers to an active session. If the SDB Server determines that the sessionId is invalid, then the SDB Server shall respond with a SDBProgramSelectConfirm message with the response field set to rspNoSession.

If the sessionId is valid, then the SDB Server shall verify that the SDBProgramSelectRequest message is encoded correctly, especially the length fields are encoded consistently. If the SDB Server detects encoding errors in the SDBProgramSelect message, then it shall reply with a SDBProgramSelectConfirm message with the response field set to rspFormatError.

If sessionId is valid and encoding of the message is correct, then the SDB Server shall verify that the broadcastProgramId field represents a valid broadcast program. If the requested broadcast program is not available (either currently or permanently), then the SDB Server shall either reply with a SDBProgramSelectConfirm message with the response field set to rspBcProgramOutOfService, or redirect to another broadcast program. In the latter case, the response field in the SDBProgramSelectConfirm message shall be set to rspRedirect.

If broadcastProgramId is valid, then the SDB Server may verify that the Client is entitled to receive the broadcast program identified by the broadcastProgramId. If the Client is not entitled to receive the broadcast program, then the SDB Server shall either reply with a SDBProgramSelectConfirm message with the response field set to rspEntitlementFailure, or redirect to another broadcast program. In the latter case, the response field in the SDBProgramSelectConfirm message shall be set to rspRedirect.

If the Client is entitled to receive the broadcast program channel, but the SDB Server does not have enough internal resources to provide the broadcast program to the Client, then the SDB Server shall reply with a SDBProgramSelectConfirm message with the response field set to rspNoServerResource.

If the SDB Server has enough internal resources to provide the broadcast program to the Client, then the SDB Server shall attempt to provide the broadcast program to the Client (the actual means, how this is achieved, are outside of the scope of this part of ISO/IEC 13818). If there are not enough network resources available, then the SDB Server shall respond to the Client with a SDBProgramSelectConfirm message with the response field set to rspNoNetworkResource.

If the SDB Server can provide the requested broadcast program to the Client, then the SDB Server shall send a SDBProgramSelectConfirm with response field set to rspOk to the Client. The value of the sessionId field shall be set by the SDB Server to the same value received in the SDBProgramSelectRequest message. The value of the privateDataCount field shall be equal to the number of privateDataBytes present in the remainder of the message. It is assumed that these privateDataByte fields contain a description of the new resources, if needed, to provide the broadcast program.

Step 3 Client:

On receipt of SDBProgramSelectConfirm, the Client shall verify that the response field is set to rspOk or rspRedirect. If the response field is set to rspOk or rspRedirect, then this is an indication that the new broadcast program is available. Otherwise, exception processing shall be performed according to the SDLs (see Normative Annex A).

10.3.2 SDB Server Initiated Program Select Command Sequence

Figure 10-2 illustrates the procedure for program selection initiated by the SDB Server.

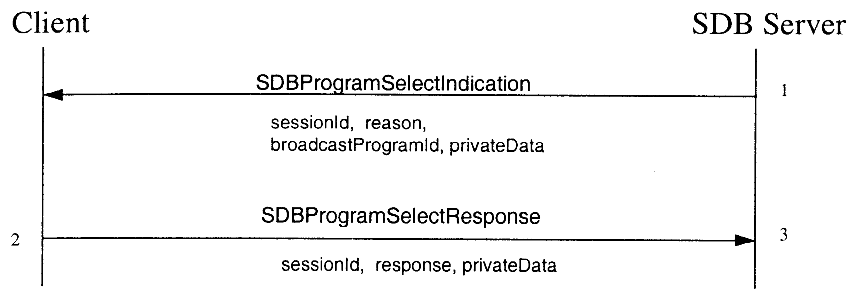

Client
SDB Server
SDBProgramSelectIndication
1
sessionId, reason,
broadcastProgramId, privateData
SDBProgramSelectResponse
2
3
sessionId, response, privateData

Figure 10-2 Scenario for SDB Server Initiated Program Select Sequence

Step 1 SDB Server:

If the SDB Server determines to switch to another broadcast program (e.g., a pre-subscribed Pay-per-View event starts or the subscriber is no more entitled to receive a broadcast program), then the SDB Server shall provide the new broadcast program to the Client. If the network resources are not sufficient to new program, then the SDB Server shall abort the scenario. Otherwise, the SDB Server shall send a SDBProgramSelectIndication to the Client to indicate that a broadcast program will be provided, and start timer tMsg. The sessionId field shall be set to the same value as it was negotiated during the UN-Session-Setup. The broadcastProgramId shall be set to a value which identifies the broadcast program, which is provided to the Client.

If timer tMsg expires before a SDBProgramSelectResponse is received, then the SDB Server may repeat the SDBProgramSelectIndication message and start timer tMsg again. If tMsg expires repeatedly without a SDBProgramSelectResponse message being received, then the SDB Server may initiate an audit with the Network.

Step 2 Client:

On receipt of a SDBProgramSelectIndication, the Client shall verify that the sessionId field refers to an active session. If the Client determines that the sessionId is invalid, then the Client shall respond with a SDBProgramSelectConfirm message with the response field set to rspNoSession.

If the sessionId is valid, the Client shall verify that the SDBProgramSelectIndication message is encoded correctly, especially the length fields are encoded consistently and, if needed, the privateDataByte fields contain enough information to receive the broadcast program. If the Client detects encoding errors in the SDBProgramSelectIndication message, then it shall reply with a SDBProgramSelectResponse message with the response field set to rspFormatError.

If sessionId is valid and encoding of the message is correct, then the Client shall accept the new broadcast program and reply with a SDBProgramSelectResponse message with the response field set to rspOk.

Step 3 SDB Server:

On receipt of SDBProgramSelectResponse the SDB Server shall verify that the response field is set to rspOk. The SDB Server can now consider the new broadcast program to be accepted by the Client.

Note: If the Client does not want to receive the indicated broadcast program, then it may either release the U-N session with the SDB Server or generate a SDBProgramSelectRequest message after the SDBProgramSelectResponse message.

10.4 SDB Reason and Response Codes

10.4.1 SDB Reason Codes

Table 10-9 gives the list of reason codes used within the SDB Channel Change Protocol.

Table 10-9 DSM-CC SDB Reason Codes

Reason Code	Value	Description
rsnOk	0x0000	Indicates that a Broadcast Program has been started as a normal action (e.g. pre-subscribed Pay Per View event starts).
rsnNormal	0x0001	Indicates that a Broadcast Program has been discontinued due to Session Release.
rsnSeEntitlementFailure	0x0002	Indicates that a Broadcast Program has been discontinued due to entitlement failure.
reserved	0x0003 - 0x7FFF	ISO/IEC 13818-6 reserved
private use	0x8000 - 0xFFFF	For private use

10.4.2 SDB Response Codes

Table 10-10 gives the list of response codes used within the SDB Channel Change Protocol.

Table 10-10 DSM-CC SDB Response Codes

Response Code	Value	Description
rspOk	0x0000	Indicates a positive acknowledgment.
rspFormatError	0x0001	Indicates that the condition is due to invalid format (e.g., missing parameter) detected.
rspNoSession	0x0002	Indicates that the either the Client or the Server rejects a SDBProgramSelect command as the referenced session is not established.
rspNoNetworkResource	0x0003	Indicates that the Network does not have enough resources to support the SDB service according to the Server's request.
rspNoServerResource	0x0004	Indicates that the SDB Server does not have enough resources to support the SDB service.
rspEntitlementFailure	0x0005	Indicates that a Broadcast Program could not be delivered to a Client due to entitlement failure.
rspBcProgramOutOfService	0x0006	Indicates that a Broadcast Program cannot be delivered because it is out of service.
rspRedirect	0x0007	Indicates that instead of the requested broadcast program, an alternate program has been provided.
reserved	0x0008 - 0x7FFF	ISO/IEC 13818-6 reserved
private use	0x8000 - 0xFFFF	For private use

10.5 SDB State Machine

Each SDB service session requires a state machine in the Client and in the SDB Server. This subclause gives a high level description of both Client and Server state machines. See Normative Annex A for the SDL diagrams which define the SDB CCP state machines.

10.5.1 SDB State Machine for the Client Side

The following states are defined at the Client side for the Switched Digital Broadcast Service:

Cidle	The SDB service session is not established.
CprogramInactive	The SDB service session is established, but a broadcast program is not expected.
CprogramActive	The SDB service session is established and a broadcast program is expected.
CprogramRequest	The Client has requested a broadcast program and is waiting for an acknowledgment from the SDB Server

The possible state transitions are described in Figure 10-3. The SDL in Normative Annex A provides details on state transitions.

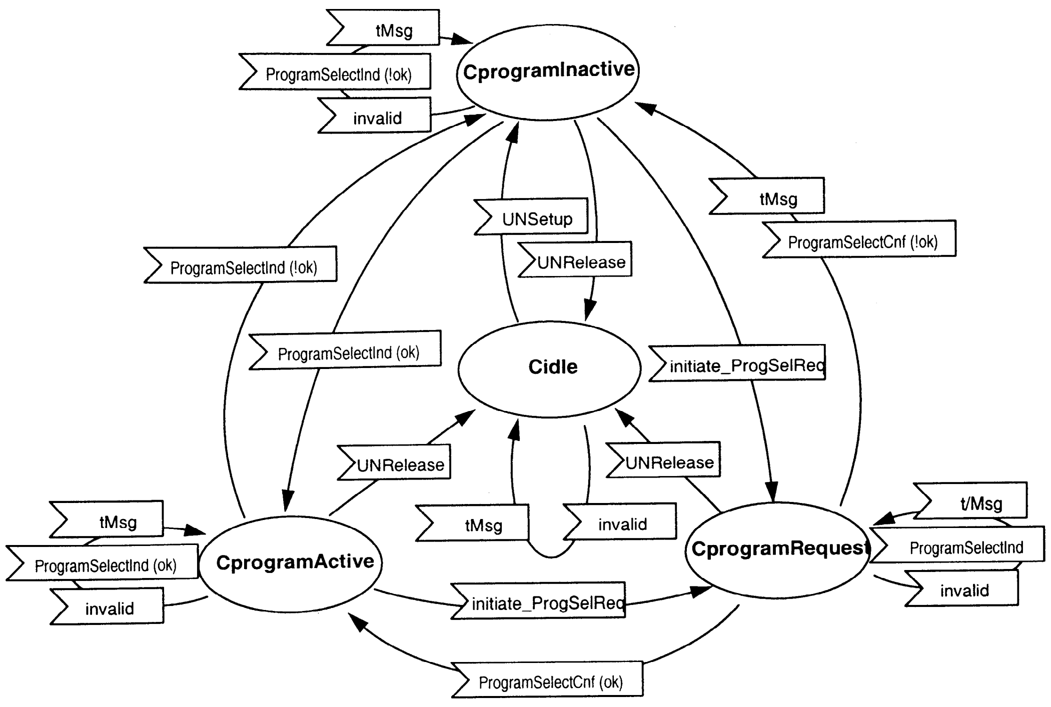


Figure 10-3 State-Event Diagram for Client SDB States

The following internal and external events are defined at the Client side for the SDB State machine.

Internal events:

[initiate-ProgSelReq]	The application requests the state machine to generate a SDBProgramSelectRequest message.
[UNSetup]	The state machine is informed that a SDB service session has been established.
[UNRelease]	The state machine is informed that the SDB service session has been released.
[Tmsg-timeOut]	The state machine is informed that the message response timer tMsg has expired without a response being received.

External events:

ProgramSelectCnf	A SDBProgramSelectConfirm message is received from the SDB Server.
ProgramSelectIndication	A SDBProgramSelectIndication message is received from the SDB Server.

The following conditions are defined for the Client side SDB State Machine:

encoding valid	The entire encoding of a received SDBProgramSelectIndication or a SDBProgramSelectConfirm message is syntactically correct and all information needed to process the event is available within the message.
response == *rspCode*	The response field in a received SDBProgramSelectIndication or a SDBProgramSelectConfirm message is equal to *rspCode.*
noProgram selected	The bpId field contained in a SDBProgramSelect message refers to the "noProgram", i.e., the release of the current broadcast program is requested.
max. retry exceeded	The maximum number of message retransmission attempts has been exceeded already.

The following reactions are defined for the Client side SDB State machine:

SDBProgSelReq	The Client sends a SDBProgramSelectRequest message to the SDB Server.
SDBProgSelRsp: *rspCode*	The Client sends a SDBProgramSelectResponse message to the SDB Server, which contains the response field encoded to *rspCode.*
UNStatusReq: *rsnCode*	The SDB state machine triggers the UN Client state machine to send a UNStatusRequest message to the SRM the reason field encoded to *rsnCode.*
StartTimerTmsg	Start the timer tMsg to check for a response from the SDB Server.
StopTimerTmsg	Stop the timer tMsg again.

Note that a UNStatusRequest message may be used to determine if there is a state inconsistency between Client and SDB Server.

10.5.2 State machine for the SDB Server Side

The following states are defined at the SDB Server side:

SIdle	The SDB service session is not established.
SprogramInactive	The SDB service session is established, but a broadcast program is not being provided.
SprogramActive	The SDB service session is established and a broadcast program is being provided.
SprogramIndication	The SDB Server has requested the Client to switch to a new broadcast program and is waiting for an acknowledgment from the Client.

The possible state transitions are described in Figure 10-4. The SDL in Normative Annex A provides details on state transitions.

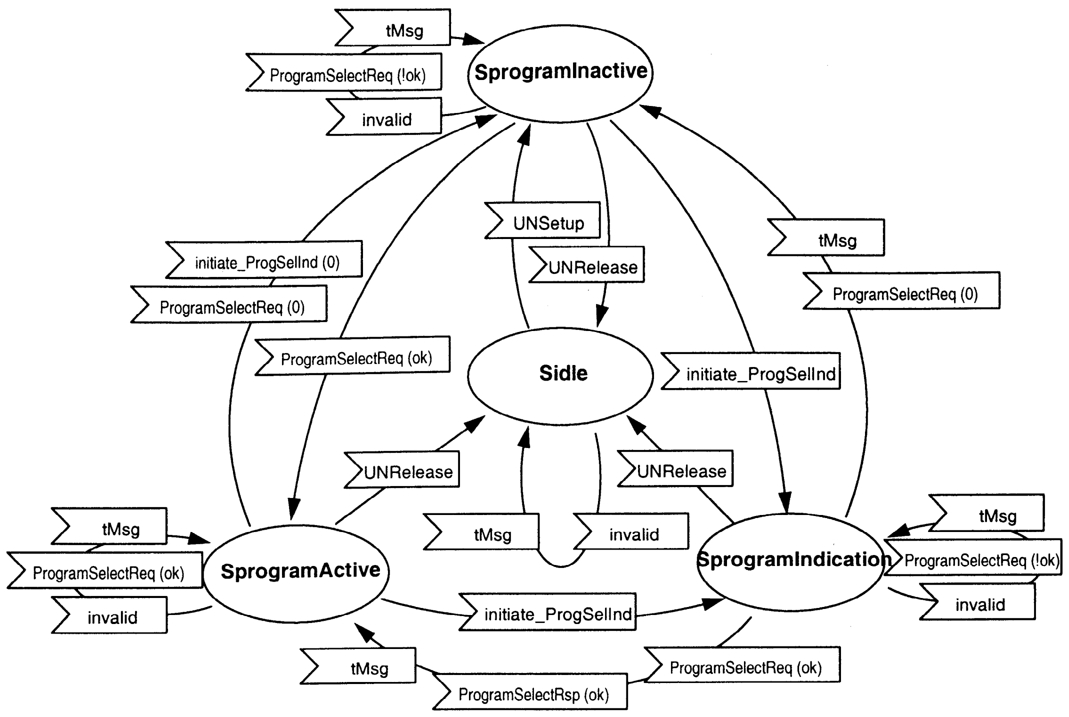


Figure 10-4 State-Event Diagram for Server SDB States

The following internal and external events are defined at the Server side for the SDB Service State Machine.

Internal events:

[initiate-ProgSelInd]	The application requests the state machine to generate a SDBProgram-SelectIndication message and enter new state accordingly.
[UNSetup]	The state machine is informed that a UN-Session has been established for Switched Digital Broadcast Service.
[UNRelease]	The state machine is informed that the SDB UN-Session has been released.
[Tmsg-timeOut]	The operation system informs the state machine that the message response timer tMsg has expired without a response being received.

External events:

ProgramSelectReq	A SDBProgramSelectRequest message is received from the Client.
ProgramSelectRsp	A SDBProgramSelectResponse message is received from the Client.

The following conditions are defined for the SDB Server side SDB State machine:

encoding valid	The entire encoding of a received SDBProgramSelectRequest or a SDBProgramSelectIndication message is syntactically correct.
program available	The SDB Server determines that a requested broadcast program is available, i.e. not out of service.
SDB resources available	The SDB Server has enough internal resources to its disposal to process a SDBProgramSelectRequest message.
net resources available	There are enough network resources available to provide a broadcast program on request of the SDB Server to a Client.
entitlement failed	The SDB Server determines that a Client is not entitled to receive the requested broadcast program.
response == *rspCode*	The response field in a received SDBProgramSelectResponse message is equal to *rspCode*.
noProgram selected	The bpId field contained in a SDBProgramSelect message refers to the "noProgram", i.e., the release of the current broadcast program is requested.
max. retry exceeded	The maximum number of message retransmission attempts has been exceeded already.
transactionId valid	The transactionId value which was contained in a SDBProgramSelectResponse message, matches the expected value.

The following reactions are defined for the SDB Server side SDB State machine:

SDBProgSelInd	The SDB Server sends a SDBProgramSelectIndication message to the Client.
SDBProgSelCnf: *rspCode*	The SDB Server sends a SDBProgramSelectConfirm message to the Client, which contains the response field encoded to *rspCode*.
UNStatusReq: *rsnCode*	The SDB state machine triggers the UN SDB Server state machine to send a UNStatusRequest message to the SRM the reason field encoded to *rsnCode*.
(provide program)	The SDB Server initiates providing the broadcast program to the Client.
(continue program)	The SDB Server continues providing the old broadcast program to the Client.
(stop program)	The SDB Server stops providing any broadcast program to the Client.
StartTimerTmsg	Start the timer tMsg to check for a response from the Client.
StopTimerTmsg	Stop the timer tMsg again.

1) The SDB Server may decide to assign a different channel than the requested one to the Client. In this case, the bpId field in the SDBProgramSelectConfirm message has to be set accordingly.

2) Optionally, the SDB Server may initiate the exchange of UNStatusRequest messages with the SRM to determine if there is a state inconsistency between Client and SDB Server.

3) The SDB Server may use the reason code rsnSeEntitlementFailure to indicate that the Client is no longer entitled to receive the previously displayed BC program channel.

11. U-U Object Carousel

11.1 Introduction

The U-U Application Portability Interface (API) provides Clients a standardized mechanism to access a collection of U-U objects (e.g., Directories, Files, and Streams). For interactive networks, the U-U API is closely coupled to the Service Interoperability interfaces for interoperability purposes. However, the use of the U-U API is not limited to interactive networks and may be used to access objects broadcast in a broadcast network. In order to do so, the Client has to implement a subset of the User-to-User functionality locally and has to access the broadcast network for new object representations when necessary.

By broadcasting U-U objects in a Network, the Broadcast Server may offer simultaneous access to the objects to a large population of Clients. This functionality may be used to support applications that operate in a pure broadcast environment, for load distribution of interactive servers, or for the publication of content of interactive servers.

The U-U Object Carousel protocol defines the transmission of U-U objects in broadcast networks, and defines how U-U objects are encoded on-the-wire so that implementations may offer Clients access to these objects via the U-U API defined in clause 5. The U-U Object Carousel builds on the Data Carousel scenario of the U-N Download protocol defined in clause 7. Figure 11-1 shows the position of the U-U Object Carousel protocol in a complete implementation which includes a Client, the Local Object Implementation, and the protocol stack used to broadcast the objects.

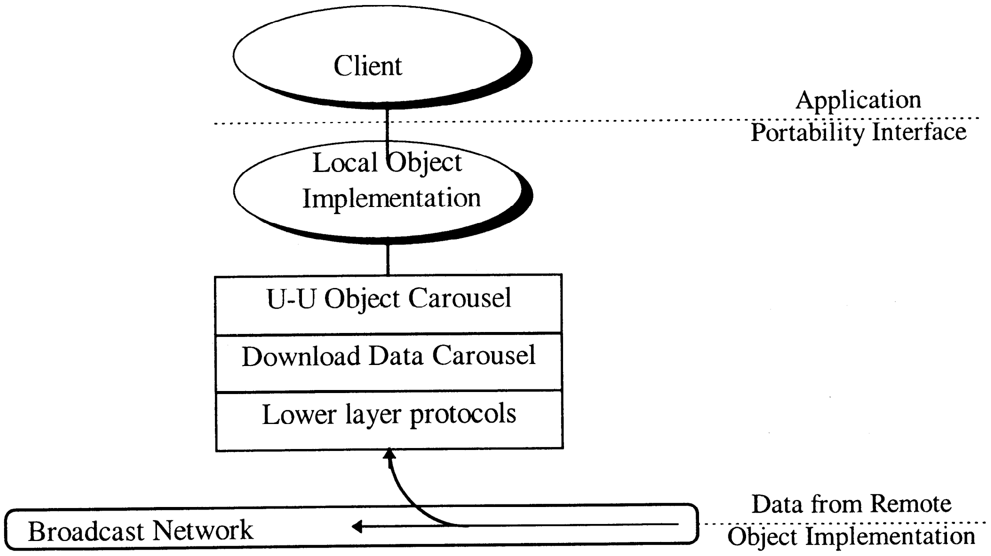

Client
Application Portability Interface
Local Object Implementation
U-U Object Carousel
Download Data Carousel
Lower layer protocols
Broadcast Network
Data from Remote Object Implementation

Figure 11-1 Location of the U-U Object Carousel in a protocol stack

The U-U Object Carousel protocol is compatible with the User-to-User protocol of clause 5 and with the Object Request Broker (ORB) framework as defined by CORBA. Within the User-to-User system environment, a single instance of a U-U Object Carousel represents a particular Service Domain. The Service Domain has a Service Gateway that presents a graph of services names to the Clients. To support this kind of Service Domain, DSM-CC defines an Inter-ORB Protocol that is henceforth referred to as the Broadcast Inter-ORB Protocol (BIOP). The BIOP protocol is network independent and can be applied on top of any type of broadcast network. The network independence is achieved by using the concept of Taps as defined in clause 5. A Tap facilitates a reference to a particular network connection by means of an association tag. The BIOP protocol consists of three parts:

1. BIOP Profile Body definition. BIOP defines a Profile Body which is used to construct a unique object reference for each object that is broadcast in the Network. The BIOP object references use the Inter-operable Object Reference (IOR) format defined by CORBA. BIOP is therefore compatible with any Service Domain that uses the IOR format.
2. BIOP Message Formats. U-U Object Carousels broadcast the U-U objects in so-called BIOP messages. Each supported U-U object is encoded in a dedicated BIOP message which is built from a generic message format. The

messages are described using OMG IDL and are converted to bits-on-the-wire by means of a data representation standard. The default data representation standard for BIOP is Common Data Representation Lite (CDR-Lite) as specified in clause 5.

3. BIOP Transport definitions. To facilitate Clients access to U-U objects in broadcast networks the BIOP messages have to be broadcast repetitively in the broadcast network. The U-U Object Carousel protocol is therefore based on the Data Carousel scenario of the U-N Download protocol, which implements this functionality. The BIOP messages are carried in the Modules of the Data Carousel. The Modules are fragmented into Blocks (as defined by U-N Download) and are transported in DownloadDataBlock() messages. The associated delivery parameters of the Module (moduleSize, blockSize, Time-outs) are specified in the associated DownloadInfoIndication() messages. The U-U Object Carousel uses the DownloadServerInitiate() messages of U-N Download to broadcast the IOR of the Service Gateway of the carousel.

In addition to the BIOP protocol, this clause defines the semantics of the U-U API for U-U Object Carousel. The semantics of the U-U API differ slightly from the semantics of the U-U API for interactive networks because of the broadcast nature of the network. These changed semantics include the of a 20 byte identifier that shall be used to uniquely identify U-U Object Carousels. This identifier is compatible with the E.164 NSAP addresses used in clause 4, User-to-Network Session Messages and clause 5, User-to-User Interface.

Finally, the U-U Object Carousel defines three descriptors that may be used if the U-U Object Carousel is implemented in broadcast networks based on MPEG-2 Transport Streams. The descriptors facilitate the implementation of U-U Object Carousels using multiple MPEG-2 programs.

11.2 Concepts

11.2.1 Supported U-U Objects and Interfaces

The U-U API is based on a number of interfaces which U-U objects may inherit. Within the U-U Object Carousel, four U-U Objects are supported that shall support the following READER interfaces: namely

Table 11-1 Objects supported within the U-U Object Carousel

U-U Object	Supported READER Interfaces
DSM::Directory	Access, Directory
DSM::File	Base, Access, File
DSM::Stream	Base, Access, Stream
DSM::ServiceGateway	Access, ServiceGateway, Directory, Session

The U-U objects are defined in clause 5. In general the semantics of the U-U API will differ slightly from the semantics of the U-U API for interactive networks because of the broadcast nature of the network. For example, with the Stream interface a pause("now") command may freeze the image on screen but can not pause the delivery of the (broadcast) stream. Hence, a subsequent resume("now") command will probably imply that part of the stream content is not shown. Also, the Access interface will return attributes which are set to default values because the broadcast of these attributes is not defined in BIOP. In particular, the Hist_T, Lock_T, and Perms_T structures shall default to the following values:

- Hist_T: Version and DateTime shall have entries all equal to 0.
- Lock_T: readLock shall be 0 and writeLock shall be 0.
- Perms_T: all permission fields shall indicate read-only permission; the a Password string shall be NULL;
- The rAuthData field shall be empty; the allSecure flag shall be zero.

The U-U Object Carousel protocol is extensible to support the broadcasting of general objects. Informative Annex F illustrates this functionality by showing support for enhanced Stream objects that inherit the Event interface.

11.2.2 Service Domain and Service Gateway

Each instance of a U-U Object Carousel represents a Service Domain. Each Service Domain has a globally unique identifier that identifies a particular instance of a carousel. Called the Carousel NSAP address, this identifier is compatible with the E.164 NSAP serverIds used in clause 4, User-to-Network Session Messages and clause 5, User-to-

User Interfaces. The U-U Object Carousel protocol defines the syntax of this identifier to include a 1 byte AFI field, a 1-byte Type field, a 4 byte carousel id (carouselId) field, a 4-byte specifier field, and 10 bytes of private data (see Figure 11-2).

AFI	Type	carouselId	specifier	privateData
1-byte	1-byte	4-byte	4-byte	10-byte

Figure 11-2 Format of Carousel NSAP address

The semantics of the fields of the Carousel NSAP address are as follows:

The **AFI** (authority and format identifier) shall be set to 0x00. This value is defined in ISO 8348 Annex B as NSAP addresses reserved for private use. As such, the rest of the NSAP address fields are available for private definition.

The **type** field shall be set to 0x00 when the Carousel NSAP address points to a U-U Object Carousel. The values in the range 0x01 to 0x7F shall be reserved to ISO/IEC 13818-6. The values in the range 0x80 to 0xFF shall be user private and their use is outside the scope of this part of ISO/IEC 13818.

The **carouselId** is a 32 bit field that uniquely identifies the carousel with a particular network specific scope.

The **specifier** field is the composite of the specifierType field and specifierData() structure that is defined in clause 6, User Compatibility. The specifier (that is, the organization uniquely identified by the specifierData structure) shall specify the syntax and semantics of the private data field in such a way that the Carousel NSAP address becomes an unique identifier of the carousel within the network of the specifier.

The **privateData** is a 10 Byte field. Its use is not defined by this part of ISO/IEC 13818.

The Service Domain has a Service Gateway which provides the root Directory of the content that is broadcast by the U-U Object Carousel. The IOR of the Service Gateway is broadcast by means of DownloadServerInitiate() messages. Along with the IOR of the ServiceGateway, the DownloadServerInitiate() message may carry download Taps that indicate that a non-flow-controlled download should occur as part of Session attach().

The network connection on which the DownloadServerInitiate() message is delivered should be either well-known to Clients (for example, delivered via U-N Config) or Network specific methods should exist that facilitate the resolution of the Carousel NSAP address to a network connection.

11.2.3 Object References

BIOP uses the Inter-operable Object Reference (IOR) format defined by CORBA. The IOR of an object that resides inside the U-U Object Carousel contains the mandatory LiteComponents BIOP::ObjectLocation and DSM::ConnBinder. The syntax of these LiteComponents are defined in clause 5.

The BIOP::ObjectLocation LiteComponent uniquely locates the object in the U-U Object Carousel by means of the triple carouselId, moduleId, and objectKey. The carouselId provides an identifier of the U-U Object Carousel that is unique within a Network specific scope. The carouselId provides a context for the moduleId field which identifies the Module in which the object is broadcast. The objectKey uniquely identifies the object within the Module. Objects can only belong to one U-U Object Carousel.

The BIOP::ConnBinder Component contains a sequence of Taps that identify the delivery of the object in the broadcast network. At least one Tap shall point to a DownloadInfoIndication() message that contain the module delivery parameters of the Module (such as size, version, blocksize, time-outs) in which the object is delivered. The DownloadInfoIndication() messages point subsequently to the network connections that are used to broadcast the Modules.

11.2.4 Transport of BIOP Messages

BIOP messages are transported in Modules of DSM-CC Data Carousels. Multiple BIOP messages may be carried in one Module. The Modules of the Data Carousel are fragmented into Blocks. These Blocks are transported in DownloadDataBlock() messages (described in clause 7). Figure 11-3 illustrates the applied encapsulation and fragmentation methods.

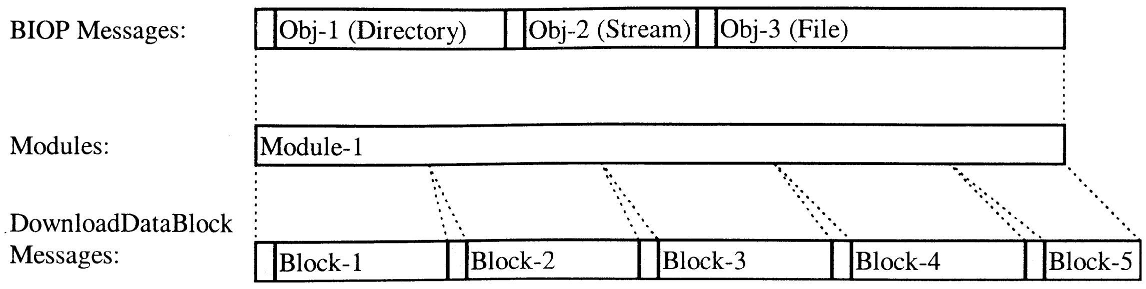


Figure 11-3 Encapsulation of BIOP Messages in Module

11.2.5 Module Delivery Parameters

Modules are broadcast in the broadcast network. The parameters that describe the delivery of a particular Module in the broadcast network are called the module delivery parameters. In U-U Object Carousels, the module delivery parameters are conveyed in DownloadInfoIndication() messages. The module delivery parameters contain all transport parameters that are necessary to acquire the Module from the broadcast network. This information consists of the size and version of the Module, the applied block size, and various time-out values. In addition, the DownloadInfoIndication() message contains the information about the Tap (described below) on which the Modules are being broadcast. Each DownloadInfoIndication() messages can describe multiple Modules.

11.2.6 Taps

The BIOP protocol is network independent and is thus applicable for any type of broadcast network. The network independence is achieved by using the concept of Taps as defined in clause 5. A Tap facilitates a reference to a particular network connection by means of an association tag. BIOP defines the use of the following TapUse values (TapUse values are defined in clause 5).

1. BIOP_DELIVERY_PARA_USE: The Inter-operable Object Reference (IOR) of an object includes such Taps to indicate the connections at which the DownloadInfoIndication() messages are broadcast that describe the module delivery parameters of the Module in which the object is conveyed. The syntax and the semantics of the selector field of these Taps are specified in this clause.
2. BIOP_OBJECT_USE: Used in the DownloadInfoIndication() messages which convey the module delivery parameters of the Modules to indicate the connections on which the Modules are broadcast. These Taps are optionally carried in the IOR of objects to allow fast acquisition procedures. The syntax and the semantics of the selector field of these Taps are specified in this clause.
3. BIOP_PROGRAM_USE, BIOP_ES_USE: The Stream object contains Taps to indicate the stream(s) that are associated with the object. The syntax and the semantics of the selector field of these Taps are specified in subclause 11.3.2.4.
4. STR_STATUS_AND_EVENT_USE, STR_EVENT_USE, STR_STATUS_USE, NPT_USE: The Stream object contains Taps to indicate the various sub-streams that are associated with the object. The syntax and the semantics of the selector field of these Taps are specified in subclause 11.3.2.4.
5. DOWNLOAD_CTRL_DOWN_USE, DOWNLOAD_DATA_DOWN_USE: The DownloadServerInitiate() messages that carry the IOR of the ServiceGateway use these Taps to indicate the various connections at which DownloadInfoIndication() messages are broadcast that described the Modules that should be download. The syntax and the semantics of the selector field of these Taps are specified in this clause.

In the course of resolving an object, Clients have to associate the Taps to the connections or to the broadcast network. Clients need, therefore, an association table that makes the association between the Taps and the connections of the broadcast network. The syntax and communication of such association tables is outside the scope of this part of ISO/IEC 13818. However, to support the implementation of U-U Object Carousels in broadcast networks based on MPEG-2 Transport Streams, this part of ISO/IEC 13818 does define three descriptors that can be inserted into MPEG-2 PSI (as defined in ISO/IEC 13818-1). In other cases, the association tables may be communicated to the Clients by using User-to-Network messages or by using network specific methods.

11.3 Broadcast Inter ORB Protocol

11.3.1 Inter-operable Object Reference (IOR)

11.3.1.1 Profile Body Definition

BIOP uses the Inter-operable Object Reference format (IOR) defined by CORBA. An IOR contains Profile Bodies that encapsulate all the basic information that a particular protocol stack needs to identify the object. A single Profile Body holds enough information to drive a complete invocation of the object using that protocol. The syntax of the IOR is specified in clause 5.

The Profile Body that shall be used for objects that are broadcast in the U-U Object Carousel is the BIOP Profile Body. The BIOP Profile Body is defined in clause 5 and consist of the DSM::LiteComponentProfile structure. The BIOP Profile Body is labeled by the ProfileId TAG_BIOP. The syntax of the DSM::LiteComponentProfile and TAG_BIOP ProfileId are defined in clause 5.

The BIOP Profile Body shall contain at least the LiteComponents BIOP::ObjectLocation and DSM::ConnBinder. The semantics of the LiteComponents are described below.

11.3.1.1.1 Object Location Component

The BIOP::ObjectLocation component uniquely locates the object in the broadcast network. The presence of the LiteComponent BIOP::ObjectLocation in the BIOP Profile Body is identified by the component tag TAG_BIOP_ObjectLocation. The BIOP::ObjectLocation structure shall directly be inserted into the component_data field of the LiteComponent (i.e., by using the rules for CDR-Lite encapsulations). A BIOP Profile Body shall contain exactly one component of type BIOP::ObjectLocation.

The semantics of the fields of the BIOP::ObjectLocation structure are described below:

The **version** describes the version of BIOP::ObjectLocation protocol. The major version for ISO/IEC 13818-6, IS, is 1; the minor version is 0.

The **carouselId** field provides a context for the moduleId field. It uniquely identifies the carousel within the broadcast network and allows the distributed implementation of the carousel.

The **moduleId** identifies the module in which the object is conveyed within the carousel.

The **objectKey** identifies the object within the module in which it is broadcast. This field contains a length field followed by a series of bytes (see clause 5 for the IDL definition of objectKey). The Server supplies the objectKey field value.

11.3.1.1.2 ConnBinder Component

The DSM::ConnBinder component contains a number of Taps. The presence of the LiteComponent DSM::ConnBinder in the BIOP Profile Body is identified by the component tag TAG_ConnBinder. The DSM::ConnBinder structure shall be inserted directly into the component_data field of the LiteComponent. A BIOP Profile Body shall contain exactly one component of type DSM::ConnBinder. The DSM::ConnBinder shall contain at least one Tap (with a TapUse value of BIOP_DELIVERY_PARA_USE) that points to a DownloadInfoIndication() message that conveys the Module delivery parameters. Optionally, Taps may be present with a TapUse value of BIOP_OBJECT_USE.

The semantics of the fields of a Tap with a TapUse value of BIOP_DELIVERY_PARA_USE are described below:

- The use field indicates the use of the Tap. The value of this field shall be BIOP_DELIVERY_PARA_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.

- The assocTag identifies the connection on which the DownloadInfoIndication() message is broadcast. The communication to the Clients of the associations between the used association_tag values and the used connections is outside the scope of this part of ISO/IEC 13818.
- The selector field shall contain a selectorType of value 0x01 and the DSM::MessageSelector structure. The DSM::MessageSelector structure contains a transactionId field and a timeout field as defined in clause 5. The value of the transactionId field shall be set to the transactionId of the DownloadInfoIndication() message that contains the module delivery parameters. The timeout field shall indicate the time-out period in microseconds to be used to time out the acquisition of the DownloadInfoIndication() message.

The semantics of the fields of a Tap with a TapUse value of BIOP_OBJECT_USE are described below:

- The use field indicates the use of the Tap. The value of this field shall be BIOP_OBJECT _USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the broadcast channel on which the Modules are broadcast. The communication to the Clients of the associations between the used association_tag values and the used connections is outside the scope of BIOP.
- The selector field shall be of 0 length.

11.3.2 Message Set Definition

BIOP conveys the U-U objects Directory, File, Stream, and ServiceGateway in messages. These BIOP messages are all derived from a generic object message format. BIOP defines four messages, namely:

1. a Directory message. The Directory message is used to convey a U-U object of the type Directory. It contains references to other Directory, File, Stream, and ServiceGateway objects.
2. a File message. The File message is used to convey a U-U object of the type File. It just contains just the data of the file data.
3. a Stream message. The Stream message is used to convey a U-U object of the type Stream. It contains a reference to the stream in the broadcast network.
4. a ServiceGateway message. The ServiceGateway message is used to convey a U-U object of the type ServiceGateway. It contains references to other Directory, File, Stream, and ServiceGateway objects. There shall be no more than one ServiceGateway message in one U-U Object Carousel.

11.3.2.1 Generic Object Message Format

The generic object message format is used to encapsulate the data and attributes of a single object. The message consists of a header, a sub-header, and a message body. The syntax and semantics of the generic object message format are defined below.

```
module BIOP {
    typedef unsigned long ServiceID;
    struct ServiceContext {
        ServiceID                 context_id;
        sequence<octet,65535>     context_data;
    };
    struct MessageHeader {
        char              magic[4];
        Version           biop_version;
        boolean           byte_order;
        octet             message_type;
        unsigned long     message_size;
    };
    struct MessageSubHeader {
        sequence<octet,255>               objectKey;
        sequence<octet>                   objectKind;
        sequence<octet,65535>             objectInfo;
        sequence <ServiceContext,255>     serviceContextList;
    };
    //
    struct GenericObjectMessage {
        MessageHeader                     messageHeader;
        MessageSubHeader                  messageSubHeader;
        sequence<octet>                   messageBody;
    };
};
```

The semantics of the fields of the BIOP::MessageHeader are as follows:

The **magic** field identifies the BIOP message. The value of this field is always "BIOP" encoded in ISO Latin-1.

The **biop_version** field contains the version number of the BIOP protocol used in this message. The version number applies to the structure and encoding of the message only. Therefore it is not equivalent with the version of the BIOP::ObjectLocation, though it has the same structure. The major version of this specification is 1; the minor version is 0.

The **byte_order** field indicates the byte ordering used for the following subsequent elements of the message (including message_size). A value of FALSE (0) indicates big-endian byte ordering, and TRUE (1) indicates little endian ordering.

The **message_type** field indicates the type of the message. The value of this field shall be set to 0x00. The values in the range from 0x01 to 0xFF are reserved for ISO/IEC 13818-6.

The **message_size** field contains the length of the message following the message header in bytes. This count includes any alignment gaps that may be introduced by the data encoding standard.

The semantics of the fields BIOP::MessageSubHeader are as follows:

The **objectKey** field identifies the object that is conveyed in this message. It is identical to the objectKey that is present in the BIOP::ObjectLocation component of the IOR of the object. The value of the objectKey is only meaningful to the Broadcast Server and is not interpreted by the Client.

The **objectKind** field identifies the kind of the object that is conveyed in this message. It is identical to the Kind string that is present in the IOR of the object. The value of the objectKind defines the syntax and semantics of the objectInfo field and the messageBody field.

The **objectInfo** field contains some or all of the attributes of this object. The syntax and semantics of this field are dependent of the value of the objectKind field.

The **service_context** field contains ORB service data that is passed from the Broadcast Server to the Client. The use of this field is outside the scope of this part of ISO/IEC 13818. This field can be used by emerging specifications that require service-specific context information to be passed by each acquisition.

Note: The BIOP ServiceId should not be confused with the DSM-CC ServerId. The ServiceId identifies some application contextual information, whereas the DSM-CC ServerId is a Server identifier as specified by the U-N Message protocols.

The semantics of the messageBody field are as follows:

The **messageBody** field contains the header, sub-header, and file data of the object. The syntax and semantics of this field are dependent of the value of the objectKind field.

To instantiate the generic object message format into a dedicated object message the semantics of the objectInfo and messageBody fields have to be defined. The objectInfo field is intended to carry the attributes of the object, while the messageBody is intended to carry the data of the object. As described later, the instantiated Directory message has also an objectInfo field that may be used to carry the attributes of the bound object. By carrying the proper attributes in that field the quick browsing through Directories and object attributes is supported. Hence the instantiation of the generic object message format for a particular objectKind should also specify which attributes are carried in the objectInfo field of the (parent) Directory message.

In the following the generic message format is instantiated for the Directory object, the File object, the Stream Message, and the ServiceGateway object.

11.3.2.2 Directory Message Format

The BIOP Directory message is an instantiation of the generic object format. The following rules define this instantiation.

1. The objectKind field shall contain the string "DSM::Directory" or "dir".
2. The Access attributes of the Directory object are not encapsulated in the objectInfo field of the Directory message and are neither encapsulated in the objectInfo field of the parent Directory object (if any). Hence, the objectInfo field shall be empty.
3. The messageBody field shall contain the BIOP::DirectoryMessageBody structure. The syntax and semantics of the BIOP::DirectoryMessageBody are defined below.

```
Module BIOP {
    typedef string<255> Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    //
    typedef sequence<NameComponent,255> Name;
    typedef octet BindingType;
    const BindingType nobject = 1;
    const BindingType ncontext = 2;
    const BindingType composite = 3;
    //
    struct Binding {
        Name                        bindingName;
        octet                       bindingType;
        IOP::IOR                    objectRef;
        sequence <octet,65535>      objectInfo;
    };
    typedef sequence <Binding,65535> DirectoryMessageBody};
};
```

The BIOP::DirectoryMessageBody structure consists of a loop of Bindings. A binding correlates an object name (i.e. bindingName) to an IOR and provides additional information about the object. The IOR must include the BIOP Profile Body when the referenced object belongs to the Carousel.

The semantics of the fields of the BIOP::DirectoryMessageBody are defined below:

The **bindingName** field (i.e. id and kind) contains the path specification of the object (as defined by CosNaming).

The **bindingType** field indicates the type of the object binding. Binding can either be of type 'nobject' when the name is not bound to a Directory or 'ncontext' when the name is bound to a Directory object. 'composite' is defined for compatibility with the User-to-User Composite bindings, as described in clause 5.

The **objectRef** field contains the IOR of the object.

The **objectInfo** field may contain some of the attributes of the bound object as well as user private information about the object. If attributes of the bound object are carried in this field they shall be the first structures that are encapsulated in this field. The use of this field for user private purposes is outside the scope of this part of ISO/IEC 13818.

11.3.2.3 File Message Format

The BIOP File message is an instantiation of the generic object format. The following rules define this instantiation.

1. The objectKind field shall contain the string "DSM::File" or "fil".
2. The Access attributes and the DSM::File::Content attribute of the File object are not encapsulated in neither the objectInfo field of the File message or the objectInfo field of the (parent) Directory message. The DSM::File::ContentSize attribute shall be inserted at the beginning of both the objectInfo field of the File message and the objectInfo field of the (parent) Directory message.
3. The messageBody field shall contain the BIOP::FileMessageBody structure. The syntax and semantics of the BIOP::FileMessageBody are defined below.

```
module BIOP {
    typedef        sequence <octet>           FileMessageBody;
};
```

The **FileMessageBody** contains the file data as an octet stream.

11.3.2.4 Stream Message Format

The BIOP Stream message is an instantiation of the generic object format. The following rules define this instantiation.

1. The objectKind field shall contain the string "DSM::Stream" or "str".
2. The Access attributes of the Stream object are not encapsulated in either the objectInfo field of the File message nor the objectInfo field of the (parent) Directory message. The DSM::Stream::Info_T attribute shall be inserted at the beginning of the objectInfo field of the File message.
3. The messageBody field shall contain the BIOP::StreamMessageBody structure. The syntax and semantics of the BIOP::StreamMessageBody are defined below.

```
module BIOP {
    struct StreamMessageBody {
          sequence <Tap,255>                stream;
    };
};
```

The BIOP::StreamMessageBody consists a sequence of Taps that are associated with the stream object. The semantics of the stream field are defined below.

The **stream** field contains one or more Taps that are associated with this stream object. Regarding the content of the stream, either one or more Taps are present with a TapUse value of BIOP_ES_USE or one Tap is present with a TapUse value of BIOP_PROGRAM_USE. In the first case, the stream consists of a number of elementary streams, while in the second case the stream consists of an MPEG-2 program.

The semantics of the fields of a Tap that point to an elementary stream are described below:

- The use field indicates the use of the Tap. The value of this field shall be BIOP_ES_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the connection on which the elementary stream is broadcast. The communication to the Clients of the associations between the used association_tag values and the used connections is outside the scope of this part of ISO/IEC 13818.
- The selector field shall be empty.

The semantics of the fields of a Tap that point to an MPEG-2 program are described below:

- The use field indicates the use of the Tap. The value of this field shall be BIOP_PROGRAM_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the connection on which the MPEG-2 Program Map Table is broadcast.
- The selector field shall be empty.

The **stream** field may also contain one or several additional Taps referring to NPT, Event and Mode descriptors.

The semantics of the fields of a Tap pointing to a NPT descriptor are described below:

- The use field indicates the use of the Tap. The value of this field shall be STR_NPT_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the connection on which the NPT descriptor is broadcast.
- The selector field shall be empty.

The semantics of the fields of a Tap pointing to Stream Mode and Stream Event descriptors are described below:

- The use field indicates the use of the Tap. The value of this field shall be STR_STATUS_AND_EVENT_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the connection on which the Stream Mode and the Stream Event descriptors are broadcast.
- The selector field shall be empty.

The semantics of the fields of a Tap pointing to a Stream Event descriptor are described below:

- The use field indicates the use of the Tap. The value of this field shall be STR_EVENT_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the connection on which the Stream Event descriptor is broadcast.
- The selector field shall be empty.

The semantics of the fields of a Tap pointing to a Stream Mode descriptor are described below:

- The use field indicates the use of the Tap. The value of this field shall be STR_STATUS_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the connection on which the Stream Mode descriptor is broadcast.
- The selector field shall be empty.

11.3.2.5 Service Gateway Message Format

The BIOP Service Gateway message is an instantiation of the generic object format. The following rules define this instantiation.

1. The objectKind field shall contain the string "DSM::ServiceGateway" or "srg".
2. The Access attributes of the Service Gateway object are not encapsulated in the objectInfo field of the ServcieGateway message.
3. The messageBody field shall contain the BIOP::DirectoryMessageBody structure. The syntax and semantics of the BIOP::DirectoryMessageBody are defined in subclause 11.3.2.2.

11.3.3 Transport Definitions

11.3.3.1 BIOP Messages

BIOP messages are transported in Modules of a DSM-CC Data Carousel. Modules may convey multiple non-fragmented BIOP messages. The start of a BIOP message shall coincide with the start of the Module. The Modules of the Data Carousel are fragmented into Blocks. These Blocks are transported in DownloadDataBlock() messages.

The following semantics and constraints are imposed on the transport of the Blocks in the DownloadDataBlock() messages:

The downloadId field of the DownloadDataBlock() messages shall have the same value as the carouselId field of the U-U Object Carousel.

The moduleId field of the DownloadDataBlock() messages shall have the same value as the moduleId field of the U-U Object Carousel.

The moduleVersion field of the DownloadDataBlock() messages shall have the same value as the moduleVersion field of the DownloadInfoIndication() message that describes this Module.

11.3.3.2 Module Delivery Parameters

The delivery parameters of the module in the broadcast network are conveyed in a DownloadInfoIndication() message. One DownloadInfoIndication() message can convey the module delivery parameters of multiple Modules of the same U-U Object Carousel. The following semantics apply on the fields of the DownloadInfoIndication() message:

The **transactionId** field shall have the same value as the **transactionId** value encapsulated in the selector of the BIOP_DELIVERY_PARA_USE Taps of the IORs of the objects that are carried in Modules described in this message.

The **downloadId** field shall have the same value as the **downloadId** field of the DownloadDataBlock() messages which carry the Blocks of the Modules described in this message. Consequently, the value of this field shall be equal to the carouselId of the U-U Object Carousel.

The **blockSize** field contains the block size of all the DownloadDataBlock() messages which convey the Blocks of the Modules described in this message.

The **windowSize, ackPeriod, tCDownloadWIndow,** and **tCDownloadScenario** fields are not used and are set to zero.

The **compatibilityDescriptor()** field is not used and has a zero length.

The **moduleId, moduleSize,** and **moduleVersion** fields have the same semantics as with the Download Data Carousel scenario as described in clause 7.

The **moduleInfoLength** field defines the length in bytes of the moduleInfo field for the described module.

The **moduleInfoBytes** field shall contain the BIOP::ModuleInfo structure. The BIOP::ModuleInfo structure provides additional delivery parameters and the Taps that are used to broadcast the Modules in the network. The syntax and semantics of the BIOP::ModuleInfo structure are shown below.

```
module BIOP {
    struct ModuleInfo {
        unsigned long                   moduleTimeOut;
        unsigned long                   blockTimeOut;
        unsigned long                   minBlockTime;
        sequence <DSM::Tap,255>         Taps;
        sequence <octet,255>            userInfo;
    };
};
```

The **moduleTimeOut** field gives the time out value in microseconds that may be used to time out the acquisition of all Blocks of the Module.

The **blockTimeOut** field gives the time out value in microseconds that may be used to time out the reception of the next Block of the after a Block has been acquired.

The **minBlockTime** field indicates the minimum time period that exists between the delivery of two subsequent Blocks of the described Module. Clients may use this value to adjust their acquisition procedures for optimization purposes.

The **Taps** field of BIOP::ModuleInfo shall contain at least one Tap with the TapUse value of BIOP_OBJECT _USE. This Tap shall point to the network connection on which the Modules are broadcast. The semantics of the fields of this Tap are described below.

The **userInfo** field of BIOP::ModuleInfo is not specified by this part of ISO/IEC 13818. In general, this field may encapsulate additional information that is necessary to describe the delivery of the Module in the network.

The **privateDataLength** and **privateDataByte** fields have the same semantics as the Download Data Carousel scenario does, as described in clause 7.

11.3.3.3 IOR of Service Gateway

The IOR of the Service Gateway is broadcast by means of DownloadServerInitiate() messages. The use of the DownloadServerInitiate() messages for the carriage of the IOR of the ServiceGateway is such that U-N Download (non-flow controlled) can be employed as a part of the Session attach() functionality. In particular, the DownloadServerInitiate() messages may also contain Taps (with TapUse values set to either DOWNLOAD_CTRL_DOWN_USE or DOWNLOAD_DATA_DOWN_USE) that point to network connections on which the DownloadInfoIndication() and DownloadDataBlock() messages are broadcast. that contains the descriptions of the modules and modules themselves that have to be downloaded.

The following semantics apply on the fields of the DownloadServerInitiate() message:

The **serverId** field shall contain the 20 byte Carousel NSAP address of the U-U Object Carousel. The Carousel Specifier is defined in subclause 11.2.2.

The **compatibilityDescriptor()** field is not used and has a zero length.

The **privateDataLength** field of the DownloadServerInitiate() message defines the length in bytes of the privateDataByte fields that follow this field.

The data in the **privateDataByte** field of the DownloadServerInitiate() message shall contain the BIOP::ServiceGatewayInfo structure. The syntax and semantics of the BIOP::ServiceGatewayInfo structure are defined below:

```
module BIOP {
    struct ServiceGatewayInfo {
        IOP::IOR                        objectRef;
        sequence <DSM::Tap,255>         Taps;
        sequence <ServiceContext,255>   serviceContextList;
        sequence <octet,65535>          userInfo;
    };
};
```

The **objectRef** field contains the IOR of the ServiceGateway.

The **Taps** field shall contain zero or more Taps which have either a DOWNLOAD_CTRL_DOWN_USE value or a DOWNLOAD_DATA_DOWN_USE value. The Taps with a TapUse of DOWNLOAD_CTRL_DOWN_USE shall point to the network connections on which the DownloadInfoIndication() messages are broadcast that describe the modules that should be download before proceeding with the ServiceGateway attach() functionality. The Taps with a TapUse of DOWNLOAD_DATA_DOWN_USE shall point to the network connections on which the DownloadDataBlock() messages are broadcast that carry the Modules that should be downloaded before proceeding.

The semantics of the fields of a Tap with a TapUse value of DOWNLOAD_CTRL_DOWN_USE are described below:

- The use field indicates the use of the Tap. The value of this field shall be DOWNLOAD_CTRL_DOWN_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the connection on which the DownloadInfoIndication() messages are broadcast. The communication to the Clients of the associations between the used association_tag values and the used connections is outside the scope of this part of ISO/IEC 13818.
- The selector field shall contain a selectorType of value 0x01 and the DSM::MessageSelector structure. The BIOP::MessageSelector structure contains a transactionId field and a timeout field as defined in clause 5. The value of the transactionId field shall be set to the transactionId of the DownloadInfoIndication() message that contains the module descriptions. The timeout field shall indicate the time-out period in microseconds to be used to time out the acquisition of the DownloadInfoIndication() message.

The semantics of the fields of a Tap with a TapUse value of DOWNLOAD_CTRL_DOWN_USE are described below:

- The use field indicates the use of the Tap. The value of this field shall be DOWNLOAD_DATA_DOWN_USE.
- The value of the id field is not defined by this part of ISO/IEC 13818.
- The assocTag identifies the connection on which the DownloadDataBlock() messages are broadcast. The communication to the Clients of the associations between the used association_tag values and the used connections is outside the scope of this part of ISO/IEC 13818.
- The selector field shall be empty.

The userInfo field of BIOP::ServiceGatewayInfo is not defined in this part of ISO/IEC 13818 and is user-private.

11.4 MPEG-2 Descriptors

The U-U Object Carousel protocol is network independent and is applicable for any type of broadcast network. Network independence is achieved by using the Tap concept of clause 5. A Tap facilitates a reference to a particular network connection by means of an association tag. In the course of resolving an object, Clients have to associate the Taps to broadcast connections of the network. Clients need, therefore, to maintain the associations between the Taps and the connections of the broadcast network.

When U-U Object Carousels are used on top of broadcast networks which implement the DSM-CC User-to-Network Session Protocol, the associations can be delivered to the Client by means of the Session Setup sequence.

When U-U Object Carousels are used on top of broadcast networks which use MPEG-2 Transport Streams, but do not provide the association capability of a DSM-CC U-N session, additional functionality is required. In particular, mechanisms are necessary that facilitate

1. the association of a MPEG-2 program, via the Transport Stream Program Map Table (PMT), with a U-U Object Carousel,
2. the association of a DSM-CC Tap with the Transport Stream packet identifier (PID) of the MPEG-2 bit stream which is transporting the Service Gateway,
3. Client local object mapping of the PID on which the IOR of the Service Gateway is broadcast, and
4. the distributed implementation of a U-U Object Carousel on top of multiple MPEG-2 programs.

In order to provide the above mechanisms, three U-U Object Carousel descriptors are defined within the MPEG-2 Systems descriptor_tag table. Table 11-2 contains the DSM-CC descriptor_tag assignments and, for reference purposes, lists the assignment space for the MPEG-2 Systems defined values. For details of the MPEG-2 Systems defined values, see ISO/IEC 13818-1, Table 2-39, Program and program element descriptors. Note also that DSM-CC defines other descriptor tags in this space.

Table 11-2 MPEG-2 Systems descriptor_tag assignments for DSM-CC

descriptor_tag	TS	PS	DSMCC Section Type
0-18	n/a	n/a	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 defined
19	X	X	carousel_identifier_descriptor
20	X	X	association_tag_descriptor
21	X	X	deferred_association_tags_descriptor
22	X	X	ISO/IEC 13818-6 reserved
23	X	X	DSM-CC NPT Reference descriptor (see clause 8, Stream Descriptors)
24	X	X	DSM-CC NPT Endpoint descriptor (see clause 8, Stream Descriptors)
25	X	X	DSM-CC Stream Mode descriptor (see clause 8, Stream Descriptors)
26	X	X	DSM-CC Stream Event descriptor (see clause 8, Stream Descriptors)
27-63	n/a	n/a	ITU-T Rec. H.222.0 | ISO/IEC 13818-1 defined
64-255	n/a	n/a	User Private per ITU-T Rec. H.222.0 | ISO/IEC 13818-1

These program descriptors may be used when U-U Object Carousels are implemented on top of broadcast networks which use MPEG-2 Transport Streams and implement the DSMCC_section syntax (defined in clause 9), a backwards compatible extension to the MPEG-2 Transport Stream private_section syntax (specifically, that DownloadDataBlock() messages and DownloadInfoIndication() messages are transported by means of DSMCC_sections).

11.4.1 Carousel identifier descriptor

The carousel identifier descriptor facilitates the association between a MPEG-2 program and a U-U Object Carousel. The intended location of the carousel_identifier_descriptor() is the first descriptor loop within the PMT of the MPEG-2 program. The syntax and semantics of the carousel_identifier_descriptor() are described below:

Table 11-3 carousel_identifier_descriptor

Syntax	No. of bits	Mnemonic
carousel_identifier_descriptor() {		
descriptor_tag	**8**	**uimsbf**
descriptor_length	**8**	**uimsbf**
carousel_id	**32**	**uimsbf**
for (n=0;n<N;n++) {		
private_data_byte	**8**	**uimsbf**
}		
}		

The **descriptor_tag** field is an 8-bit field. For the carousel_identifier_descriptor, the value shall be 19 (decimal).

The **descriptor_length** field specifies the length of the descriptor in bytes.

The **carousel_id** field is a 32 bit field. Its value shall be identical to the carouselId of the U-U Object Carousel.

The **private_data_byte** field is user private and is not specified by this part of ISO/IEC 13818.

11.4.2 Association tag descriptor

The association tag descriptor facilitates the association between an association_tag and an MPEG-2 elementary/PSI bit stream. In the case of the U-U Carousel, it may be used to identify the streams on which download data (e.g., ServiceGateway) are transported. To relate a bit stream (and hence its PID) to a particular association_tag value, the Broadcast Server may insert the association_tag descriptor in the program descriptor loop of the bit stream.

The syntax and semantics of the association_tag_descriptor are described below:

Table 11-4 association_tag_descriptor

Syntax	No. of bits	Mnemonic
association_tag_descriptor() {		
descriptor_tag	8	**uimsbf**
descriptor_length	8	**uimsbf**
association_tag	16	**uimsbf**
use	16	**uimsbf**
selector_byte_length	8	**uimsbf**
for (n=0;n<N1;n++) {		
selector_byte	8	**uimsbf**
}		
for (n=0;n<N2;n++) {		
private_data_byte	8	**uimsbf**
}		
}		

The **descriptor_tag** field is an 8-bit field. For the association_tag_descriptor, the value shall be 20 (decimal).

The **descriptor_length** field specifies the length of the descriptor in bytes.

The **association_tag** field contains a 16-bit value that identifies uniquely a related group of resources in order to assist in end-to-end correlation of these resources. In the case of the U-U Carousel, it may be used to identify the PID of a bit stream which is transporting download data. The Broadcast Server shall ensure the uniqueness of the association_tag within the scope of the MPEG-2 programs that are used to implement the U-U Object Carousel.

The **use** field indicates the contents of the bit stream in which the association_tag descriptor is contained. The value of the use field shall specify the syntax and semantics of the selector_byte field which follows. Table 11-5 contains the assigned values of the use field. If the use value equals 0x0000, this indicates that DownloadServerInitiate messages which carry the IOR of the Service Gateway are contained in the bit stream identified by this PID. In this case, the data in the selector_byte fields shall contain the fields that are shown Table 11-6. If the use value equal 0x0001, this indicates that general object carousel data are contained in the bit stream, and selector_byte shall be set to 0.

Table 11-5 assigned values for the use field

value of use field	Description	selector_byte field content
0x0000	ServiceGateway	transaction_id, timeout
0x0001	General Object Carousel Data	0
0x0002-0x00FF	reserved for ISO/IEC 13818-6	reserved for ISO/IEC 13818-6
0x0100-0xFFFF	user private	user private

The **selector_byte_length** field defines the length in bytes of the following selector_byte fields.

The meaning of **selector_byte** field is dependent upon the value of the use field. In the case when the use field has the value of 0x0000, the data in the selector_byte fields shall contain the fields that are shown Table 11-6.

Table 11-6 transaction_id and timeout fields

transaction_id	32	uimsbf
timeout	32	uimsbf

The semantics of the transaction_id and timeout fields are as follows.

The value of the **transaction_id** field shall correspond to the transaction_id of the DownloadServerInitiate() message that conveys the IOR of the Service Gateway of the U-U Object Carousel. If this value is set to 0xFFFFFFFF, then this indicates to the receiver that the transaction_id value is not known.

The **timeout** field shall indicate the time-out period in microseconds that may be used to time out the acquisition of the DownloadServerInitiate() message. If this value is set to 0xFFFFFFFF, then this indicates to the receiver that the timeout value is not known.

The **private_data_byte** fields is user private and is not specified by this part of ISO/IEC 13818.

11.4.3 Deferred association tags descriptor

A U-U Object Carousel may use multiple bit streams (therefore, multiple PIDs), programs, and Transport Streams to broadcast the objects and associated control information. To facilitate Clients with the local mapping of all association_tags that are used in the different MPEG-2 programs for the U-U Carousel, a descriptor is defined that may be inserted in the first descriptor loop of the PMTs of the MPEG-2 programs that implement the U-U Object Carousel. The deferred_assocation_tags_descriptor() contains all association_tags that are used within the U-U Object Carousel but that are not associated with a PID in the PMT in which the descriptor resides. The deferred_association_tags_descriptor() contains, therefore, a forward reference to a MPEG-2 program that does contain the PID to which the association tag is linked. Multiple deferred_assocation_tags_descriptor()'s may be inserted in a PMT if necessary.

The syntax and semantics of the deferred_association_tags_descriptor() are described below:

Table 11-7 deferred_association_tags_descriptor

Syntax	No. of bits	Mnemonic
deferred_association_tags_descriptor() {		
descriptor_tag	8	uimsbf
descriptor_length	8	uimsbf
association_tags_loop_length	8	uimsbf
for (n=0;n<N1;n++) {		
association_tag	16	uimsbf
}		
transport_stream_id	16	uimsbf
program_number	16	uimsbf
for (n-0;n<N2;n++) {		
private_data_byte	8	uimsbf
}		
}		

The **descriptor_tag** field is an 8-bit field. For the deferred_association_tags_descriptor, the value shall be 21 (decimal).

The **descriptor_length** field specifies the length of the descriptor in bytes.

The **association_tags_loop_length** field defines the length in bytes of the loop of association tags that follows this field.

The **association_tag** field contains the association_tag that is part of the U-U Object Carousel, but not associated with a PID in the PMT in which the descriptor resides (i.e., not associated with this MPEG-2 program).

The **transport_stream_id** field indicates the Transport Stream in which the MPEG-2 program resides that contains the PIDs that are associated with the enlisted association tags.

The **program_number** field indicates the program_number of the MPEG-2 program that contains the PIDs that are associated with enlisted association tags.

The **private_data_byte** field is user private and is not specified by this part of ISO/IEC 13818.

12. User-to-Network Pass-Thru Messages

12.1 Overview and the General Message Format

The User-to-Network (U-N) Pass-Thru messages are used for sending messages between Users. These messages are assumed to be part of a larger protocol stack and are designed to be carried on a lower layer transport protocol (e.g., UDP/IP, TCP/IP, AAL5, Serial). Constraints on specific lower level protocols are given in clause 9.

All U-N Pass-Thru messages are sent from a User (either a Client or a Server) through the Network which delivers the message to the User (either a Client or a Server) specified by the sender of the message. This clause describes which messages are available, the format of these messages, and scenarios describing how these messages are used.

The syntax of these messages is extensible beyond those defined in this part of ISO/IEC 13818. Additional messages for a specific implementation are outside of the scope of this part of ISO/IEC 13818. If any of the messages or scenarios defined in this part of ISO/IEC 13818 are used, then they shall be implemented exactly as defined in this part of ISO/IEC 13818.

All Pass-Thru messages have a common message format. Table 12-1 defines the User-to-Network Pass-Thru Message format. This format is called the unPassThruMessage ().

Table 12-1 General Format of DSM-CC User-Network Pass-Thru Message

Syntax

unPassThruMessage () {
    dsmccMessageHeader()
    MessagePayload()
}

The **dsmccMessageHeader** is defined in clause 2. For Pass-Thru messages, the dsmccType field shall be set to 0x05.

The **MessagePayload** is constructed from data fields and differs in structure depending on the function of the particular message. Subclause 12.2 defines the DSM-CC User-to-Network Pass-Thru Messages.

12.2 Pass-Thru Messages

Table 12-2 lists the message id's which have been defined for the User-to-Network Pass-Thru messages:

Table 12-2 DSM-CC U-N Pass-Thru messageIds

messageId	Message Name	Description
0x0000	Reserved	ISO/IEC 13818-6 Reserved.
0x0001	PassThruRequest	Sent from a User to the Network to request that the Network deliver PassThruData() to another User. There is no response to this message from the Network.
0x0002	PassThruIndication	Sent from the Network to a User to deliver PassThruData() from another User. There is no response to this message from the recipient.
0x0003	PassThruReceiptRequest	Sent from a User to the Network to request that the Network delivers PassThruData() to another User and request that the recipient respond upon receipt of the data.
0x0004	PassThruReceiptConfirm	Sent from the Network to a User in response to a PassThruReceiptRequest message. This message contains PassThruData() sent from the recipient of the initial message.
0x0005	PassThruReceiptIndication	Sent from the Network to a User to deliver PassThruData() from another User and request that the recipient respond upon receipt of the data.
0x0006	PassThruReceiptResponse	Sent from a User to the Network in response to a PassThruReceiptIndication message. This message contains PassThruData() which will be delivered to the originator of the PassThruMessage sequence.
0x0007 - 0x7FFF	Reserved	ISO/IEC 13818-6 Reserved.
0x8000 - 0xFFFF	User Defined	User Defined U-N Pass-Thru message.

12.2.1 Use of PassThruData() structure in Pass-Thru messages

Pass-Thru and Pass-Thru Receipt messages contain a PassThruData() field which contains privateData. The definition of this data is outside of the scope of this part of ISO/IEC 13818. Table 12-3 defines the format of the PassThruData() which is transported in Pass-Thru messages.

Table 12-3 DSM-CC U-N PassThruData format

Syntax	Num. of Bytes
PassThruData(){	
passThruDataLength	2
for(i=0; i<passThruDataLength; i++) {	
passThruDataByte	1
}	
}	

The **passThruDataLength** field defines the total number of passThruDataBytes.

The **passThruDataBytes** contain private data. The format and usage of this data is outside of the scope of this part of ISO/IEC 13818.

12.2.2 Pass-Thru message definitions

12.2.2.1 PassThruRequest

This message is sent from a User to the Network to request that the network deliver a message to the requested User. Table 12-4 defines the syntax of the PassThruRequest message.

Table 12-4 DSM-CC U-N PassThruRequest message

Syntax	Num. of Bytes
PassThruRequest(){	
dsmccMessageHeader()	
userId	**20**
passThruType	**2**
PassThruData()	
}	

The **userId** field indicates the User to which the message is being sent. This value is supplied by the sending User.

The **passThruType** field shall be used to indicate the type of PassThruData() that is being sent. This value is supplied by the sending User.

The **PassThruData()** structure contains private data which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 12.2.1 for more information on this data. This data is supplied by the sending User.

12.2.2.2 PassThruIndication

This message is sent from the Network to a User to deliver a message from the indicated User. Table 12-5 defines the syntax of the PassThruIndication message.

Table 12-5 DSM-CC U-N PassThruIndication message

Syntax	Num. of Bytes
PassThruIndication(){	
dsmccMessageHeader()	
userId	**20**
passThruType	**2**
PassThruData()	
}	

The **userId** field indicates the User from which the message was sent. This value is set by the Network.

The **passThruType** field shall be used to indicate the type of PassThruData() that is being sent. This value is supplied by the sending User.

The **PassThruData()** structure contains private data which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 12.2.1 for more information on this data. This data is supplied by the sending User.

12.2.2.3 PassThruReceiptRequest

This message is sent from a User to the Network to send a message to the indicated User and request a receipt message from that User. Table 12-6 defines the syntax of the PassThruReceiptRequest message.

Table 12-6 DSM-CC U-N PassThruReceiptRequest message

Syntax	Num. of Bytes
PassThruReceiptRequest(){	
dsmccMessageHeader()	
sourceUserId	20
destinationUserId	20
passThruType	2
PassThruData()	
}	

The **sourceUserId** field indicates the User which is sending the message. This value is supplied by the sending User and is the address to which the Network will send the response.

The **destinationUserId** field indicates the User to which the message is being sent. This value is supplied by the sending User.

The **passThruType** field shall be used to indicate the type of PassThruData() that is being sent. This value is supplied by the sending User.

The **PassThruData()** structure contains private data which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 12.2.1 for more information on this data. This data is supplied by the sending User.

12.2.2.4 PassThruReceiptConfirm

This message is sent from the Network to the sending User in response to a PassThruReceiptRequest message. Table 12-7 defines the syntax of the PassThruReceiptConfirm message.

Table 12-7 DSM-CC U-N PassThruReceiptConfirm message

Syntax	Num. of Bytes
PassThruReceiptConfirm(){	
dsmccMessageHeader()	
response	2
PassThruData()	
}	

The **response** field shall be used to indicate the status of the Pass-Thru message. If a receipt message is received from the receiving User, the Network shall set this field to the value received from that User. If a receipt message is not received or the message scenario is terminated by the Network, the Network shall set this field to indicate the reason for the failure.

The **PassThruData()** structure contains private data which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 12.2.1 for more information on this data. This data is supplied by the receiving User. If the Network terminates the scenario, as indicated by the response field, then the PassThruData() fields shall be set to 0 to indicate that no data is present.

12.2.2.5 PassThruReceiptIndication

This message is sent from the Network to a receiving User to deliver a message from the indicated sending User and request a receipt message from the receiving User. Table 12-8 defines the syntax of the PassThruReceiptIndication message.

Table 12-8 DSM-CC U-N PassThruReceiptIndication message

Syntax	Num. of Bytes
PassThruReceiptIndication(){	
dsmccMessageHeader()	
userId	**20**
passThruType	**2**
PassThruData()	
}	

The **userId** field indicates the User which sent the message. This value is supplied by the Network.

The **passThruType** field shall be used to indicate the type of PassThruData() that is being sent. This value is supplied by the sending User.

The **PassThruData()** structure contains private data which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 12.2.1 for more information on this data. This data is supplied by the sending User.

12.2.2.6 PassThruReceiptResponse

This message is sent from the receiving User to the Network in response to a PassThruReceiptIndication message. Table 12-9 defines the syntax of the PassThruReceiptResponse message.

Table 12-9 DSM-CC U-N PassThruReceiptResponse message

Syntax	Num. of Bytes
PassThruReceiptResponse(){	
dsmccMessageHeader()	
response	**2**
PassThruData()	
}	

The **response** field shall be set by the receiving User to indicate the status of the Pass-Thru message.

The **PassThruData()** structure contains private data which is outside of the scope of this part of ISO/IEC 13818. Refer to subclause 12.2.1 for more information on this data. This data shall be supplied by the receiving User.

12.3 User-to-Network Pass-Thru Message Field Data Types

Table 12-10 defines the data fields used in the User-to-Network Pass-Thru Messages.

Table 12-10 User-to-Network Session Message Field Data Types

Field Name	Length (Bytes)	Range	Description
passThruType	2	0x0000 - 0xFFFF	This field is used to indicate the type of PassThruData(). Table 12-12 defines the possible values for this field.
response	2	0x0000 - 0xFFFF	This field indicates the response to a Pass-Thru Receipt message. This response is passed back to the requesting User. Table 12-11 defines the possible values for this field.

Field Name	Length (Bytes)	Range	Description
userId	20	As specified by OSI NSAP.	A globally unique OSI NSAP address which identifies a User. This address must be a specific address or be able to be resolved to a specific address by the Network.

12.4 Pass-Thru Message Scenario

Users may communicate between themselves using the Pass-Thru commands which pass a message payload through the Network. The Pass-Thru messages may be sent from any User to any other User in a Network. In these scenarios, the sender of the message shall be considered to be the sending User and the recipient shall be considered to be the receiving User. The format of the User Data payload of these messages is defined by the Users and is outside of the scope of this part of ISO/IEC 13818. Since Pass-Thru messages are not confirmed, these messages shall be considered to provide unreliable transport at the User-to-Network level.

12.4.1 Pass-Thru Message scenario

Figure 12-1 describes a Pass-Thru message scenario.

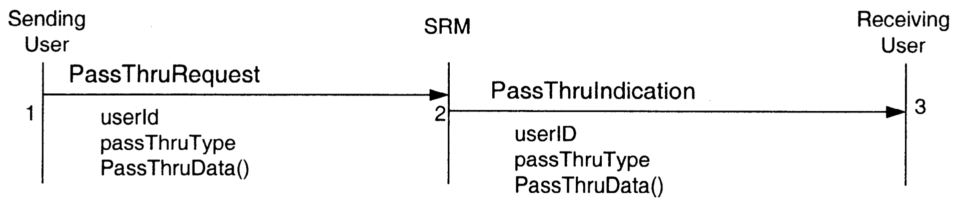


Figure 12-1 Scenario for Pass-Thru Message

12.4.1.1 The Sending User sends a PassThruRequest

Step 1 (Sending User)

The Sending User creates a PassThruRequest message which contains the userId of the recipient of the message, a passThruType, and the PassThruData() payload.

Step 2 (SRM)

The Network validates the senders address and the recipients userId. If the receiving userId's address can be resolved by the Network it creates a PassThruIndication message and delivers it to the indicated User. The userId field shall contain the Id of the User which sent the PassThruRequest message. The passThruType and PassThruData() shall be identical to the data received in the PassThruRequest message.

Step 3 (Receiving User)

The Receiving User validates the message and, if it is applicable, processes the payload.

12.5 Pass-Thru Receipt Message Scenario

The PassThruReceipt messages allow a User to send a message to another User and request that the recipient respond to the message. The Pass-Thru Receipt commands pass a message payload through the Network. The format of the payload of these messages is defined by the User and is outside of the scope of this part of ISO/IEC 13818. The Initiator of a PassThruReceipt message is considered to be the sending User and the Recipient is considered to be the receiving User.

12.5.1 Pass-Thru Receipt Message scenario

Figure 12-2 describes a Pass-Thru Receipt message scenario.

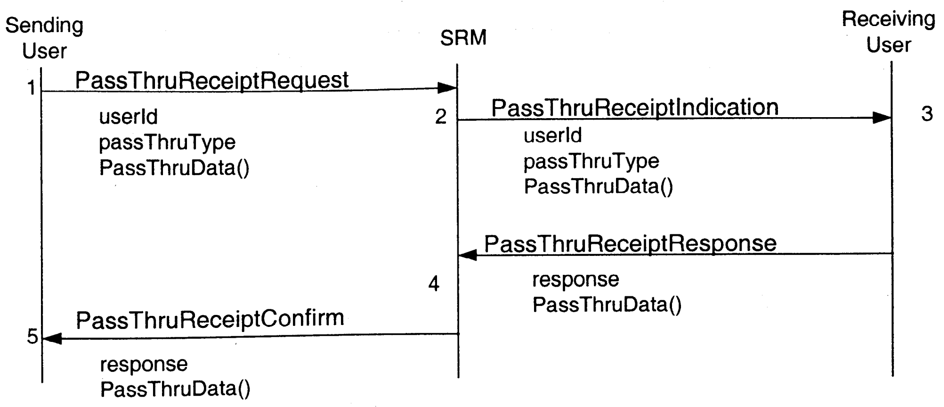


Figure 12-2 Scenario for Pass-Thru Receipt Message

12.5.1.1 The Sending User sends a PassThruReceiptRequest

Step 1 (Sending User)

The sending User creates a PassThruReceiptRequest message which contains the userId of the recipient of the message, a passThruType, and the PassThruData() payload.

The sending User starts timer tMsg. If this timer expires before the PassThruReceiptConfirm message is received, the scenario terminates. If the PassThruReceiptConfirm message is received after the timer has expired, this response shall be discarded.

Step 2 (SRM)

The Network validates the sending userId and the recipients address. If the recipient's address can be resolved by the Network it creates a PassThruReceiptIndication message and delivers it to the receiving User. The userId field shall contain the Id of the User which sent the PassThruReceiptRequest message. The passThruType and PassThruData() shall be identical to the data received in the PassThruReceiptRequest message.

The SRM starts timer tMsg. If this timer expires before the PassThruReceiptResponse message is received, the SRM shall send the PassThruReceiptConfirm to the sending user to indicate that no response was received and the scenario terminates. If the PassThruReceiptResponse message is received after the timer has expired, this response shall be discarded.

Step 3 (Receiving User)

The receiving User validates the message and, if it is applicable, processes the payload. The receiving User generates a PassThruReceiptResponse message and sends it to the Network. The response field indicates how the message was handled by the receiving User. The PassThruData() is generated by the receiving User and shall be passed to the sending User which generated the request. The content of the PassThruData is outside of the scope of this part of ISO/IEC 13818.

Step 4 (SRM)

Upon receipt of the PassThruReceiptResponse message, the Network cancels timer tMsg. The Network generates a PassThruReceiptConfirm message and sends it to the sending User. The response and PassThruData() fields shall be identical to the data received in the PassThruReceiptResponse message.

Step 5 (Sending User)

Upon receipt of the PassThruReceiptConfirm message, the sending User cancels timer tMsg and processes the message according to the response and PassThruData() fields.

12.6 Pass-Thru Response Codes

Table 12-11 defines the response codes that are defined for use by the U-N Pass-Thru messages:

Table 12-11 User-to-Network Pass-Thru message response codes

Response	Value	Description
rspOK	0x0000	May be used the receiving User to indicate that the message was accepted.
rspNoUser	0x0001	Indicates that the Network was unable to deliver the PassThruReceipt message because the indicated userId was invalid.
rspNoReceipt	0x0002	Indicates that the receiving User of a Pass-Thru message did not respond to a passThruReceiptIndication message within tMsg time period.
reserved	0x0003 - 0x7FFF	ISO/IEC 13818-6 reserved.
User defined	0x8000 - 0xFFFF	These response codes are defined by the User and are outside of the scope of this part of ISO/IEC 13818.

12.7 Pass-Thru Type Codes

Table 12-12 defines the type codes that are defined for use by the U-N Pass-Thru messages:

Table 12-12 User-to-Network Pass-Thru type values

Value	Description
0x0000	ISO/IEC 13818-6 reserved
0x0001 - 0x0015	ITU-T Rec. T.120-series reserved
0x0016 - 0x7FFF	ISO/IEC 13818-6 reserved
0x8000 - 0xFFFF	User Defined. The use of these values are outside of the scope of this part of ISO/IEC 13818

12.8 State Machine

See Normative Annex A for the SDL diagrams which define the Pass-Thru state machines.

Annex A
(normative)
User-Network Protocol State Machines

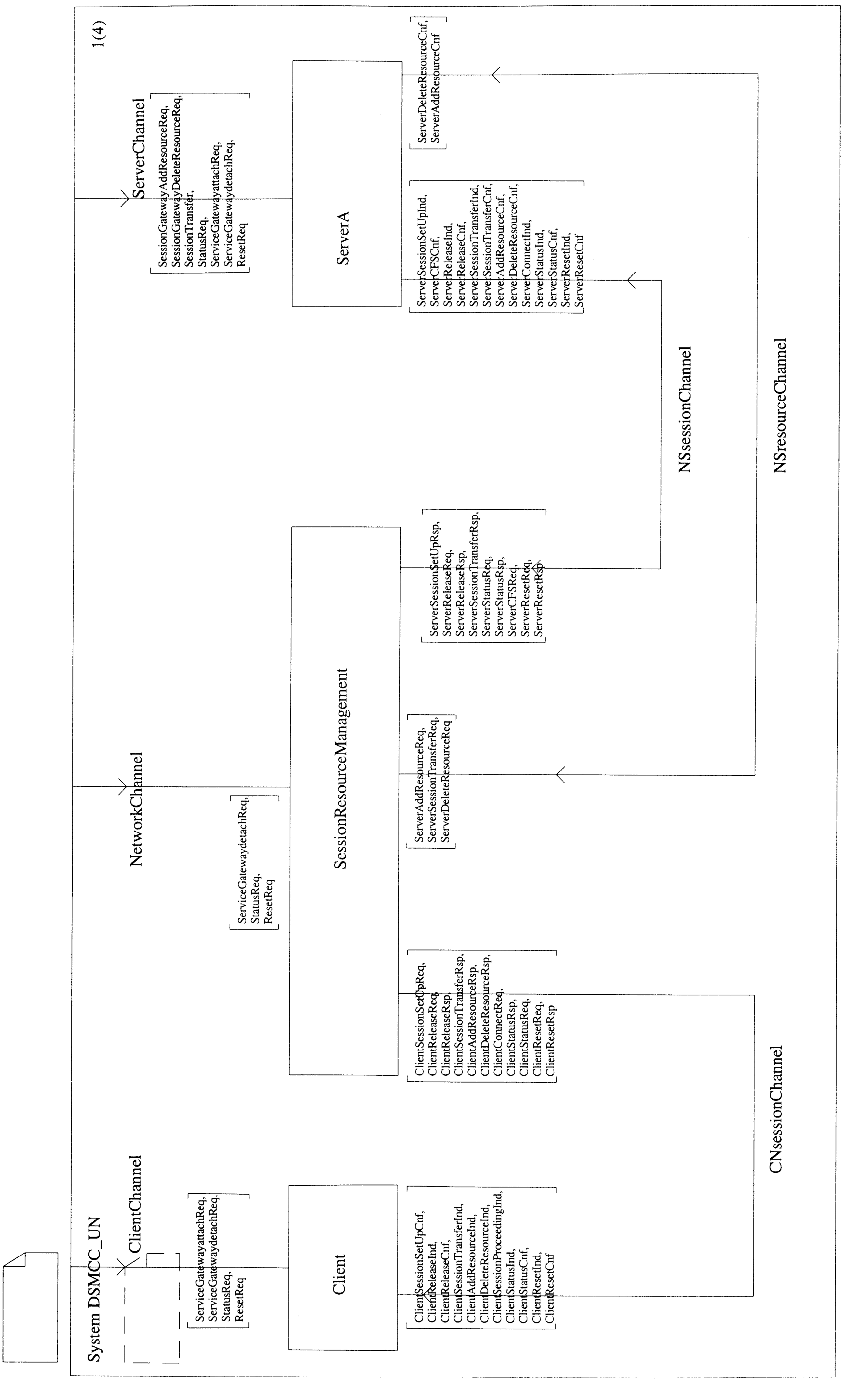
A.1 Introduction

State machines for the User-to-Network protocols are described using the Specification and Description language. SDLs are included for the Session, Download (flow controlled scenario only), SDB-CCP, and Pass-Thru protocols.

A.2 U-N Session

The following pages contain the U-N Session protocol SDLs.

Figure A-1 U-N Session Specification and Description Language


System DSMCC_UN
1(4)
ClientChannel
ServiceGatewayattachReq, ServiceGatewaydetachReq, StatusReq, ResetReq
NetworkChannel
ServiceGatewaydetachReq, StatusReq, ResetReq
ServerChannel
SessionGatewayAddResourceReq, SessionGatewayDeleteResourceReq, SessionTransfer, StatusReq, ServiceGatewayattachReq, ServiceGatewaydetachReq, ResetReq
Client
SessionResourceManagement
ServerA
ClientSessionSetUpCnf, ClientReleaseInd, ClientReleaseCnf, ClientSessionTransferInd, ClientAddResourceInd, ClientDeleteResourceInd, ClientSessionProceedingInd, ClientStatusInd, ClientStatusCnf, ClientResetInd, ClientResetCnf
ClientSessionSetUpReq, ClientReleaseReq, ClientReleaseRsp, ClientSessionTransferRsp, ClientAddResourceRsp, ClientDeleteResourceRsp, ClientConnectReq, ClientStatusRsp, ClientStatusReq, ClientResetReq, ClientResetRsp
CNsessionChannel
ServerAddResourceReq, ServerSessionTransferReq, ServerDeleteResourceReq
ServerSessionSetUpRsp, ServerReleaseReq, ServerReleaseRsp, ServerSessionTransferRsp, ServerStatusReq, ServerStatusRsp, ServerCFSReq, ServerResetReq, ServerResetRsp
NSsessionChannel
ServerSessionSetUpInd, ServerCFSCnf, ServerReleaseInd, ServerReleaseCnf, ServerSessionTransferInd, ServerSessionTransferCnf, ServerAddResourceCnf, ServerDeleteResourceCnf, ServerConnectInd, ServerStatusInd, ServerStatusCnf, ServerResetInd, ServerResetCnf
NSresourceChannel
ServerDeleteResourceCnf, ServerAddResourceCnf

System DSMCC_UN

2(4)

```
SYNONYM messageRetryCount integer = 1;
SYNONYM maxsession integer        = 3;
SYNONYM maxinSC integer           = 10;
SYNONYM maxaPrincipal integer     = 10;
SYNONYM maxdownloadInfo integer= 10;
SYNONYM maxaContext integer       = 10;
SYNONYM maxknownId integer = 10;
SYNONYM maxresource integer = 1;
SYNONYM maxservice integer = 10;
SYNONYM maxForwardCount integer = 10;
SYNONYM maxclients integer = 3;
```

```
/* is used to identify a session troughout its life cycle */
SYNTYPE sessionIdType = Integer
CONSTANTS 0:maxsession
ENDSYNTYPE;
/* authentication information */
SYNTYPE SCType = Integer
CONSTANTS 0:maxinSC
ENDSYNTYPE;
```

```
NEWTYPE NSAPType
LITERALS     ServerANSAP,ServerBNSAP,clientNSAP,CFSserverNSAP,viodNSAP
ENDNEWTYPE NSAPType;
NEWTYPE serviceGatewayIdType
LITERALS     serviceGatewayA,serviceGatewayB
ENDNEWTYPE serviceGatewayIdType;
NEWTYPE serviceIdType
LITERALS     Films,
             Sports,
             singleUser,
             multiUser,
             Stillpicture,
             Videobased
ENDNEWTYPE serviceIdType;
NEWTYPE rPathSpecType STRUCT
serviceGatewayName    serviceGatewayIdType;
serviceName           serviceIdType;
ENDNEWTYPE rPathSpecType;
```

```
NEWTYPE reasonType
LITERALS rsnNoTransaction,
      rsnClNoResource,
      rsnNormal,
      rsnClFormatError,
      rsnSeNoSession,
      rsnNeNoSession,
      rsnClProcError,
      rsnRetrans,
      rsnClTransferReject,
      rsnOK,
      rsnNeResourceFailed,
      rsnSeTimerExpired,
      rsnNeTimerExpired
ENDNEWTYPE reasonType;
NEWTYPE responseType
LITERALS   rspOK,
      rspClRejResource,
      rspNeInvalidServer,
      rspSeInvalidServer,
      rspClInvalidServer,
      rspNeInvalidClient,
      rspSeInvalidClient,
      rspClInvalidClient,
      rspNeNoCalls,
      rspSeNoCalls,
      rspNeProcError,
      rspNeNoSession,
      rspSeNoSession,
      rspClNoSession,
      rspNeNoResource,
      rspSeNoResource,
      rspNeResourceOK,
      rspForward,
      rspClNoResource,
      rspNeForwardFailed,
      rspClTransferReject,
      rspNeTransferFail,
      rspSeTransferReject,
      rspNeTransferReject,
      rspSeTransferNoResource,
      rspSeTransferNoRes,
      rspNeResourceFailed,
      rspClResourceFailed,
      rspSeResourceFailed,
      rspNeFormatError,
      rspClProcError,
      rspNeTimerExpired,
      rspClTimerExpired,
      rspSeTimerExpired
ENDNEWTYPE responseType;
```

```
/* used in Session and Resource Manager */
NEWTYPE forwardCountType STRUCT
forwardCount integer;
forwardServerId NSAPType;
ENDNEWTYPE forwardCountType;
NEWTYPE stat_type
LITERALS OCC,UNOCC
ENDNEWTYPE stat_type;
NEWTYPE elem STRUCT
stat        stat_type;
forwardServerId NSAPType;
ENDNEWTYPE elem;
/* used in the client */
NEWTYPE element STRUCT
stat stat_type;
id PId;
ENDNEWTYPE element;
```

System DSMCC_UN

3(4)

```
/* used in the server */
NEWTYPE processes STRUCT
Id        serviceGatewayIdType;
ProcessId PId;
ENDNEWTYPE processes;
NEWTYPE sgtableType Array(sessionIdType,processes)
ENDNEWTYPE sgtableType;
NEWTYPE NSAPtable STRUCT
clientId  NSAPType;
serverId  NSAPType;
ENDNEWTYPE NSAPtable;
NEWTYPE NSAPtableType Array(sessionIdType,NSAPtable)
ENDNEWTYPE NSAPtableType;
NEWTYPE sessionelem STRUCT
stat stat_type;
sessionId sessionIdType;
cfssessionId sessionIdType;
ENDNEWTYPE sessionelem;
NEWTYPE sessiontableType Array(sessionIdType,sessionelem);
~~DEFAULT(.(. UNOCC,0,0 .).);ENDNEWTYPE sessiontableType~~;
```

```
/* RESOURCE DEFINITIONS */
NEWTYPE resourceDescriptorTypeType LITERALS
                        ContinuousFeedSession,
                        AtmConnection,
                        MpegProgram,
                        PhysicalChannel,
                        TSUpstreamBandwidth,
                        TSDownstreamBandwidth,
                        AtmSvcConnection,
                        AtmConnectionNotify,
                        IP,
                        ClientTDMAAssigment,
                        Reserved,
                        SharedResource,
                        SharedReqId,
                        UserDefined,
                        TypeOwner
ENDNEWTYPE resourceDescriptorTypeType;
NEWTYPE resourceDataByteType STRUCT
downstreamBandwidth       real;  /* User Cell rate Mbit/s */
upstreamBandwidth    real;  /* User Cell rate kbit/s */
ENDNEWTYPE resourceDataByteType;
NEWTYPE resourceDescriptorType STRUCT       /* chapter 4.5.4 in the spec */
resourceReqId             integer;  /* is set by the user and then compared in the confirm message */
resourceDescriptorType    resourceDescriptorTypeType;  /* defines the specific resource being requested */
resourceNum                    integer;  /* resourceNumberAssignor+resourceNumber */
/*  associationTag tbd  */
/*  resourceAllocator tbd  */
resourceAttribute               integer;  /* is set by the user,indicates if the entire resourceReq must be satisfied or not */
/*  resourceView tbd  */
/*  resourceStatus tbd  */
/*  resourceLength tbd  */
resourceDataValueByte     resourceDataByteType;
ENDNEWTYPE resourceDescriptorType;
```

```
NEWTYPE resourceCountType
Array(resourceCount,resourceDescriptorType);
ENDNEWTYPE resourceCountType;
NEWTYPE GenericIdentifierTransport STRUCT
sessionId            sessionIdType;
resourceCount  resourceCountType;
ENDNEWTYPE GenericIdentifierTransport;
NEWTYPE statusDataBytesType STRUCT
sessionId sessionIdType;
serverId NSAPType;
resource resourceCountType;
ENDNEWTYPE statusDataBytesType;
NEWTYPE statusCountType STRUCT
statusCount integer;
statusDataBytes statusDataBytesType;
~~ENdNEWTYPE statusCountType;~~
```

```
NEWTYPE userdataType STRUCT
sc SCType;
aP aPrincipalType;
dow downloadInfoType;
aC aContextType;
rP rPathSpecType;
ENDNEWTYPE userdataType;
```

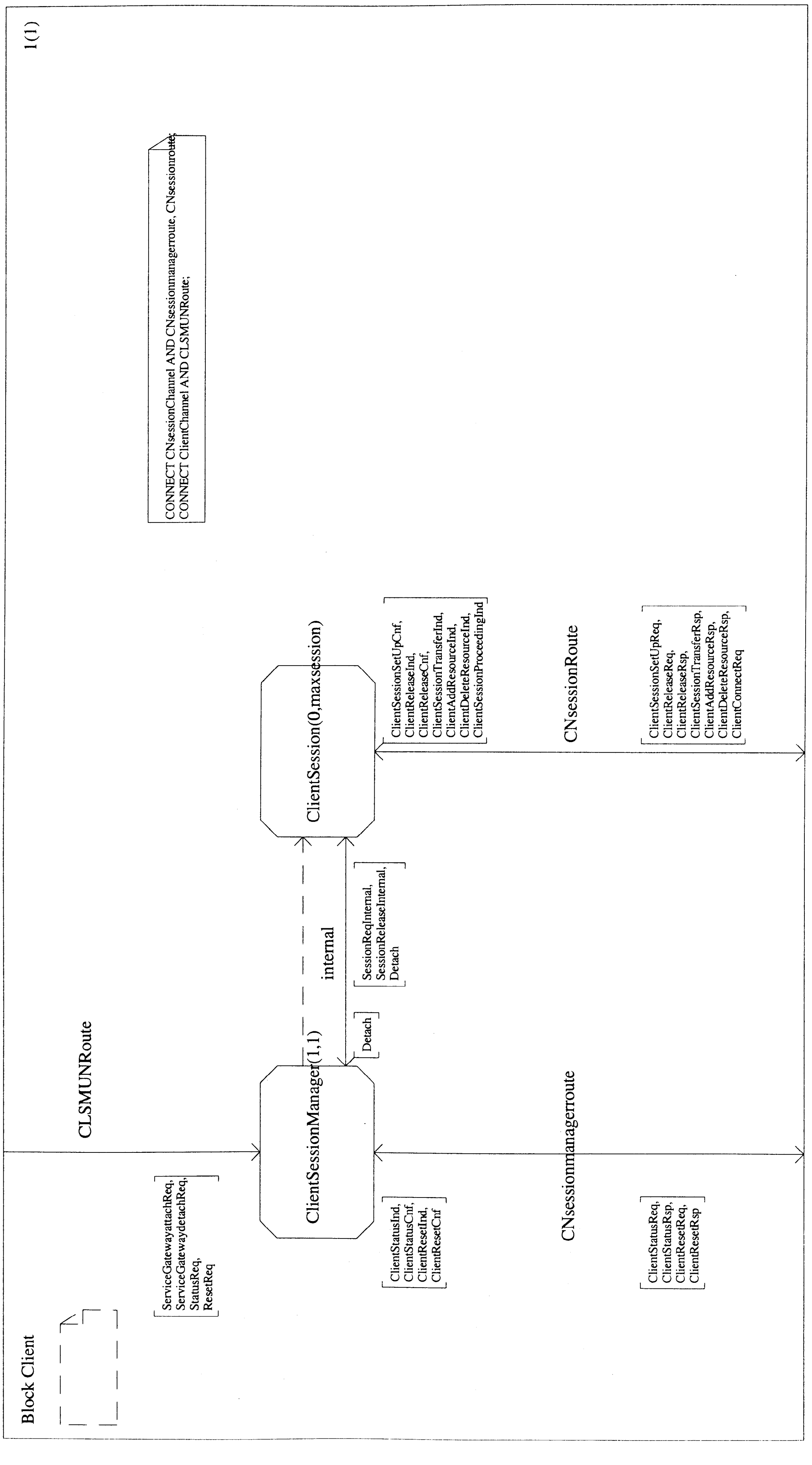
System DSMCC_UN

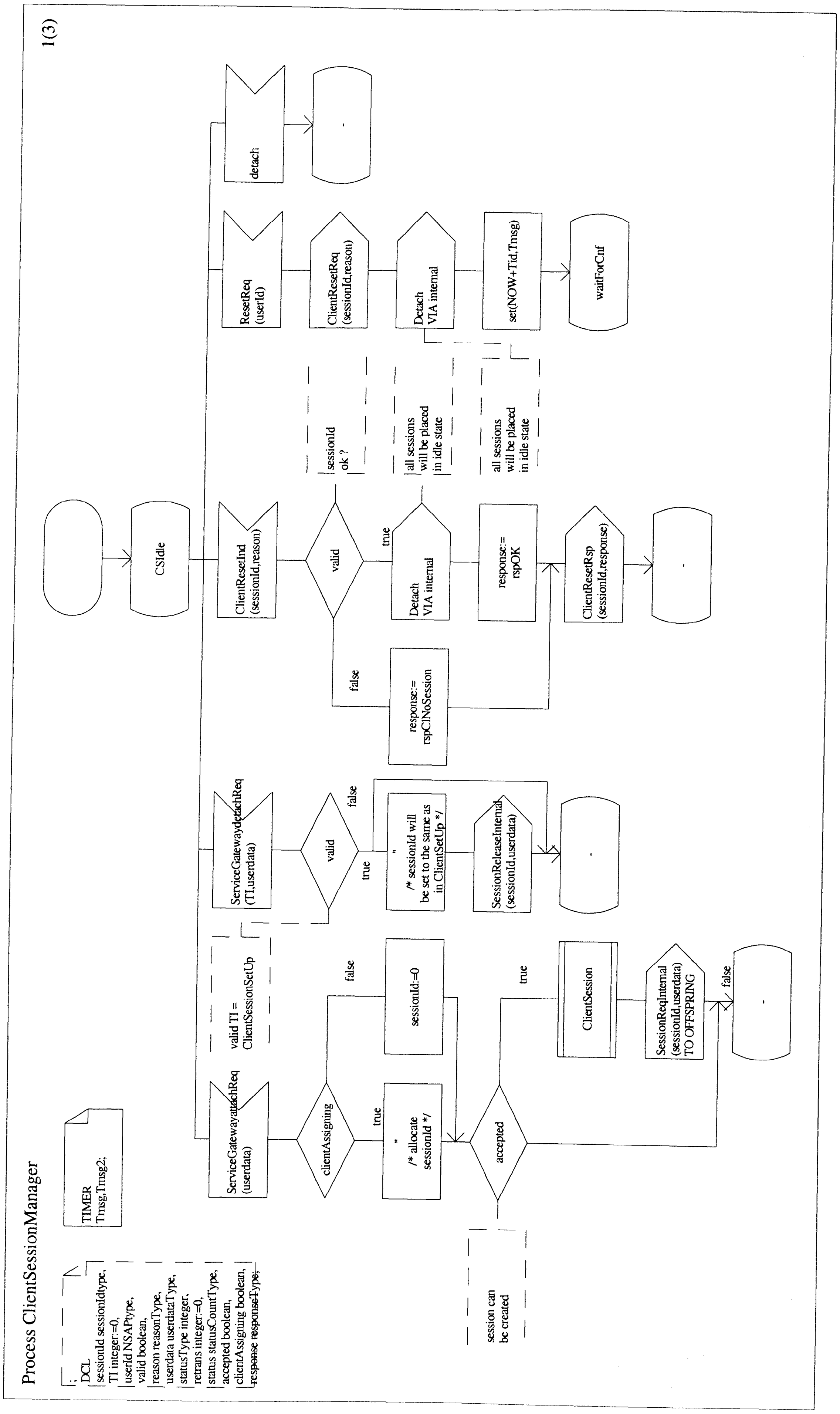
4(4)

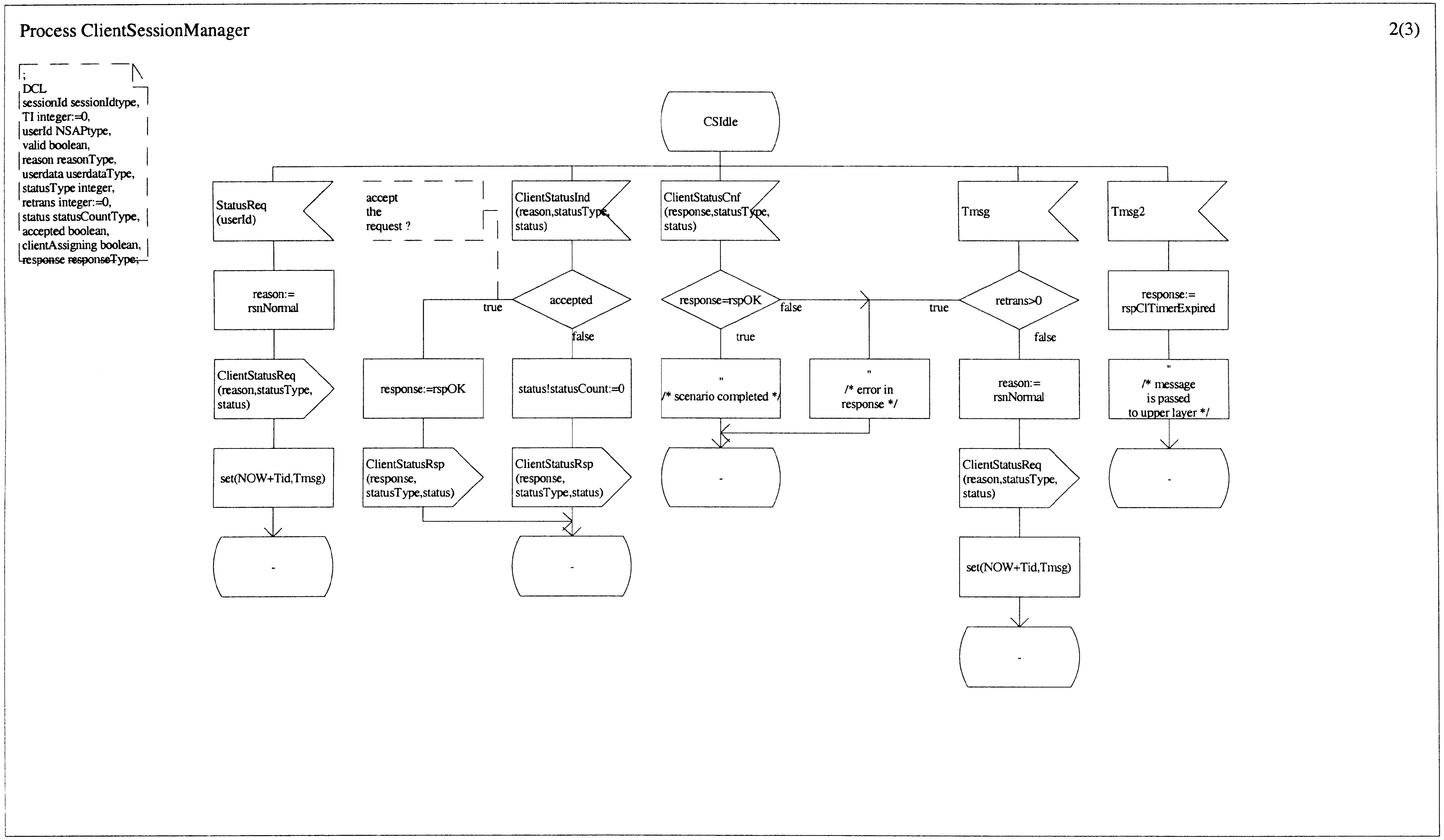
```
SIGNAL
/* this signals is the client sending to SRM */
ClientSessionSetUpReq(sessionIdType,NSAPType,NSAPType,userdataType),
ClientReleaseReq(sessionIdType,reasonType,userdataType),
ClientReleaseRsp(sessionIdType,responseType,userdataType),
ClientAddResourceRsp(sessionIdType,responseType,resourceCountType),
ClientDeleteResourceRsp(sessionIdType,responseType,userdataType),
ClientConnectReq(sessionIdType,userdataType),
/* this signals is the client receiving from SRM */
ClientSessionSetUpCnf(sessionIdType,NSAPType,responseType,resourceCountType,userdataType),
ClientReleaseInd(sessionIdType,reasonType,userdataType),
ClientReleaseCnf(sessionIdType,responseType,userdataType),
ClientAddResourceInd(sessionIdType,resourceCountType),
ClientDeleteResourceInd(sessionIdType,reasonType,resourceCountType,userdataType),
ClientSessionProceedingInd(reasonType),
/* this signals is the server sending to SRM */
ServerReleaseReq(sessionIdType,reasonType,userdataType),
ServerReleaseRsp(sessionIdType,responseType,userdataType),
ServerAddResourceReq(sessionIdType,resourceCountType),
ServerSessionSetUpRsp(sessionIdType,NSAPType,responseType,NSAPType,resourceCountType,userdataType),
ServerDeleteResourceReq(sessionIdType,reasonType,resourceCountType,userdataType),
/* this signals is the SRM sending to the server */
ServerReleaseInd(sessionIdType,reasonType,userdataType),
ServerReleaseCnf(sessionIdType,responseType,userdataType),
ServerAddResourceCnf(sessionIdType,responseType,resourceCountType),
ServerSessionSetUpInd(sessionIdType,NSAPType,NSAPType,forwardCountType,userdataType),
ServerDeleteResourceCnf(sessionIdType,responseType,userdataType),
ServerConnectINd(sessionIdType,userdataType),
/* this signals is the client receiving from the environment */
ServiceGatewayattachReq(userdataType),
ServiceGatewaydetachReq(integer,userdataType);
```

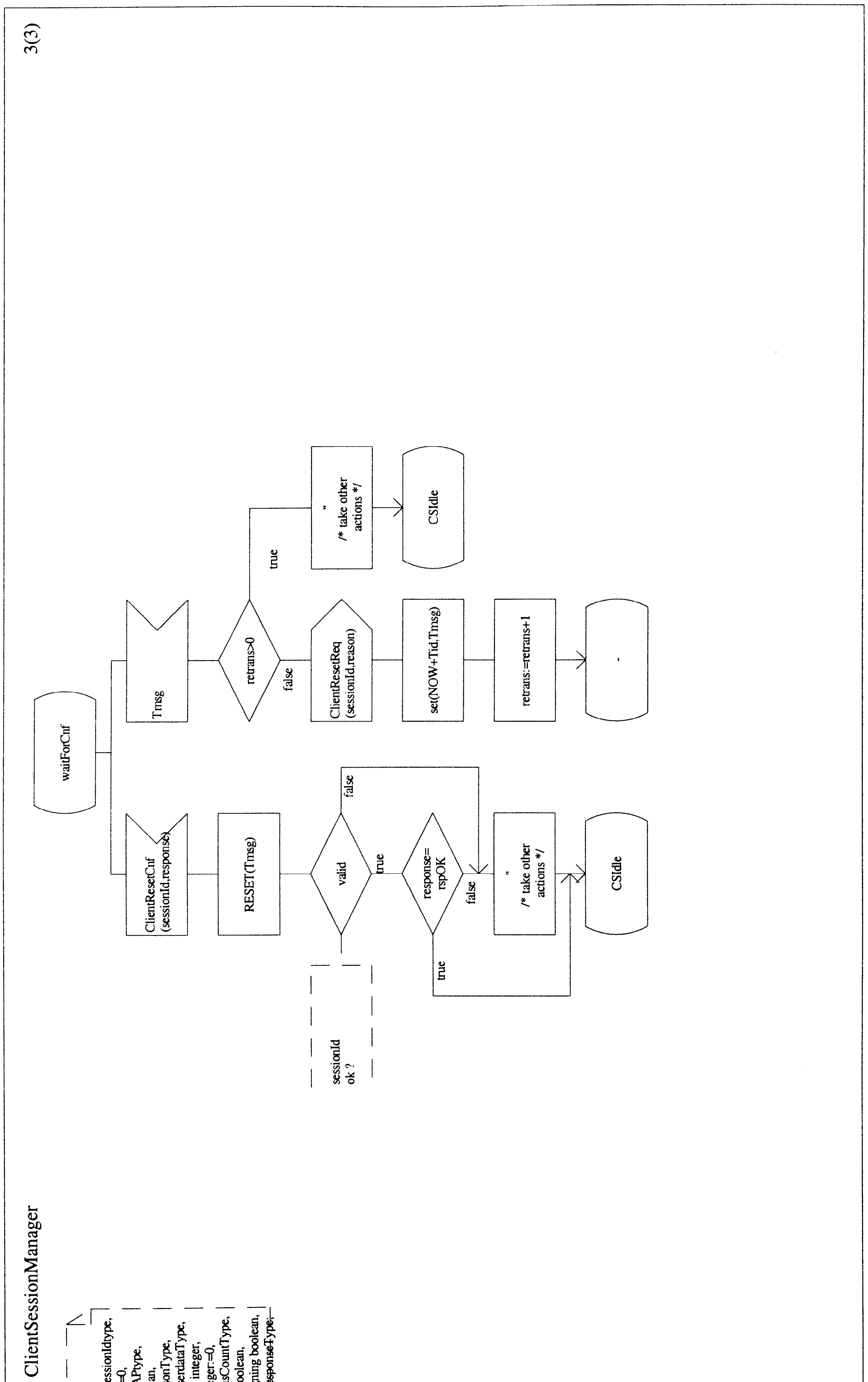
```
SIGNAL
/* internal signals in SRM */
deleteHandler,
SRMCFS(sessionIdType,NSAPType,resourceCountType,responseType),
ReqInternal(sessionIdType,NSAPType,NSAPType,userdataType),
ServerAddResourceInternal(sessionIdType,resourceCountType),
activateResourceHandler,
detachResourceHandler,
ServerSessionTransferInternal(sessionIdType,NSAPType,NSAPType,userdataType),
ServerDeleteResourceInternal(sessionIdType,reasonType,resourceCountType,userdataType),
/* internal signals in the client */
Detach,
SessionReqInternal(sessionIdType,userdataType),
SessionReleaseInternal(sessionIdType,userdataType),
/* internal signals in the server */ SessionGatewayaddResourceReq(sessionIdType,resourceCountType),
ServerSetUp(sessionIdType,userdataType,resourceCountType),
ServerInternal(sessionIdType,NSAPType,resourceCountType),
SessionTransfer(sessionIdType,NSAPType,NSAPType,NSAPType,resourceCountType,userdataType),
SessionGatewaydeleteResourceReq(sessionIdType,reasonType,resourceCountType,userdataType);
```

```
SIGNAL
/* status signals */
ClientStatusInd (reasonType,integer,statusCountType),
ClientStatusReq (reasonType,integer,statusCountType),
ClientStatusCnf (responseType,integer,statusCountType),
ClientStatusRsp (responseType,integer,statusCountType),
ServerStatusReq (reasonType,integer,statusCountType),
ServerStatusInd (reasonType,integer,statusCountType),
ServerStatusCnf (responseType,integer,statusCountType),
ServerStatusRsp (responseType,integer,statusCountType),
/* input from env */ StatusReq(NSAPtype),
ClientResetReq (sessionIdType,reasonType),
ClientResetInd (sessionIdType,reasonType),
ClientResetCnf (sessionIdType,responseType),
ClientResetRsp (sessionIdType,responseType),
ServerResetReq (sessionIdType,reasonType),
ServerResetInd (sessionIdType,reasonType),
ServerResetCnf (sessionIdType,responseType),
ServerResetRsp (sessionIdType,responseType),
/* input from env */ ResetReq(NSAPtype),
/* continuous feed session */
ServerCFSReq(sessionIdType,NSAPType,resourceCountType),
ServerCFSCnf(sessionIdType,responseType,resourceCountType),
/* signals used at server session tranfer command sequence */
ServerSessionTransferReq(sessionIdType,NSAPtype,NSAPtype,userdataType),
ServerSessionTransferCnf(sessionIdType,responseType,userdataType),
ServerSessionTransferInd(sessionIdType,NSAPtype,NSAPtype,NSAPtype,resourceCountType,userdataType),
ServerSessionTransferRsp(sessionIdType,responseType,resourceCountType,userdataType),
ClientSessionTransferInd(sessionIdType,NSAPtype,NSAPtype,NSAPtype,resourceCountType,userdataType),
ClientSessionTransferRsp(sessionIdType,responseType,userdataType);
```

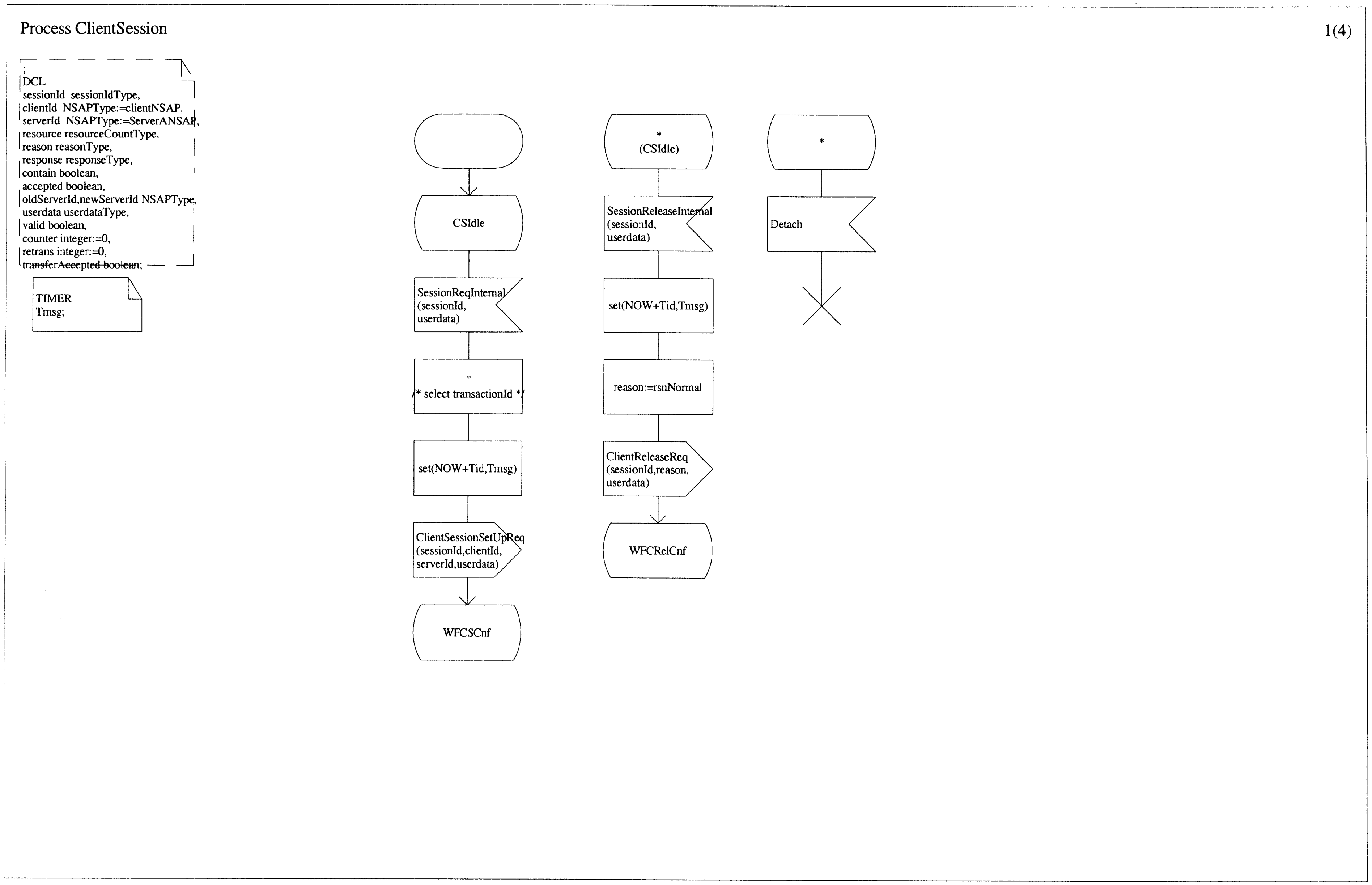

Block Client
1(1)
CONNECT CNsessionChannel AND CNsessionmanagerroute, CNsessionroute;
CONNECT ClientChannel AND CLSMUNRoute;
CLSMUNRoute
ServiceGatewayattachReq,
ServiceGatewaydetachReq,
StatusReq,
ResetReq
ClientSessionManager(1,1)
ClientStatusInd,
ClientStatusCnf,
ClientResetInd,
ClientResetCnf
CNsessionmanagerroute
ClientStatusReq,
ClientStatusRsp,
ClientResetReq,
ClientResetRsp
Detach
internal
SessionReqInternal,
SessionReleaseInternal,
Detach
ClientSession(0,maxsession)
ClientSessionSetUpCnf,
ClientReleaseInd,
ClientReleaseCnf,
ClientSessionTransferInd,
ClientAddResourceInd,
ClientDeleteResourceInd,
ClientSessionProceedingInd
CNsessionRoute
ClientSessionSetUpReq,
ClientReleaseReq,
ClientReleaseRsp,
ClientSessionTransferRsp,
ClientAddResourceRsp,
ClientDeleteResourceRsp,
ClientConnectReq


Process ClientSessionManager
1(3)
DCL
sessionId sessionIdtype,
TI integer:=0,
userId NSAPtype,
valid boolean,
reason reasonType,
userdata userdataType,
statusType integer,
retrans integer:=0,
status statusCountType,
accepted boolean,
clientAssigning boolean,
~~response responseType;~~
TIMER
Tmsg,Tmsg2;
CSIdle
ServiceGatewayattachReq (userdata)
valid TI = ClientSessionSetUp
clientAssigning
true
false
/* allocate sessionId */
sessionId:=0
accepted
true
false
session can be created
ClientSession
SessionReqInternal (sessionId,userdata) TO OFFSPRING
ServiceGatewaydetachReq (TI,userdata)
valid
true
false
/* sessionId will be set to the same as in ClientSetUp */
SessionReleaseInternal (sessionId,userdata)
ClientResetInd (sessionId,reason)
valid
true
false
sessionId ok ?
Detach VIA internal
all sessions will be placed in idle state
response:= rspOK
response:= rspClNoSession
ClientResetRsp (sessionId,response)
ResetReq (userId)
ClientResetReq (sessionId,reason)
Detach VIA internal
all sessions will be placed in idle state
set(NOW+Tid,Tmsg)
waitForCnf
detach
-


Process ClientSessionManager
2(3)
;
DCL
sessionId sessionIdtype,
TI integer:=0,
userId NSAPtype,
valid boolean,
reason reasonType,
userdata userdataType,
statusType integer,
retrans integer:=0,
status statusCountType,
accepted boolean,
clientAssigning boolean,
~~response responseType;~~
CSIdle
StatusReq (userId)
reason:= rsnNormal
ClientStatusReq (reason,statusType, status)
set(NOW+Tid,Tmsg)
-
accept the request ?
ClientStatusInd (reason,statusType, status)
accepted
true
false
response:=rspOK
status!statusCount:=0
ClientStatusRsp (response, statusType,status)
ClientStatusRsp (response, statusType,status)
-
ClientStatusCnf (response,statusType, status)
response=rspOK
true
false
" /* scenario completed */
" /* error in response */
-
Tmsg
retrans>0
true
false
reason:= rsnNormal
ClientStatusReq (reason,statusType, status)
set(NOW+Tid,Tmsg)
-
Tmsg2
response:= rspClTimerExpired
" /* message is passed to upper layer */
-


Process ClientSessionManager
3(3)
waitForCnf
ClientResetCnf (sessionId,response)
RESET(Tmsg)
valid
true
false
sessionId ok ?
response= rspOK
true
false
" /* take other actions */
CSIdle
Tmsg
retrans>0
true
false
" /* take other actions */
CSIdle
ClientResetReq (sessionId,reason)
set(NOW+Tid,Tmsg)
retrans:=retrans+1
-

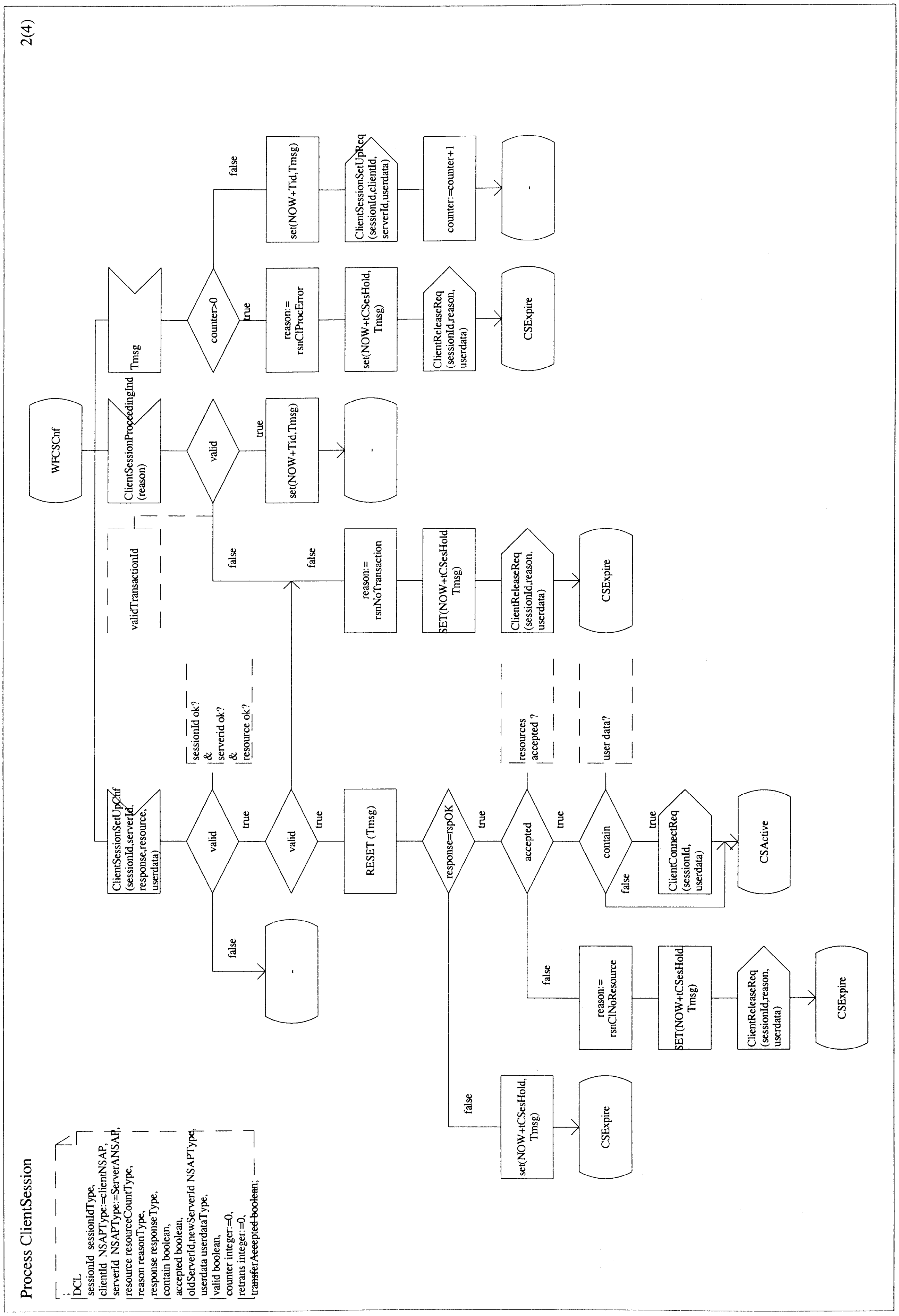
```
;
DCL
sessionId sessionIdtype,
TI integer:=0,
userId NSAPtype,
valid boolean,
reason reasonType,
userdata userdataType,
statusType integer,
retrans integer:=0,
status statusCountType,
accepted boolean,
clientAssigning boolean,
~~response responseType;~~
```

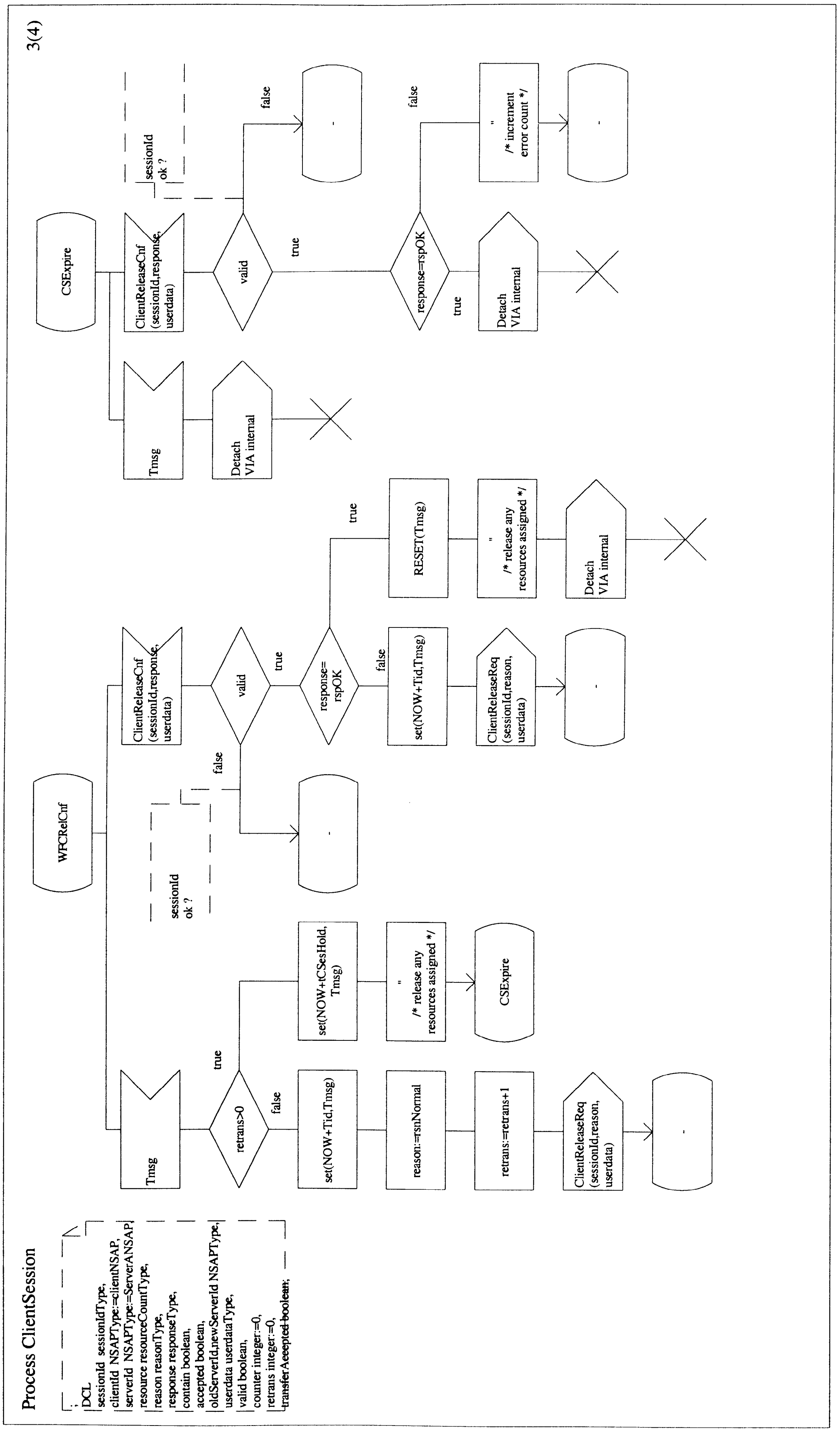

Process ClientSession

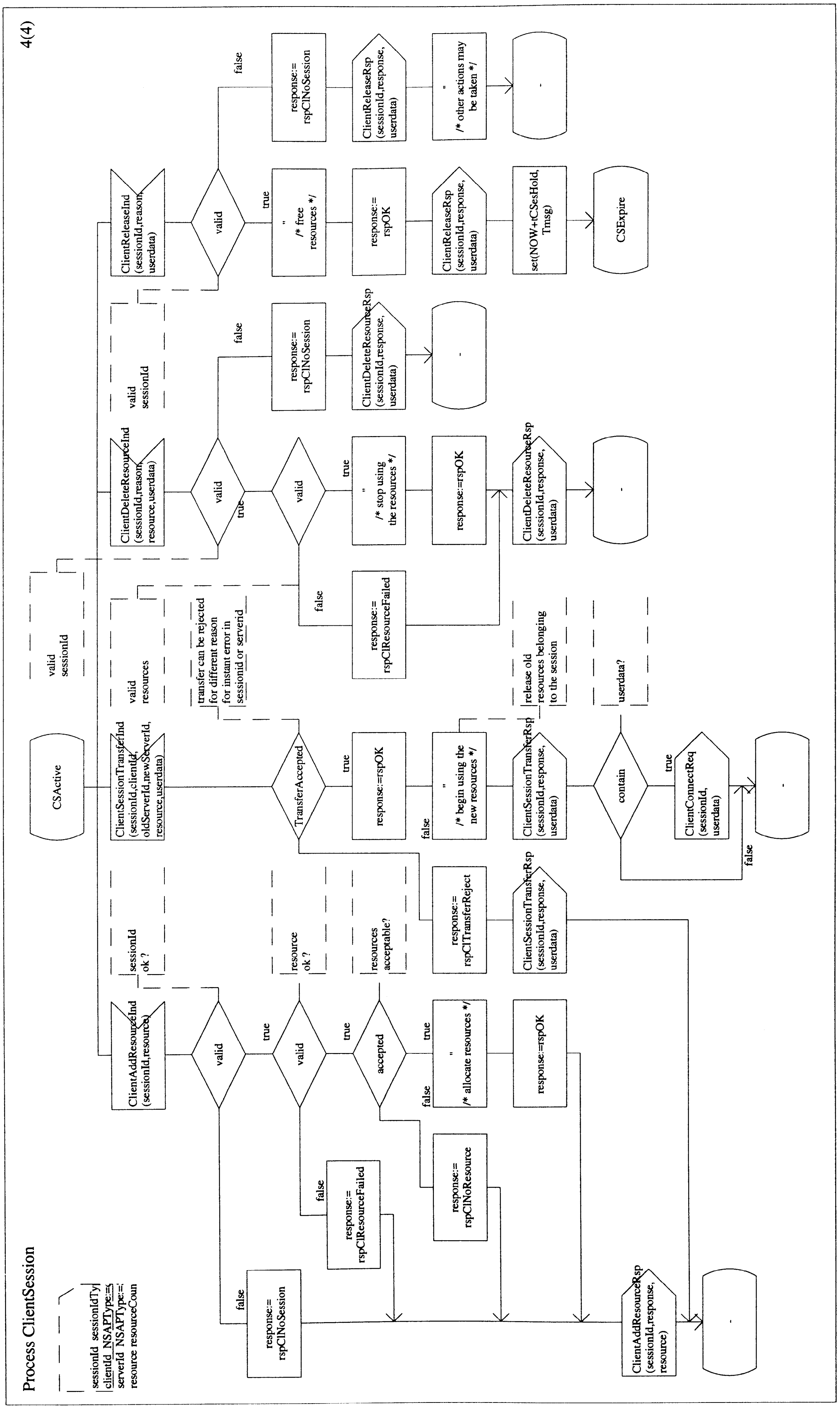
1(4)

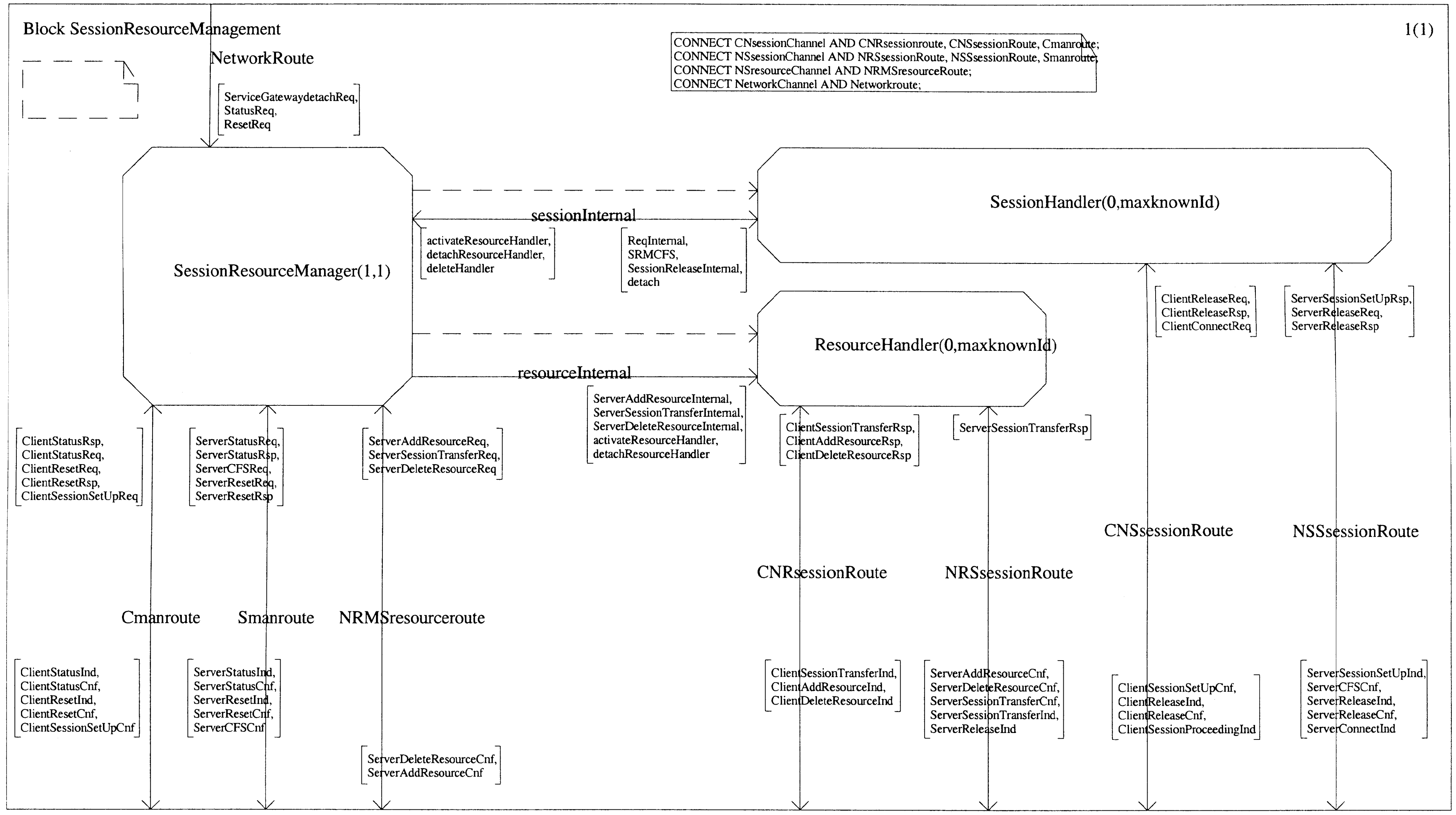
```
;
DCL
sessionId  sessionIdType,
clientId  NSAPType:=clientNSAP,
serverId  NSAPType:=ServerANSAP,
resource resourceCountType,
reason reasonType,
response responseType,
contain boolean,
accepted boolean,
oldServerId,newServerId NSAPType,
userdata userdataType,
valid boolean,
counter integer:=0,
retrans integer:=0,
~~transferAccepted boolean~~;
```

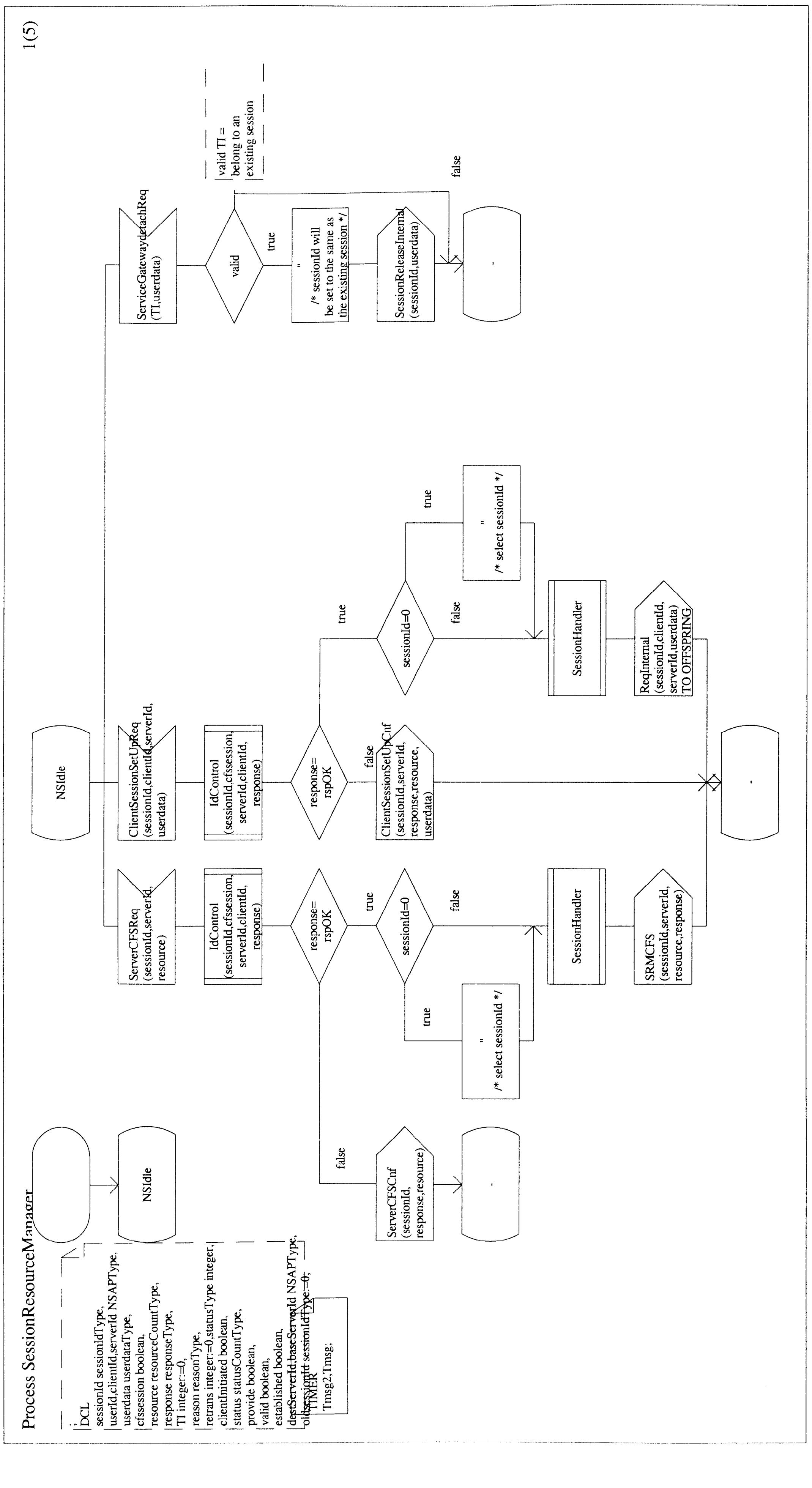
```
TIMER
Tmsg;
```

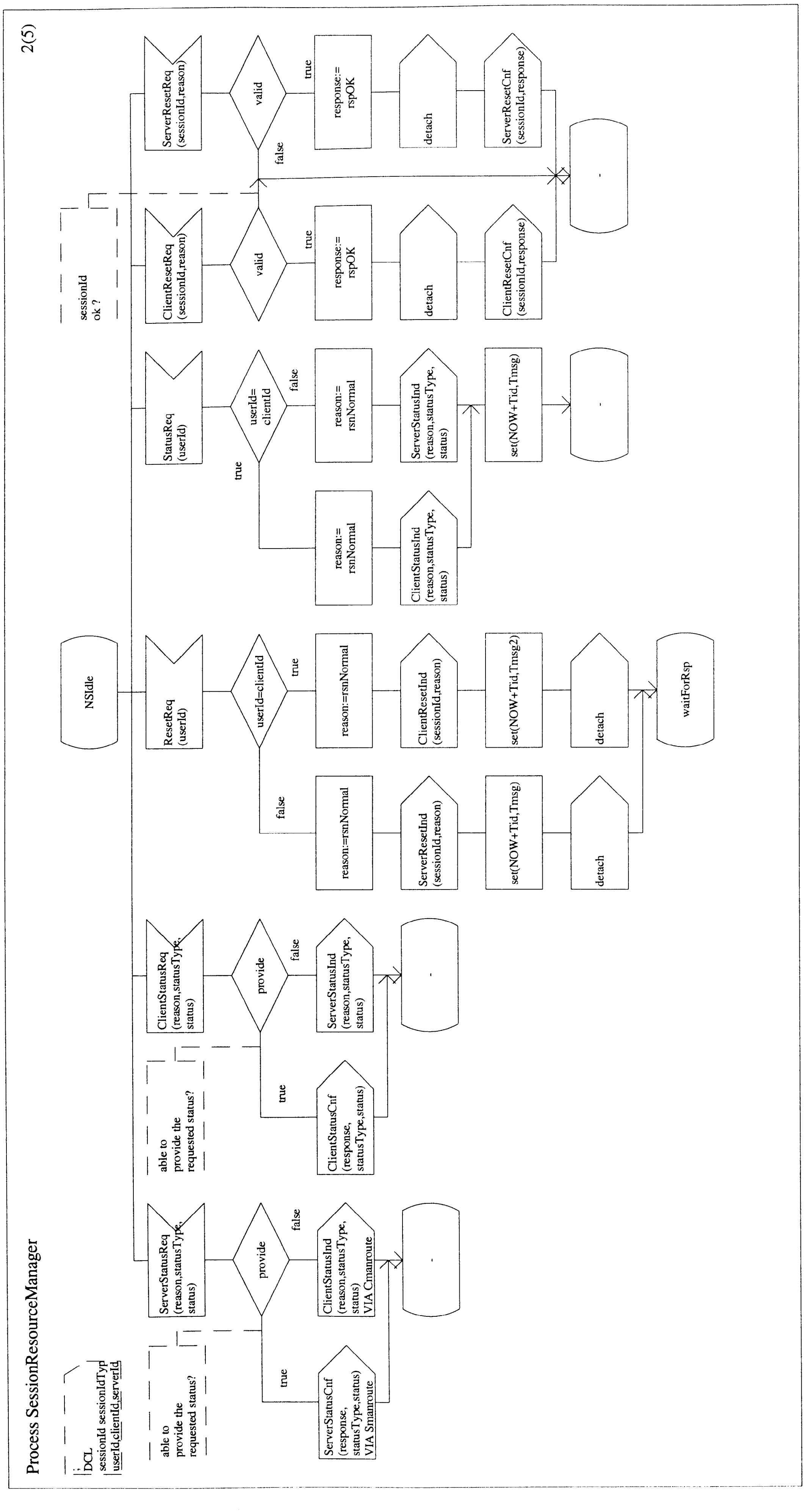

Process ClientSession
2(4)
DCL
sessionId sessionIdType,
clientId NSAPType:=clientNSAP,
serverId NSAPType:=ServerANSAP,
resource resourceCountType,
reason reasonType,
response responseType,
contain boolean,
accepted boolean,
oldServerId,newServerId NSAPType,
userdata userdataType,
valid boolean,
counter integer:=0,
retrans integer:=0,
~~transferAccepted boolean~~;
WFCSCnf
ClientSessionSetUpCnf (sessionId,serverId, response,resource, userdata)
ClientSessionProceedingInd (reason)
Tmsg
validTransactionId
sessionId ok? & serverid ok? & resource ok?
valid
true
false
valid
true
false
-
RESET (Tmsg)
response=rspOK
true
false
set(NOW+tCSesHold, Tmsg)
CSExpire
resources accepted ?
accepted
true
false
reason:= rsnClNoResource
SET(NOW+tCSesHold Tmsg)
ClientReleaseReq (sessionId,reason, userdata)
CSExpire
user data?
contain
true
false
ClientConnectReq (sessionId, userdata)
CSActive
reason:= rsnNoTransaction
SET(NOW+tCSesHold Tmsg)
ClientReleaseReq (sessionId,reason, userdata)
CSExpire
valid
true
false
set(NOW+Tid,Tmsg)
-
counter>0
true
false
reason:= rsnClProcError
set(NOW+tCSesHold, Tmsg)
ClientReleaseReq (sessionId,reason, userdata)
CSExpire
set(NOW+Tid,Tmsg)
ClientSessionSetUpReq (sessionId,clientId, serverId,userdata)
counter:=counter+1
-


Process ClientSession
3(4)
DCL
sessionId sessionIdType,
clientId NSAPType:=clientNSAP,
serverId NSAPType:=ServerANSAP,
resource resourceCountType,
reason reasonType,
response responseType,
contain boolean,
accepted boolean,
oldServerId,newServerId NSAPType,
userdata userdataType,
valid boolean,
counter integer:=0,
retrans integer:=0,
~~transferAccepted boolean;~~
WFCRelCnf
Tmsg
retrans>0
true
false
set(NOW+tCSesHold, Tmsg)
/* release any resources assigned */
CSExpire
set(NOW+Tid,Tmsg)
reason:=rsnNormal
retrans:=retrans+1
ClientReleaseReq (sessionId,reason, userdata)
-
ClientReleaseCnf (sessionId,response, userdata)
valid
true
false
sessionId ok ?
-
response= rspOK
true
false
RESET(Tmsg)
/* release any resources assigned */
Detach VIA internal
set(NOW+Tid,Tmsg)
ClientReleaseReq (sessionId,reason, userdata)
-
CSExpire
Tmsg
Detach VIA internal
ClientReleaseCnf (sessionId,response, userdata)
valid
true
false
sessionId ok ?
-
response=rspOK
true
false
Detach VIA internal
/* increment error count */
-

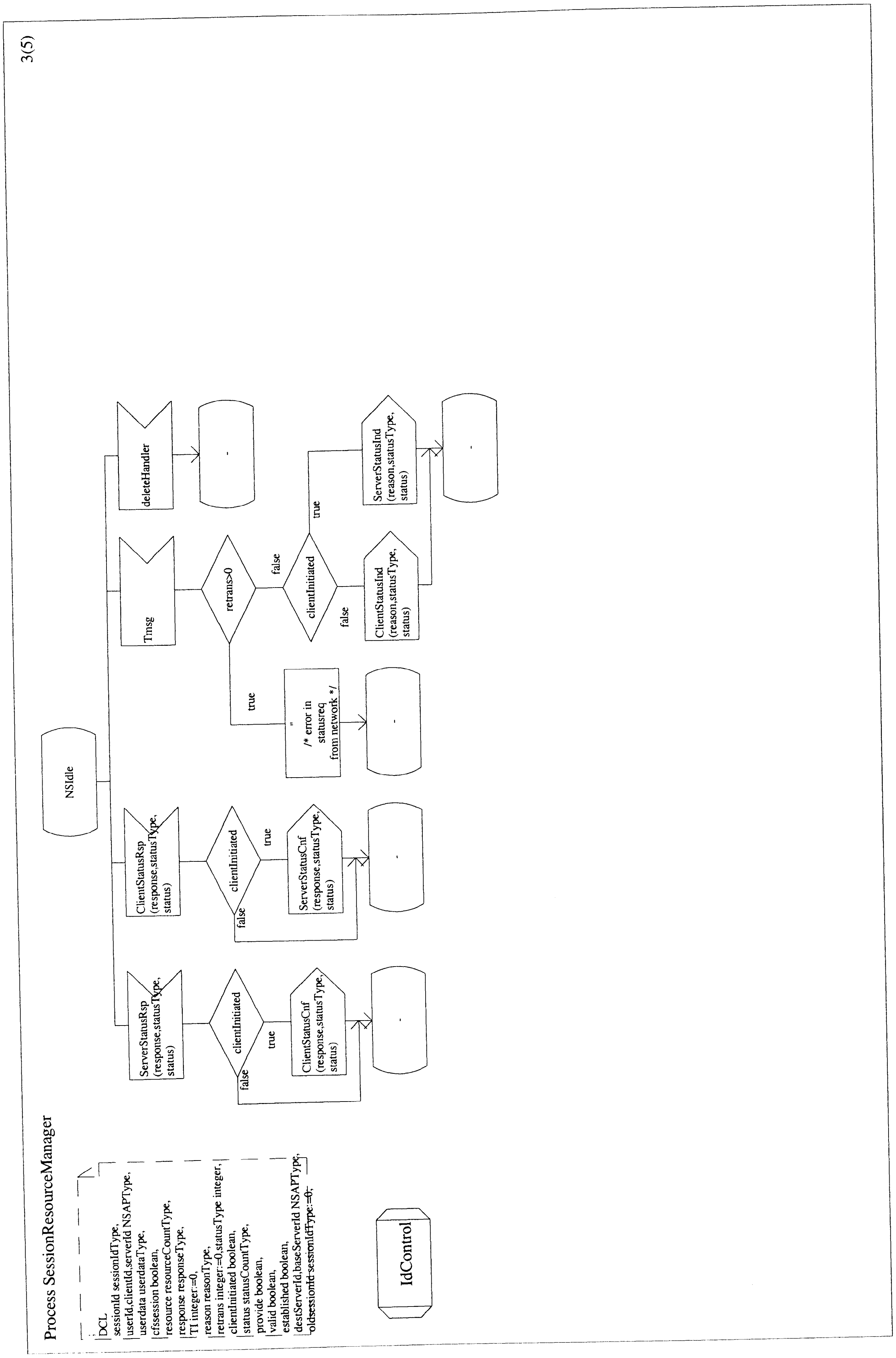

Process ClientSession
4(4)
sessionId sessionIdTy
clientId NSAPType:=
serverId NSAPType:=
resource resourceCoun
CSActive
ClientAddResourceInd (sessionId,resource)
sessionId ok ?
valid
false
response:= rspClNoSession
true
resource ok ?
valid
false
response:= rspClResourceFailed
true
resources acceptable?
accepted
false
response:= rspClNoResource
true
/* allocate resources */
response:=rspOK
ClientAddResourceRsp (sessionId,response, resource)
-
ClientSessionTransferInd (sessionId,clientId, oldServerId,newServerId, resource,userdata)
valid resources
transfer can be rejected for different reason for instant error in sessionid or serverid
TransferAccepted
true
response:=rspOK
false
response:= rspClTransferReject
ClientSessionTransferRsp (sessionId,response, userdata)
/* begin using the new resources */
release old resources belonging to the session
ClientSessionTransferRsp (sessionId,response, userdata)
userdata?
contain
true
ClientConnectReq (sessionId, userdata)
false
-
valid sessionId
ClientDeleteResourceInd (sessionId,reason, resource,userdata)
valid
true
valid
true
/* stop using the resources */
response:=rspOK
false
response:= rspClResourceFailed
ClientDeleteResourceRsp (sessionId,response, userdata)
-
valid sessionId
false
response:= rspClNoSession
ClientDeleteResourceRsp (sessionId,response, userdata)
-
ClientReleaseInd (sessionId,reason, userdata)
valid
true
/* free resources */
response:= rspOK
ClientReleaseRsp (sessionId,response, userdata)
set(NOW+tCSesHold, Tmsg)
CSExpire
false
response:= rspClNoSession
ClientReleaseRsp (sessionId,response, userdata)
/* other actions may be taken */
-


Block SessionResourceManagement
1(1)
CONNECT CNsessionChannel AND CNRsessionroute, CNSsessionRoute, Cmanroute;
CONNECT NSsessionChannel AND NRSsessionRoute, NSSsessionRoute, Smanroute;
CONNECT NSresourceChannel AND NRMSresourceRoute;
CONNECT NetworkChannel AND Networkroute;
NetworkRoute
ServiceGatewaydetachReq, StatusReq, ResetReq
SessionResourceManager(1,1)
SessionHandler(0,maxknownId)
ResourceHandler(0,maxknownId)
sessionInternal
activateResourceHandler, detachResourceHandler, deleteHandler
ReqInternal, SRMCFS, SessionReleaseInternal, detach
resourceInternal
ServerAddResourceInternal, ServerSessionTransferInternal, ServerDeleteResourceInternal, activateResourceHandler, detachResourceHandler
ClientReleaseReq, ClientReleaseRsp, ClientConnectReq
ServerSessionSetUpRsp, ServerReleaseReq, ServerReleaseRsp
ClientSessionTransferRsp, ClientAddResourceRsp, ClientDeleteResourceRsp
ServerSessionTransferRsp
ClientStatusRsp, ClientStatusReq, ClientResetReq, ClientResetRsp, ClientSessionSetUpReq
ServerStatusReq, ServerStatusRsp, ServerCFSReq, ServerResetReq, ServerResetRsp
ServerAddResourceReq, ServerSessionTransferReq, ServerDeleteResourceReq
CNSsessionRoute
NSSsessionRoute
CNRsessionRoute
NRSsessionRoute
Cmanroute
Smanroute
NRMSresourceroute
ClientStatusInd, ClientStatusCnf, ClientResetInd, ClientResetCnf, ClientSessionSetUpCnf
ServerStatusInd, ServerStatusCnf, ServerResetInd, ServerResetCnf, ServerCFSCnf
ServerDeleteResourceCnf, ServerAddResourceCnf
ClientSessionTransferInd, ClientAddResourceInd, ClientDeleteResourceInd
ServerAddResourceCnf, ServerDeleteResourceCnf, ServerSessionTransferCnf, ServerSessionTransferInd, ServerReleaseInd
ClientSessionSetUpCnf, ClientReleaseInd, ClientReleaseCnf, ClientSessionProceedingInd
ServerSessionSetUpInd, ServerCFSCnf, ServerReleaseInd, ServerReleaseCnf, ServerConnectInd

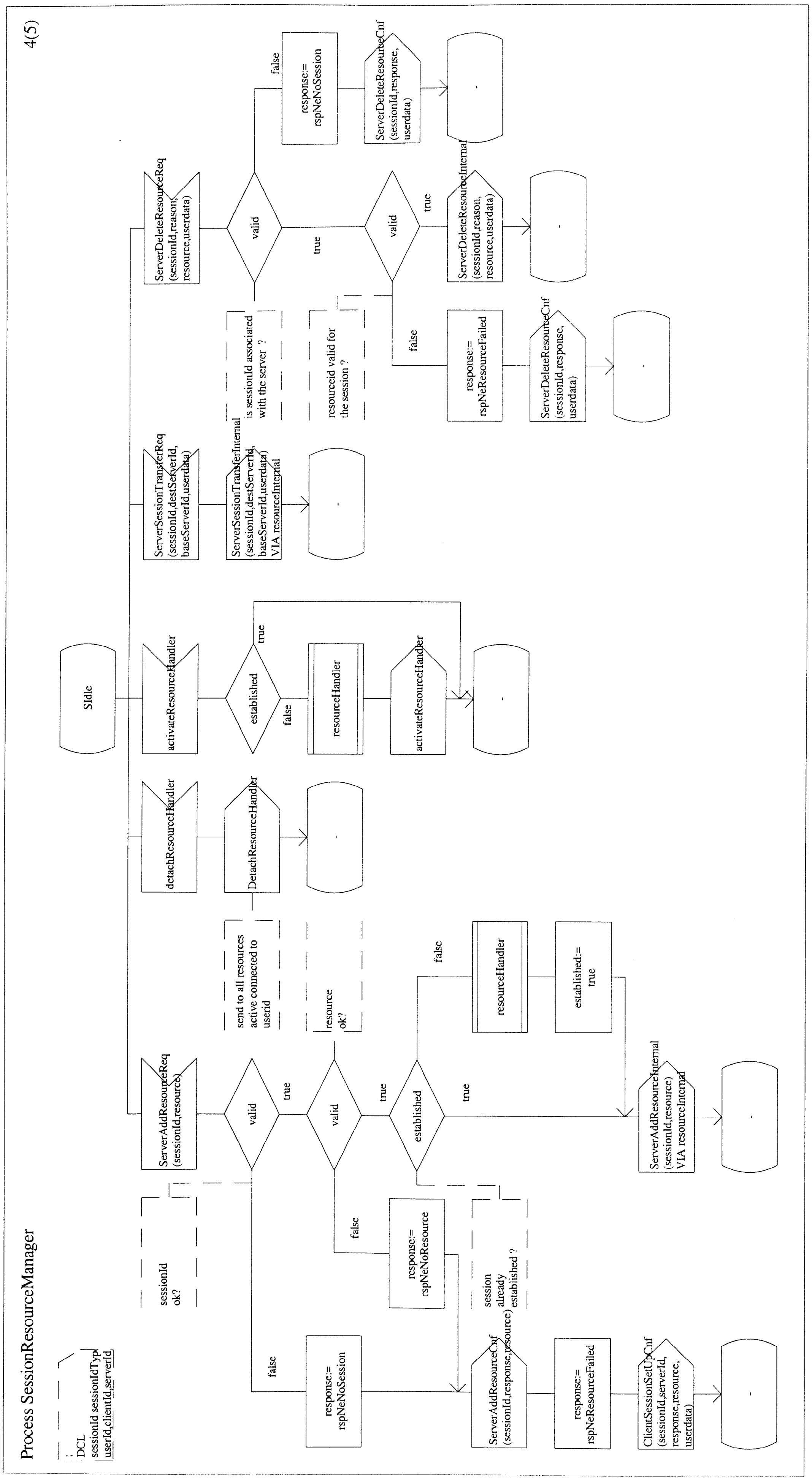

Process SessionResourceManager
1(5)
;
DCL
sessionId sessionIdType,
userId,clientId,serverId NSAPType,
userdata userdataType,
cfssession boolean,
resource resourceCountType,
response responseType,
TI integer:=0,
reason reasonType,
retrans integer:=0,statusType integer,
clientInitiated boolean,
status statusCountType,
provide boolean,
valid boolean,
established boolean,
destServerId,baseServerId NSAPType,
oldsessionId sessionIdType:=0;
TIMER
Tmsg2,Tmsg;
NSIdle
NSIdle
ServerCFSReq (sessionId,serverId, resource)
IdControl (sessionId,cfssession, serverId,clientId, response)
response= rspOK
true
false
sessionId=0
true
false
'' /* select sessionId */
SessionHandler
SRMCFS (sessionId,serverId, resource,response)
ServerCFSCnf (sessionId, response,resource)
-
ClientSessionSetUpReq (sessionId,clientId,serverId, userdata)
IdControl (sessionId,cfssession, serverId,clientId, response)
response= rspOK
false
true
ClientSessionSetUpCnf (sessionId,serverId, response,resource, userdata)
sessionId=0
true
false
'' /* select sessionId */
SessionHandler
ReqInternal (sessionId,clientId, serverId,userdata) TO OFFSPRING
-
ServiceGatewaydetachReq (TI,userdata)
valid
true
false
valid TI = belong to an existing session
'' /* sessionId will be set to the same as the existing session */
SessionReleaseInternal (sessionId,userdata)
-

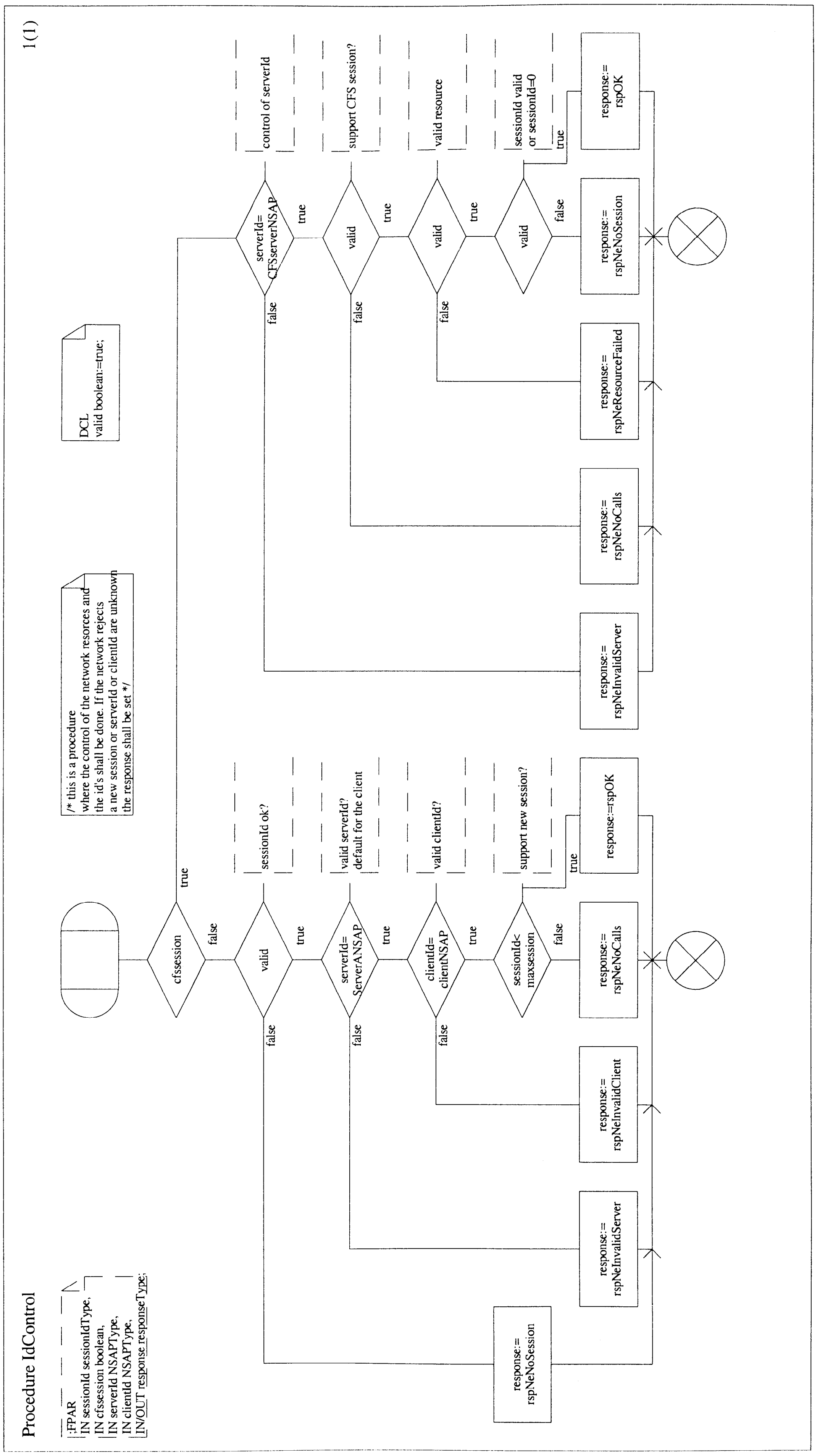


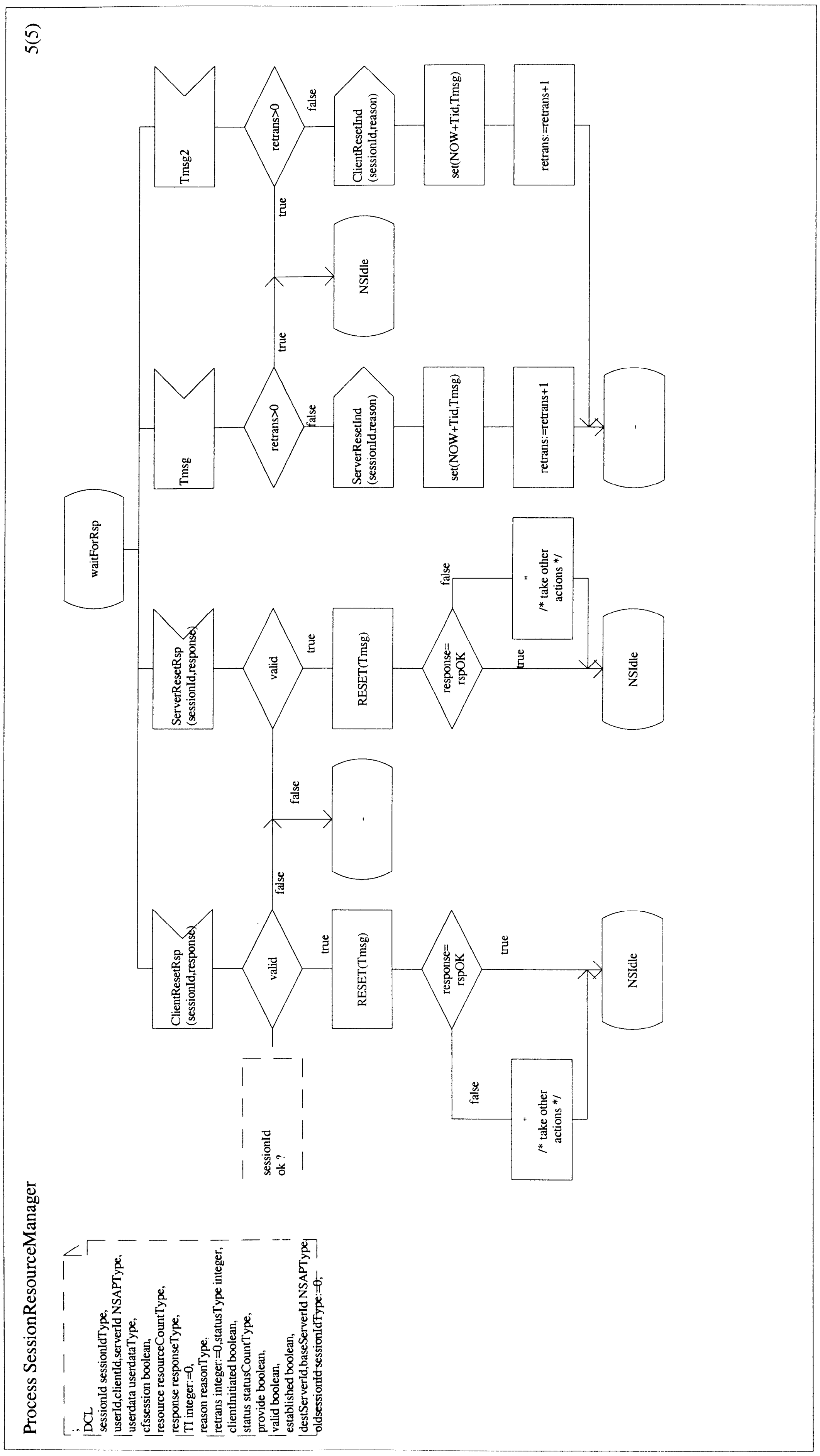
Process SessionResourceManager
2(5)
;
DCL
sessionId sessionIdTyp
userId,clientId,serverId
NSIdle
ServerStatusReq
(reason,statusType,
status)
able to
provide the
requested status?
provide
true
false
ServerStatusCnf
(response,
statusType,status)
VIA Smanroute
ClientStatusInd
(reason,statusType,
status)
VIA Cmanroute
-
ClientStatusReq
(reason,statusType,
status)
ClientStatusCnf
(response,
statusType,status)
ServerStatusInd
(reason,statusType,
status)
ResetReq
(userId)
userId=clientId
reason:=rsnNormal
ClientResetInd
(sessionId,reason)
ServerResetInd
(sessionId,reason)
set(NOW+Tid,Tmsg2)
set(NOW+Tid,Tmsg)
detach
waitForRsp
StatusReq
(userId)
userId=
clientId
reason:=
rsnNormal
sessionId
ok ?
ClientResetReq
(sessionId,reason)
valid
response:=
rspOK
ClientResetCnf
(sessionId,response)
ServerResetReq
(sessionId,reason)
ServerResetCnf
(sessionId,response)


3(5)
Process SessionResourceManager
NSIdle
ServerStatusRsp (response,statusType, status)
clientInitiated
true
false
ClientStatusCnf (response,statusType, status)
-
ClientStatusRsp (response,statusType, status)
clientInitiated
true
false
ServerStatusCnf (response,statusType, status)
-
Tmsg
retrans>0
true
'' /* error in statusreq from network */
-
false
clientInitiated
true
false
ServerStatusInd (reason,statusType, status)
ClientStatusInd (reason,statusType, status)
-
deleteHandler
-
IdControl

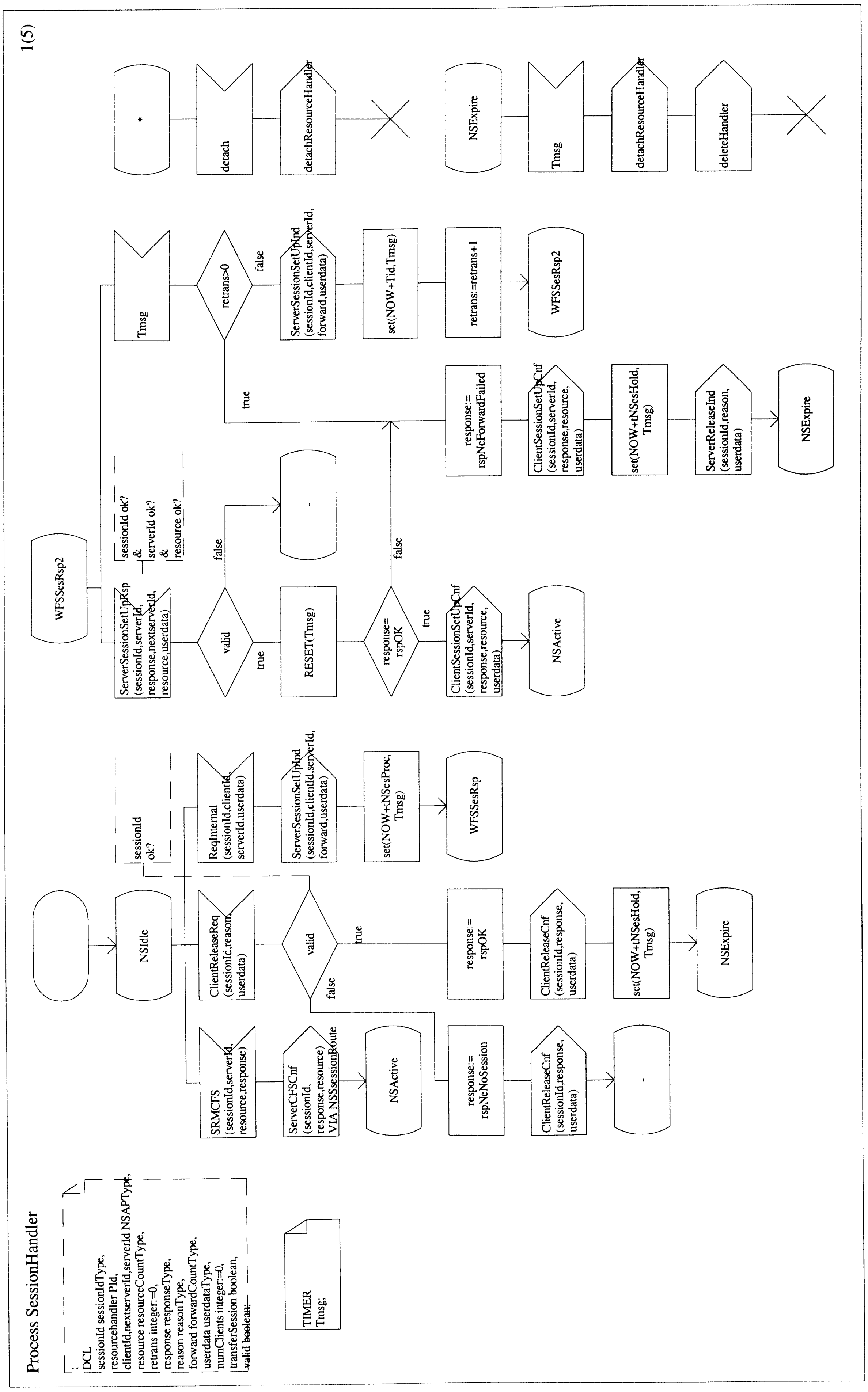
```
DCL
sessionId sessionIdType,
userId,clientId,serverId NSAPType,
userdata userdataType,
cfssession boolean,
resource resourceCountType,
response responseType,
TI integer:=0,
reason reasonType,
retrans integer:=0,statusType integer,
clientInitiated boolean,
status statusCountType,
provide boolean,
valid boolean,
established boolean,
destServerId,baseServerId NSAPType,
oldsessionId sessionIdType:=0;
```

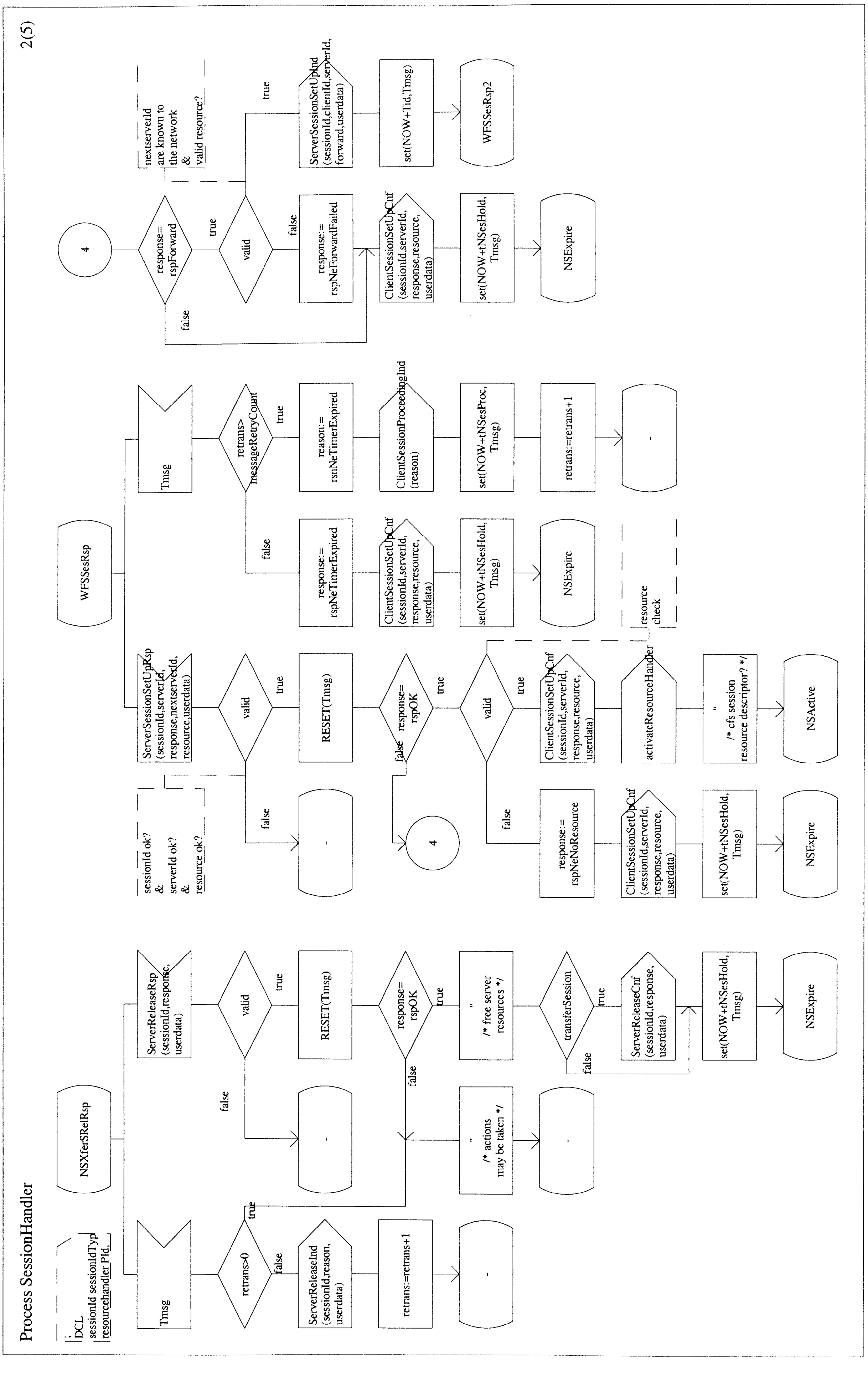

Process SessionResourceManager
4(5)
DCL
sessionId sessionIdType
userId,clientId,serverId
SIdle
ServerAddResourceReq (sessionId,resource)
valid
sessionId ok?
false
response:= rspNeNoSession
true
valid
resource ok?
false
response:= rspNeNoResource
ServerAddResourceCnf (sessionId,response,resource)
response:= rspNeResourceFailed
ClientSessionSetUpCnf (sessionId,serverId, response,resource, userdata)
true
established
session already established ?
false
resourceHandler
established:= true
true
ServerAddResourceInternal (sessionId,resource) VIA resourceInternal
send to all resources active connected to userid
detachResourceHandler
DetachResourceHandler
activateResourceHandler
established
true
false
resourceHandler
activateResourceHandler
ServerSessionTransferReq (sessionId,destServerId, baseServerId,userdata)
ServerSessionTransferInternal (sessionId,destServerId, baseServerId,userdata) VIA resourceInternal
ServerDeleteResourceReq (sessionId,reason, resource,userdata)
valid
is sessionId associated with the server ?
false
response:= rspNeNoSession
ServerDeleteResourceCnf (sessionId,response, userdata)
true
valid
resourceid valid for the session ?
true
ServerDeleteResourceInternal (sessionId,reason, resource,userdata)
false
response:= rspNeResourceFailed
ServerDeleteResourceCnf (sessionId,response, userdata)
-

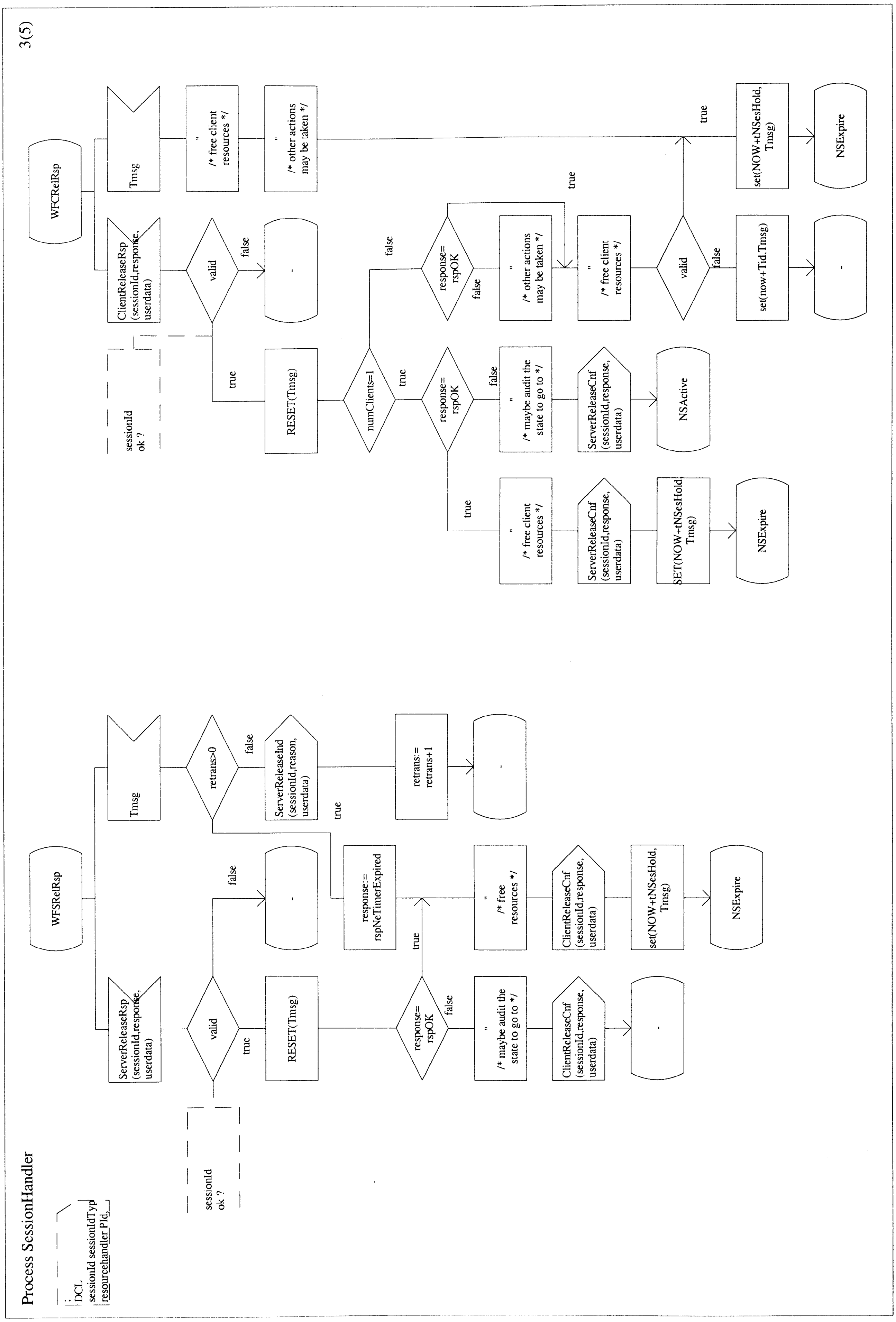

Procedure IdControl
1(1)
;FPAR
IN sessionId sessionIdType,
IN cfssession boolean,
IN serverId NSAPType,
IN clientId NSAPType,
IN/OUT response responseType;
DCL
valid boolean:=true;
/* this is a procedure
where the control of the network resorces and
the id's shall be done. If the network rejects
a new session or serverId or clientId are unknown
the response shall be set */
cfssession
true
false
valid
serverId=
ServerANSAP
clientId=
clientNSAP
sessionId<
maxsession
sessionId ok?
valid serverId?
default for the client
valid clientId?
support new session?
response:=
rspNeNoSession
response:=
rspNeInvalidServer
response:=
rspNeInvalidClient
response:=
rspNeNoCalls
response:=rspOK
serverId=
CFSserverNSAP
control of serverId
support CFS session?
valid resource
sessionId valid
or sessionId=0
response:=
rspNeInvalidServer
response:=
rspNeNoCalls
response:=
rspNeResourceFailed
response:=
rspNeNoSession
response:=
rspOK

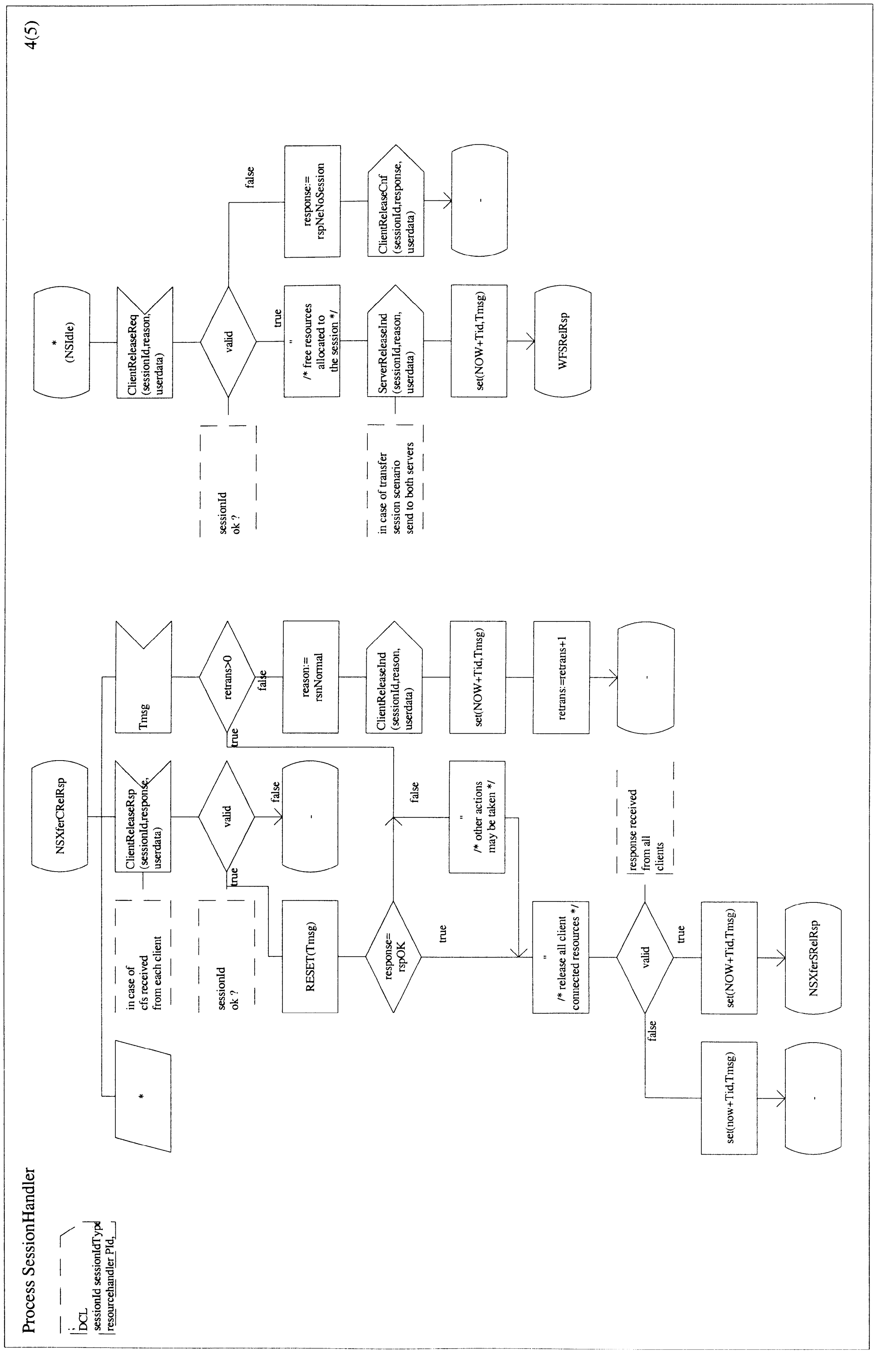


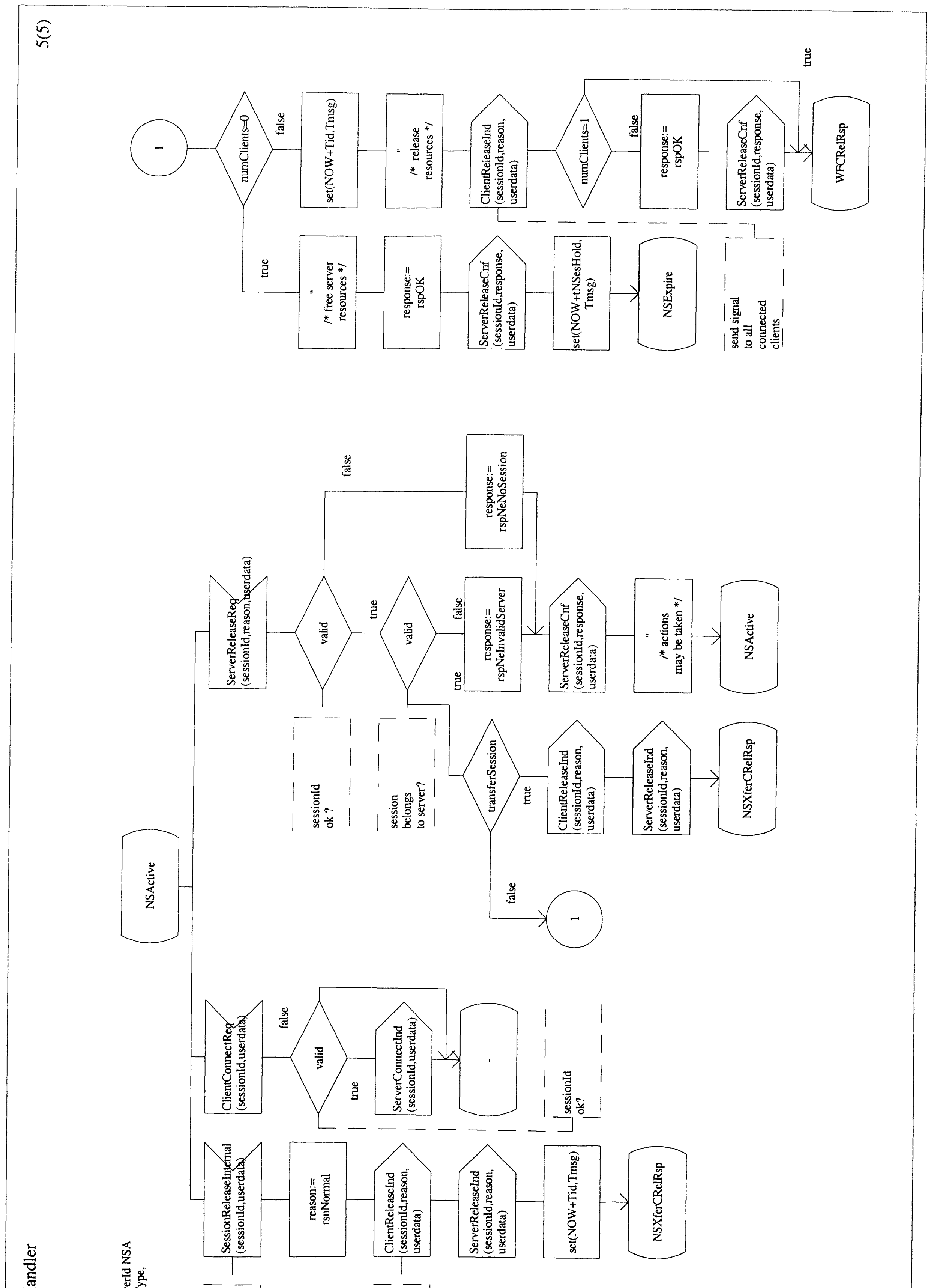
Process SessionResourceManager
5(5)
DCL
sessionId sessionIdType,
userId,clientId,serverId NSAPType,
userdata userdataType,
cfssession boolean,
resource resourceCountType,
response responseType,
TI integer:=0,
reason reasonType,
retrans integer:=0,statusType integer,
clientInitiated boolean,
status statusCountType,
provide boolean,
valid boolean,
established boolean,
destServerId,baseServerId NSAPType,
old~~sessionId sessionIdType:=0,~~
waitForRsp
ClientResetRsp (sessionId,response)
sessionId ok ?
valid
true
false
RESET(Tmsg)
response= rspOK
true
false
'' /* take other actions */
NSIdle
-
ServerResetRsp (sessionId,response)
valid
true
false
RESET(Tmsg)
response= rspOK
true
false
'' /* take other actions */
NSIdle
Tmsg
retrans>0
false
true
ServerResetInd (sessionId,reason)
set(NOW+Tid,Tmsg)
retrans:=retrans+1
-
NSIdle
Tmsg2
retrans>0
false
true
ClientResetInd (sessionId,reason)
set(NOW+Tid,Tmsg)
retrans:=retrans+1

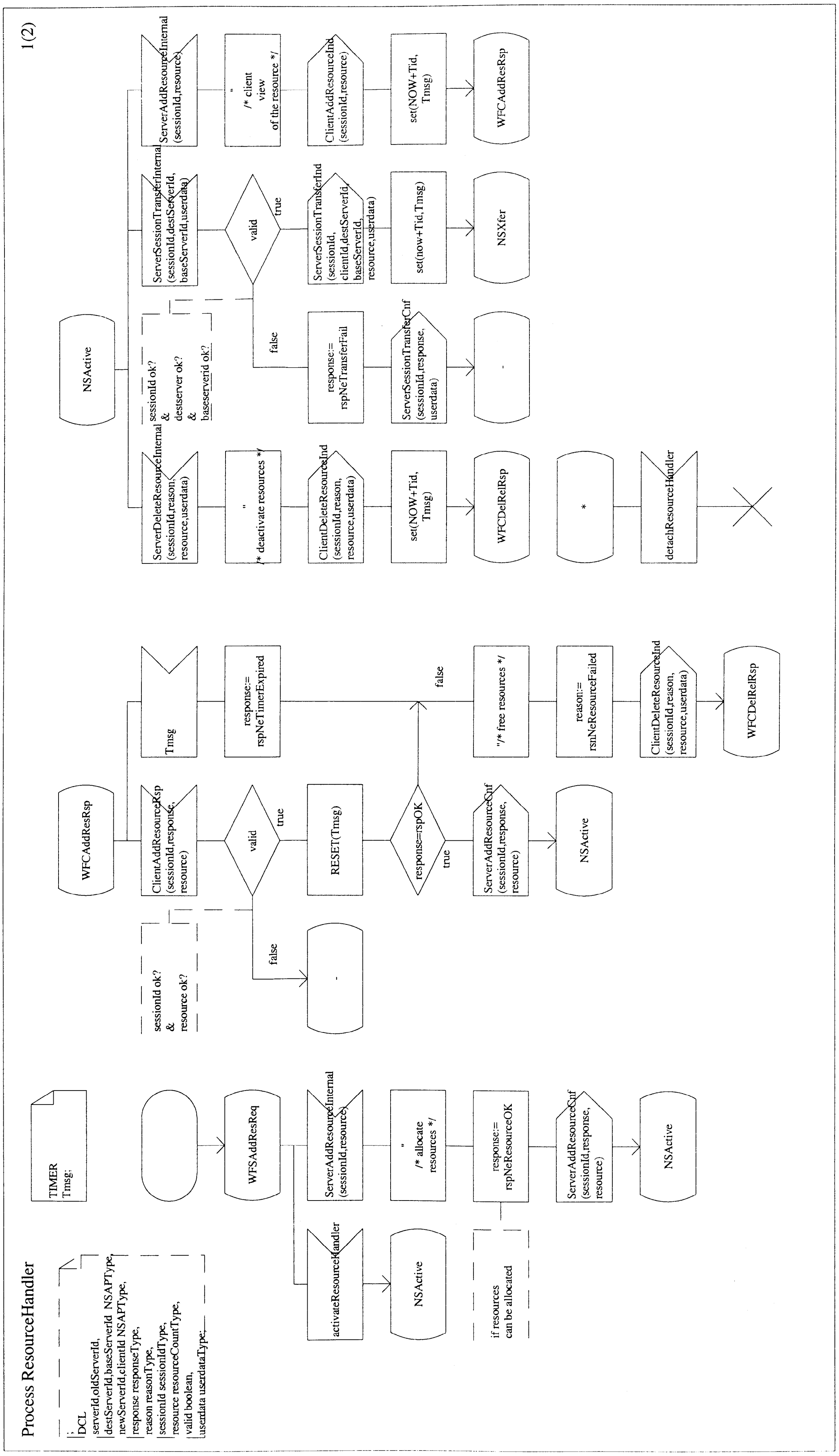

Process SessionHandler
1(5)
DCL
sessionId sessionIdType,
resourcehandler PId,
clientId,nextserverId,serverId NSAPType,
resource resourceCountType,
retrans integer:=0,
response responseType,
reason reasonType,
forward forwardCountType,
userdata userdataType,
numClients integer:=0,
transferSession boolean,
~~valid boolean;~~
TIMER
Tmsg;
NSIdle
SRMCFS (sessionId,serverId, resource,response)
ServerCFSCnf (sessionId, response,resource) VIA NSSessionRoute
NSActive
ClientReleaseReq (sessionId,reason, userdata)
valid
true
false
response:= rspOK
ClientReleaseCnf (sessionId,response, userdata)
set(NOW+tNSesHold, Tmsg)
NSExpire
response:= rspNeNoSession
ClientReleaseCnf (sessionId,response, userdata)
-
sessionId ok?
ReqInternal (sessionId,clientId, serverId,userdata)
ServerSessionSetUpInd (sessionId,clientId,serverId, forward,userdata)
set(NOW+tNSesProc, Tmsg)
WFSSesRsp
WFSSesRsp2
sessionId ok? & serverId ok? & resource ok?
ServerSessionSetUpRsp (sessionId,serverId, response,nextserverId, resource,userdata)
valid
true
false
RESET(Tmsg)
response= rspOK
true
false
ClientSessionSetUpCnf (sessionId,serverId, response,resource, userdata)
NSActive
-
Tmsg
retrans>0
true
false
ServerSessionSetUpInd (sessionId,clientId,serverId, forward,userdata)
set(NOW+Tid,Tmsg)
retrans:=retrans+1
WFSSesRsp2
response:= rspNeForwardFailed
ClientSessionSetUpCnf (sessionId,serverId, response,resource, userdata)
set(NOW+tNSesHold, Tmsg)
ServerReleaseInd (sessionId,reason, userdata)
NSExpire
*
detach
detachResourceHandler
NSExpire
Tmsg
detachResourceHandler
deleteHandler

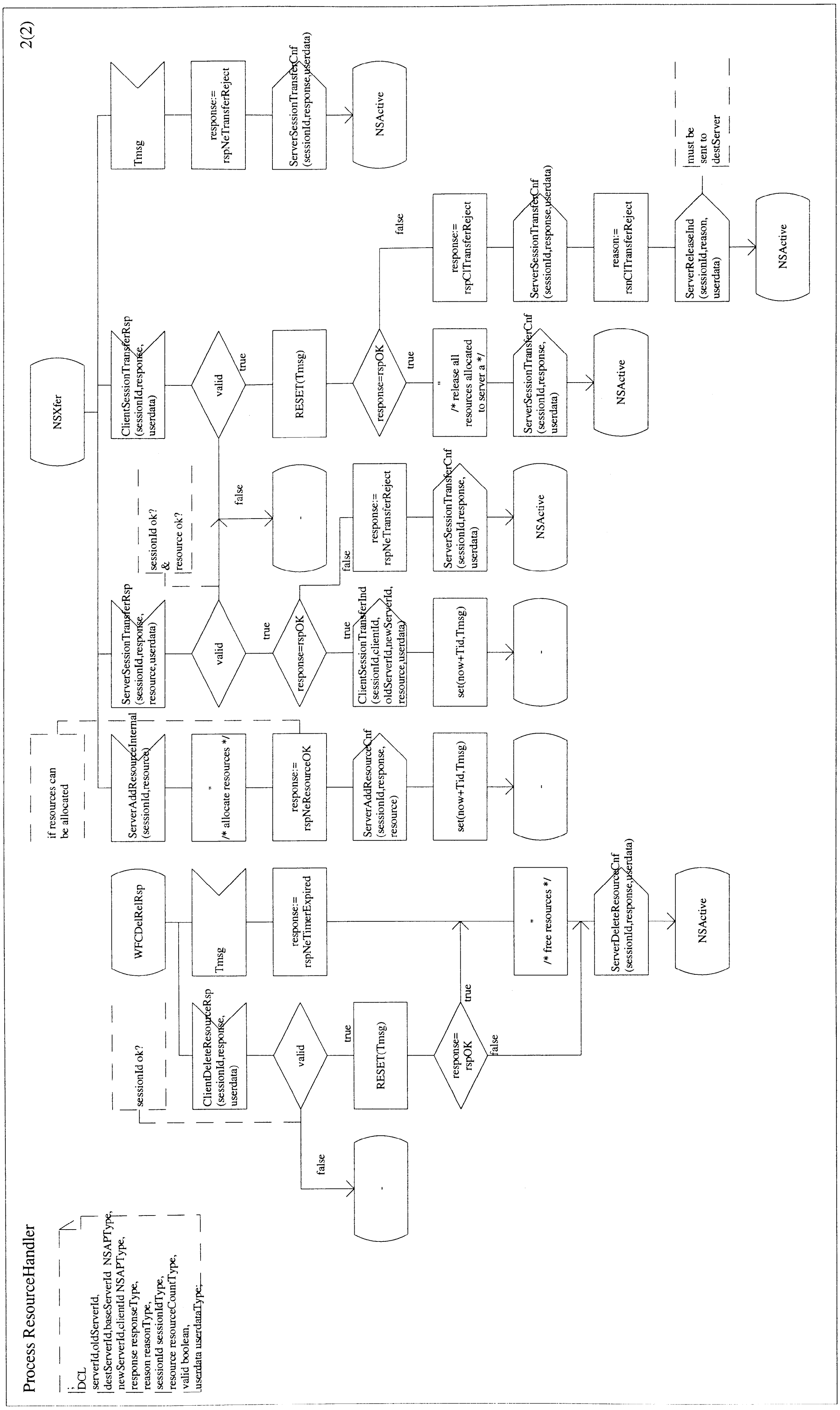

Process SessionHandler
2(5)
;
DCL
sessionId sessionIdTyp
resourcehandler PId
Tmsg
retrans>0
true
false
ServerReleaseInd (sessionId,reason, userdata)
retrans:=retrans+1
-
NSXferSRelRsp
ServerReleaseRsp (sessionId,response, userdata)
valid
true
false
RESET(Tmsg)
response= rspOK
true
false
/* free server resources */
/* actions may be taken */
transferSession
ServerReleaseCnf (sessionId,response, userdata)
set(NOW+tNSesHold, Tmsg)
NSExpire
sessionId ok? & serverId ok? & resource ok?
WFSSesRsp
ServerSessionSetUpRsp (sessionId,serverId, response,nextserverId, resource,userdata)
4
ClientSessionSetUpCnf (sessionId,serverId, response,resource, userdata)
activateResourceHandler
/* cfs session resource descriptor? */
NSActive
response:= rspNeNoResource
resource check
response:= rspNeTimerExpired
retrans> messageRetryCount
reason:= rsnNeTimerExpired
ClientSessionProceedingInd (reason)
set(NOW+tNSesProc, Tmsg)
response= rspForward
response:= rspNeForwardFailed
nextserverId are known to the network & valid resource?
ServerSessionSetUpInd (sessionId,clientId,serverId, forward,userdata)
set(NOW+Tid,Tmsg)
WFSSesRsp2

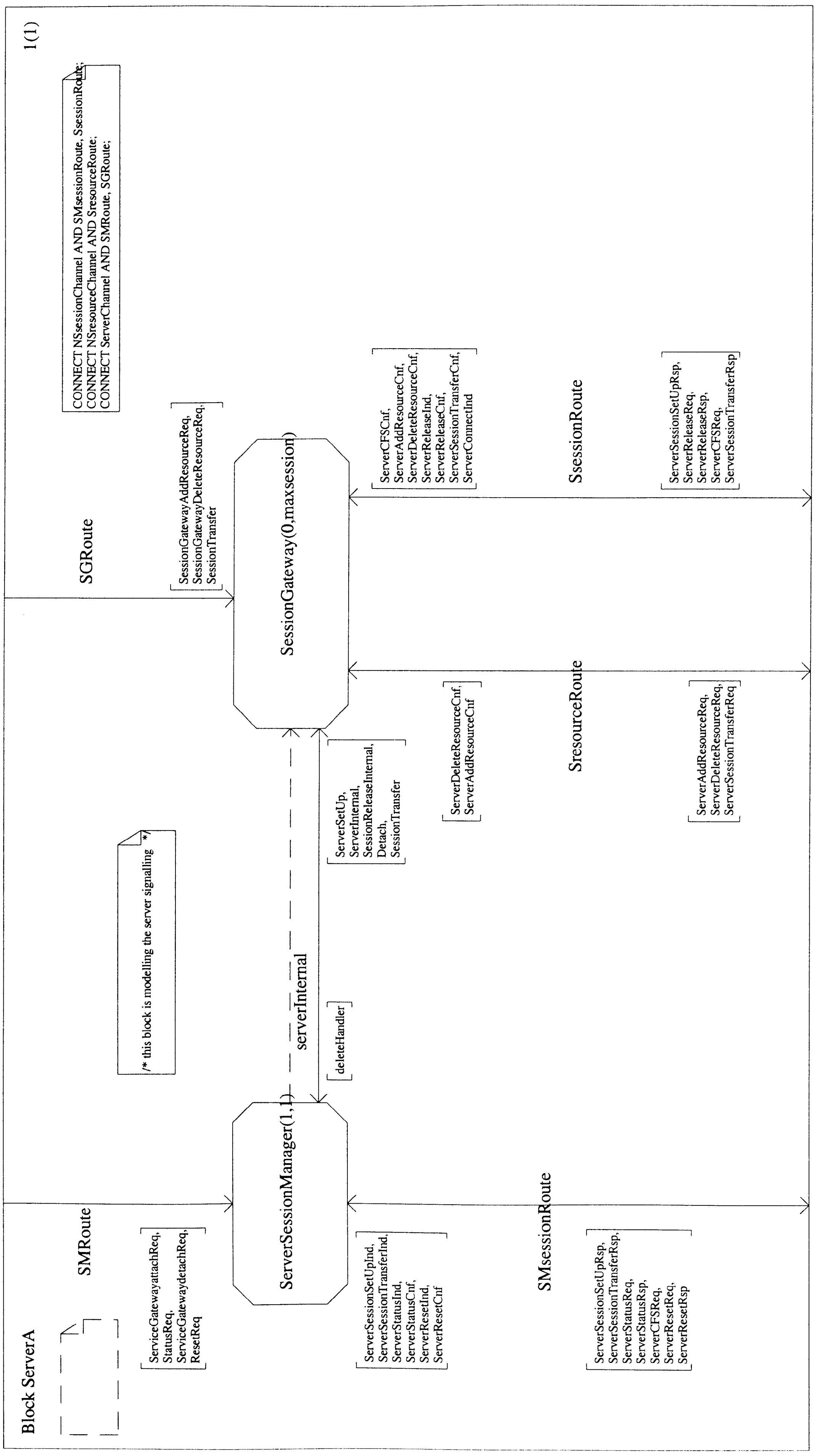

Process SessionHandler
3(5)
DCL
sessionId sessionIdTyp
resourcehandler PId,
WFSRelRsp
ServerReleaseRsp (sessionId,response, userdata)
sessionId ok ?
valid
true
false
RESET(Tmsg)
response= rspOK
/* maybe audit the state to go to */
ClientReleaseCnf (sessionId,response, userdata)
-
Tmsg
retrans>0
ServerReleaseInd (sessionId,reason, userdata)
retrans:= retrans+1
response:= rspNeTimerExpired
/* free resources */
set(NOW+tNSesHold, Tmsg)
NSExpire
WFCRelRsp
ClientReleaseRsp (sessionId,response, userdata)
numClients=1
NSActive
ServerReleaseCnf (sessionId,response, userdata)
/* free client resources */
SET(NOW+tNSesHold, Tmsg)
/* other actions may be taken */
set(now+Tid,Tmsg)

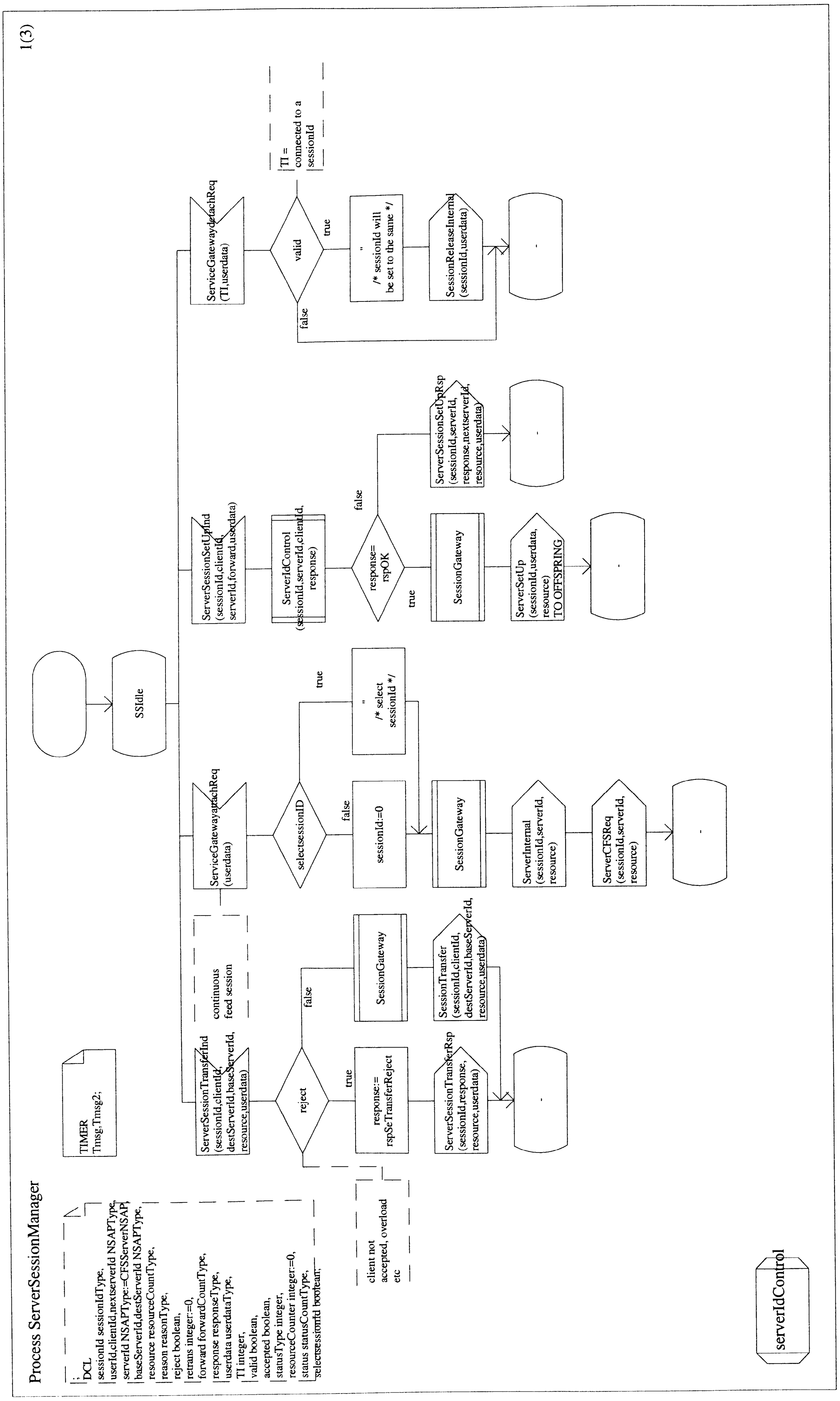

Process SessionHandler
4(5)
;
DCL
sessionId sessionIdType
resourcehandler PId,
NSXferCRelRsp
ClientReleaseRsp (sessionId,response, userdata)
Tmsg
*
in case of cfs received from each client
valid
false
true
-
sessionId ok ?
retrans>0
false
true
reason:= rsnNormal
ClientReleaseInd (sessionId,reason, userdata)
set(NOW+Tid,Tmsg)
retrans:=retrans+1
-
RESET(Tmsg)
response= rspOK
true
false
/* other actions may be taken */
/* release all client connected resources */
response received from all clients
valid
true
false
set(NOW+Tid,Tmsg)
NSXferSRelRsp
set(now+Tid,Tmsg)
-
*
(NSIdle)
ClientReleaseReq (sessionId,reason, userdata)
valid
true
false
sessionId ok ?
/* free resources allocated to the session */
response:= rspNeNoSession
ServerReleaseInd (sessionId,reason, userdata)
in case of transfer session scenario send to both servers
ClientReleaseCnf (sessionId,response, userdata)
set(NOW+Tid,Tmsg)
WFSRelRsp
-

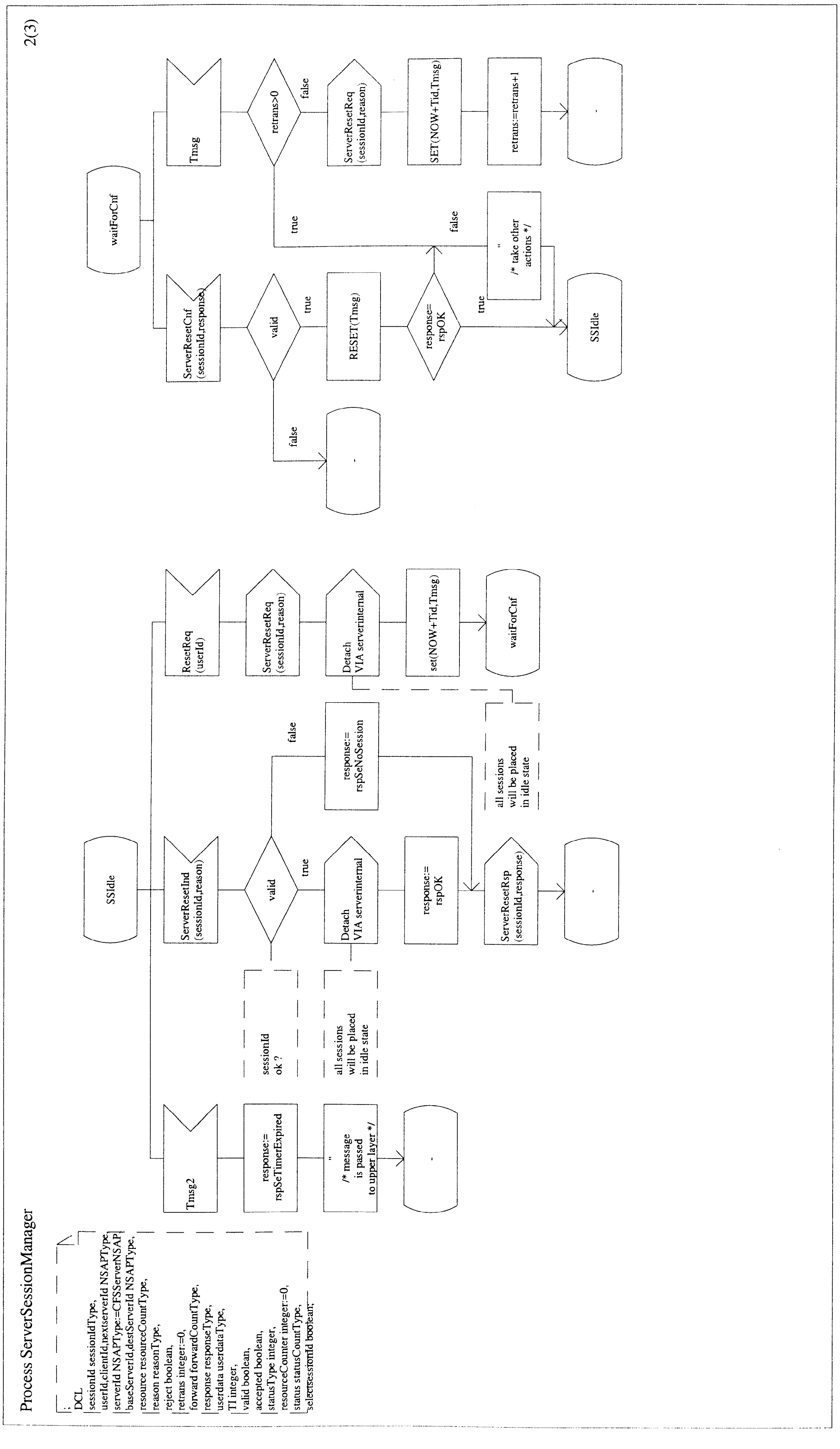

Process SessionHandler
5(5)
NSActive
SessionReleaseInternal (sessionId,userdata)
network initiated release
reason:= rsnNormal
ClientReleaseInd (sessionId,reason, userdata)
in case of cfs send to each client
ServerReleaseInd (sessionId,reason, userdata)
set(NOW+Tid,Tmsg)
NSXferCRelRsp
ClientConnectReq (sessionId,userdata)
valid
true
false
ServerConnectInd (sessionId,userdata)
-
sessionId ok?
ServerReleaseReq (sessionId,reason,userdata)
sessionId ok ?
session belongs to server?
response:= rspNeInvalidServer
response:= rspNeNoSession
ServerReleaseCnf (sessionId,response, userdata)
/* actions may be taken */
transferSession
1
numClients=0
/* free server resources */
response:= rspOK
set(NOW+tNSesHold, Tmsg)
NSExpire
send signal to all connected clients
set(NOW+Tid,Tmsg)
/* release resources */
numClients=1
WFCRelRsp

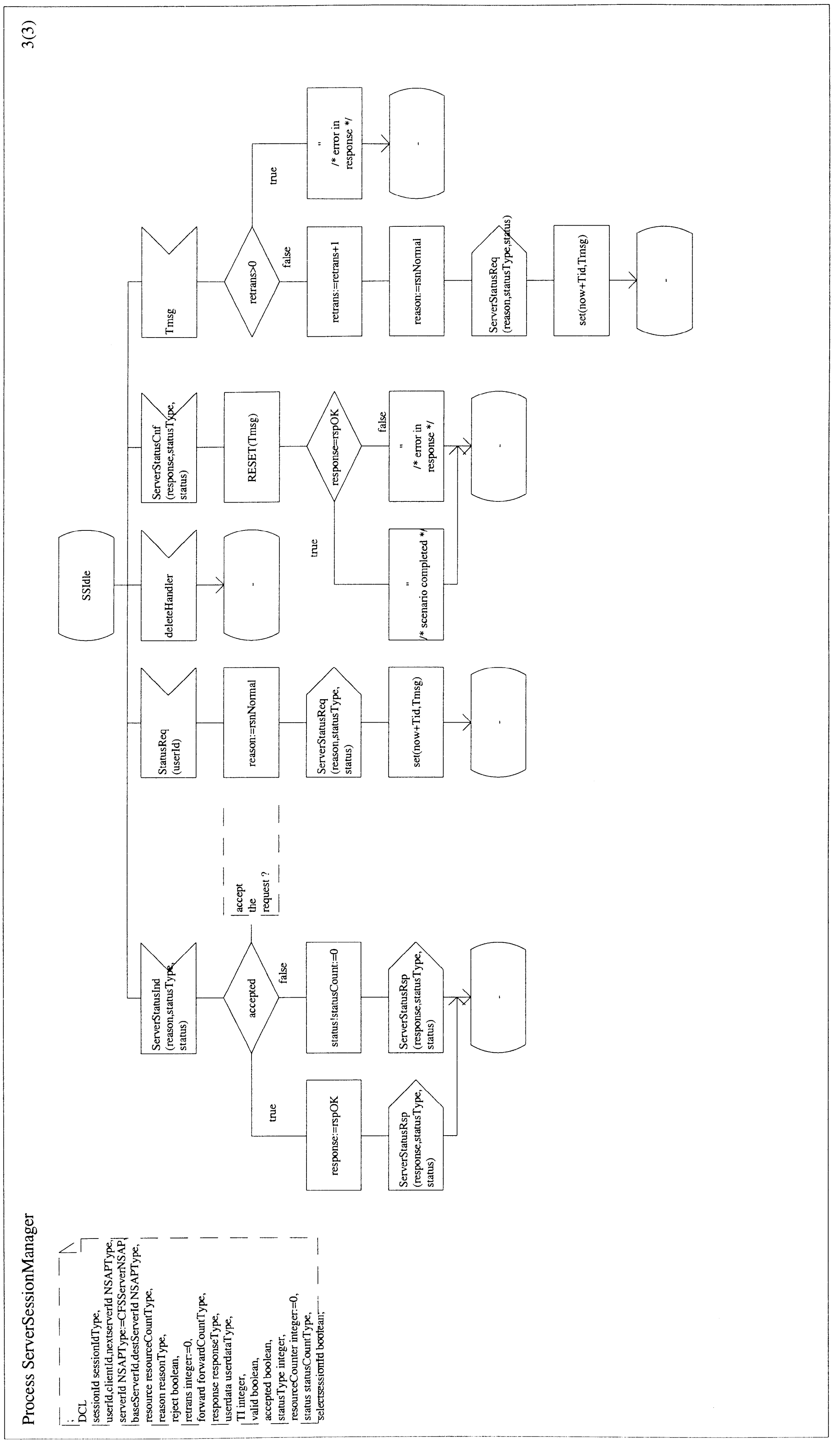

Process ResourceHandler
1(2)
DCL
serverId,oldServerId,
destServerId,baseServerId NSAPType,
newServerId,clientId NSAPType,
response responseType,
reason reasonType,
sessionId sessionIdType,
resource resourceCountType,
valid boolean,
userdata userdataType;
TIMER
Tmsg;
WFSAddResReq
ServerAddResourceInternal (sessionId,resource)
/* allocate resources */
response:= rspNeResourceOK
if resources can be allocated
ServerAddResourceCnf (sessionId,response, resource)
NSActive
activateResourceHandler
NSActive
WFCAddResRsp
ClientAddResourceRsp (sessionId,response, resource)
sessionId ok? & resource ok?
valid
true
false
-
RESET(Tmsg)
response=rspOK
true
false
ServerAddResourceCnf (sessionId,response, resource)
NSActive
Tmsg
response:= rspNeTimerExpired
/* free resources */
reason:= rsnNeResourceFailed
ClientDeleteResourceInd (sessionId,reason, resource,userdata)
WFCDelRelRsp
NSActive
ServerDeleteResourceInternal (sessionId,reason, resource,userdata)
/* deactivate resources */
ClientDeleteResourceInd (sessionId,reason, resource,userdata)
set(NOW+Tid, Tmsg)
WFCDelRelRsp
*
detachResourceHandler
sessionId ok? & destserver ok? & baseserverid ok?
ServerSessionTransferInternal (sessionId,destServerId, baseServerId,userdata)
valid
true
false
response:= rspNeTransferFail
ServerSessionTransferCnf (sessionId,response, userdata)
-
ServerSessionTransferInd (sessionId, clientId,destServerId, baseServerId, resource,userdata)
set(now+Tid,Tmsg)
NSXfer
ServerAddResourceInternal (sessionId,resource)
/* client view of the resource */
ClientAddResourceInd (sessionId,resource)
set(NOW+Tid, Tmsg)
WFCAddResRsp

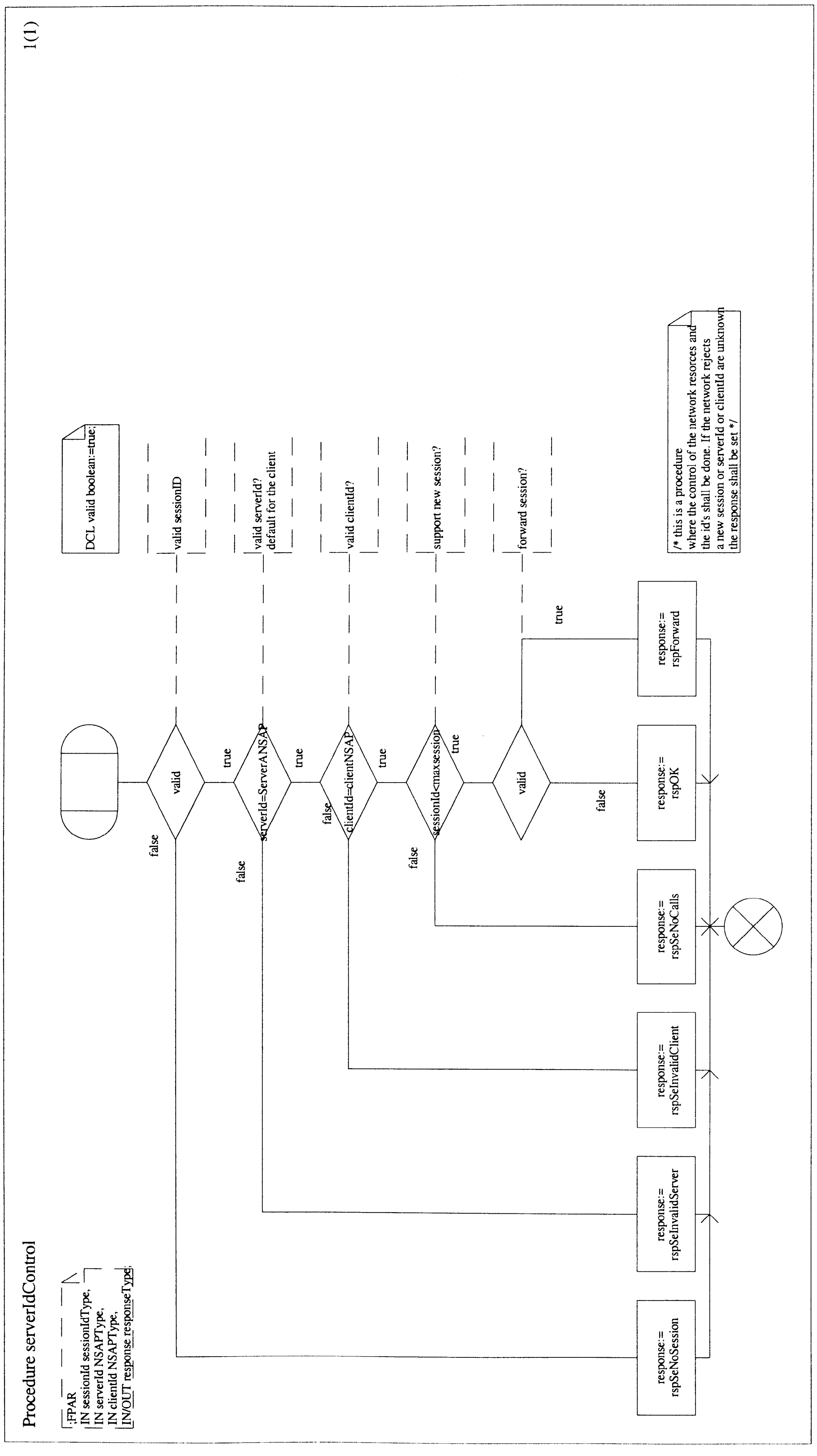

Process ResourceHandler
2(2)
;
DCL
serverId,oldServerId,
destServerId,baseServerId NSAPType,
newServerId,clientId NSAPType,
response responseType,
reason reasonType,
sessionId sessionIdType,
resource resourceCountType,
valid boolean,
userdata userdataType;
WFCDelRelRsp
Tmsg
response:= rspNeTimerExpired
/* free resources */
ServerDeleteResourceCnf (sessionId,response,userdata)
NSActive
sessionId ok?
ClientDeleteResourceRsp (sessionId,response, userdata)
valid
true
false
RESET(Tmsg)
response= rspOK
-
if resources can be allocated
ServerAddResourceInternal (sessionId,resource)
/* allocate resources */
response:= rspNeResourceOK
ServerAddResourceCnf (sessionId,response, resource)
set(now+Tid,Tmsg)
-
ServerSessionTransferRsp (sessionId,response, resource,userdata)
response=rspOK
ClientSessionTransferInd (sessionId,clientId, oldServerId,newServerId, resource,userdata)
sessionId ok? & resource ok?
response:= rspNeTransferReject
ServerSessionTransferCnf (sessionId,response, userdata)
NSXfer
ClientSessionTransferRsp (sessionId,response, userdata)
RESET(Tmsg)
/* release all resources allocated to server a */
response:= rspClTransferReject
ServerSessionTransferCnf (sessionId,response,userdata)
reason:= rsnClTransferReject
ServerReleaseInd (sessionId,reason, userdata)
must be sent to destServer
Tmsg
response:= rspNeTransferReject
ServerSessionTransferCnf (sessionId,response,userdata)


Block ServerA
1(1)
CONNECT NSessionChannel AND SMsessionRoute, SsessionRoute;
CONNECT NSresourceChannel AND SresourceRoute;
CONNECT ServerChannel AND SMRoute, SGRoute;
/* this block is modelling the server signalling */
SMRoute
ServiceGatewayattachReq, StatusReq, ServiceGatewaydetachReq, ResetReq
ServerSessionManager(1,1)
SMsessionRoute
ServerSessionSetUpInd, ServerSessionTransferInd, ServerStatusInd, ServerStatusCnf, ServerResetInd, ServerResetCnf
ServerSessionSetUpRsp, ServerSessionTransferRsp, ServerStatusReq, ServerStatusRsp, ServerCFSReq, ServerResetReq, ServerResetRsp
serverInternal
deleteHandler
ServerSetUp, ServerInternal, SessionReleaseInternal, Detach, SessionTransfer
SGRoute
SessionGatewayAddResourceReq, SessionGatewayDeleteResourceReq, SessionTransfer
SessionGateway(0,maxsession)
SresourceRoute
ServerDeleteResourceCnf, ServerAddResourceCnf
ServerAddResourceReq, ServerDeleteResourceReq, ServerSessionTransferReq
SsessionRoute
ServerCFSCnf, ServerAddResourceCnf, ServerDeleteResourceCnf, ServerReleaseInd, ServerReleaseCnf, ServerSessionTransferCnf, ServerConnectInd
ServerSessionSetUpRsp, ServerReleaseReq, ServerReleaseRsp, ServerCFSReq, ServerSessionTransferRsp

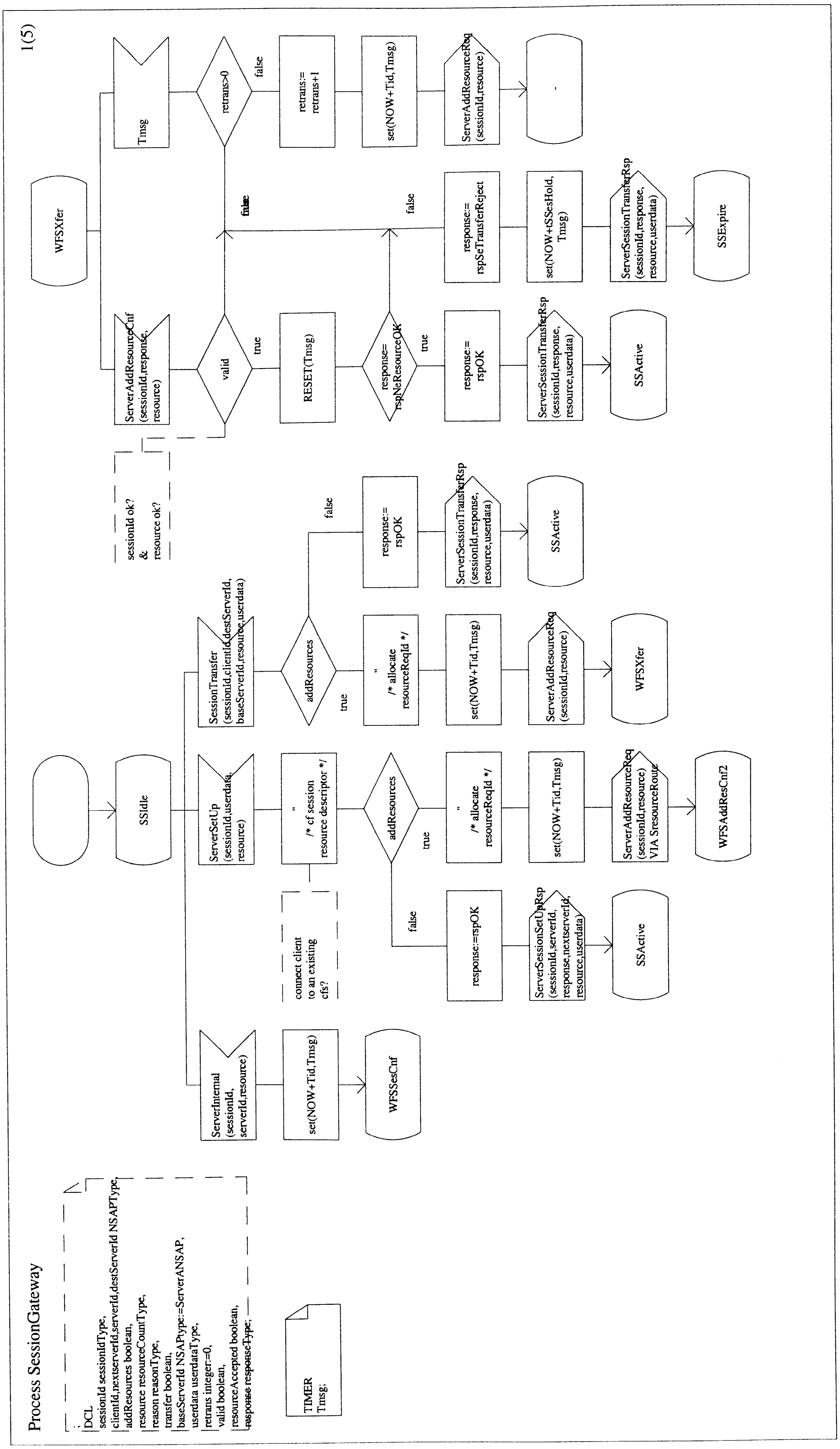

Process ServerSessionManager
1(3)
DCL
sessionId sessionIdType,
userId,clientId,nextserverId NSAPType,
serverId NSAPType:=CFSServerNSAP,
baseServerId,destServerId NSAPType,
resource resourceCountType,
reason reasonType,
reject boolean,
retrans integer:=0,
forward forwardCountType,
response responseType,
userdata userdataType,
TI integer,
valid boolean,
accepted boolean,
statusType integer,
resourceCounter integer:=0,
status statusCountType,
selectsessionId boolean;
TIMER
Tmsg,Tmsg2;
SSIdle
ServerSessionTransferInd
(sessionId,clientId,
destServerId,baseServerId,
resource,userdata)
continuous
feed session
reject
true
false
response:=
rspSeTransferReject
ServerSessionTransferRsp
(sessionId,response,
resource,userdata)
SessionGateway
SessionTransfer
(sessionId,clientId,
destServerId,baseServerId,
resource,userdata)
client not
accepted, overload
etc
ServiceGatewayAttachReq
(userdata)
selectsessionID
false
true
sessionId:=0
/* select
sessionId */
SessionGateway
ServerInternal
(sessionId,serverId,
resource)
ServerCFSReq
(sessionId,serverId,
resource)
ServerSessionSetUpInd
(sessionId,clientId,
serverId,forward,userdata)
ServerIdControl
(sessionId,serverId,clientId,
response)
response=
rspOK
true
false
SessionGateway
ServerSetUp
(sessionId,userdata,
resource)
TO OFFSPRING
ServerSessionSetUpRsp
(sessionId,serverId,
response,nextserverId,
resource,userdata)
ServiceGatewayDetachReq
(TI,userdata)
valid
true
false
/* sessionId will
be set to the same */
TI =
connected to a
sessionId
SessionReleaseInternal
(sessionId,userdata)
-
serverIdControl

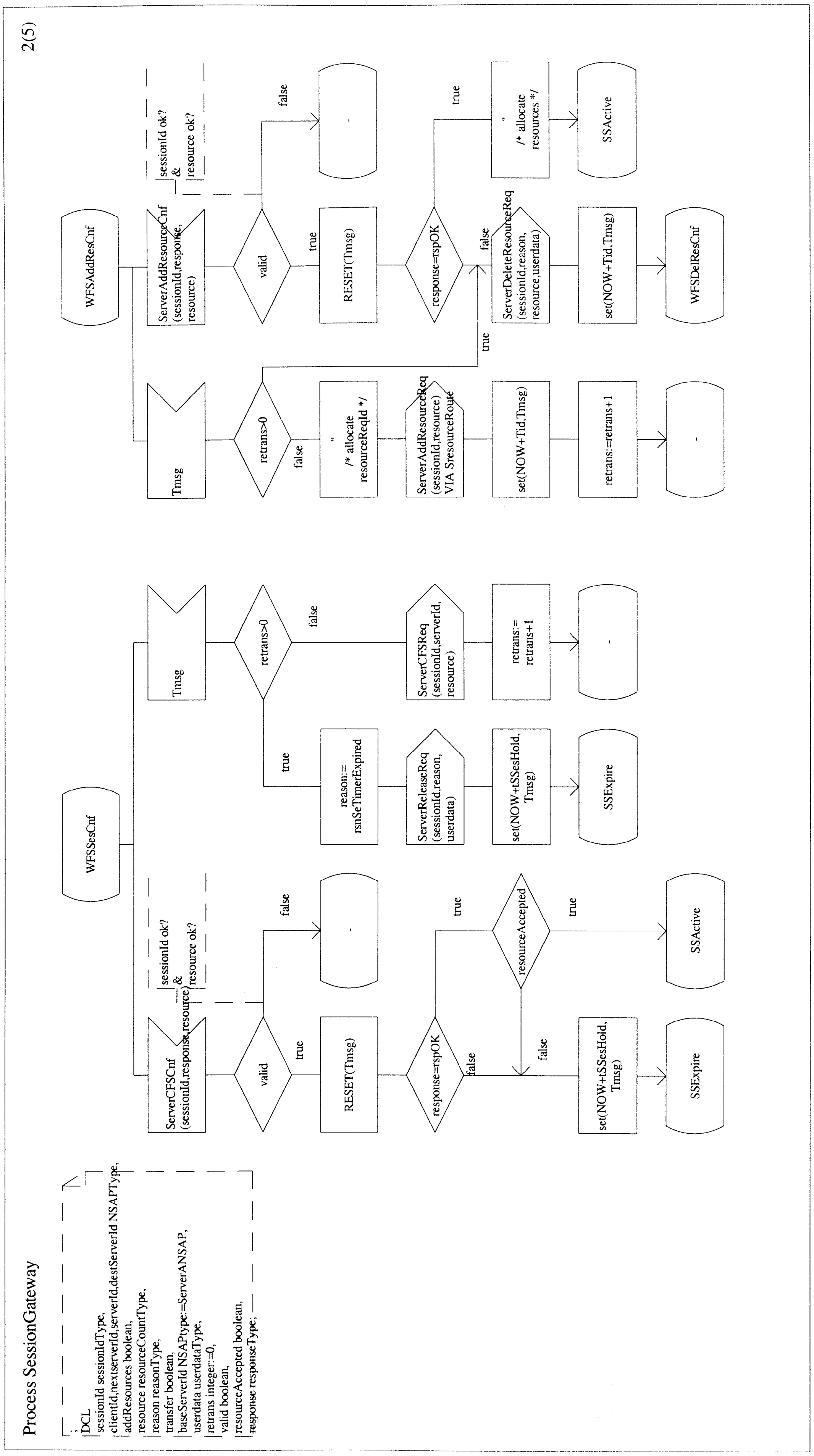

Process ServerSessionManager
2(3)
;
DCL
sessionId sessionIdType,
userId,clientId,nextserverId NSAPType,
serverId NSAPType:=CFSServerNSAP,
baseServerId,destServerId NSAPType,
resource resourceCountType,
reason reasonType,
reject boolean,
retrans integer:=0,
forward forwardCountType,
response responseType,
userdata userdataType,
TI integer,
valid boolean,
accepted boolean,
statusType integer,
resourceCounter integer:=0,
status statusCountType,
~~selectsessionId boolean;~~
SSIdle
Tmsg2
response:= rspSeTimerExpired
'' /* message is passed to upper layer */
-
ServerResetInd (sessionId,reason)
valid
sessionId ok ?
true
false
Detach VIA serverinternal
all sessions will be placed in idle state
response:= rspOK
response:= rspSeNoSession
ServerResetRsp (sessionId,response)
-
ResetReq (userId)
ServerResetReq (sessionId,reason)
Detach VIA serverinternal
all sessions will be placed in idle state
set(NOW+Tid,Tmsg)
waitForCnf
waitForCnf
ServerResetCnf (sessionId,response)
valid
true
false
-
RESET(Tmsg)
response= rspOK
true
false
'' /* take other actions */
SSIdle
Tmsg
retrans>0
true
false
ServerResetReq (sessionId,reason)
SET(NOW+Tid,Tmsg)
retrans:=retrans+1
-

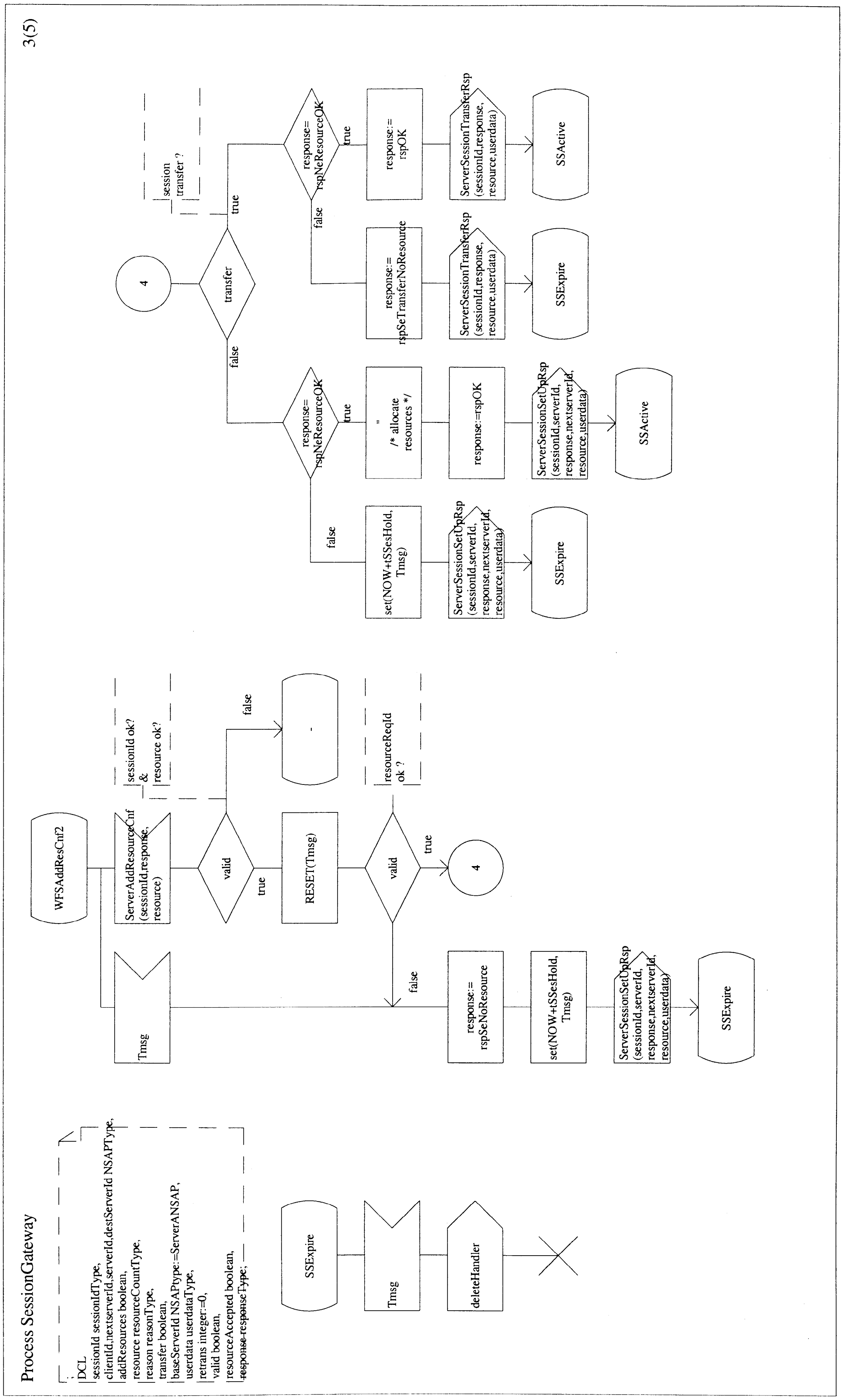

Process ServerSessionManager
3(3)
;
DCL
sessionId sessionIdType,
userId,clientId,nextserverId NSAPType,
serverId NSAPType:=CFSServerNSAP,
baseServerId,destServerId NSAPType,
resource resourceCountType,
reason reasonType,
reject boolean,
retrans integer:=0,
forward forwardCountType,
response responseType,
userdata userdataType,
TI integer,
valid boolean,
accepted boolean,
statusType integer,
resourceCounter integer:=0,
status statusCountType,
selectsessionId boolean;
SSIdle
ServerStatusInd (reason,statusType, status)
accepted
true
false
accept the request ?
response:=rspOK
ServerStatusRsp (response,statusType, status)
status!statusCount:=0
ServerStatusRsp (response,statusType, status)
-
StatusReq (userId)
reason:=rsnNormal
ServerStatusReq (reason,statusType, status)
set(now+Tid,Tmsg)
-
deleteHandler
-
ServerStatusCnf (response,statusType, status)
RESET(Tmsg)
response=rspOK
true
false
'' /* scenario completed */
'' /* error in response */
-
Tmsg
retrans>0
true
false
'' /* error in response */
-
retrans:=retrans+1
reason:=rsnNormal
ServerStatusReq (reason,statusType,status)
set(now+Tid,Tmsg)
-

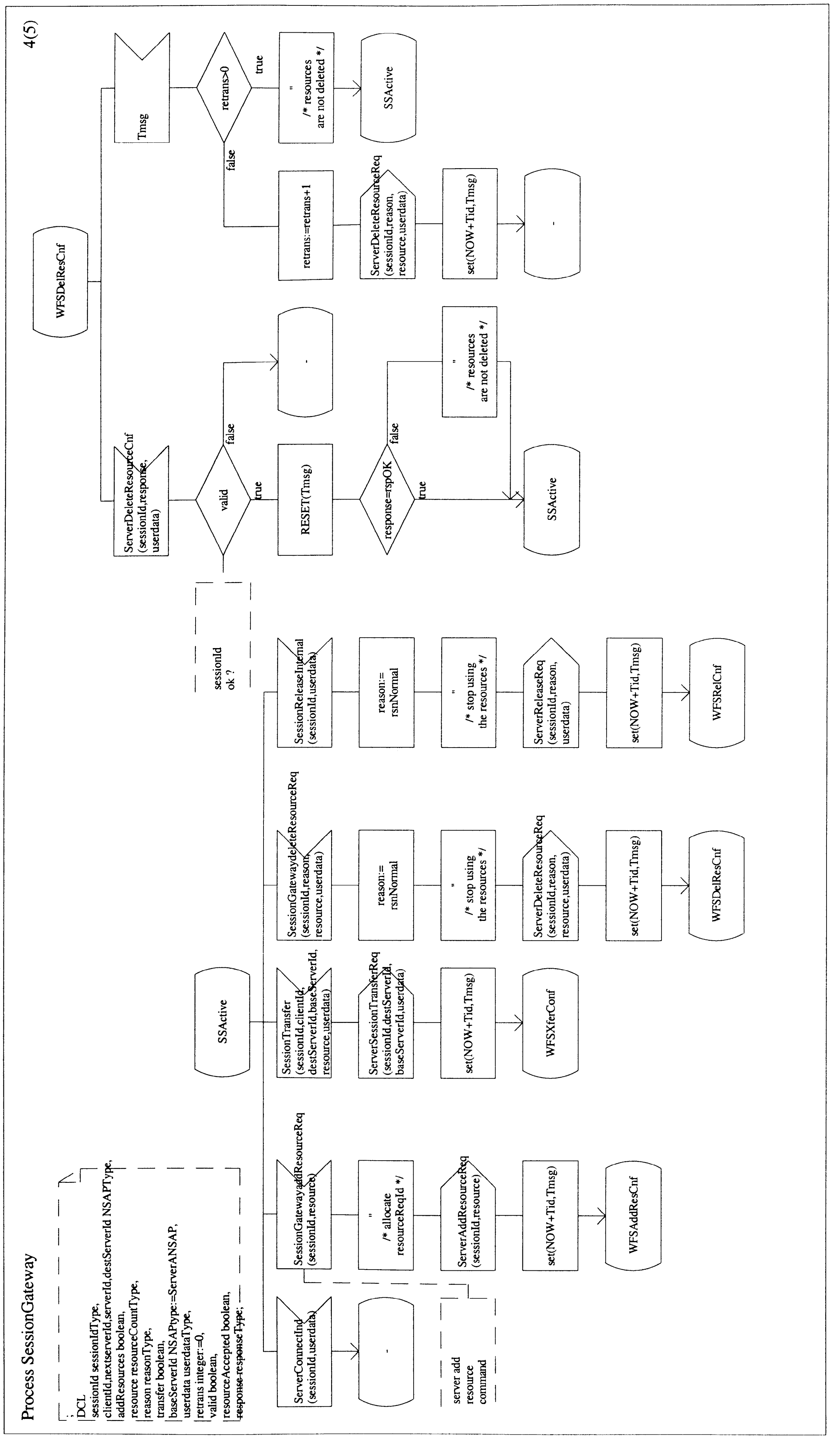


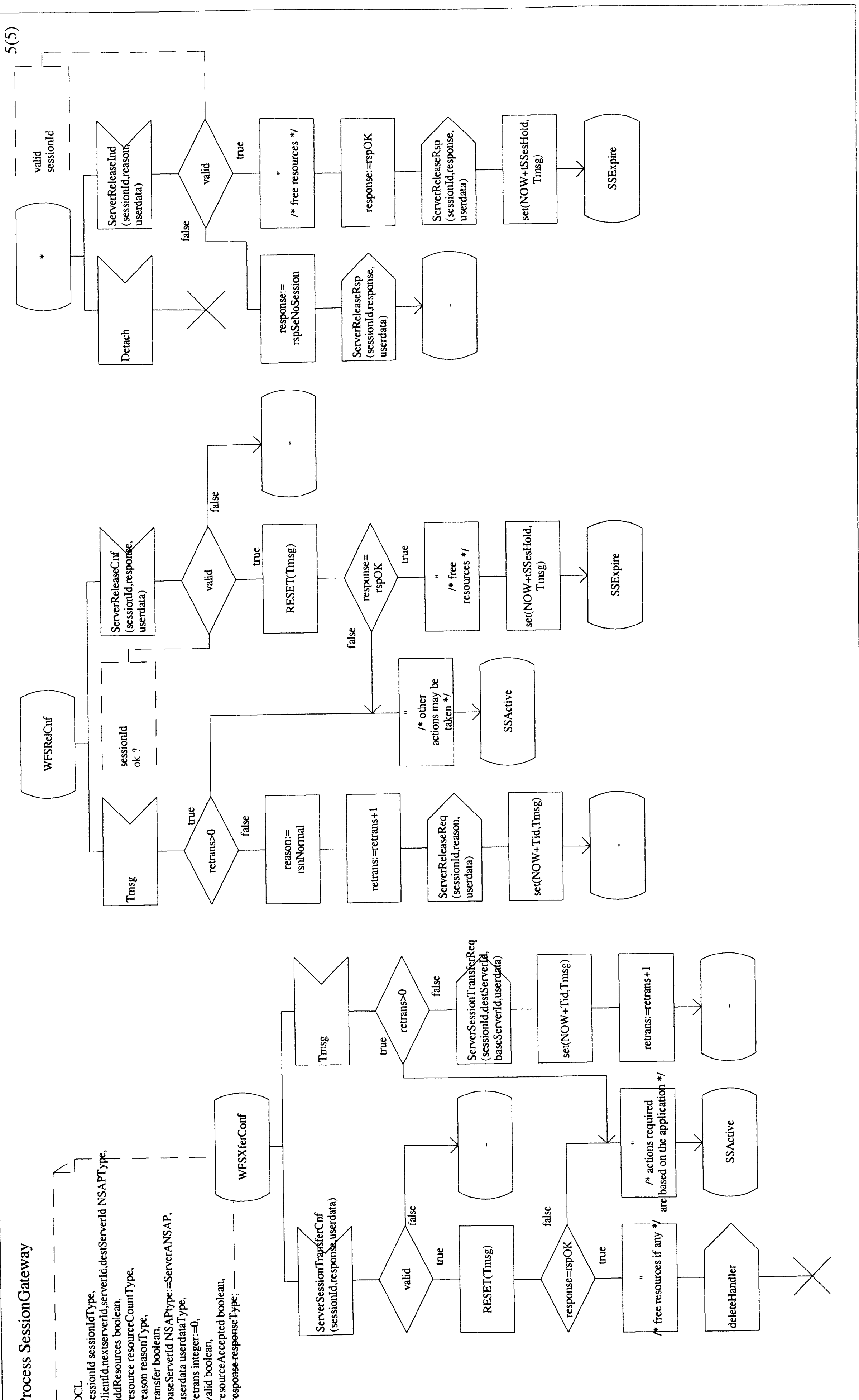
Procedure serverIdControl
1(1)
;FPAR
IN sessionId sessionIdType,
IN serverId NSAPType,
IN clientId NSAPType,
IN/OUT response responseType;
DCL valid boolean:=true;
valid sessionID
valid serverId?
default for the client
valid clientId?
support new session?
forward session?
/* this is a procedure
where the control of the network resorces and
the id's shall be done. If the network rejects
a new session or serverId or clientId are unknown
the response shall be set */
valid
serverId=ServerANSAP
clientId=clientNSAP
sessionId<maxsession
valid
true
false
response:=
rspForward
response:=
rspOK
response:=
rspSeNoCalls
response:=
rspSeInvalidClient
response:=
rspSeInvalidServer
response:=
rspSeNoSession


Process SessionGateway
1(5)
DCL
sessionId sessionIdType,
clientId,nextserverId,serverId,destServerId NSAPType,
addResources boolean,
resource resourceCountType,
reason reasonType,
transfer boolean,
baseServerId NSAPtype:=ServerANSAP,
userdata userdataType,
retrans integer:=0,
valid boolean,
resourceAccepted boolean,
~~response~~ response~~Type~~;
TIMER
Tmsg;
SSIdle
ServerInternal (sessionId, serverId,resource)
set(NOW+Tid,Tmsg)
WFSSesCnf
ServerSetUp (sessionId,userdata, resource)
/* cf session resource descriptor */
connect client to an existing cfs?
addResources
true
false
/* allocate resourceReqId */
set(NOW+Tid,Tmsg)
ServerAddResourceReq (sessionId,resource) VIA SresourceRoute
WFSAddResCnf2
response:=rspOK
ServerSessionSetUpRsp (sessionId,serverId, response,nextserverId, resource,userdata)
SSActive
SessionTransfer (sessionId,clientId,destServerId, baseServerId,resource,userdata)
addResources
true
false
/* allocate resourceReqId */
set(NOW+Tid,Tmsg)
ServerAddResourceReq (sessionId,resource)
WFSXfer
response:= rspOK
ServerSessionTransferRsp (sessionId,response, resource,userdata)
SSActive
WFSXfer
sessionId ok? & resource ok?
ServerAddResourceCnf (sessionId,response, resource)
valid
true
false
RESET(Tmsg)
response= rspNeResourceOK
true
false
response:= rspOK
ServerSessionTransferRsp (sessionId,response, resource,userdata)
SSActive
response:= rspSeTransferReject
set(NOW+tSSesHold, Tmsg)
ServerSessionTransferRsp (sessionId,response, resource,userdata)
SSExpire
Tmsg
retrans>0
false
retrans:= retrans+1
set(NOW+Tid,Tmsg)
ServerAddResourceReq (sessionId,resource)
-


Process SessionGateway
2(5)
DCL
sessionId sessionIdType,
clientId,nextserverId,serverId,destServerId NSAPType,
addResources boolean,
resource resourceCountType,
reason reasonType,
transfer boolean,
baseServerId NSAPtype:=ServerANSAP,
userdata userdataType,
retrans integer:=0,
valid boolean,
resourceAccepted boolean,
~~response responseType~~;
WFSSesCnf
ServerCFSCnf (sessionId,response,resource)
sessionId ok? & resource ok?
valid
true
false
-
RESET(Tmsg)
response=rspOK
true
false
resourceAccepted
true
false
SSActive
set(NOW+tSSesHold, Tmsg)
SSExpire
Tmsg
retrans>0
true
false
reason:= rsnSeTimerExpired
ServerReleaseReq (sessionId,reason, userdata)
set(NOW+tSSesHold, Tmsg)
SSExpire
ServerCFSReq (sessionId,serverId, resource)
retrans:= retrans+1
-
WFSAddResCnf
ServerAddResourceCnf (sessionId,response, resource)
sessionId ok? & resource ok?
valid
true
false
-
RESET(Tmsg)
response=rspOK
true
false
'' /* allocate resources */
SSActive
ServerDeleteResourceReq (sessionId,reason, resource,userdata)
set(NOW+Tid,Tmsg)
WFSDelResCnf
Tmsg
retrans>0
true
false
'' /* allocate resourceReqId */
ServerAddResourceReq (sessionId,resource) VIA SresourceRoute
set(NOW+Tid,Tmsg)
retrans:=retrans+1
-


Process SessionGateway
3(5)
DCL
sessionId sessionIdType,
clientId,nextserverId,serverId,destServerId NSAPType,
addResources boolean,
resource resourceCountType,
reason reasonType,
transfer boolean,
baseServerId NSAPtype:=ServerANSAP,
userdata userdataType,
retrans integer:=0,
valid boolean,
resourceAccepted boolean,
~~response responseType~~;
SSExpire
Tmsg
deleteHandler
WFSAddResCnf2
ServerAddResourceCnf (sessionId,response, resource)
Tmsg
valid
sessionId ok? & resource ok?
true
false
-
RESET(Tmsg)
valid
resourceReqId ok ?
true
false
4
response:= rspSeNoResource
set(NOW+tSSesHold, Tmsg)
ServerSessionSetUpRsp (sessionId,serverId, response,nextserverId, resource,userdata)
SSExpire
4
transfer
session transfer ?
true
false
response= rspNeResourceOK
true
false
response:= rspOK
ServerSessionTransferRsp (sessionId,response, resource,userdata)
SSActive
response:= rspSeTransferNoResource
ServerSessionTransferRsp (sessionId,response, resource,userdata)
SSExpire
response= rspNeResourceOK
true
false
/* allocate resources */
response:=rspOK
ServerSessionSetUpRsp (sessionId,serverId, response,nextserverId, resource,userdata)
SSActive
set(NOW+tSSesHold, Tmsg)
ServerSessionSetUpRsp (sessionId,serverId, response,nextserverId, resource,userdata)
SSExpire

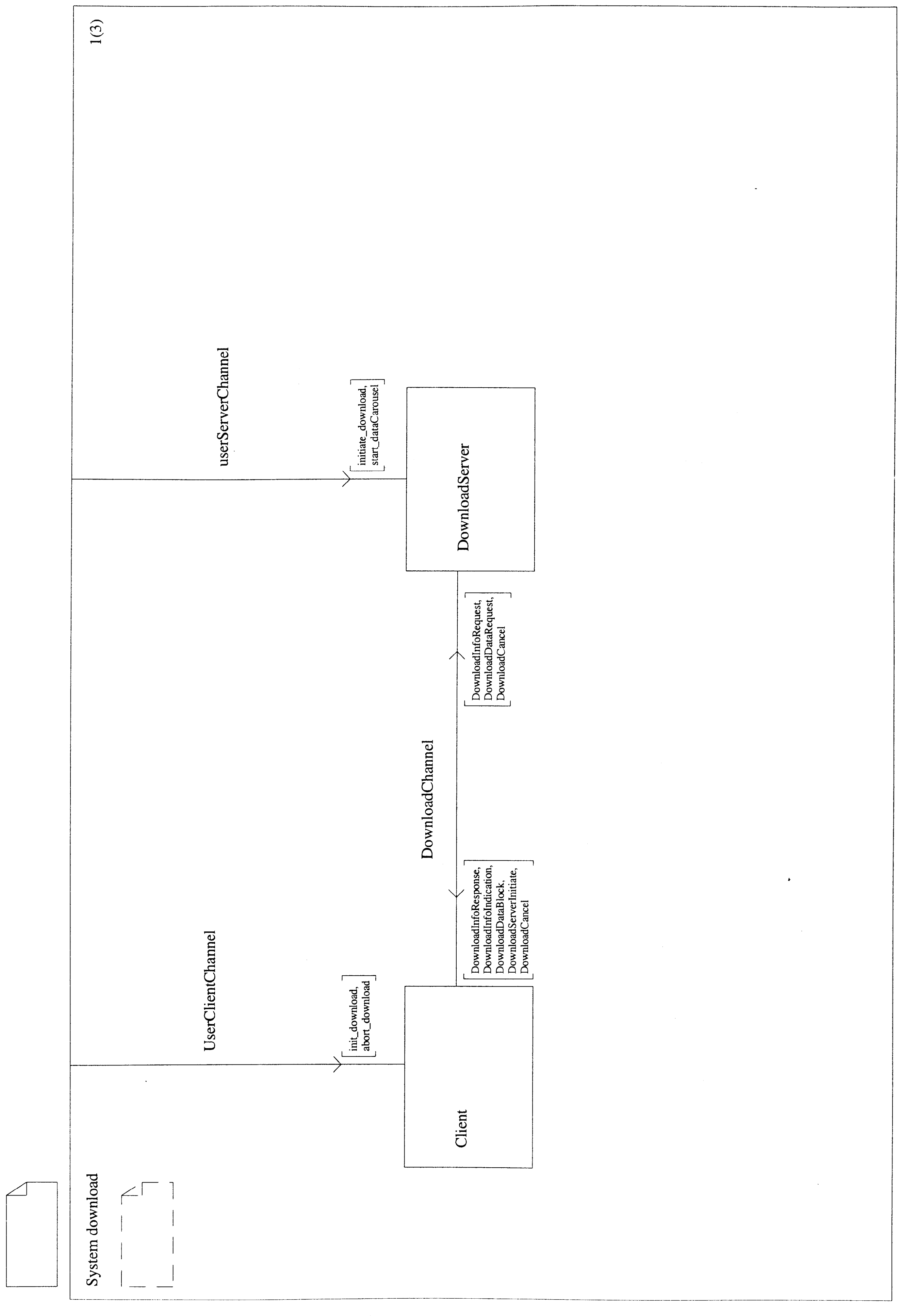

Process SessionGateway
4(5)
;
DCL
sessionId sessionIdType,
clientId,nextserverId,serverId,destServerId NSAPType,
addResources boolean,
resource resourceCountType,
reason reasonType,
transfer boolean,
baseServerId NSAPtype:=ServerANSAP,
userdata userdataType,
retrans integer:=0,
valid boolean,
resourceAccepted boolean,
~~response response Type~~;
SSActive
ServerConnectInd (sessionId,userdata)
-
server add resource command
SessionGatewayaddResourceReq (sessionId,resource)
/* allocate resourceReqId */
ServerAddResourceReq (sessionId,resource)
set(NOW+Tid,Tmsg)
WFSAddResCnf
SessionTransfer (sessionId,clientId, destServerId,baseServerId, resource,userdata)
ServerSessionTransferReq (sessionId,destServerId, baseServerId,userdata)
set(NOW+Tid,Tmsg)
WFSXferConf
SessionGatewaydeleteResourceReq (sessionId,reason, resource,userdata)
reason:= rsnNormal
/* stop using the resources */
ServerDeleteResourceReq (sessionId,reason, resource,userdata)
set(NOW+Tid,Tmsg)
WFSDelResCnf
SessionReleaseInternal (sessionId,userdata)
reason:= rsnNormal
/* stop using the resources */
ServerReleaseReq (sessionId,reason, userdata)
set(NOW+Tid,Tmsg)
WFSRelCnf
sessionId ok ?
WFSDelResCnf
ServerDeleteResourceCnf (sessionId,response, userdata)
valid
true
false
-
RESET(Tmsg)
response=rspOK
true
false
/* resources are not deleted */
SSActive
Tmsg
retrans>0
true
false
/* resources are not deleted */
SSActive
retrans:=retrans+1
ServerDeleteResourceReq (sessionId,reason, resource,userdata)
set(NOW+Tid,Tmsg)
-


Process SessionGateway
5(5)
;
DCL
sessionId sessionIdType,
clientId,nextserverId,serverId,destServerId NSAPType,
addResources boolean,
resource resourceCountType,
reason reasonType,
transfer boolean,
baseServerId NSAPtype:=ServerANSAP,
userdata userdataType,
retrans integer:=0,
valid boolean,
resourceAccepted boolean,
~~response responseType~~;
WFSXferConf
ServerSessionTransferCnf (sessionId,response,userdata)
valid
true
false
RESET(Tmsg)
response=rspOK
true
false
/* free resources if any */
deleteHandler
-
/* actions required are based on the application */
SSActive
Tmsg
retrans>0
true
false
ServerSessionTransferReq (sessionId,destServerId, baseServerId,userdata)
set(NOW+Tid,Tmsg)
retrans:=retrans+1
-
WFSRelCnf
sessionId ok ?
Tmsg
retrans>0
true
false
reason:= rsnNormal
retrans:=retrans+1
ServerReleaseReq (sessionId,reason, userdata)
set(NOW+Tid,Tmsg)
-
ServerReleaseCnf (sessionId,response, userdata)
valid
true
false
RESET(Tmsg)
response= rspOK
true
false
/* free resources */
set(NOW+tSSesHold, Tmsg)
SSExpire
/* other actions may be taken */
SSActive
-
*
valid sessionId
ServerReleaseInd (sessionId,reason, userdata)
valid
true
false
/* free resources */
response:=rspOK
ServerReleaseRsp (sessionId,response, userdata)
set(NOW+tSSesHold, Tmsg)
SSExpire
response:= rspSeNoSession
ServerReleaseRsp (sessionId,response, userdata)
-
Detach

A.3 U-N Download – Flow Controlled Scenario

The following pages contain the U-N Download (flow controlled scenario only) SDLs.

Figure A-2 U-N Download Flow Controlled Specification and Description Language


System download
1(3)
userServerChannel
initiate_download,
start_dataCarousel
DownloadServer
DownloadChannel
DownloadInfoRequest,
DownloadDataRequest,
DownloadCancel
DownloadInfoResponse,
DownloadInfoIndication,
DownloadDataBlock,
DownloadServerInitiate,
DownloadCancel
UserClientChannel
init_download,
abort_download
Client

System download

2(3)

```
SYNONYM maxNoOfClRecBytes Integer = 0; /* 2^32-1 */
SYNONYM maxNoOfprivateDataBytes Integer = 0; /* 2^16-1 */
SYNONYM downloadIdLimit Integer = 0; /* 2^32-1 */
SYNONYM blockSizeLimit Integer = 0; /* 2^16-1 */
SYNONYM windowSizeLimit Integer = 0; /* 2^16-1 */
SYNONYM ackPeriodLimit Integer = 0; /* 2^8-1 */
SYNONYM tCDownloadWindowLimit Duration = 0; /* 2^32-1 microseconds */
SYNONYM tCDownloadScenarioLimit Duration = 0; /* 2^32-1 microseconds */
SYNONYM maxNoOfModules Integer = 0; /* 2^16-1 */
SYNONYM ModuleIdLimit Integer = 0; /* 2^16-1 */
SYNONYM ModuleSizeLimit Integer = 0; /* 2^32-1 */
SYNONYM moduleInfoLengthLimit Integer = 0;
SYNONYM ModuleVersionLimit Integer = 0; /* 2^8-1 */
SYNONYM maxNoOfmoduleInfoBytes Integer = 0; /* 2^8-1 */
SYNONYM blockNumberLimit Integer = 0; /* 2^16-1 */
SYNONYM N Integer = 0;
SYNONYM tCDIRDuration Duration = 100; /* in milliseconds */
```

```
SYNTYPE bufferSizeType = Integer
  CONSTANTS 0:maxNoOfClRecBytes
ENDSYNTYPE;
SYNTYPE privateDataLenType = Integer
  CONSTANTS 0:maxNoOfprivateDataBytes
ENDSYNTYPE;
SYNTYPE downloadIdType = Integer
  CONSTANTS 0:downloadIdLimit
ENDSYNTYPE;
SYNTYPE blockSizeType = Integer
  CONSTANTS 0:blockSizeLimit
ENDSYNTYPE;
SYNTYPE tCDWDurationType = Duration
  CONSTANTS 0:tCDownloadWindowLimit
ENDSYNTYPE;
SYNTYPE tCDSDurationType = Duration
  CONSTANTS 0:tCDownloadScenarioLimit
ENDSYNTYPE;
```

```
SYNTYPE windowSizeType = Integer
  CONSTANTS 0:windowSizeLimit
ENDSYNTYPE;
SYNTYPE ackPeriodType = Integer
  CONSTANTS 0:ackPeriodLimit
ENDSYNTYPE;
/*
SYNTYPE tCDWDurationType = Integer
  CONSTANTS 0:tCDownloadWindowLimit
ENDSYNTYPE;
SYNTYPE tCDSDurationType = Integer
  CONSTANTS 0:tCDownloadScenarioLimit
ENDSYNTYPE;
*/
SYNTYPE numberOfModulesType = Integer
  CONSTANTS 0:maxNoOfModules
ENDSYNTYPE;
SYNTYPE ModuleIdType = Integer
  CONSTANTS 0:ModuleIdLimit
ENDSYNTYPE;
SYNTYPE ModuleSizeType = Integer
  CONSTANTS 0:ModuleSizeLimit
ENDSYNTYPE;
SYNTYPE moduleInfoLengthType = Integer
  CONSTANTS 0:moduleInfoLengthLimit
ENDSYNTYPE;
SYNTYPE ModuleVersionType = Integer
  CONSTANTS 0:ModuleVersionLimit
ENDSYNTYPE;
SYNTYPE NoOfmoduleInfoBytesType = Integer
  CONSTANTS 0:maxNoOfmoduleInfoBytes
ENDSYNTYPE;
SYNTYPE reservedType = Charstring
  CONSTANTS '0xFF'
ENDSYNTYPE;
SYNTYPE blockNumberType = Integer
  CONSTANTS 0:blockNumberLimit
ENDSYNTYPE;
SYNTYPE blockSizeIndexType = Integer
  CONSTANTS 0:N
ENDSYNTYPE;
```

System download

3(3)

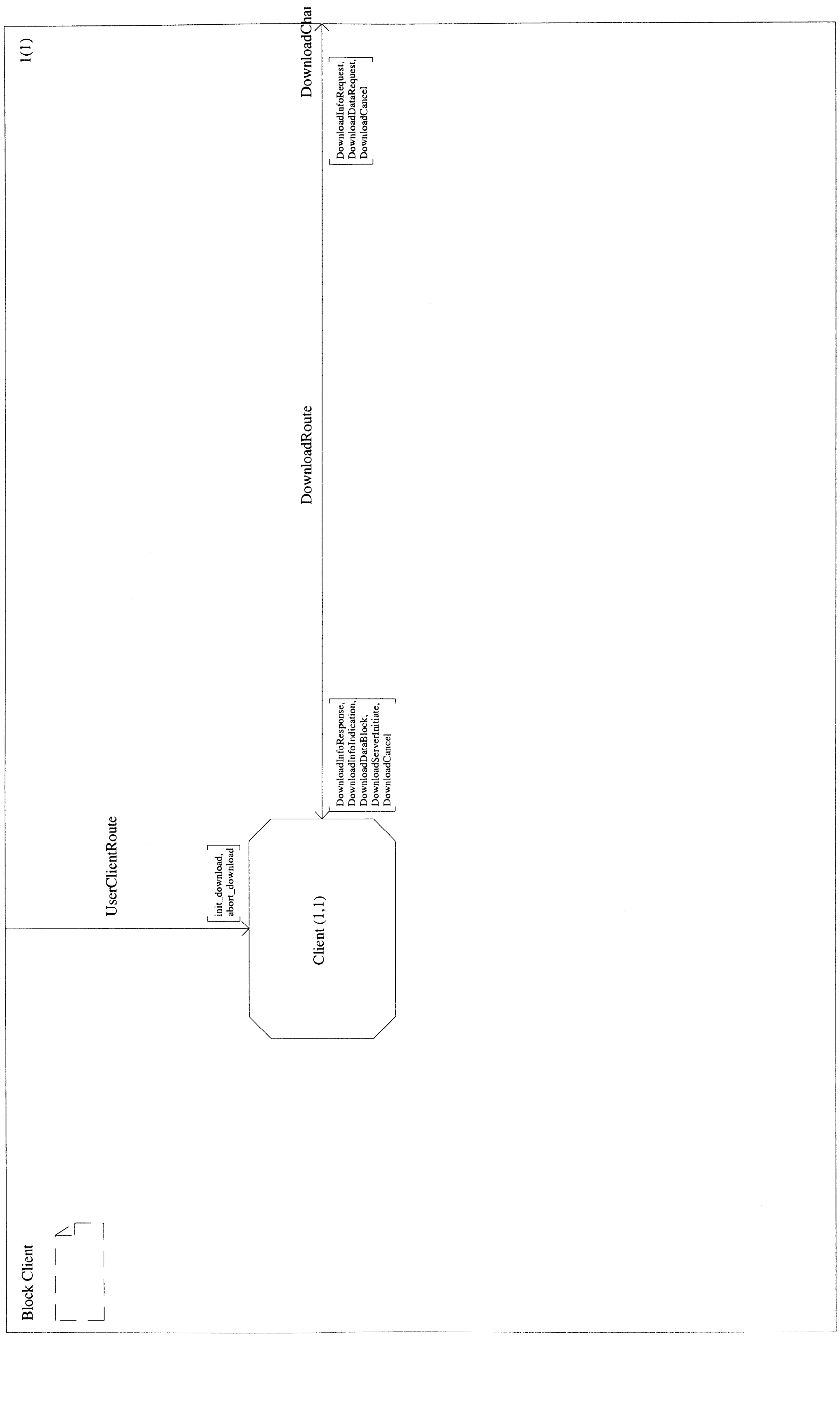
```
NEWTYPE CompatibilityDescriptorType
      /* Described in chapter 6 */
ENDNEWTYPE CompatibilityDescriptorType;
/*
NEWTYPE userCompatibilitiesType STRUCT
  CompatbilityDescriptor CompatbilityDescriptorType;
  InterfaceDescriptor InterfaceDescriptorType;
  SubDescriptor SubDescriptorType;
ENDNEWTYPE userCompatibilitiesType;
*/
NEWTYPE privateDataBytesType
  ARRAY(privateDataLenType,Character);
ENDNEWTYPE privateDataBytesType;
NEWTYPE moduleInfoBytesType
  ARRAY(NoOfmoduleInfoBytesType,Character);
ENDNEWTYPE moduleInfoBytesType;
NEWTYPE ModuleType STRUCT
  moduleId moduleIdType;
  moduleSize moduleSizeType;
  moduleVersion moduleVersionType;
  moduleInfoLength moduleInfoLengthType;
  moduleInfoBytes moduleInfoBytesType;
ENDNEWTYPE ModuleType;
NEWTYPE ModuleTableType
  ARRAY(numberOfModulesType, ModuleType);
ENDNEWTYPE ModuleTableType;
```

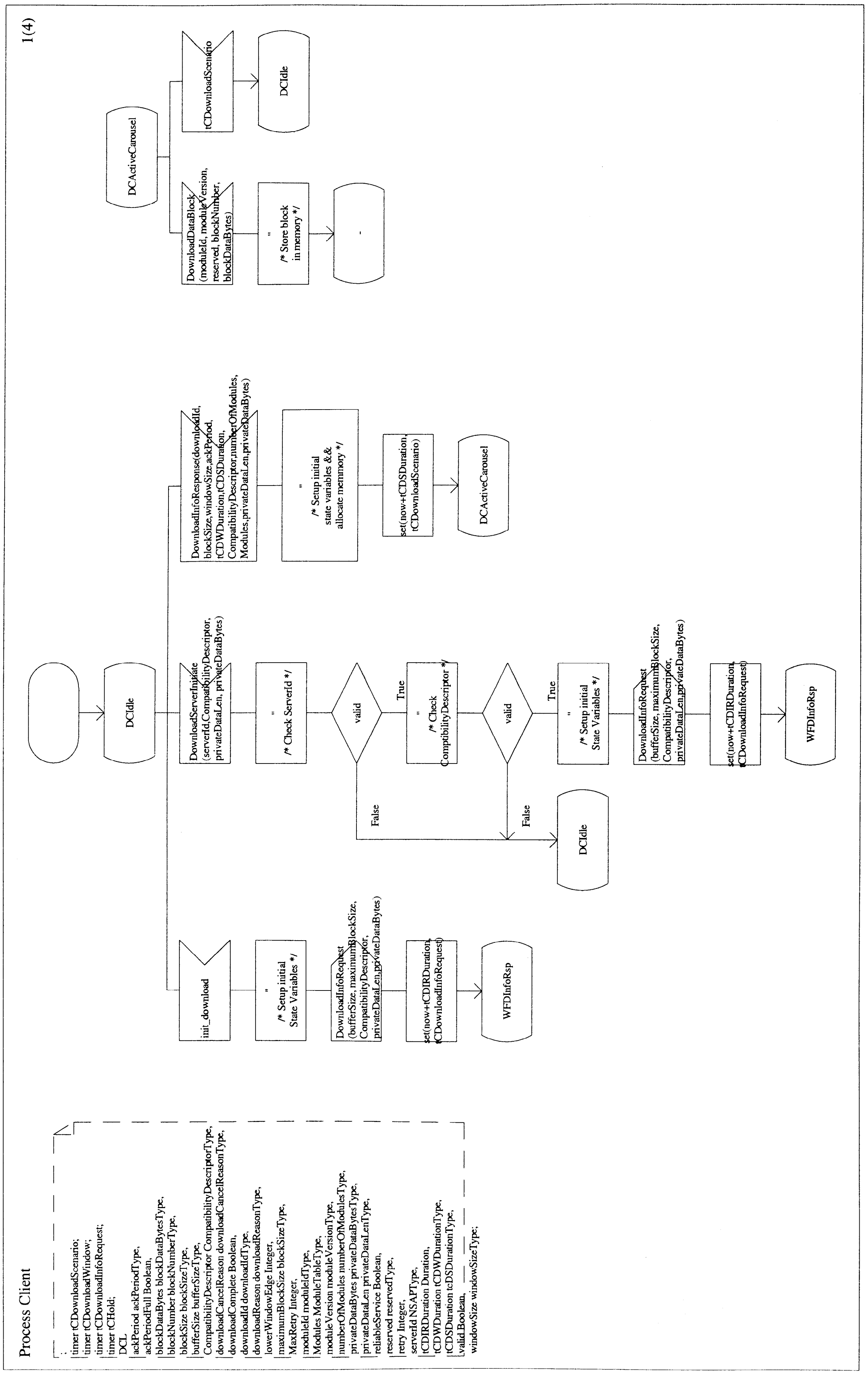
```
NEWTYPE NSAPType
  LITERALS
    ServerANSAP,
    ServerBNSAP,
    clientNSAP,
    CFSServerNSAP,
    VOID
ENDNEWTYPE NSAPType;
NEWTYPE blockDataBytesType
  ARRAY(blockSizeIndexType, Character);
ENDNEWTYPE blockDataBytesType;
NEWTYPE downloadReasonType
  LITERALS
    rsnStart,
    rsnAckCont,
    rsnNakRetransBlock,
    rsnNakRetransWindow,
    rsnEnd
ENDNEWTYPE downloadReasonType;
```

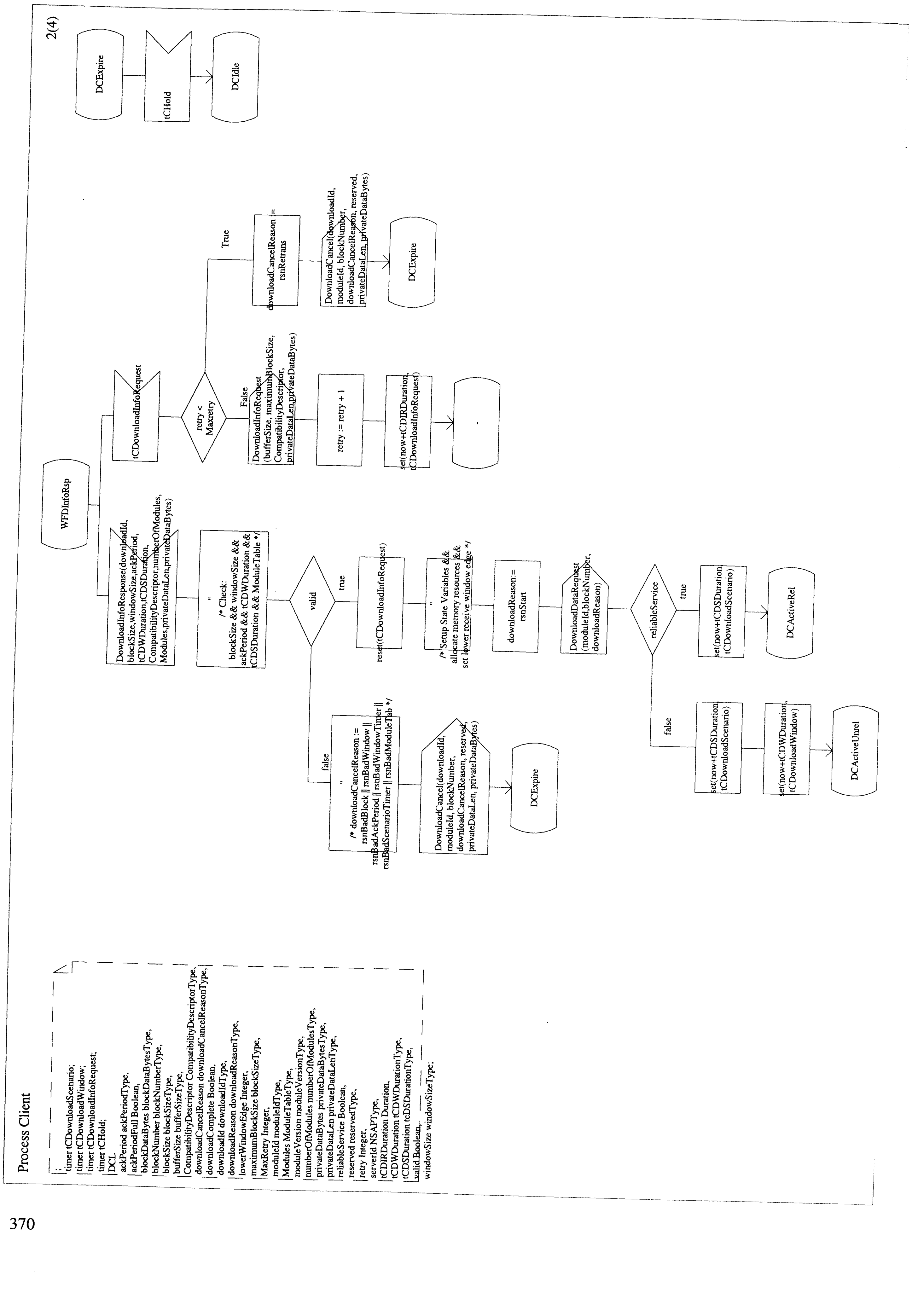
```
NEWTYPE downloadCancelReasonType
  LITERALS
    rsnScenarioTimeout,
    rsnInsuffMem,
    rsnAuthDenied,
    rsnFatal,
    rsnInfoRequestError,
    rsnCompatError,
    rsnUnreliableNetwork,
    rsnInvalidData,
    rsnInvalidBlock,
    rsnInvalidVersion,
    rsnAbort,
    rsnRetrans,
    rsnBadBlockSize,
    rsnBadWindow,
    rsnBadAckPeriod,
    rsnBadWindowTimer,
    rsnBadScenarioTimer,
    rsnBadCapabilities,
    rsnBadModuleTable
ENDNEWTYPE downloadCancelReasonType;
```

```
SIGNAL
init_download,
abort_download,
initiate_download,
start_dataCarousel;
```

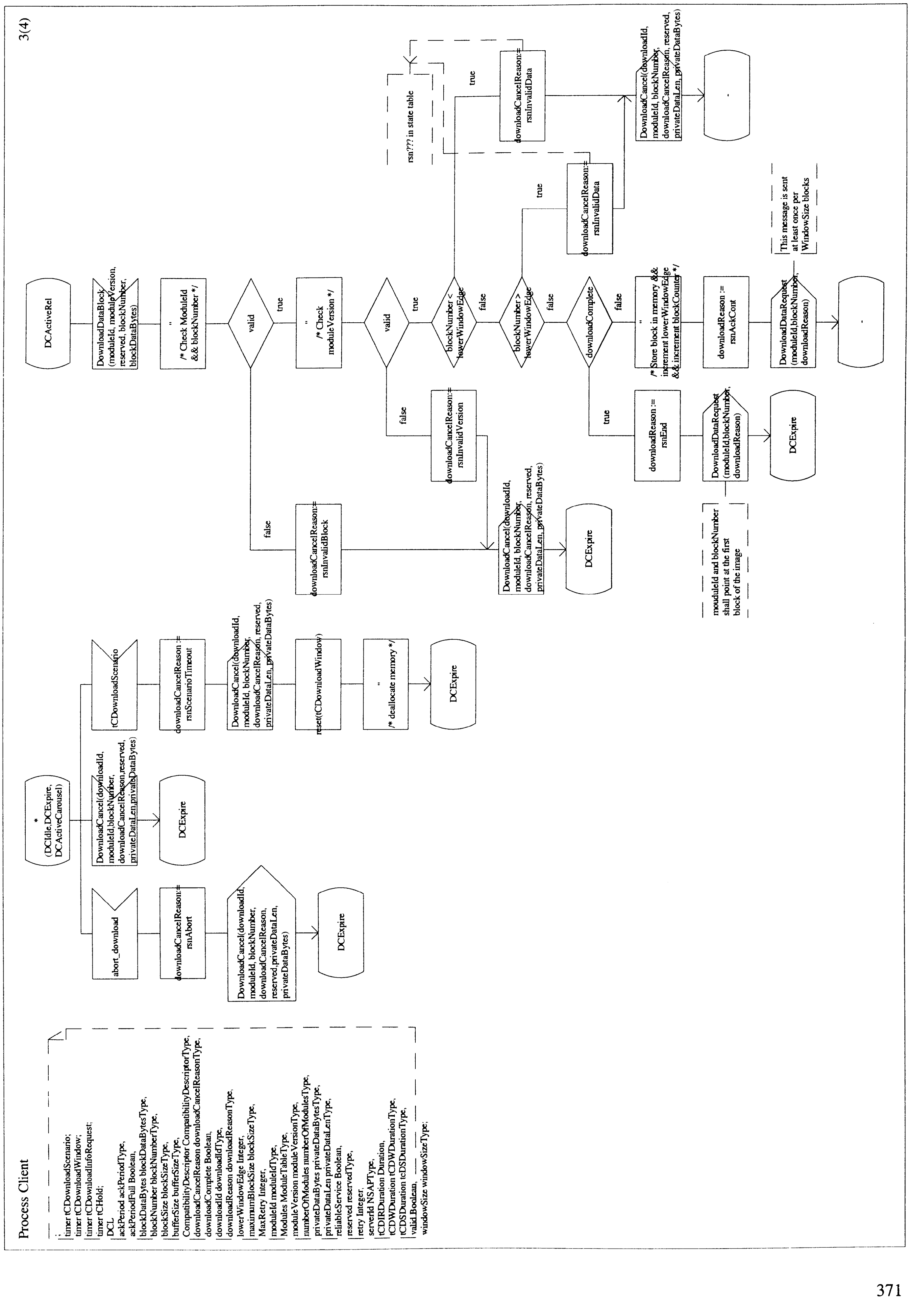
```
SIGNAL
DownloadInfoRequest(bufferSizeType, blockSizeType, CompatibilityDescriptorType, privateDataLenType, privateDataBytesType),
DownloadInfoResponse(downloadIdType, blockSizeType, windowSizeType, ackPeriodType, tCDWDurationType, tCDSDurationType,
          CompatibilityDescriptorType, numberOfModulesType, ModuleTableType, privateDataLenType, privateDataBytesType),
DownloadInfoIndication(downloadIdType, blockSizeType, windowSizeType, ackPeriodType, tCDWDurationType, tCDSDurationType,
          CompatibilityDescriptorType, numberOfModulesType, ModuleTableType, privateDataLenType, privateDataBytesType),
DownloadDataBlock(moduleIdType, moduleVersionType, reservedType, blockNumberType, blockDataBytesType),
DownloadDataRequest(moduleIdType, blockNumberType, downloadReasonType),
DownloadCancel(downloadIdType, moduleIdType, blockNumberType, downloadCancelReasonType, reservedType,  privateDataLenType, privateDataBytesType),
DownloadServerInitiate(NSAPType, CompatibilityDescriptorType, privateDataLenType, privateDataBytesType);
```

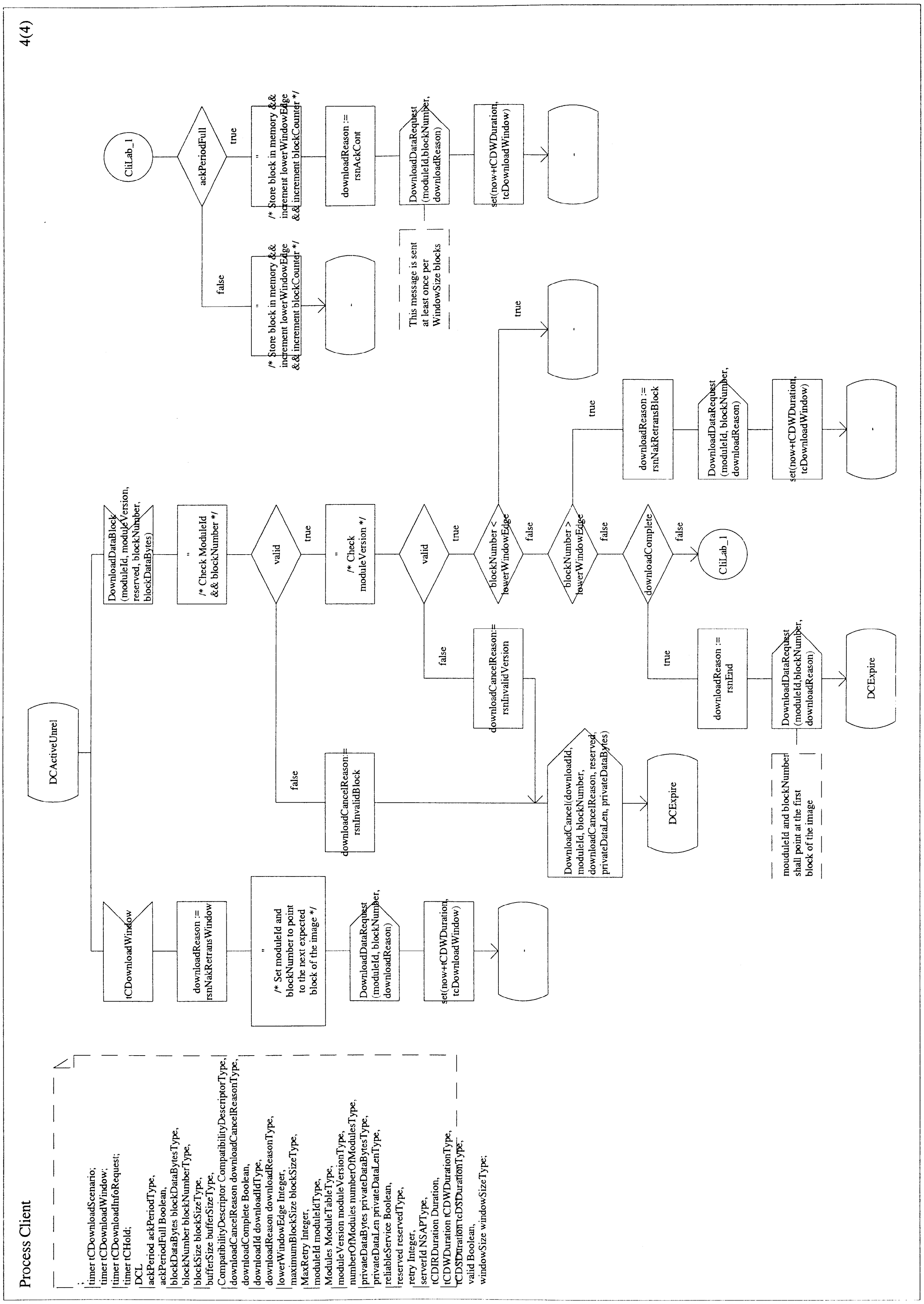

Block Client
1(1)
UserClientRoute
init_download,
abort_download
Client (1,1)
DownloadRoute
DownloadCha
DownloadInfoRequest,
DownloadDataRequest,
DownloadCancel
DownloadInfoResponse,
DownloadInfoIndication,
DownloadDataBlock,
DownloadServerInitiate,
DownloadCancel

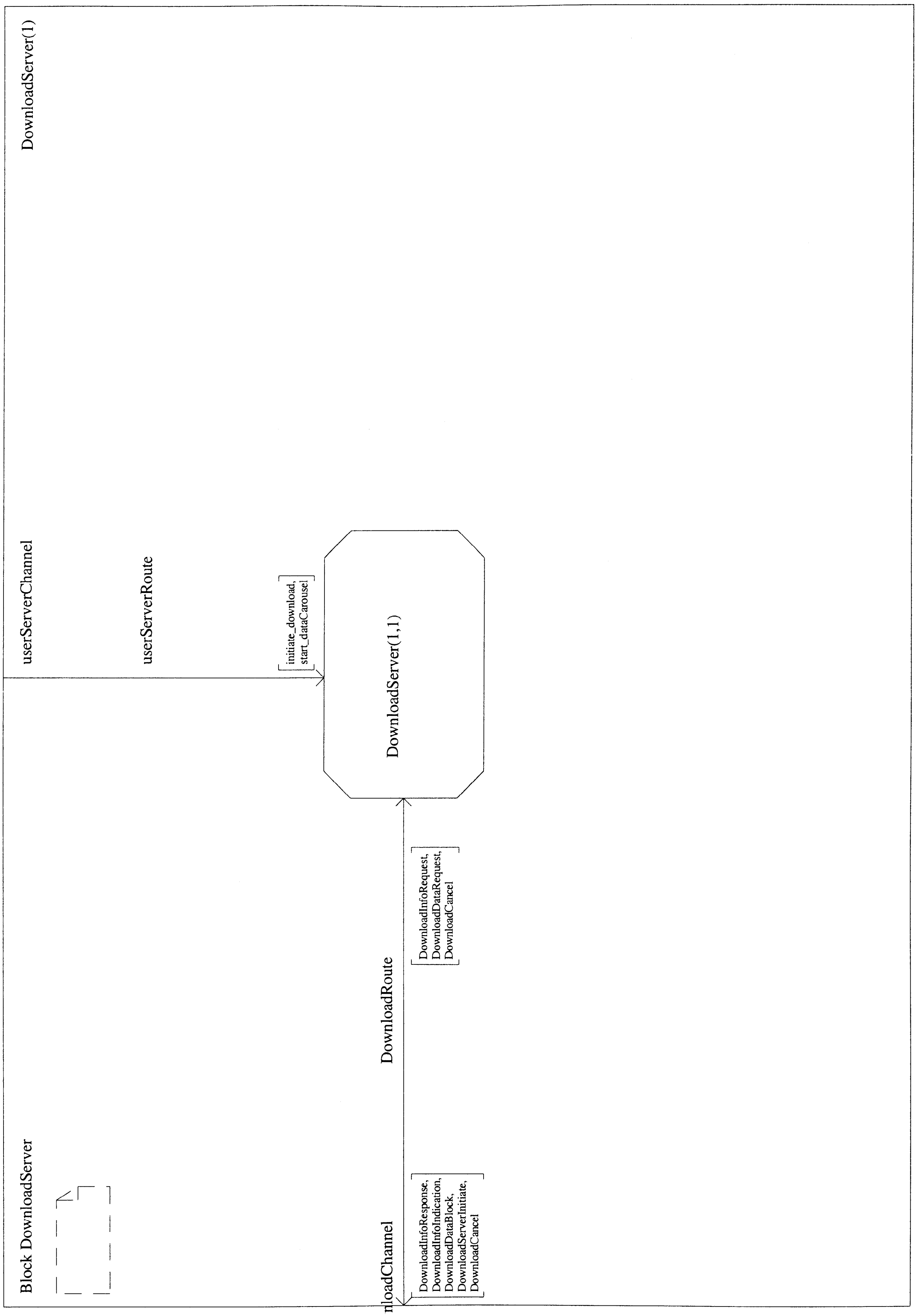

Process Client
1(4)
timer tCDownloadScenario;
timer tCDownloadWindow;
timer tCDownloadInfoRequest;
timer tCHold;
DCL
ackPeriod ackPeriodType,
ackPeriodFull Boolean,
blockDataBytes blockDataBytesType,
blockNumber blockNumberType,
blockSize blockSizeType,
bufferSize bufferSizeType,
CompatibilityDescriptor CompatibilityDescriptorType,
downloadCancelReason downloadCancelReasonType,
downloadComplete Boolean,
downloadId downloadIdType,
downloadReason downloadReasonType,
lowerWindowEdge Integer,
maximumBlockSize blockSizeType,
MaxRetry Integer,
moduleId moduleIdType,
Modules ModuleTableType,
moduleVersion moduleVersionType,
numberOfModules numberOfModulesType,
privateDataBytes privateDataBytesType,
privateDataLen privateDataLenType,
reliableService Boolean,
reserved reservedType,
retry Integer,
serverId NSAPType,
tCDIRDuration Duration,
tCDWDuration tCDWDurationType,
tCDSDuration tcDSDurationType,
valid Boolean,
windowSize windowSizeType;
DCIdle
init_download
/* Setup initial State Variables */
DownloadInfoRequest (bufferSize, maximumBlockSize, CompatibilityDescriptor, privateDataLen,privateDataBytes)
set(now+tCDIRDuration, tCDownloadInfoRequest)
WFDInfoRsp
DownloadServerInitiate (serverId,CompatibilityDescriptor, privateDataLen, privateDataBytes)
/* Check ServerId */
valid
True
False
/* Check ComptibilityDescriptor */
valid
True
False
DCIdle
/* Setup initial State Variables */
DownloadInfoRequest (bufferSize, maximumBlockSize, CompatibilityDescriptor, privateDataLen,privateDataBytes)
set(now+tCDIRDuration, tCDownloadInfoRequest)
WFDInfoRsp
DownloadInfoResponse(downloadId, blockSize,windowSize,ackPeriod, tCDWDuration,tCDSDuration, CompatibilityDescriptor,numberOfModules, Modules,privateDataLen,privateDataBytes)
/* Setup initial state variables && allocate memory */
set(now+tCDSDuration, tCDownloadScenario)
DCActiveCarousel
DCActiveCarousel
DownloadDataBlock (moduleId, moduleVersion, reserved, blockNumber, blockDataBytes)
/* Store block in memory */
-
tCDownloadScenario
DCIdle


2(4)
DCExpire
tCHold
DCIdle
WFDInfoRsp
tCDownloadInfoRequest
retry < Maxretry
True
downloadCancelReason := rsnRetrans
DownloadCancel(downloadId, moduleId, blockNumber, downloadCancelReason, reserved, privateDataLen, privateDataBytes)
DCExpire
False
DownloadInfoRequest (bufferSize, maximumBlockSize, CompatibilityDescriptor, privateDataLen,privateDataBytes)
retry := retry + 1
set(now+tCDIRDuration, tCDownloadInfoRequest)
-
DownloadInfoResponse(downloadId, blockSize,windowSize,ackPeriod, tCDWDuration,tCDSDuration, CompatibilityDescriptor,numberOfModules, Modules,privateDataLen,privateDataBytes)
/* Check: blockSize && windowSize && ackPeriod && tCDWDuration && tCDSDuration && ModuleTable */
valid
true
reset(tCDownloadInfoRequest)
/* Setup State Variables && allocate memory resources && set lower receive window edge */
downloadReason:= rsnStart
DownloadDataRequest (moduleId,blockNumber, downloadReason)
reliableService
true
set(now+tCDSDuration, tCDownloadScenario)
DCActiveRel
false
set(now+tCDSDuration, tCDownloadScenario)
set(now+tCDWDuration, tCDownloadWindow)
DCActiveUnrel
false
/* downloadCancelReason := rsnBadBlock || rsnBadWindow || rsnBadAckPeriod || rsnBadWindowTimer || rsnBadScenarioTimer || rsnBadModuleTab */
DownloadCancel(downloadId, moduleId, blockNumber, downloadCancelReason, reserved, privateDataLen, privateDataBytes)
DCExpire
Process Client

```
;
timer tCDownloadScenario;
timer tCDownloadWindow;
timer tCDownloadInfoRequest;
timer tCHold;
DCL
ackPeriod ackPeriodType,
ackPeriodFull Boolean,
blockDataBytes blockDataBytesType,
blockNumber blockNumberType,
blockSize blockSizeType,
bufferSize bufferSizeType,
CompatibilityDescriptor CompatibilityDescriptorType,
downloadCancelReason downloadCancelReasonType,
downloadComplete Boolean,
downloadId downloadIdType,
downloadReason downloadReasonType,
lowerWindowEdge Integer,
maximumBlockSize blockSizeType,
MaxRetry Integer,
moduleId moduleIdType,
Modules ModuleTableType,
moduleVersion moduleVersionType,
numberOfModules numberOfModulesType,
privateDataBytes privateDataBytesType,
privateDataLen privateDataLenType,
reliableService Boolean,
reserved reservedType,
retry Integer,
serverId NSAPType,
tCDIRDuration Duration,
tCDWDuration tCDWDurationType,
tCDSDuration tCDSDurationType,
valid Boolean,
windowSize windowSizeType;
```


Process Client
3(4)
timer tCDownloadScenario;
timer tCDownloadWindow;
timer tCDownloadInfoRequest;
timer tCHold;
DCL
ackPeriod ackPeriodType,
ackPeriodFull Boolean,
blockDataBytes blockDataBytesType,
blockNumber blockNumberType,
blockSize blockSizeType,
bufferSize bufferSizeType,
CompatibilityDescriptor CompatibilityDescriptorType,
downloadCancelReason downloadCancelReasonType,
downloadComplete Boolean,
downloadId downloadIdType,
downloadReason downloadReasonType,
lowerWindowEdge Integer,
maximumBlockSize blockSizeType,
MaxRetry Integer,
moduleId moduleIdType,
Modules ModuleTableType,
moduleVersion moduleVersionType,
numberOfModules numberOfModulesType,
privateDataBytes privateDataBytesType,
privateDataLen privateDataLenType,
reliableService Boolean,
reserved reservedType,
retry Integer,
serverId NSAPType,
tCDIRDuration Duration,
tCDWDuration tCDWDurationType,
tCDSDuration tCDSDurationType,
valid Boolean,
windowSize windowSizeType;
*
(DCIdle,DCExpire,
DCActiveCarousel)
abort_download
downloadCancelReason:=
rsnAbort
DownloadCancel(downloadId,
moduleId, blockNumber,
downloadCancelReason,
reserved,privateDataLen,
privateDataBytes)
DCExpire
DownloadCancel(downloadId,
moduleId,blockNumber,
downloadCancelReason,reserved,
privateDataLen,privateDataBytes)
DCExpire
tCDownloadScenario
downloadCancelReason :=
rsnScenarioTimeout
DownloadCancel(downloadId,
moduleId, blockNumber,
downloadCancelReason, reserved,
privateDataLen, privateDataBytes)
reset(tCDownloadWindow)
/* deallocate memory */
DCExpire
DCActiveRel
DownloadDataBlock
(moduleId, moduleVersion,
reserved, blockNumber,
blockDataBytes)
/* Check ModuleId
&& blockNumber */
valid
true
false
downloadCancelReason:=
rsnInvalidBlock
/* Check
moduleVersion */
valid
true
false
downloadCancelReason:=
rsnInvalidVersion
DownloadCancel(downloadId,
moduleId, blockNumber,
downloadCancelReason, reserved,
privateDataLen, privateDataBytes)
DCExpire
blockNumber <
lowerWindowEdge
true
false
downloadCancelReason:=
rsnInvalidData
rsn??? in state table
blockNumber >
lowerWindowEdge
true
false
downloadCancelReason:=
rsnInvalidData
DownloadCancel(downloadId,
moduleId, blockNumber,
downloadCancelReason, reserved,
privateDataLen, privateDataBytes)
-
downloadComplete
true
false
downloadReason :=
rsnEnd
DownloadDataRequest
(moduleId,blockNumber,
downloadReason)
DCExpire
moduleId and blockNumber
shall point at the first
block of the image
/* Store block in memory &&
increment lowerWindowEdge
&& increment blockCounter */
downloadReason :=
rsnAckCont
DownloadDataRequest
(moduleId,blockNumber,
downloadReason)
This message is sent
at least once per
WindowSize blocks
-

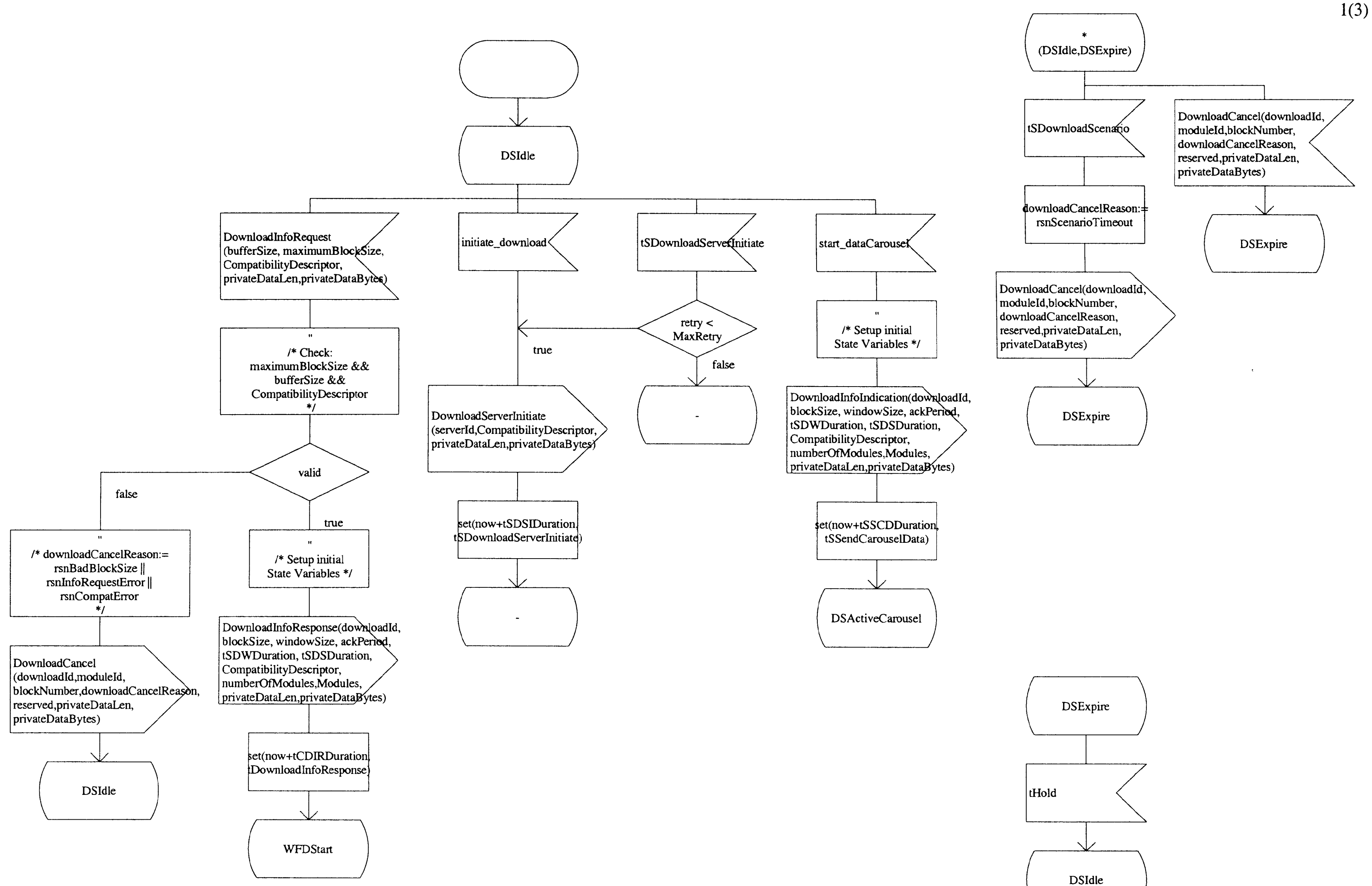

Process Client
4(4)
timer tCDownloadScenario;
timer tCDownloadWindow;
timer tCDownloadInfoRequest;
timer tCHold;
DCL
ackPeriod ackPeriodType,
ackPeriodFull Boolean,
blockDataBytes blockDataBytesType,
blockNumber blockNumberType,
blockSize blockSizeType,
bufferSize bufferSizeType,
CompatibilityDescriptor CompatibilityDescriptorType,
downloadCancelReason downloadCancelReasonType,
downloadComplete Boolean,
downloadId downloadIdType,
downloadReason downloadReasonType,
lowerWindowEdge Integer,
maximumBlockSize blockSizeType,
MaxRetry Integer,
moduleId moduleIdType,
Modules ModuleTableType,
moduleVersion moduleVersionType,
numberOfModules numberOfModulesType,
privateDataBytes privateDataBytesType,
privateDataLen privateDataLenType,
reliableService Boolean,
reserved reservedType,
retry Integer,
serverId NSAPType,
tCDIRDuration Duration,
tCDWDuration tCDWDurationType,
tCDSDuration tcDSDurationType,
valid Boolean,
windowSize windowSizeType;
DCActiveUnrel
tCDownloadWindow
downloadReason := rsnNakRetransWindow
/* Set moduleId and blockNumber to point to the next expected block of the image */
DownloadDataRequest (moduleId, blockNumber, downloadReason)
set(now+tCDWDuration, tcDownloadWindow)
-
DownloadDataBlock (moduleId, moduleVersion, reserved, blockNumber, blockDataBytes)
/* Check ModuleId && blockNumber */
valid
true
false
downloadCancelReason:= rsnInvalidBlock
/* Check moduleVersion */
valid
true
false
downloadCancelReason:= rsnInvalidVersion
DownloadCancel(downloadId, moduleId, blockNumber, downloadCancelReason, reserved, privateDataLen, privateDataBytes)
DCExpire
blockNumber < lowerWindowEdge
true
false
-
blockNumber > lowerWindowEdge
true
false
downloadReason := rsnNakRetransBlock
DownloadDataRequest (moduleId, blockNumber, downloadReason)
set(now+tCDWDuration, tcDownloadWindow)
-
downloadComplete
true
false
downloadReason := rsnEnd
DownloadDataRequest (moduleId,blockNumber, downloadReason)
DCExpire
moduleId and blockNumber shall point at the first block of the image
CliLab_1
CliLab_1
ackPeriodFull
true
false
/* Store block in memory && increment lowerWindowEdge && increment blockCounter */
-
/* Store block in memory && increment lowerWindowEdge && increment blockCounter */
downloadReason := rsnAckCont
DownloadDataRequest (moduleId,blockNumber, downloadReason)
This message is sent at least once per WindowSize blocks
set(now+tCDWDuration, tcDownloadWindow)
-


Block DownloadServer
DownloadServer(1)
userServerChannel
userServerRoute
initiate_download,
start_dataCarousel
DownloadServer(1,1)
DownloadRoute
DownloadInfoRequest,
DownloadDataRequest,
DownloadCancel
nloadChannel
DownloadInfoResponse,
DownloadInfoIndication,
DownloadDataBlock,
DownloadServerInitiate,
DownloadCancel

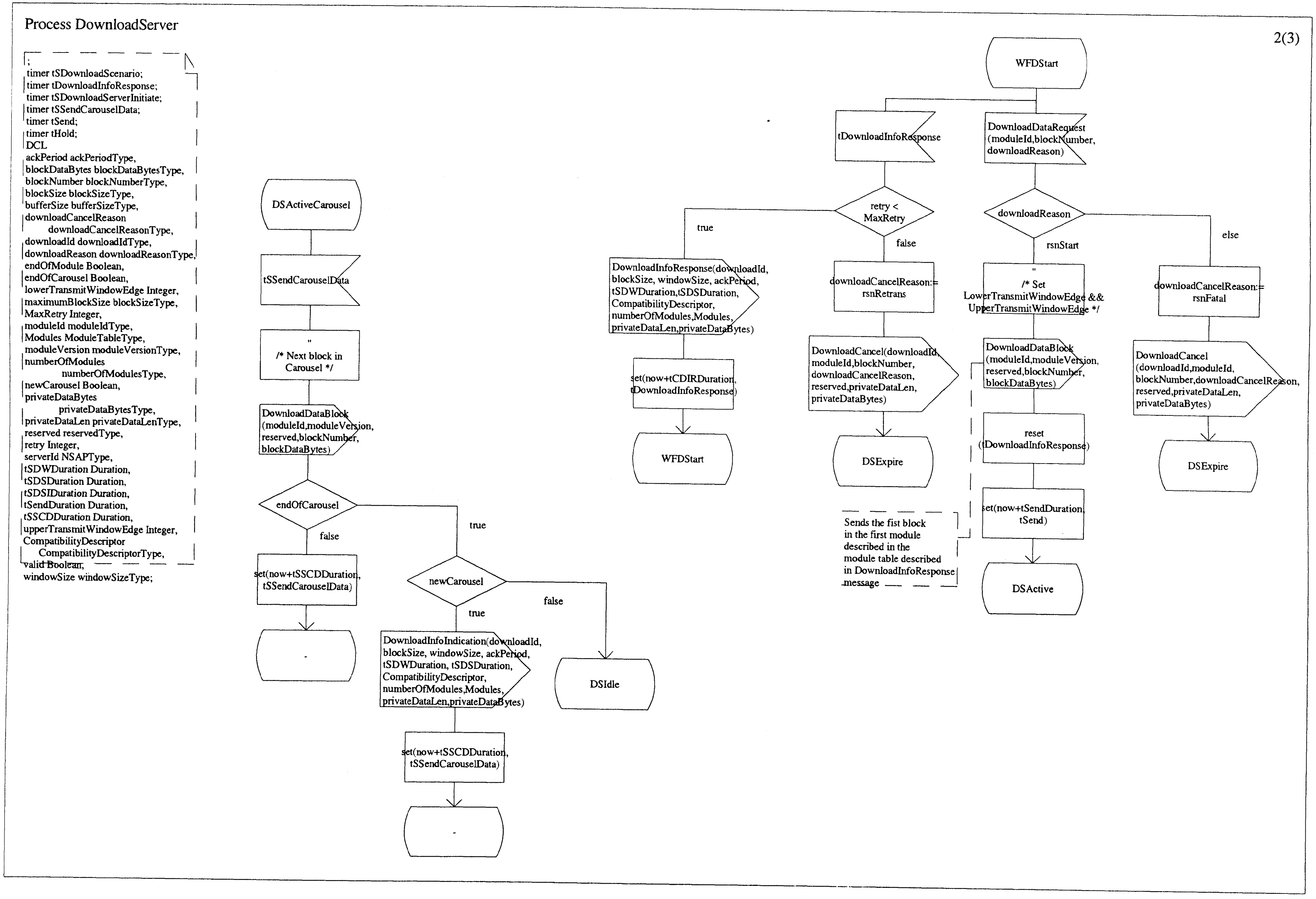
Process DownloadServer

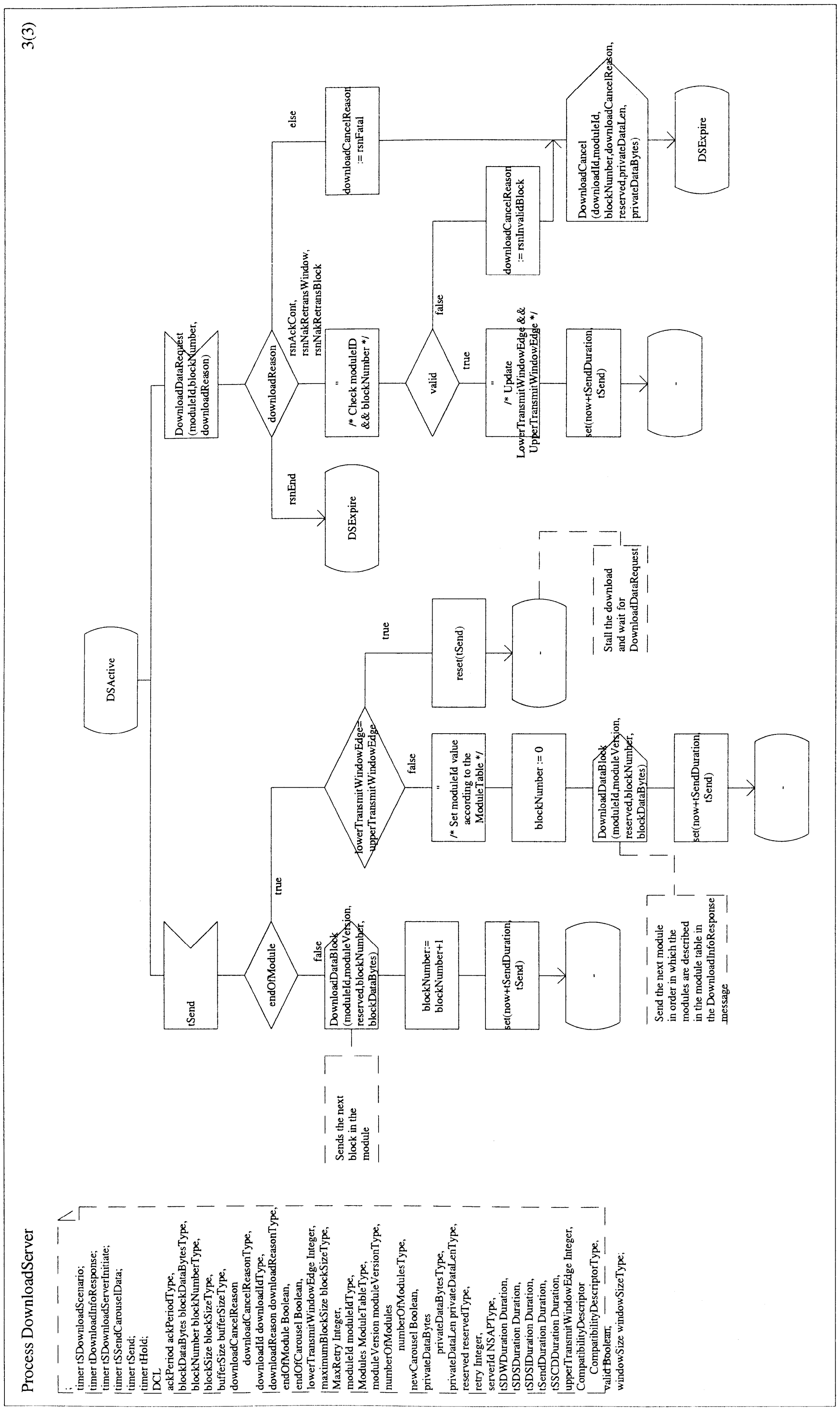
1(3)

```
;
timer tSDownloadScenario;
timer tDownloadInfoResponse;
timer tSDownloadServerInitiate;
timer tSSendCarouselData;
timer tSend;
timer tHold;
DCL
ackPeriod ackPeriodType,
blockDataBytes blockDataBytesType,
blockNumber blockNumberType,
blockSize blockSizeType,
bufferSize bufferSizeType,
downloadCancelReason
        downloadCancelReasonType,
downloadId downloadIdType,
downloadReason downloadReasonType,
endOfModule Boolean,
endOfCarousel Boolean,
lowerTransmitWindowEdge Integer,
maximumBlockSize blockSizeType,
MaxRetry Integer,
moduleId moduleIdType,
Modules ModuleTableType,
moduleVersion moduleVersionType,
numberOfModules
            numberOfModulesType,
newCarousel Boolean,
privateDataBytes
            privateDataBytesType,
privateDataLen privateDataLenType,
reserved reservedType,
retry Integer,
serverId NSAPType,
tSDWDuration Duration,
tSDSDuration Duration,
tSDSIDuration Duration,
tSendDuration Duration,
tSSCDDuration Duration,
upperTransmitWindowEdge Integer,
CompatibilityDescriptor
      CompatibilityDescriptorType,
valid Boolean,
windowSize windowSizeType;
```



DSIdle
DownloadInfoRequest (bufferSize, maximumBlockSize, CompatibilityDescriptor, privateDataLen,privateDataBytes)
" /* Check: maximumBlockSize && bufferSize && CompatibilityDescriptor */
valid
false
true
" /* downloadCancelReason:= rsnBadBlockSize || rsnInfoRequestError || rsnCompatError */
DownloadCancel (downloadId,moduleId, blockNumber,downloadCancelReason, reserved,privateDataLen, privateDataBytes)
DSIdle
" /* Setup initial State Variables */
DownloadInfoResponse(downloadId, blockSize, windowSize, ackPeriod, tSDWDuration, tSDSDuration, CompatibilityDescriptor, numberOfModules,Modules, privateDataLen,privateDataBytes)
set(now+tCDIRDuration, tDownloadInfoResponse)
WFDStart
initiate_download
true
DownloadServerInitiate (serverId,CompatibilityDescriptor, privateDataLen,privateDataBytes)
set(now+tSDSIDuration, tSDownloadServerInitiate)
-
tSDownloadServerInitiate
retry < MaxRetry
false
-
start_dataCarousel
" /* Setup initial State Variables */
DownloadInfoIndication(downloadId, blockSize, windowSize, ackPeriod, tSDWDuration, tSDSDuration, CompatibilityDescriptor, numberOfModules,Modules, privateDataLen,privateDataBytes)
set(now+tSSCDDuration, tSSendCarouselData)
DSActiveCarousel
* (DSIdle,DSExpire)
tSDownloadScenario
downloadCancelReason:= rsnScenarioTimeout
DownloadCancel(downloadId, moduleId,blockNumber, downloadCancelReason, reserved,privateDataLen, privateDataBytes)
DSExpire
DownloadCancel(downloadId, moduleId,blockNumber, downloadCancelReason, reserved,privateDataLen, privateDataBytes)
DSExpire
DSExpire
tHold
DSIdle

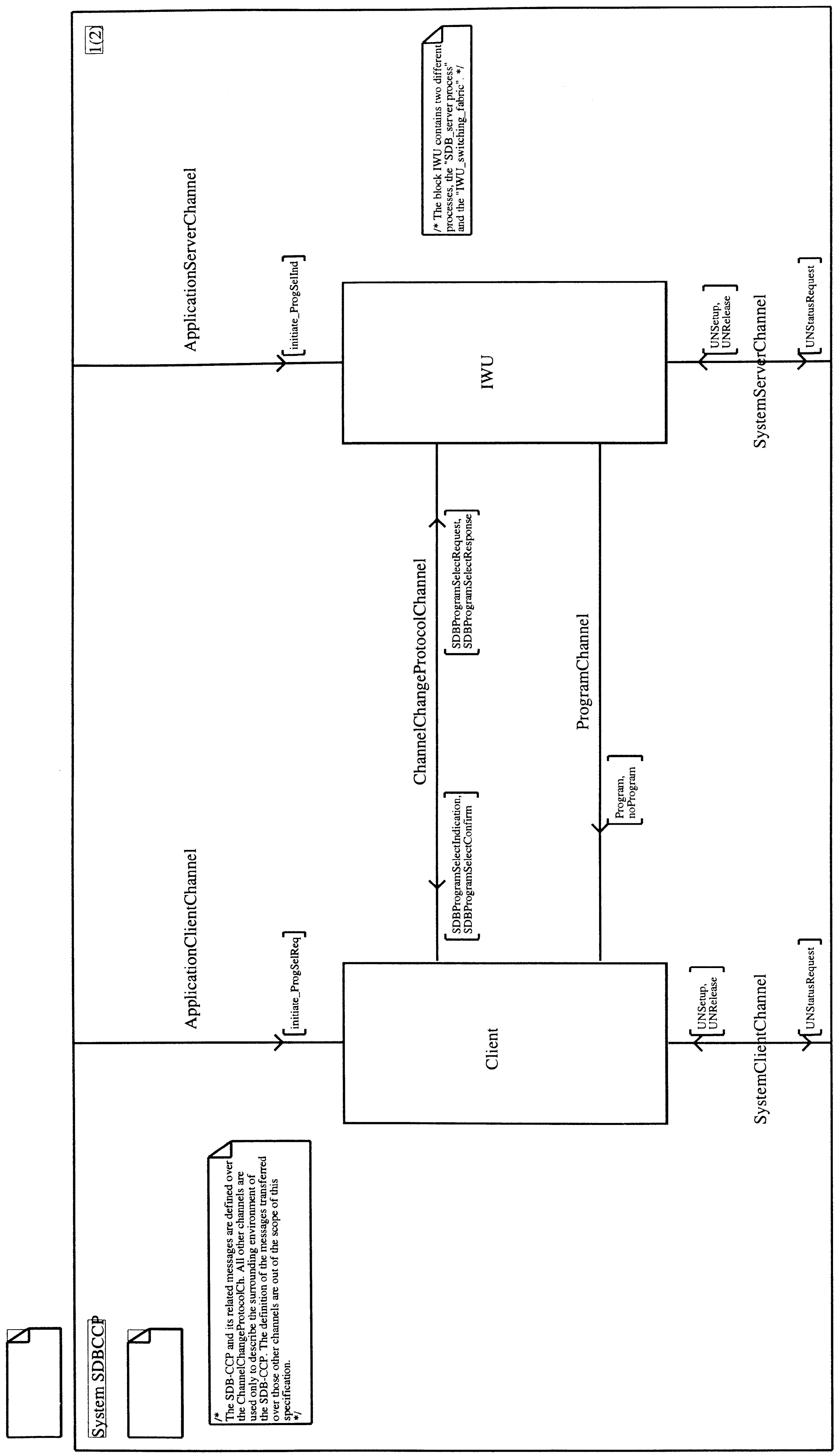

Process DownloadServer
2(3)
;
timer tSDownloadScenario;
timer tDownloadInfoResponse;
timer tSDownloadServerInitiate;
timer tSSendCarouselData;
timer tSend;
timer tHold;
DCL
ackPeriod ackPeriodType,
blockDataBytes blockDataBytesType,
blockNumber blockNumberType,
blockSize blockSizeType,
bufferSize bufferSizeType,
downloadCancelReason downloadCancelReasonType,
downloadId downloadIdType,
downloadReason downloadReasonType,
endOfModule Boolean,
endOfCarousel Boolean,
lowerTransmitWindowEdge Integer,
maximumBlockSize blockSizeType,
MaxRetry Integer,
moduleId moduleIdType,
Modules ModuleTableType,
moduleVersion moduleVersionType,
numberOfModules numberOfModulesType,
newCarousel Boolean,
privateDataBytes privateDataBytesType,
privateDataLen privateDataLenType,
reserved reservedType,
retry Integer,
serverId NSAPType,
tSDWDuration Duration,
tSDSDuration Duration,
tSDSIDuration Duration,
tSendDuration Duration,
tSSCDDuration Duration,
upperTransmitWindowEdge Integer,
CompatibilityDescriptor CompatibilityDescriptorType,
valid Boolean,
windowSize windowSizeType;
DSActiveCarousel
tSSendCarouselData
" /* Next block in Carousel */
DownloadDataBlock (moduleId,moduleVersion, reserved,blockNumber, blockDataBytes)
endOfCarousel
false
set(now+tSSCDDuration, tSSendCarouselData)
-
true
newCarousel
true
DownloadInfoIndication(downloadId, blockSize, windowSize, ackPeriod, tSDWDuration, tSDSDuration, CompatibilityDescriptor, numberOfModules,Modules, privateDataLen,privateDataBytes)
set(now+tSSCDDuration, tSSendCarouselData)
-
false
DSIdle
WFDStart
tDownloadInfoResponse
retry &lt; MaxRetry
true
DownloadInfoResponse(downloadId, blockSize, windowSize, ackPeriod, tSDWDuration,tSDSDuration, CompatibilityDescriptor, numberOfModules,Modules, privateDataLen,privateDataBytes)
set(now+tCDIRDuration, tDownloadInfoResponse)
WFDStart
false
downloadCancelReason:= rsnRetrans
DownloadCancel(downloadId, moduleId,blockNumber, downloadCancelReason, reserved,privateDataLen, privateDataBytes)
DSExpire
Sends the fist block in the first module described in the module table described in DownloadInfoResponse message
DownloadDataRequest (moduleId,blockNumber, downloadReason)
downloadReason
rsnStart
" /* Set LowerTransmitWindowEdge && UpperTransmitWindowEdge */
DownloadDataBlock (moduleId,moduleVersion, reserved,blockNumber, blockDataBytes)
reset (tDownloadInfoResponse)
set(now+tSendDuration, tSend)
DSActive
else
downloadCancelReason:= rsnFatal
DownloadCancel (downloadId,moduleId, blockNumber,downloadCancelReason, reserved,privateDataLen, privateDataBytes)
DSExpire

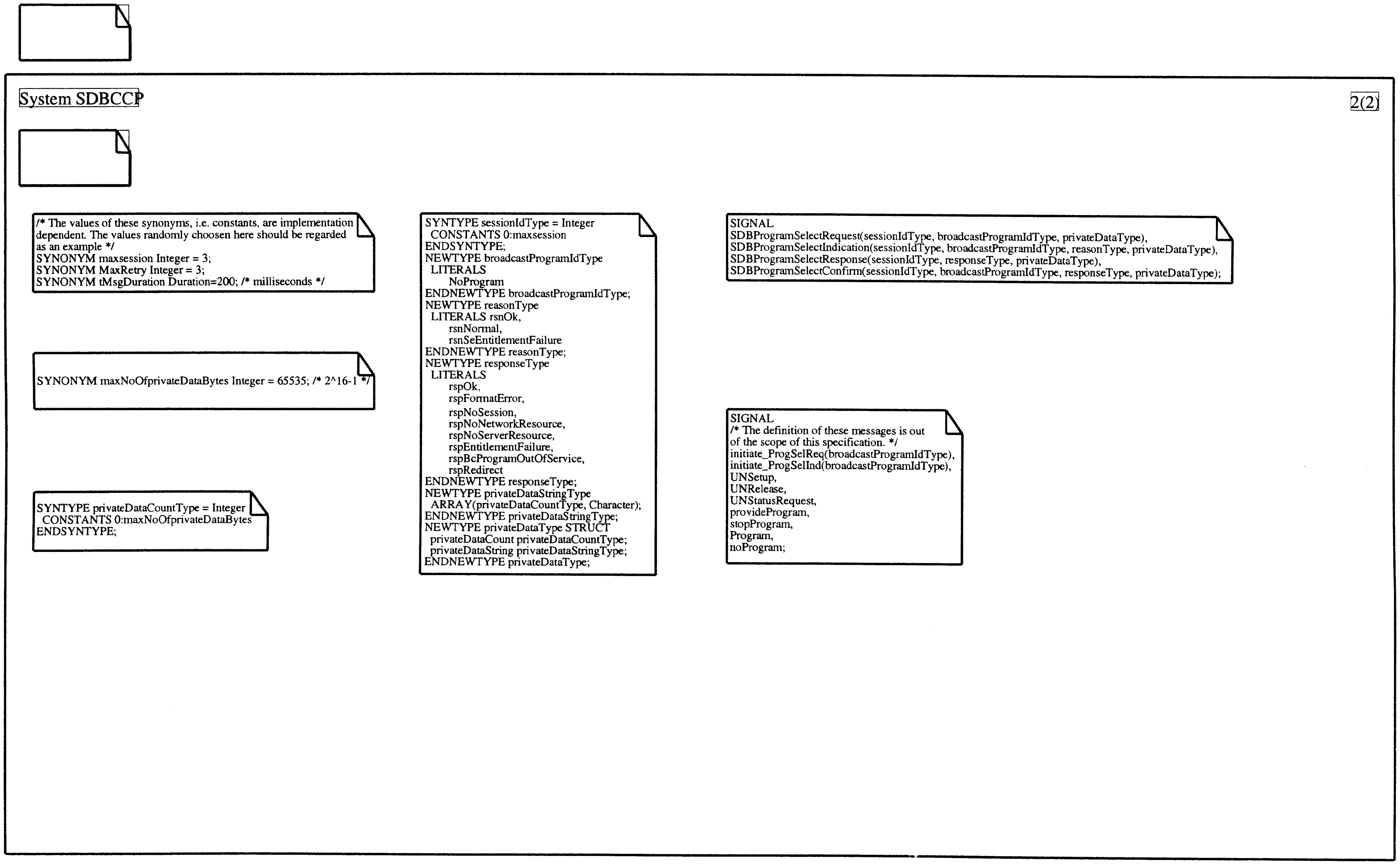

Process DownloadServer
3(3)
timer tSDownloadScenario;
timer tDownloadInfoResponse;
timer tDownloadServerInitiate;
timer tSSendCarouselData;
timer tSend;
timer tHold;
DCL
ackPeriod ackPeriodType,
blockDataBytes blockDataBytesType,
blockNumber blockNumberType,
blockSize blockSizeType,
bufferSize bufferSizeType,
downloadCancelReason downloadCancelReasonType,
downloadId downloadIdType,
downloadReason downloadReasonType,
endOfModule Boolean,
endOfCarousel Boolean,
lowerTransmitWindowEdge Integer,
maximumBlockSize blockSizeType,
MaxRetry Integer,
moduleId moduleIdType,
Modules ModuleTableType,
moduleVersion moduleVersionType,
numberOfModules numberOfModulesType,
newCarousel Boolean,
privateDataBytes privateDataBytesType,
privateDataLen privateDataLenType,
reserved reservedType,
retry Integer,
serverId NSAPType,
tSDWDuration Duration,
tSDSDuration Duration,
tSDSIDuration Duration,
tSendDuration Duration,
tSSCDDuration Duration,
upperTransmitWindowEdge Integer,
CompatibilityDescriptor CompatibilityDescriptorType,
valid Boolean,
windowSize windowSizeType;
DSActive
tSend
endOfModule
false
true
DownloadDataBlock (moduleId,moduleVersion, reserved,blockNumber, blockDataBytes)
Sends the next block in the module
blockNumber:= blockNumber+1
set(now+tSendDuration, tSend)
-
lowerTransmitWindowEdge= upperTransmitWindowEdge
false
true
/* Set moduleId value according to the ModuleTable */
blockNumber := 0
DownloadDataBlock (moduleId,moduleVersion, reserved,blockNumber, blockDataBytes)
set(now+tSendDuration, tSend)
-
Send the next module in order in which the modules are described in the module table in the DownloadInfoResponse message
reset(tSend)
-
Stall the download and wait for DownloadDataRequest
DownloadDataRequest (moduleId,blockNumber, downloadReason)
downloadReason
rsnEnd
DSExpire
rsnAckCont, rsnNakRetransWindow, rsnNakRetransBlock
/* Check moduleID && blockNumber */
valid
true
false
/* Update LowerTransmitWindowEdge && UpperTransmitWindowEdge */
set(now+tSendDuration, tSend)
-
downloadCancelReason := rsnInvalidBlock
else
downloadCancelReason := rsnFatal
DownloadCancel (downloadId,moduleId, blockNumber,downloadCancelReason, reserved,privateDataLen, privateDataBytes)
DSExpire

A.4 U-N Switched Digital Broadcast Channel Change Protocol

The following pages contain the U-N SDB-CCP SDLs.

Figure A-3 U-N SDB-CCP Specification and Description Language


System SDBCCP
1(2)
/* The SDB-CCP and its related messages are defined over the ChannelChangeProtocolCh. All other channels are used only to describe the surrounding environment of the SDB-CCP. The definition of the messages transferred over those other channels are out of the scope of this specification. */
/* The block IWU contains two different processes, the "SDB_server process" and the "IWU_switching_fabric". */
ApplicationServerChannel
[initiate_ProgSelInd]
IWU
[UNSetup, UNRelease]
[UNStatusRequest]
SystemServerChannel
ChannelChangeProtocolChannel
[SDBProgramSelectRequest, SDBProgramSelectResponse]
[SDBProgramSelectIndication, SDBProgramSelectConfirm]
ProgramChannel
[Program, noProgram]
ApplicationClientChannel
[initiate_ProgSelReq]
Client
[UNSetup, UNRelease]
[UNStatusRequest]
SystemClientChannel



```
/* The values of these synonyms, i.e. constants, are implementation
dependent. The values randomly choosen here should be regarded
as an example */
SYNONYM maxsession Integer = 3;
SYNONYM MaxRetry Integer = 3;
SYNONYM tMsgDuration Duration=200; /* milliseconds */
```

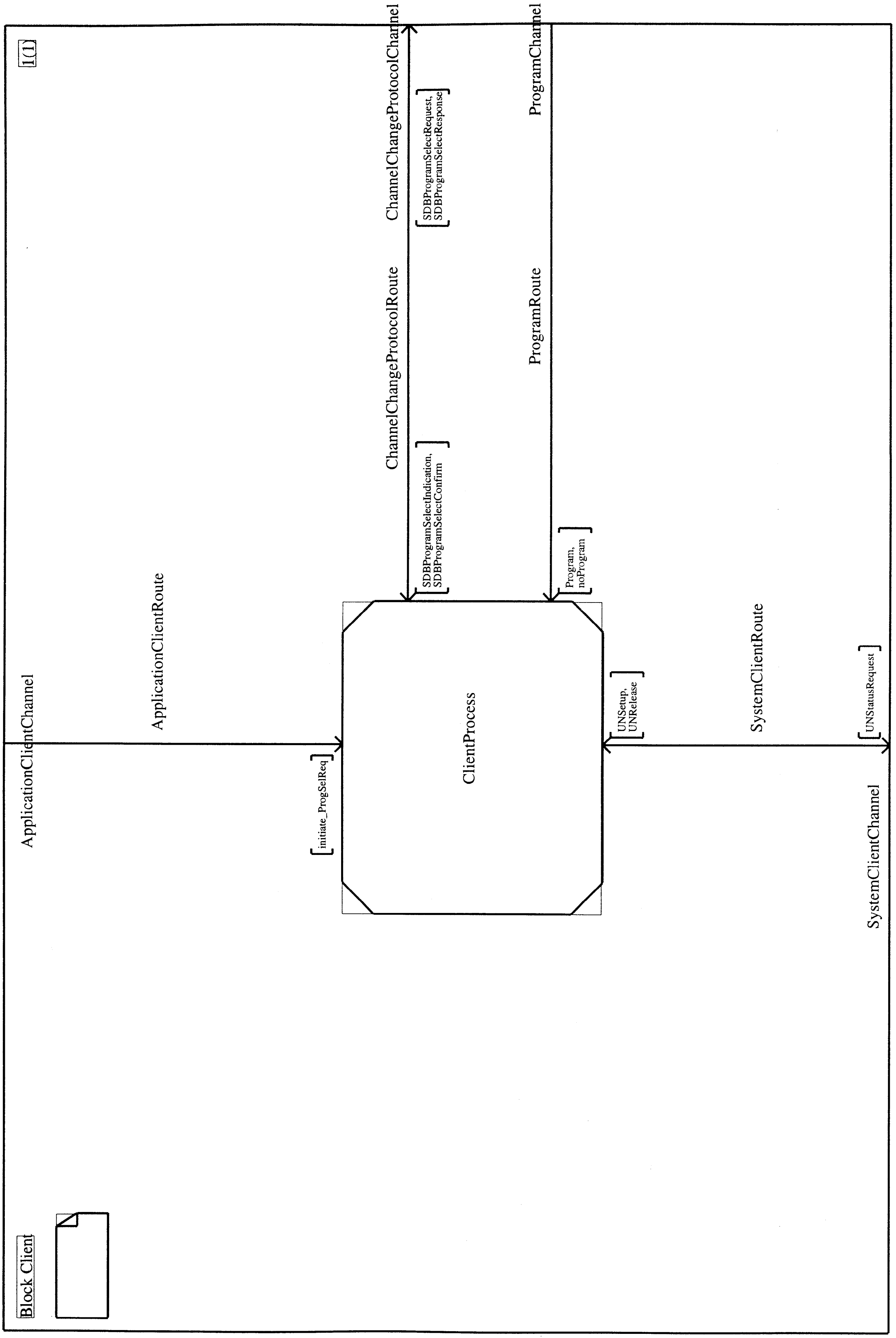
```
SYNONYM maxNoOfprivateDataBytes Integer = 65535; /* 2^16-1 */
```

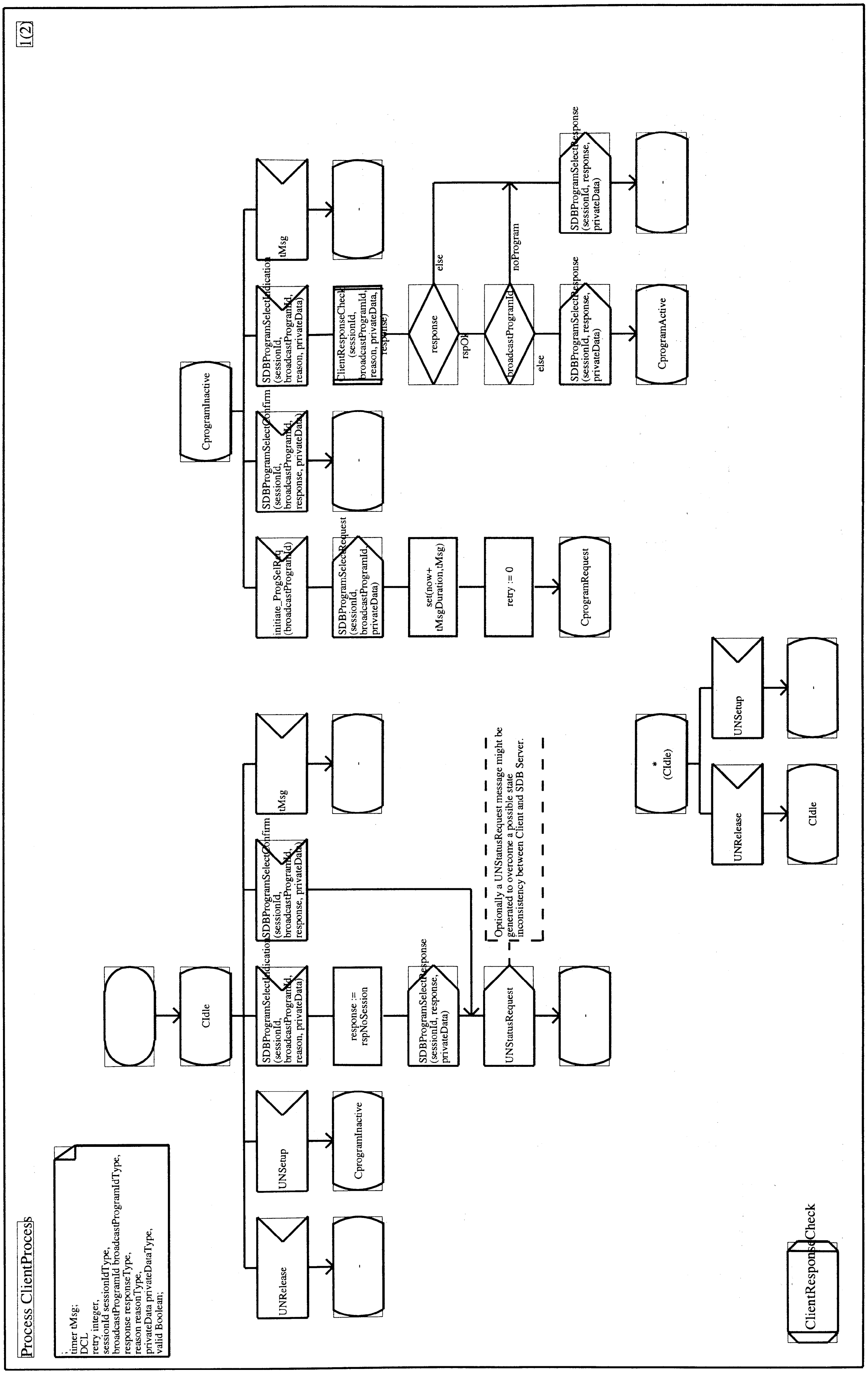
```
SYNTYPE privateDataCountType = Integer
  CONSTANTS 0:maxNoOfprivateDataBytes
ENDSYNTYPE;
```

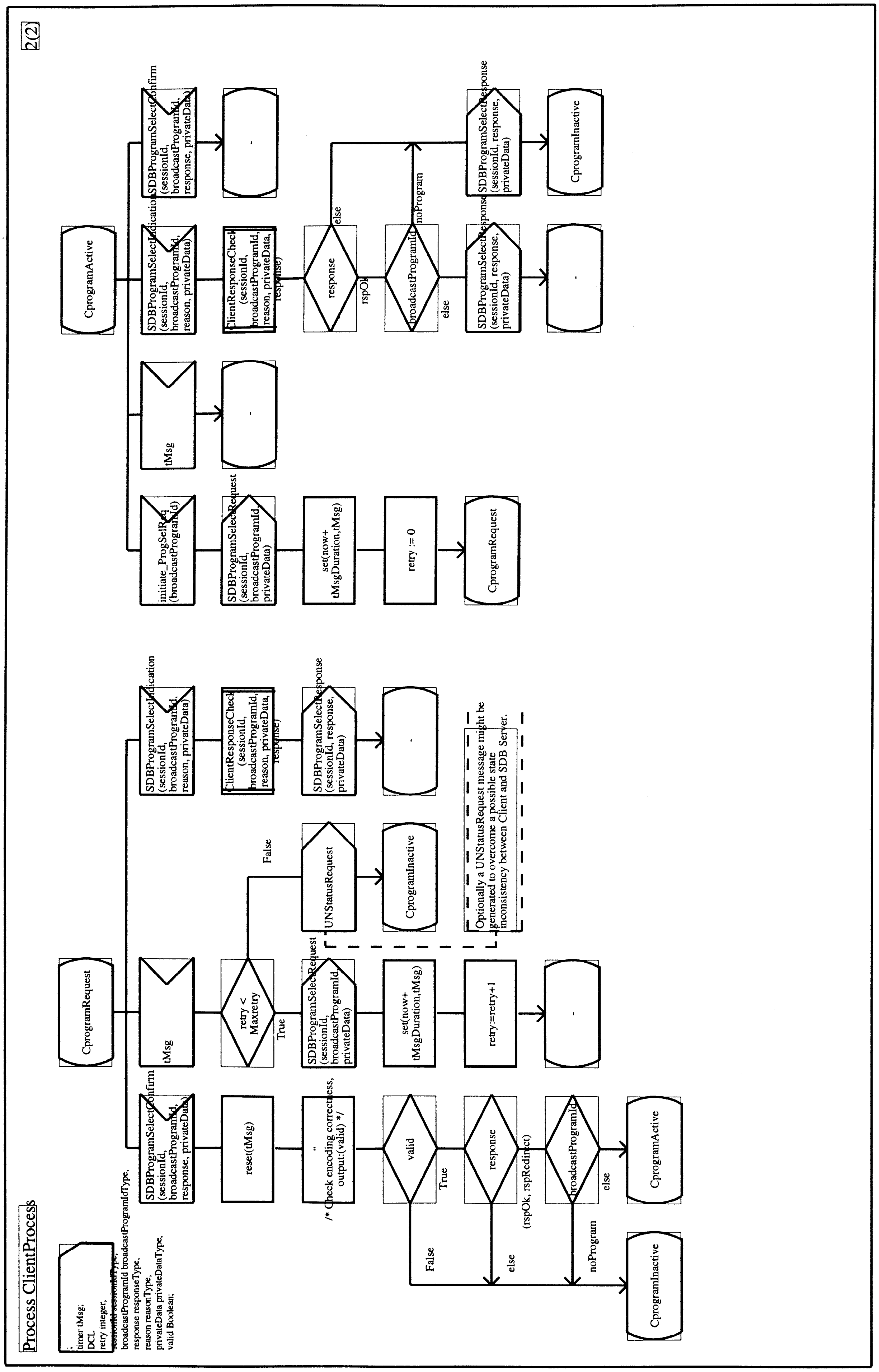
```
SYNTYPE sessionIdType = Integer
  CONSTANTS 0:maxsession
ENDSYNTYPE;
NEWTYPE broadcastProgramIdType
  LITERALS
      NoProgram
ENDNEWTYPE broadcastProgramIdType;
NEWTYPE reasonType
  LITERALS rsnOk,
      rsnNormal,
      rsnSeEntitlementFailure
ENDNEWTYPE reasonType;
NEWTYPE responseType
  LITERALS
      rspOk,
      rspFormatError,
      rspNoSession,
      rspNoNetworkResource,
      rspNoServerResource,
      rspEntitlementFailure,
      rspBcProgramOutOfService,
      rspRedirect
ENDNEWTYPE responseType;
NEWTYPE privateDataStringType
  ARRAY(privateDataCountType, Character);
ENDNEWTYPE privateDataStringType;
NEWTYPE privateDataType STRUCT
  privateDataCount privateDataCountType;
  privateDataString privateDataStringType;
ENDNEWTYPE privateDataType;
```

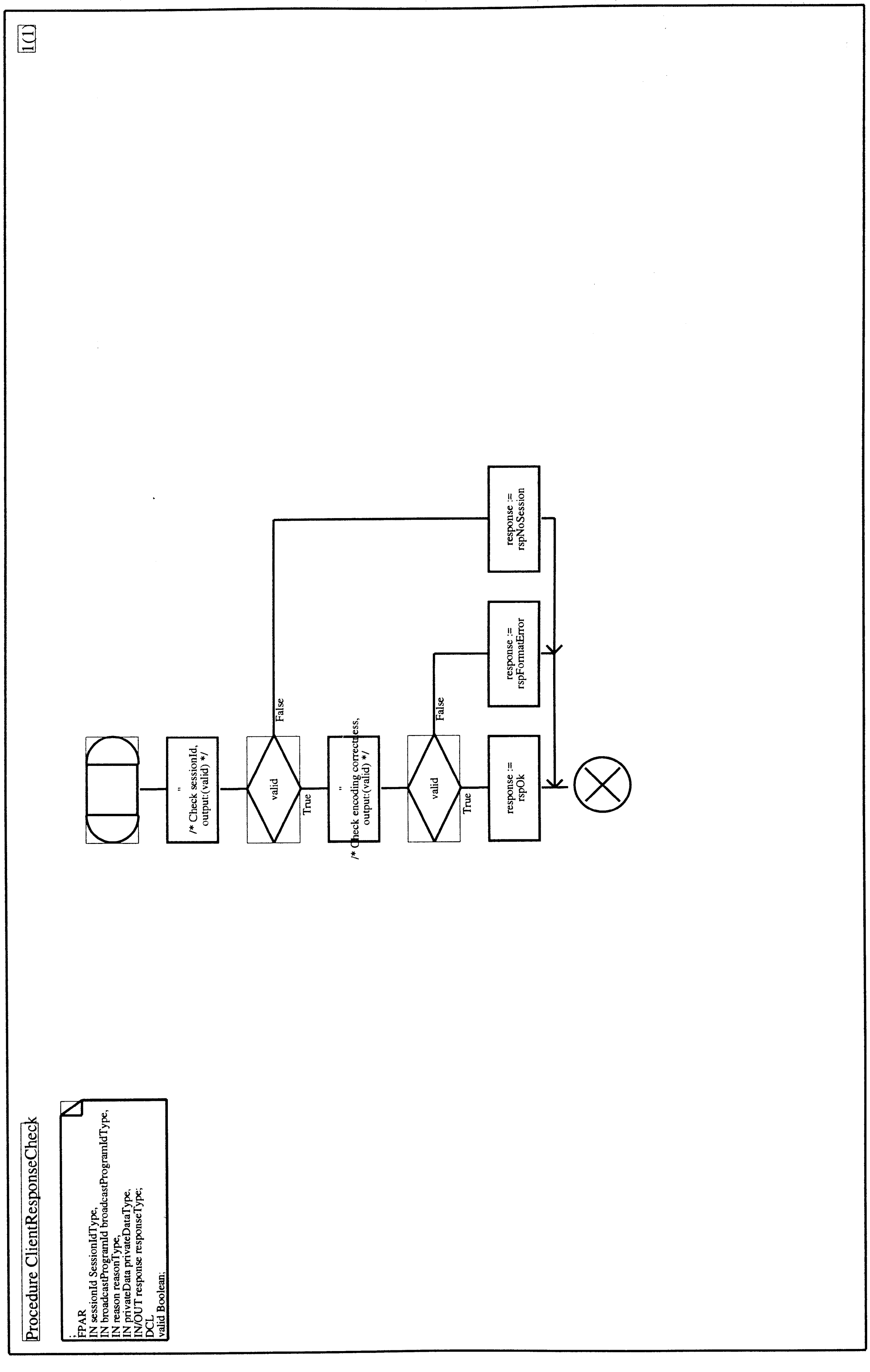
```
SIGNAL
SDBProgramSelectRequest(sessionIdType, broadcastProgramIdType, privateDataType),
SDBProgramSelectIndication(sessionIdType, broadcastProgramIdType, reasonType, privateDataType),
SDBProgramSelectResponse(sessionIdType, responseType, privateDataType),
SDBProgramSelectConfirm(sessionIdType, broadcastProgramIdType, responseType, privateDataType);
```

```
SIGNAL
/* The definition of these messages is out
of the scope of this specification. */
initiate_ProgSelReq(broadcastProgramIdType),
initiate_ProgSelInd(broadcastProgramIdType),
UNSetup,
UNRelease,
UNStatusRequest,
provideProgram,
stopProgram,
Program,
noProgram;
```


Block Client
1(1)
ApplicationClientChannel
ApplicationClientRoute
initiate_ProgSelReq
ClientProcess
ChannelChangeProtocolChannel
SDBProgramSelectRequest,
SDBProgramSelectResponse
ChannelChangeProtocolRoute
SDBProgramSelectIndication,
SDBProgramSelectConfirm
ProgramChannel
ProgramRoute
Program,
noProgram
UNSetup,
UNRelease
SystemClientRoute
UNStatusRequest
SystemClientChannel

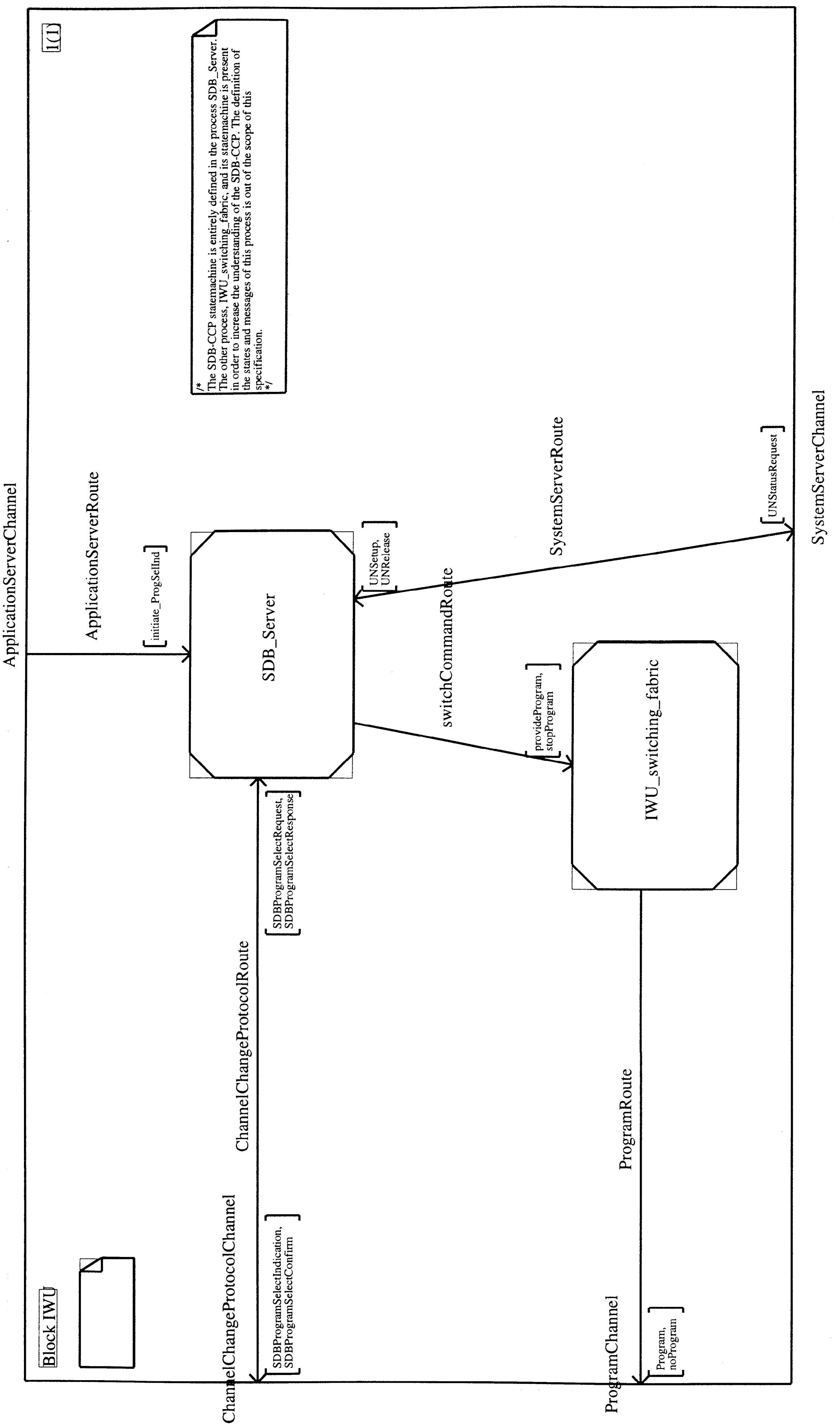

Process ClientProcess
1(2)
;
timer tMsg;
DCL
retry integer,
sessionId sessionIdType,
broadcastProgramId broadcastProgramIdType,
response responseType,
reason reasonType,
privateData privateDataType,
valid Boolean;
CIdle
UNRelease
-
UNSetup
CprogramInactive
SDBProgramSelectIndication (sessionId, broadcastProgramId, reason, privateData)
response := rspNoSession
SDBProgramSelectResponse (sessionId, response, privateData)
UNStatusRequest
-
SDBProgramSelectConfirm (sessionId, broadcastProgramId, response, privateData)
Optionally a UNStatusRequest message might be generated to overcome a possible state inconsistency between Client and SDB Server.
tMsg
-
*
(CIdle)
UNRelease
CIdle
UNSetup
-
CprogramInactive
initiate_ProgSelReq (broadcastProgramId)
SDBProgramSelectRequest (sessionId, broadcastProgramId, privateData)
set(now+ tMsgDuration,tMsg)
retry := 0
CprogramRequest
SDBProgramSelectConfirm (sessionId, broadcastProgramId, response, privateData)
-
SDBProgramSelectIndication (sessionId, broadcastProgramId, reason, privateData)
ClientResponseCheck (sessionId, broadcastProgramId, reason, privateData, response)
response
rspOK
else
broadcastProgramId
else
noProgram
SDBProgramSelectResponse (sessionId, response, privateData)
CprogramActive
SDBProgramSelectResponse (sessionId, response, privateData)
-
tMsg
-
ClientResponseCheck

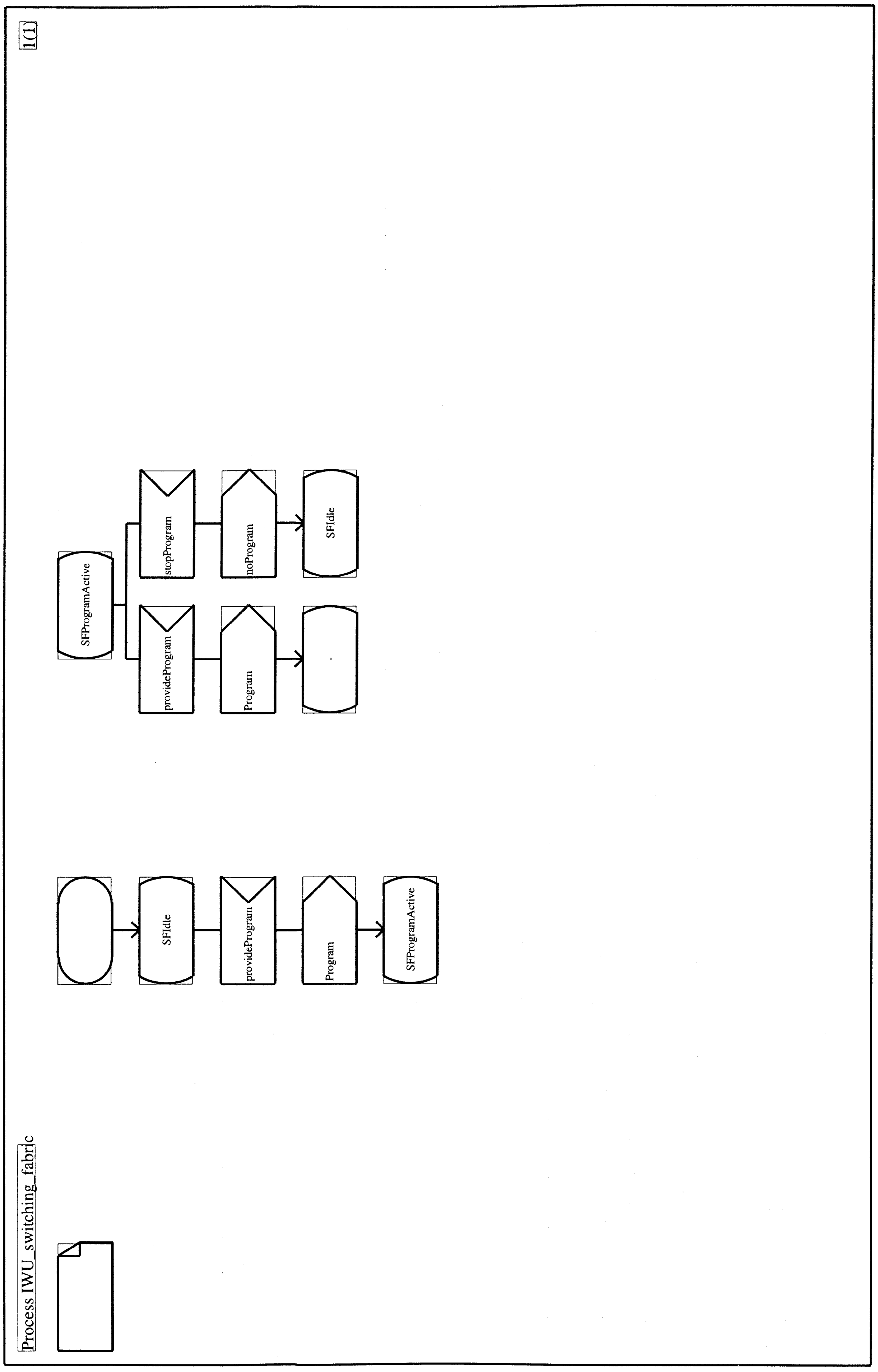

Process ClientProcess
2(2)
timer tMsg;
DCL
retry integer,
sessionId sessionIdType,
broadcastProgramId broadcastProgramIdType,
response responseType,
reason reasonType,
privateData privateDataType,
valid Boolean;
CprogramRequest
SDBProgramSelectConfirm (sessionId, broadcastProgramId, response, privateData)
reset(tMsg)
/* Check encoding correctness, output:(valid) */
valid
True
False
response
(rspOk, rspRedirect)
else
broadcastProgramId
else
noProgram
CprogramActive
CprogramInactive
tMsg
retry < Maxretry
True
False
SDBProgramSelectRequest (sessionId, broadcastProgramId, privateData)
set(now+ tMsgDuration,tMsg)
retry:=retry+1
-
UNStatusRequest
CprogramInactive
Optionally a UNStatusRequest message might be generated to overcome a possible state inconsistency between Client and SDB Server.
SDBProgramSelectIndication (sessionId, broadcastProgramId, reason, privateData)
ClientResponseCheck (sessionId, broadcastProgramId, reason, privateData, response)
SDBProgramSelectResponse (sessionId, response, privateData)
-
CprogramActive
initiate_ProgSelReq (broadcastProgramId)
SDBProgramSelectRequest (sessionId, broadcastProgramId, privateData)
set(now+ tMsgDuration,tMsg)
retry := 0
CprogramRequest
tMsg
-
SDBProgramSelectIndication (sessionId, broadcastProgramId, reason, privateData)
ClientResponseCheck (sessionId, broadcastProgramId, reason, privateData, response)
response
rspOk
else
broadcastProgramId
else
noProgram
SDBProgramSelectResponse (sessionId, response, privateData)
-
SDBProgramSelectResponse (sessionId, response, privateData)
CprogramInactive
SDBProgramSelectConfirm (sessionId, broadcastProgramId, response, privateData)
-

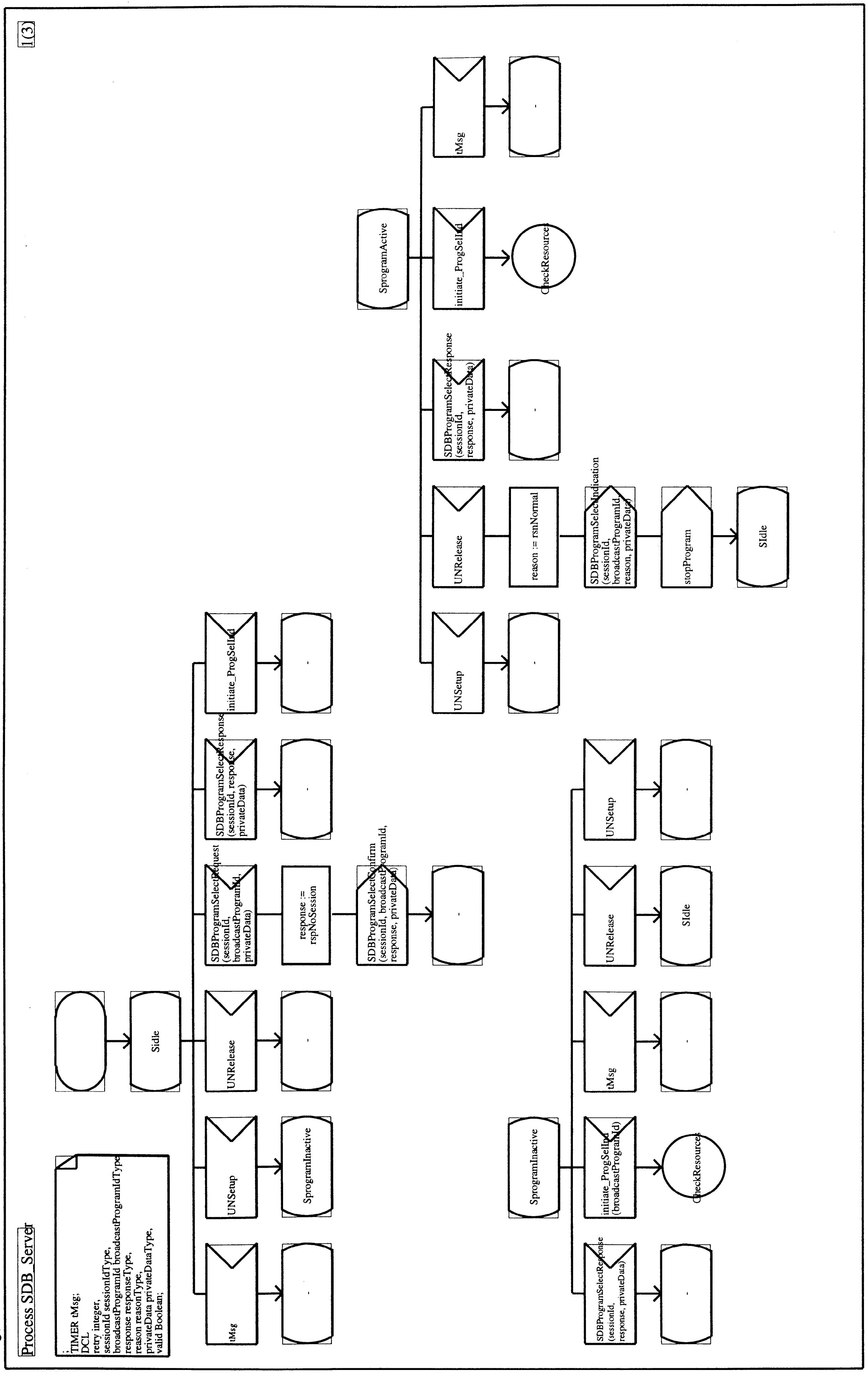

Procedure ClientResponseCheck

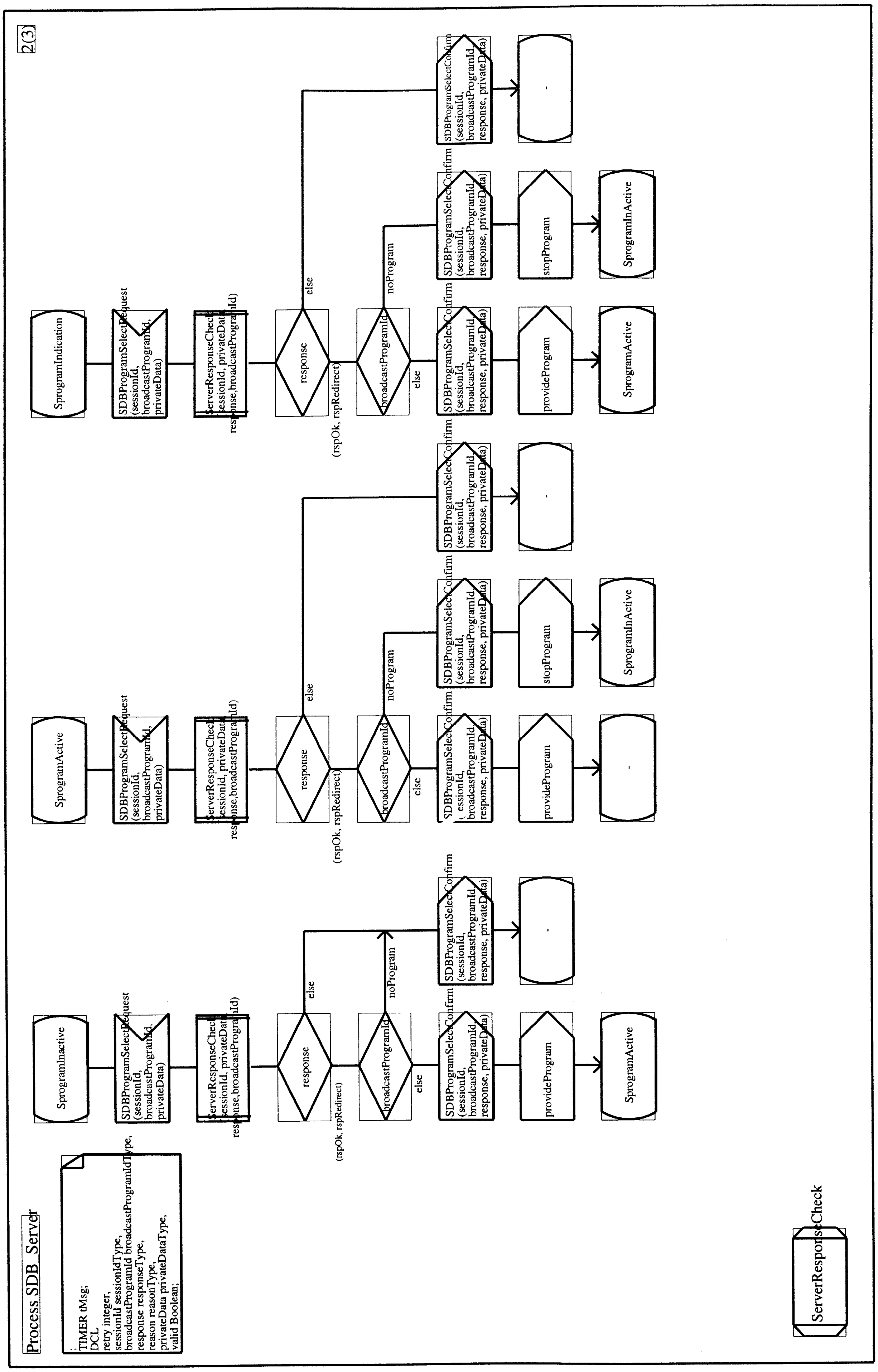
```
;
FPAR
IN sessionId SessionIdType,
IN broadcastProgramId broadcastProgramIdType,
IN reason reasonType,
IN privateData privateDataType,
IN/OUT response responseType;
DCL
valid Boolean;
```

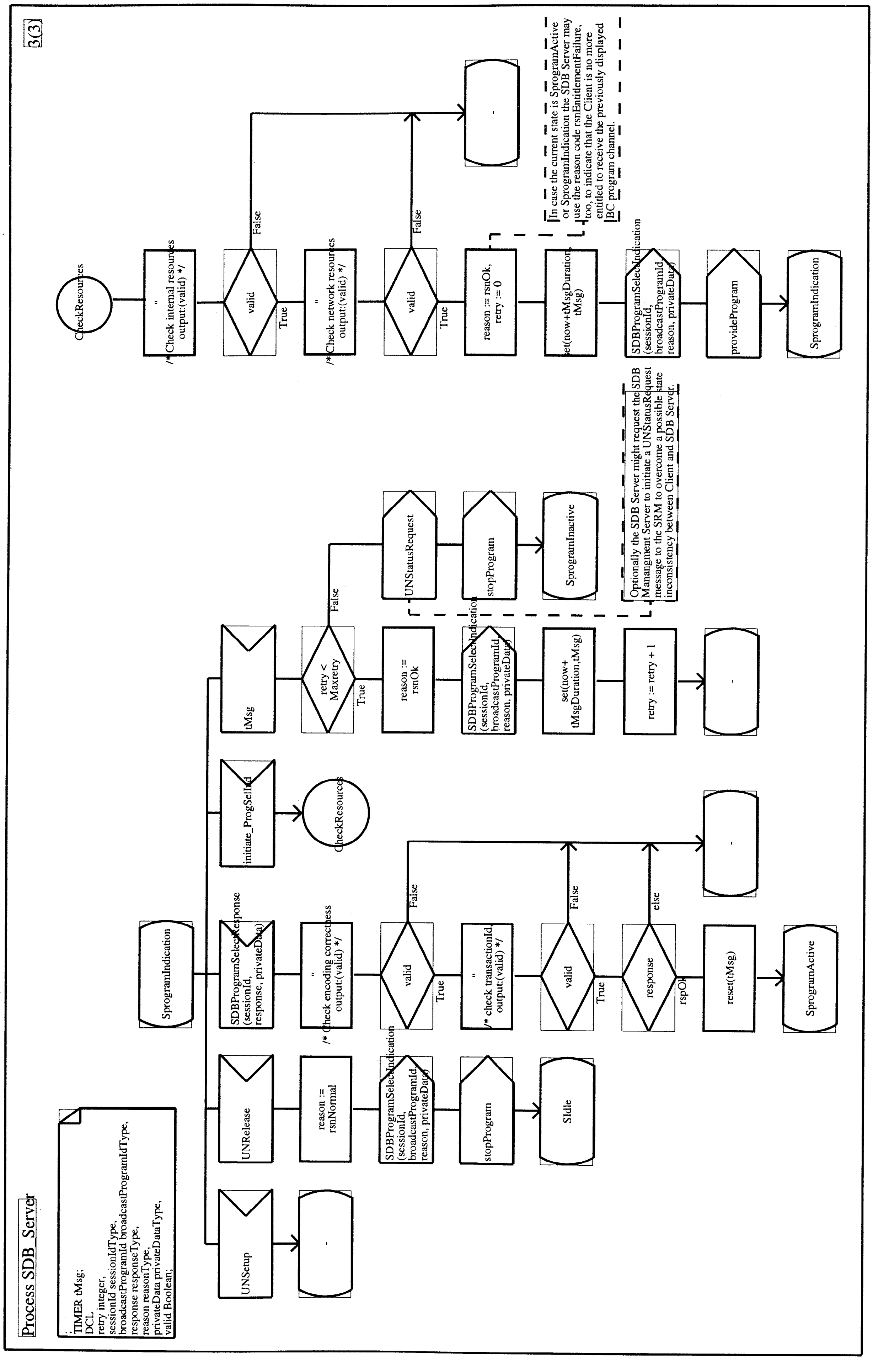
1(1)
/* Check sessionId, output:(valid) */
valid
True
False
/* Check encoding correctness, output:(valid) */
valid
True
False
response := rspOk
response := rspFormatError
response := rspNoSession

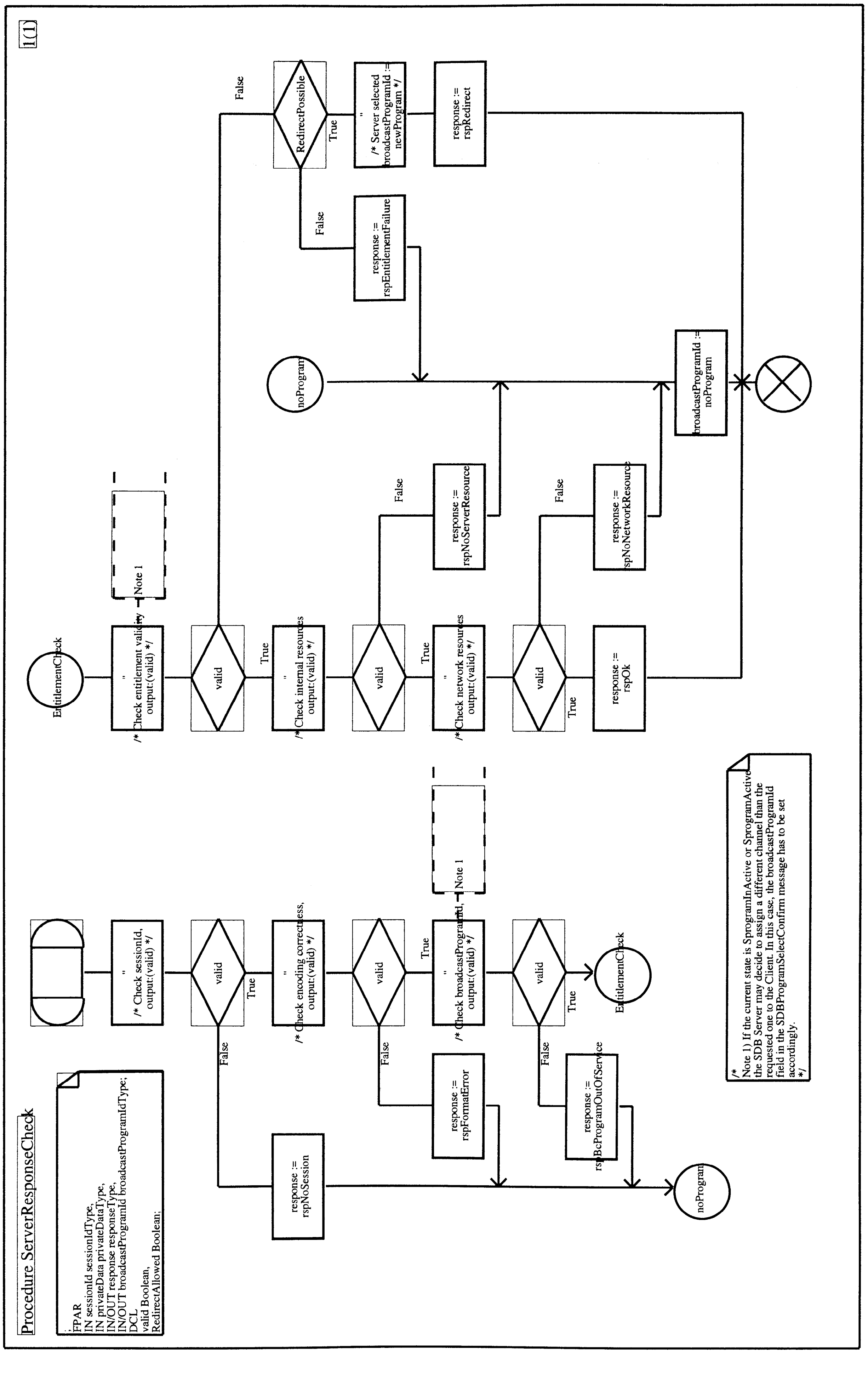

Block IWU
1(1)
/*
The SDB-CCP statemachine is entirely defined in the process SDB_Server.
The other process, IWU_switching_fabric, and its statemachine is present
in order to increase the understanding of the SDB-CCP. The definition of
the states and messages of this process is out of the scope of this
specification.
*/
ApplicationServerChannel
ApplicationServerRoute
[initiate_ProgSelInd]
SDB_Server
[UNSetup, UNRelease]
SystemServerRoute
[UNStatusRequest]
SystemServerChannel
switchCommandRoute
[provideProgram, stopProgram]
IWU_switching_fabric
[SDBProgramSelectRequest, SDBProgramSelectResponse]
ChannelChangeProtocolRoute
ChannelChangeProtocolChannel
[SDBProgramSelectIndication, SDBProgramSelectConfirm]
ProgramRoute
ProgramChannel
[Program, noProgram]


Process IWU_switching_fabric
1(1)
SFIdle
provideProgram
Program
SFProgramActive
SFProgramActive
provideProgram
Program
-
stopProgram
noProgram
SFIdle


Process SDB_Server
1(3)
;
TIMER tMsg;
DCL
retry integer,
sessionId sessionIdType,
broadcastProgramId broadcastProgramIdType,
response responseType,
reason reasonType,
privateData privateDataType,
valid Boolean;
SIdle
tMsg
-
UNSetup
SprogramInactive
UNRelease
-
SDBProgramSelectRequest (sessionId, broadcastProgramId, privateData)
response := rspNoSession
SDBProgramSelectConfirm (sessionId, broadcastProgramId, response, privateData)
-
SDBProgramSelectResponse (sessionId, response, privateData)
-
initiate_ProgSelInd
-
SprogramActive
UNSetup
-
UNRelease
reason := rsnNormal
SDBProgramSelectIndication (sessionId, broadcastProgramId, reason, privateData)
stopProgram
SIdle
SDBProgramSelectResponse (sessionId, response, privateData)
-
initiate_ProgSelInd
CheckResources
tMsg
-
SprogramInactive
SDBProgramSelectResponse (sessionId, response, privateData)
-
initiate_ProgSelInd (broadcastProgramId)
CheckResources
tMsg
-
UNRelease
SIdle
UNSetup
-


Process SDB_Server
2(3)
;
TIMER tMsg;
DCL
retry integer,
sessionId sessionIdType,
broadcastProgramId broadcastProgramIdType,
response responseType,
reason reasonType,
privateData privateDataType,
valid Boolean;
SprogramInactive
SprogramActive
SprogramIndication
SDBProgramSelectRequest (sessionId, broadcastProgramId, privateData)
ServerResponseCheck (sessionId, privateData, response,broadcastProgramId)
response
(rspOk, rspRedirect)
else
broadcastProgramId
noProgram
SDBProgramSelectConfirm (sessionId, broadcastProgramId, response, privateData)
provideProgram
stopProgram
SprogramActive
SprogramInActive
-
ServerResponseCheck


Process SDB_Server
3(3)
;
TIMER tMsg;
DCL
retry integer,
sessionId sessionIdType,
broadcastProgramId broadcastProgramIdType,
response responseType,
reason reasonType,
privateData privateDataType,
valid Boolean;
SprogramIndication
UNSetup
-
UNRelease
reason := rsnNormal
SDBProgramSelectIndication (sessionId, broadcastProgramId reason, privateData)
stopProgram
SIdle
SDBProgramSelectResponse (sessionId, response, privateData)
/* Check encoding correctness output:(valid) */
valid
True
False
/* check transactionId output:(valid) */
valid
True
False
response
rspOk
else
reset(tMsg)
SprogramActive
-
initiate_ProgSelInd
CheckResources
tMsg
retry < Maxretry
True
False
reason := rsnOk
SDBProgramSelectIndication (sessionId, broadcastProgramId reason, privateData)
set(now+ tMsgDuration,tMsg)
retry := retry + 1
-
UNStatusRequest
stopProgram
SprogramInactive
Optionally the SDB Server might request the SDB Manangment Server to initiate a UNStatusRequest message to the SRM to overcome a possible state inconsistency between Client and SDB Server.
CheckResources
/* Check internal resources output:(valid) */
valid
True
False
/* Check network resources output:(valid) */
valid
True
False
-
reason := rsnOk, retry := 0
set(now+tMsgDuration, tMsg)
SDBProgramSelectIndication (sessionId, broadcastProgramId reason, privateData)
provideProgram
SprogramIndication
In case the current state is SprogramActive or SprogramIndication the SDB Server may use the reason code rsnEntitlementFailure, too, to indicate that the Client is no more entitled to receive the previously displayed BC program channel.


1(1)
Procedure ServerResponseCheck
/* Check sessionId, output:(valid) */
valid
True
False
/* Check encoding correctness, output:(valid) */
/* Check broadcastProgramId, output:(valid) */
Note 1
EntitlementCheck
response := rspNoSession
response := rspFormatError
response := rspBcProgramOutOfService
noProgram
/* Check entitlement validity output:(valid) */
/* Check internal resources output:(valid) */
/* Check network resources output:(valid) */
response := rspOk
response := rspNoServerResource
response := rspNoNetworkResource
RedirectPossible
response := rspEntitlementFailure
/* Server selected broadcastProgramId := newProgram */
response := rspRedirect
broadcastProgramId := noProgram

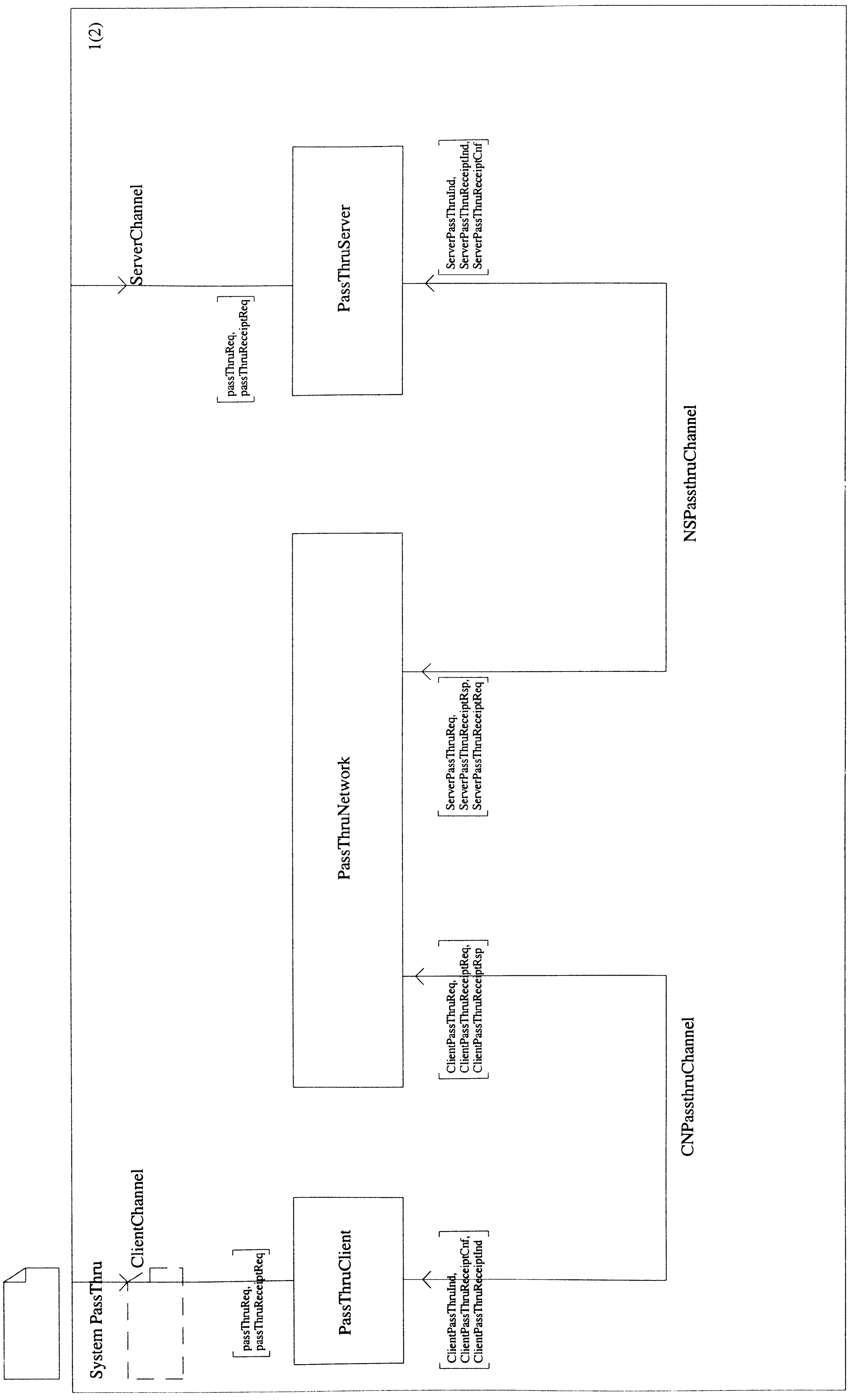
;
FPAR
IN sessionId sessionIdType,
IN privateData privateDataType,
IN/OUT response responseType,
IN/OUT broadcastProgramId broadcastProgramIdType;
DCL
valid Boolean,
RedirectAllowed Boolean;

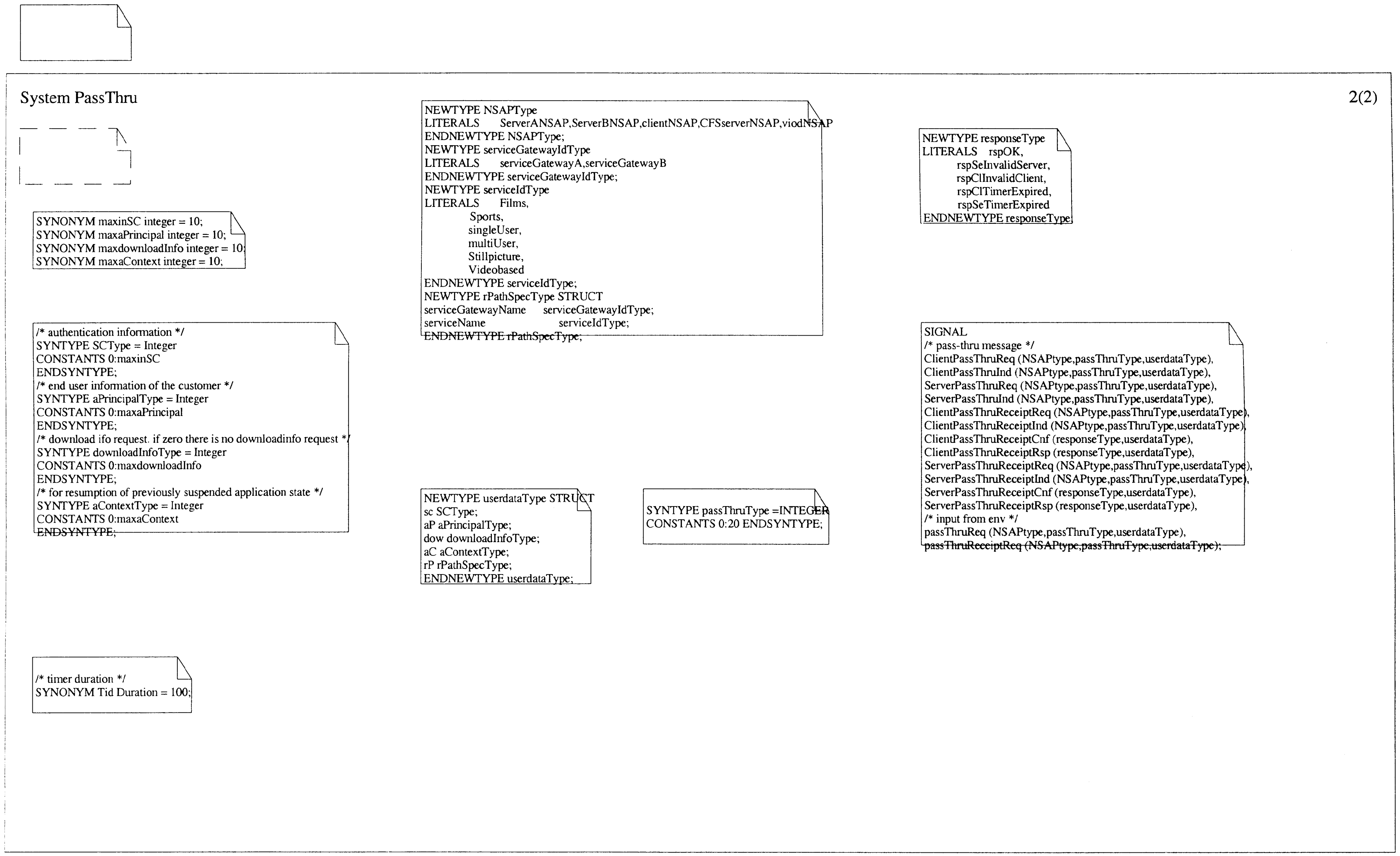
/*
Note 1) If the current state is SprogramInActive or SprogramActive the SDB Server may decide to assign a different channel than the requested one to the Client. In this case, the broadcastProgramId field in the SDBProgramSelectConfirm message has to be set accordingly.
*/

A.5 U-N Pass-Thru

The following pages contain the U-N Pass-Thru SDLs.

Figure A-4 U-N Pass-Thru Specification and Description Language


System PassThru
1(2)
ClientChannel
ServerChannel
passThruReq,
passThruReceiptReq
PassThruServer
ServerPassThruInd,
ServerPassThruReceiptInd,
ServerPassThruReceiptCnf
NSPassthruChannel
PassThruNetwork
ServerPassThruReq,
ServerPassThruReceiptRsp,
ServerPassThruReceiptReq
ClientPassThruReq,
ClientPassThruReceiptReq,
ClientPassThruReceiptRsp
CNPassthruChannel
passThruReq,
passThruReceiptReq
PassThruClient
ClientPassThruInd,
ClientPassThruReceiptCnf,
ClientPassThruReceiptInd



System PassThru

2(2)

```
SYNONYM maxinSC integer = 10;
SYNONYM maxaPrincipal integer = 10;
SYNONYM maxdownloadInfo integer = 10;
SYNONYM maxaContext integer = 10;
```

```
/* authentication information */
SYNTYPE SCType = Integer
CONSTANTS 0:maxinSC
ENDSYNTYPE;
/* end user information of the customer */
SYNTYPE aPrincipalType = Integer
CONSTANTS 0:maxaPrincipal
ENDSYNTYPE;
/* download ifo request. if zero there is no downloadinfo request */
SYNTYPE downloadInfoType = Integer
CONSTANTS 0:maxdownloadInfo
ENDSYNTYPE;
/* for resumption of previously suspended application state */
SYNTYPE aContextType = Integer
CONSTANTS 0:maxaContext
ENDSYNTYPE;
```

```
/* timer duration */
SYNONYM Tid Duration = 100;
```

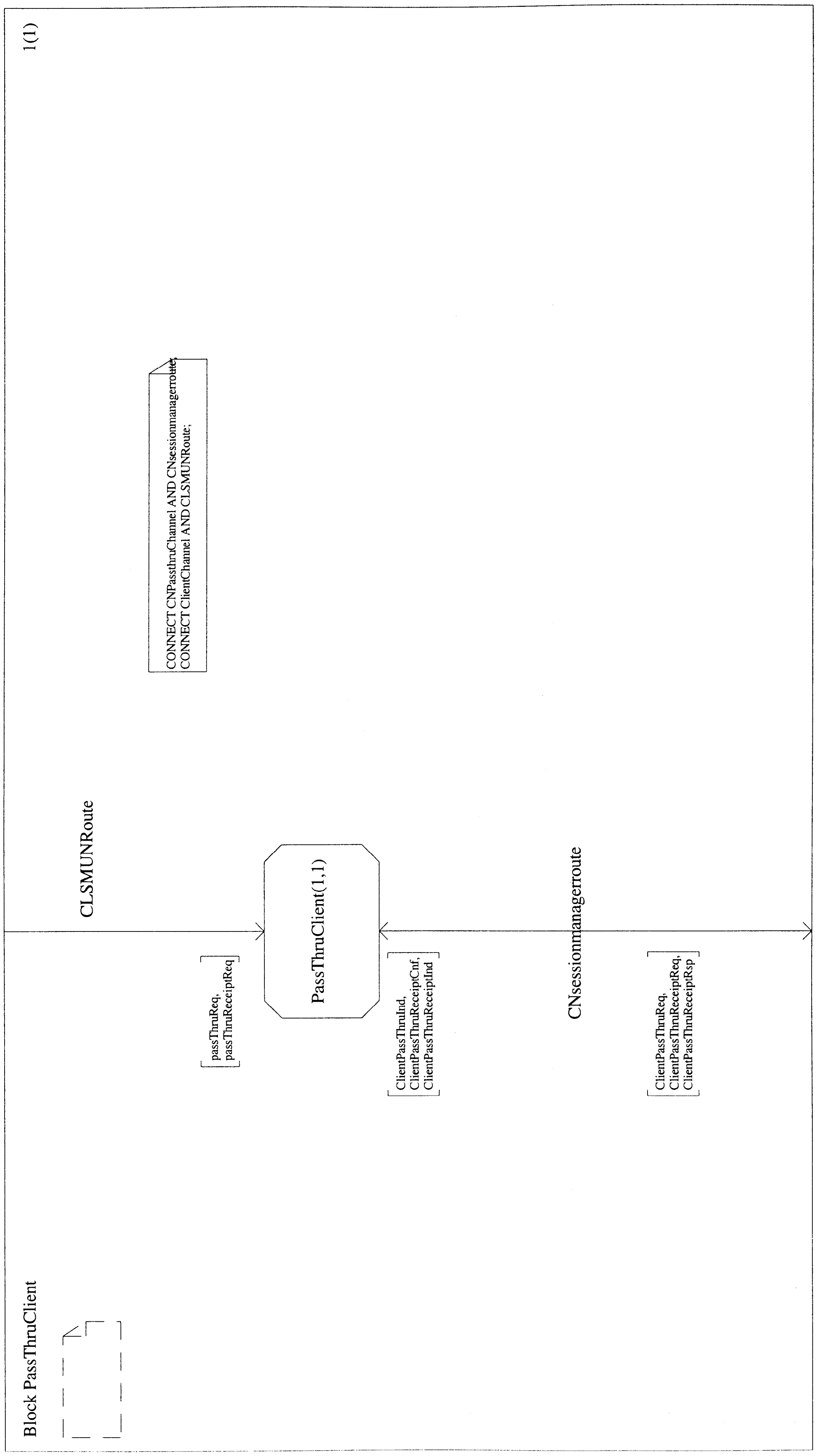
```
NEWTYPE NSAPType
LITERALS      ServerANSAP,ServerBNSAP,clientNSAP,CFSserverNSAP,viodNSAP
ENDNEWTYPE NSAPType;
NEWTYPE serviceGatewayIdType
LITERALS      serviceGatewayA,serviceGatewayB
ENDNEWTYPE serviceGatewayIdType;
NEWTYPE serviceIdType
LITERALS      Films,
          Sports,
          singleUser,
          multiUser,
          Stillpicture,
          Videobased
ENDNEWTYPE serviceIdType;
NEWTYPE rPathSpecType STRUCT
serviceGatewayName      serviceGatewayIdType;
serviceName             serviceIdType;
ENDNEWTYPE rPathSpecType;
```

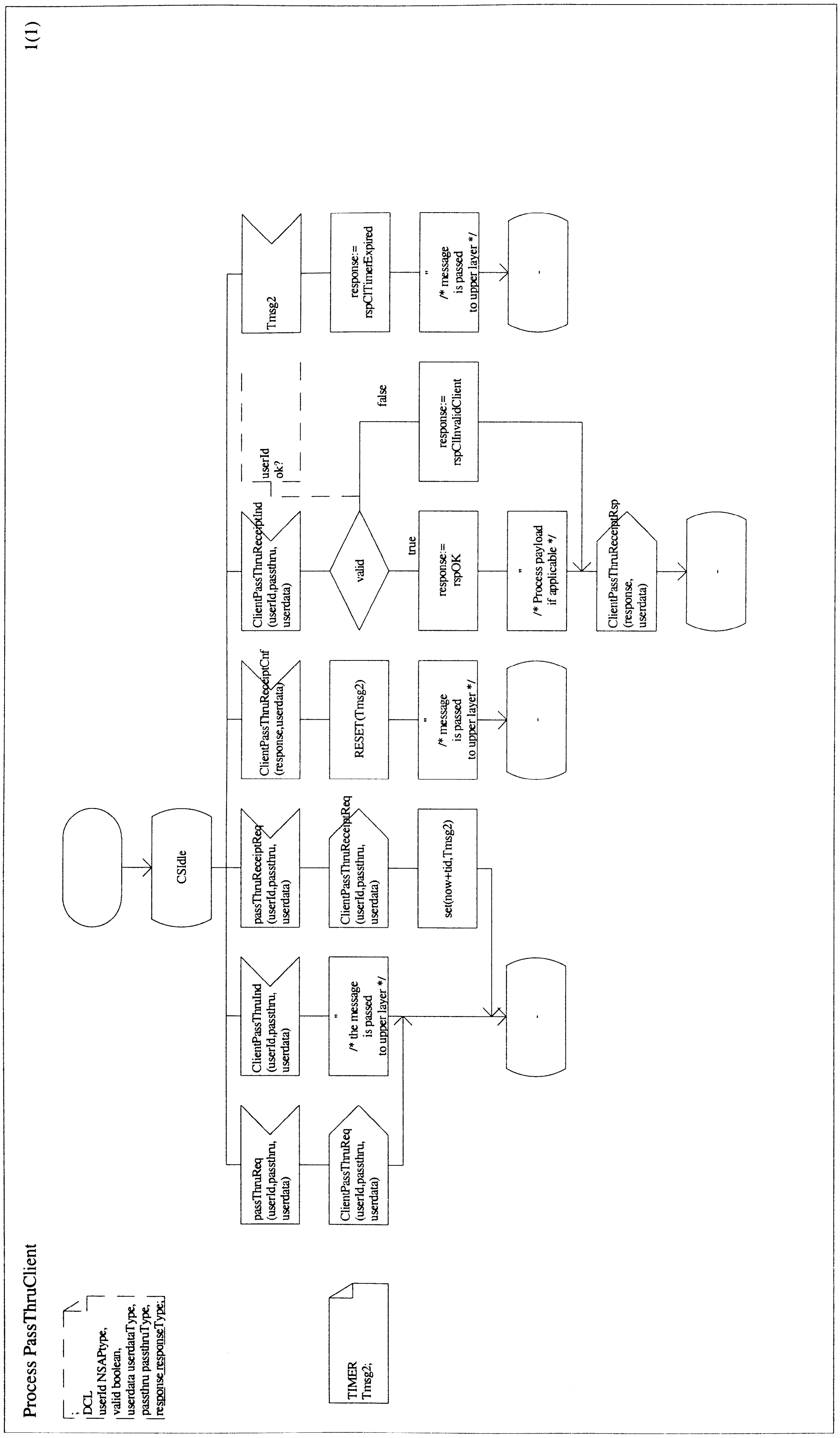
```
NEWTYPE userdataType STRUCT
sc SCType;
aP aPrincipalType;
dow downloadInfoType;
aC aContextType;
rP rPathSpecType;
ENDNEWTYPE userdataType;
```

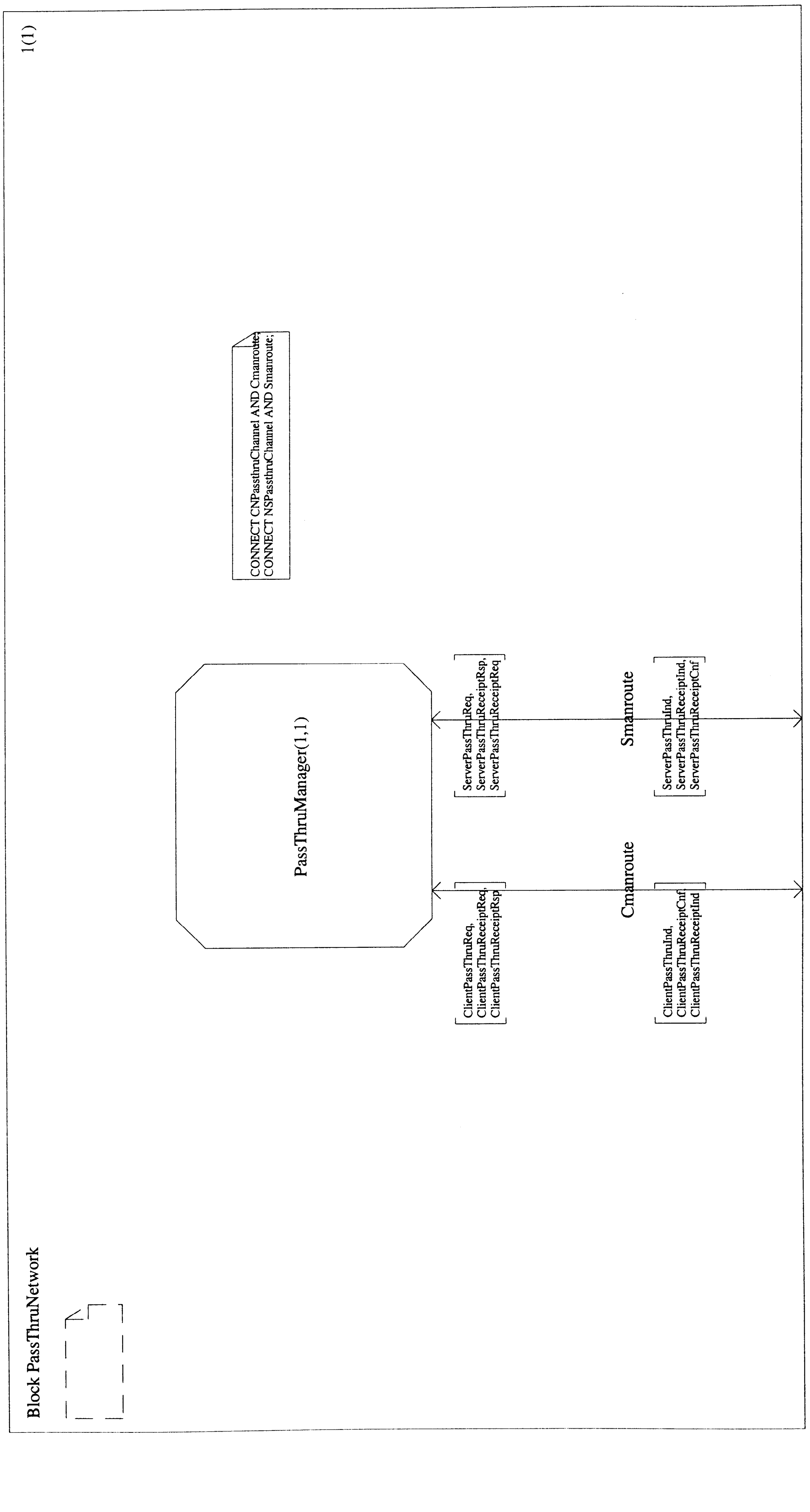
```
SYNTYPE passThruType =INTEGER
CONSTANTS 0:20 ENDSYNTYPE;
```

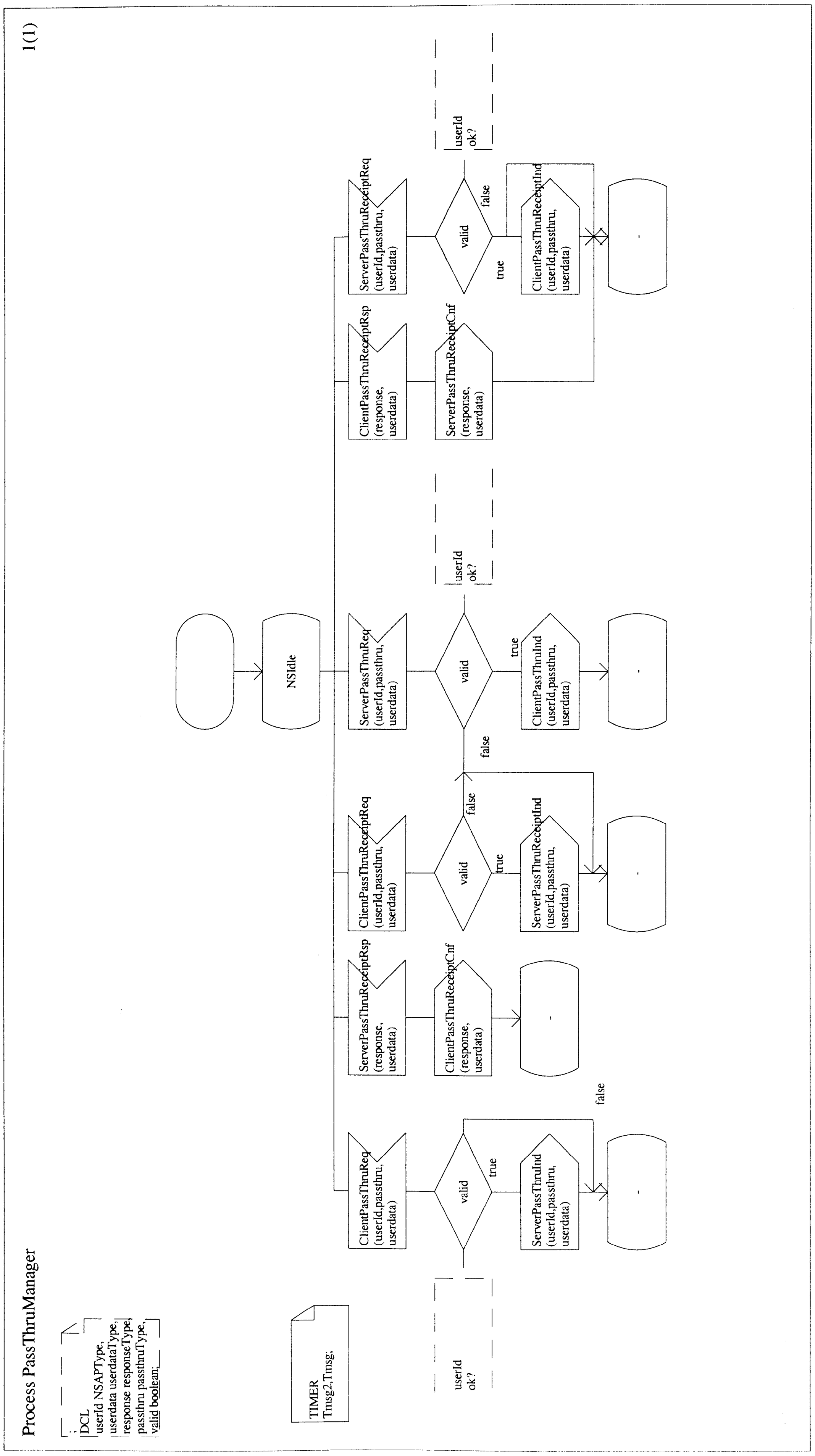
```
NEWTYPE responseType
LITERALS   rspOK,
        rspSeInvalidServer,
        rspClInvalidClient,
        rspClTimerExpired,
        rspSeTimerExpired
ENDNEWTYPE responseType
```

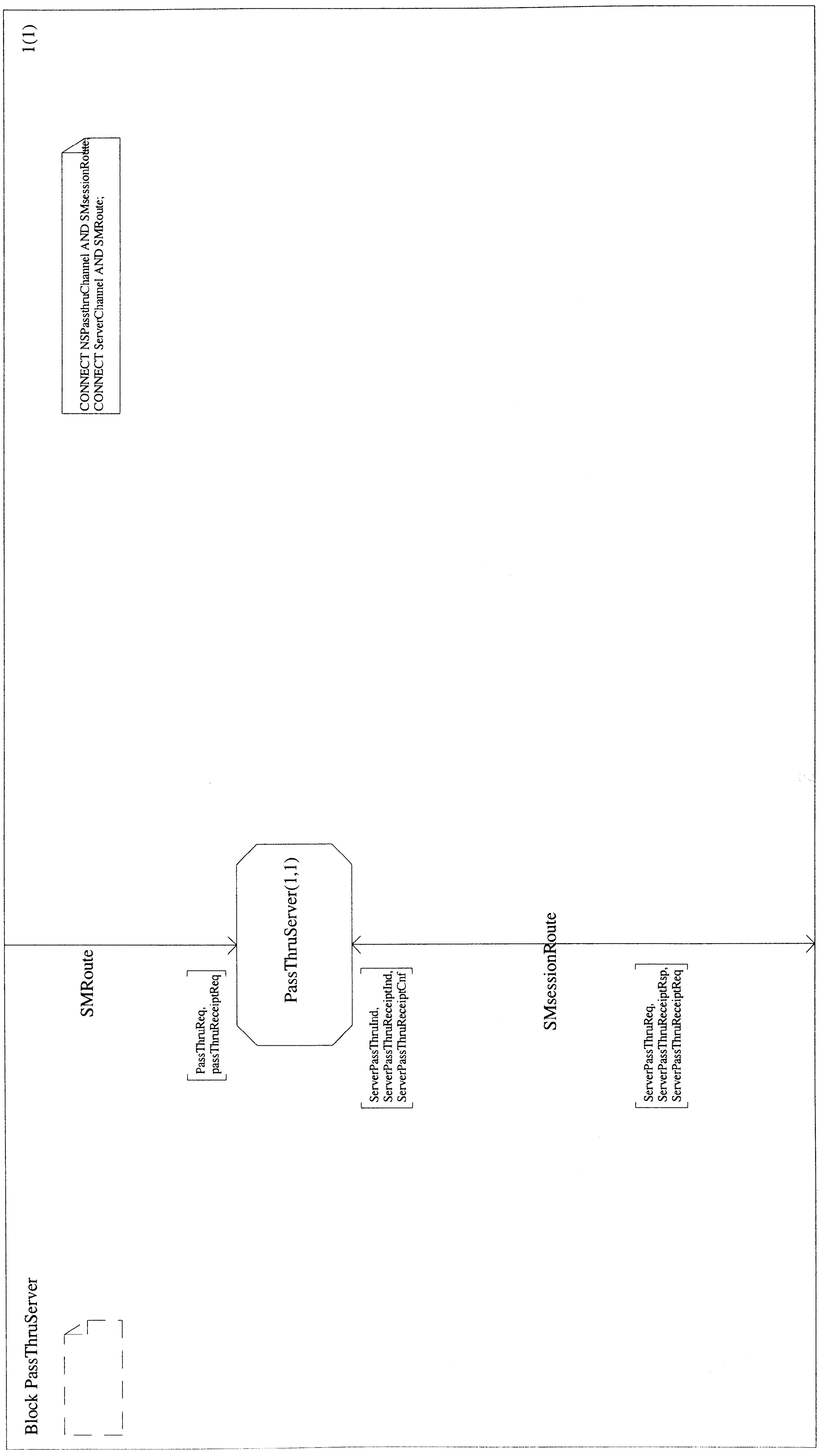
```
SIGNAL
/* pass-thru message */
ClientPassThruReq (NSAPtype,passThruType,userdataType),
ClientPassThruInd (NSAPtype,passThruType,userdataType),
ServerPassThruReq (NSAPtype,passThruType,userdataType),
ServerPassThruInd (NSAPtype,passThruType,userdataType),
ClientPassThruReceiptReq (NSAPtype,passThruType,userdataType),
ClientPassThruReceiptInd (NSAPtype,passThruType,userdataType),
ClientPassThruReceiptCnf (responseType,userdataType),
ClientPassThruReceiptRsp (responseType,userdataType),
ServerPassThruReceiptReq (NSAPtype,passThruType,userdataType),
ServerPassThruReceiptInd (NSAPtype,passThruType,userdataType),
ServerPassThruReceiptCnf (responseType,userdataType),
ServerPassThruReceiptRsp (responseType,userdataType),
/* input from env */
passThruReq (NSAPtype,passThruType,userdataType),
passThruReceiptReq (NSAPtype,passThruType,userdataType);
```

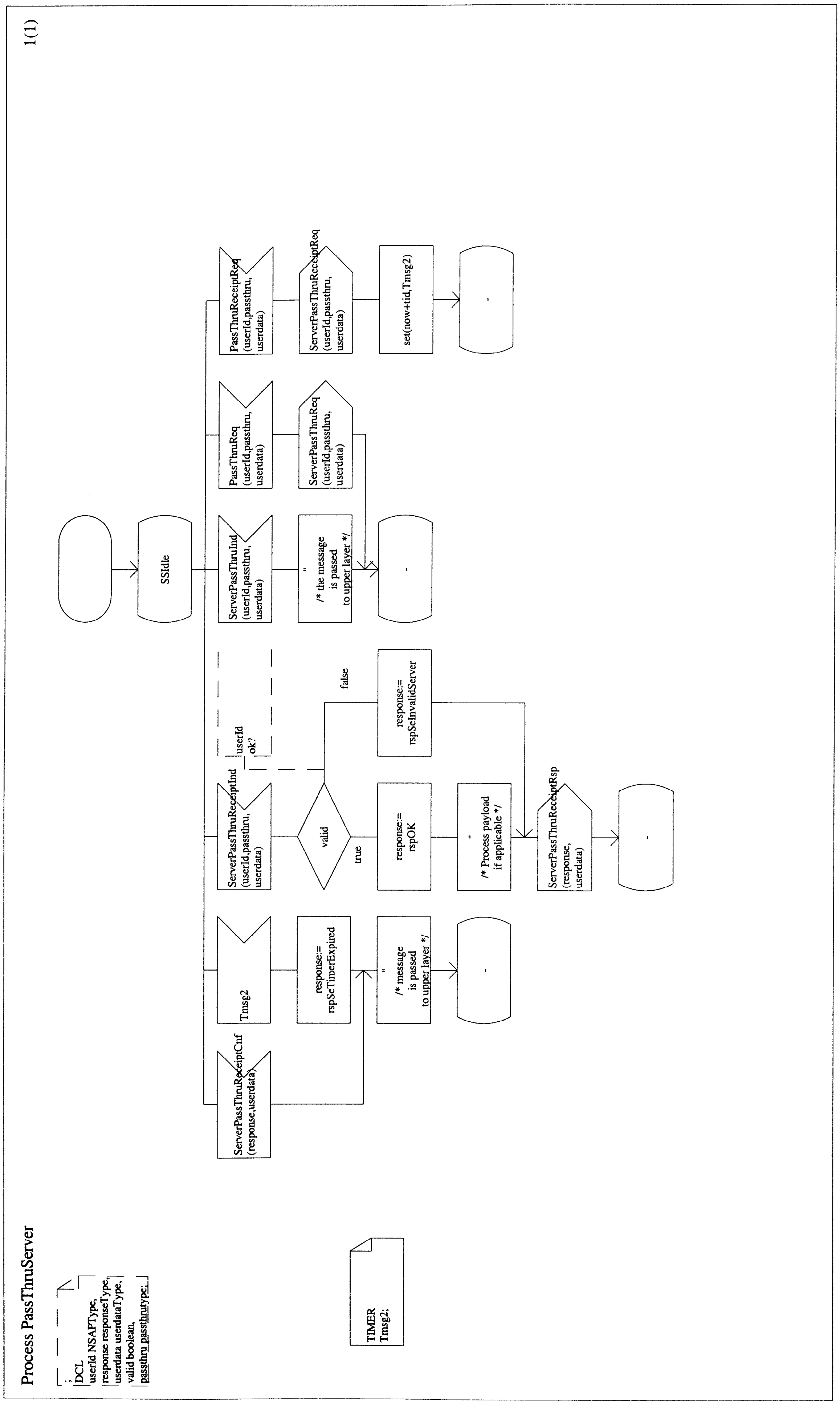

Block PassThruClient
1(1)
CONNECT CNPassthruChannel AND CNsessionmanagerroute;
CONNECT ClientChannel AND CLSMUNRoute;
CLSMUNRoute
passThruReq,
passThruReceiptReq
PassThruClient(1,1)
ClientPassThruInd,
ClientPassThruReceiptCnf,
ClientPassThruReceiptInd
CNsessionmanagerroute
ClientPassThruReq,
ClientPassThruReceiptReq,
ClientPassThruReceiptRsp


Process PassThruClient
1(1)
DCL
userId NSAPtype,
valid boolean,
userdata userdataType,
passthru passthruType,
response responseType;
TIMER
Tmsg2;
CSIdle
passThruReq
(userId,passthru,
userdata)
ClientPassThruReq
(userId,passthru,
userdata)
ClientPassThruInd
(userId,passthru,
userdata)
/* the message
is passed
to upper layer */
passThruReceiptReq
(userId,passthru,
userdata)
ClientPassThruReceiptReq
(userId,passthru,
userdata)
set(now+tid,Tmsg2)
ClientPassThruReceiptCnf
(response,userdata)
RESET(Tmsg2)
/* message
is passed
to upper layer */
-
ClientPassThruReceiptInd
(userId,passthru,
userdata)
valid
true
false
userId
ok?
response:=
rspOK
response:=
rspClInvalidClient
/* Process payload
if applicable */
ClientPassThruReceiptRsp
(response,
userdata)
-
Tmsg2
response:=
rspClTimerExpired
/* message
is passed
to upper layer */
-


Block PassThruNetwork
1(1)
CONNECT CNPassthruChannel AND Cmanroute;
CONNECT NSPassthruChannel AND Smanroute;
PassThruManager(1,1)
ServerPassThruReq,
ServerPassThruReceiptRsp,
ServerPassThruReceiptReq
Smanroute
ServerPassThruInd,
ServerPassThruReceiptInd,
ServerPassThruReceiptCnf
ClientPassThruReq,
ClientPassThruReceiptReq,
ClientPassThruReceiptRsp
Cmanroute
ClientPassThruInd,
ClientPassThruReceiptCnf,
ClientPassThruReceiptInd


Process PassThruManager
1(1)
DCL
userId NSAPType,
userdata userdataType,
response responseType,
passthru passthruType,
valid boolean;
TIMER
Tmsg2,Tmsg;
NSIdle
ClientPassThruReq (userId,passthru, userdata)
valid
true
false
ServerPassThruInd (userId,passthru, userdata)
userId ok?
ServerPassThruReceiptRsp (response, userdata)
ClientPassThruReceiptCnf (response, userdata)
ClientPassThruReceiptReq (userId,passthru, userdata)
ServerPassThruReceiptInd (userId,passthru, userdata)
ServerPassThruReq (userId,passthru, userdata)
ClientPassThruInd (userId,passthru, userdata)
ClientPassThruReceiptRsp (response, userdata)
ServerPassThruReceiptCnf (response, userdata)
ServerPassThruReceiptReq (userId,passthru, userdata)
ClientPassThruReceiptInd (userId,passthru, userdata)
-


Block PassThruServer
1(1)
CONNECT NSPassthruChannel AND SMsessionRoute;
CONNECT ServerChannel AND SMRoute;
SMRoute
PassThruReq,
passThruReceiptReq
PassThruServer(1,1)
ServerPassThruInd,
ServerPassThruReceiptInd,
ServerPassThruReceiptCnf
SMsessionRoute
ServerPassThruReq,
ServerPassThruReceiptRsp,
ServerPassThruReceiptReq


Process PassThruServer
1(1)
DCL
userId NSAPType,
response responseType,
userdata userdataType,
valid boolean,
passthru passthrutype;
TIMER
Tmsg2;
SSIdle
ServerPassThruReceiptCnf
(response,userdata)
Tmsg2
response:=
rspSeTimerExpired
/* message
is passed
to upper layer */
ServerPassThruReceiptInd
(userId,passthru,
userdata)
valid
true
false
userId
ok?
response:=
rspOK
response:=
rspSeInvalidServer
/* Process payload
if applicable */
ServerPassThruReceiptRsp
(response,
userdata)
ServerPassThruInd
(userId,passthru,
userdata)
/* the message
is passed
to upper layer */
PassThruReq
(userId,passthru,
userdata)
ServerPassThruReq
(userId,passthru,
userdata)
PassThruReceiptReq
(userId,passthru,
userdata)
ServerPassThruReceiptReq
(userId,passthru,
userdata)
set(now+tid,Tmsg2)
-
-
-
-

Annex B
(informative)
Application Examples

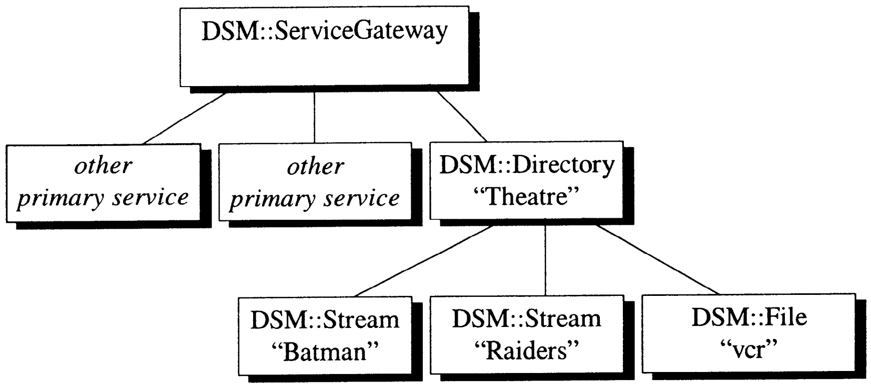
B.1 Introduction

This clause contains code examples illustrating the use of the User-to-User Interfaces. It attempts to cover various functionalities, including Video Stream Play, Building a Multimedia Directory Hierarchy, Movie Information Database, and View as a Personalized Directory.

The code examples are written in C. In order to be operational, they will need supporting header files and functions. The header files are generated by the IDL compiler for the interfaces involved. Where supporting functions are needed, function prototypes and comments are shown.

B.2 Video Stream Play

Here is test code to play a video stream. It begins after the DSM_Session_attach(). The ServiceGateway tree looks like:



Resolve and play a video stream:

```
/* function prototype to make a PathSpec of 2 NameComponents
 * and set process TRUE to return both ObjRefs
 * note: this function exists only for simplification of the example
 */
DSM_PathSpec make2StepPath(char *kind1, char *id1,
                              char *kind2, char *id2);

/* function prototype to get the nth ObjRef out of PathRefs */
DSM_ObjRef nthRef(DSM_PathRefs *refs, DSM_u_short whichone);

/* function prototype to make a simple name of one NameComponent */
CosNaming_Name makeCosSimpleName(char *kind, char *id);

/* prerequisite: resolve gatewayref through DSM_Session_attach() */

DSM_ObjRef *showHalfOfRaiders(DSM_ObjRef *gatewayref)
  {
  char *dirkind = "dir";
  char *strkind = "str";
  char *movie1 = "Batman";
  char *movie2 = "Raiders";
  char *theatre = "VoD";
  CORBA_Environment *ev;
  DSM_AppNPT = *intermission, *start;
  DSM_PathRefs = *refs;
  DSM_Scale = *scale;
  DSM_ObjRef *str, *vodsvc;
  CosNaming_Name *nextmovie;
  DSM_PathSpec *path;

  /* here alloc memory as needed for types above */
```

Play a video stream, continued:

```
  /* 60*60*60 = 21600 for one hour into the movie
   */
  start->aSeconds = 0;
  start->aMicroSeconds = 0;
  intermission->aSeconds = 216000;
  intermission->aMicroSeconds = 0;
  scale->aNumerator = 1;
  scale->aDenominator = 1;

  path = make2StepPath(dirkind, theatre, strkind, movie1);
  DSM_Directory_open(gatewayref, ev, DSM_DEPTH, path, refs);

  str = nthRef(refs, 1);
  DSM_Stream_play(str, ev, start, scale, intermission);


  /* insert wait code here
   * then change to a different movie */
  DSM_Base_close(str, ev); /* close the first movie */
  /* here insert code for error checking on ev */

  /* get your theatre object reference from the previous open*/
  vodsvc = nthRef(refs, 0);

  /* resolve and play the next movie */
  nextmovie = makeCosSimpleName(strkind, movie2);
  str = DSM_Directory_resolve(vodsvc, ev, nextmovie);
  /* here insert code for error checking on ev */

  DSM_Stream_play(str, ev, start, scale, intermission);
  /* here insert code for error checking on ev */

  printf("%s intermission will be in one hour. \n", movie2);
}
```

B.3 Building a Directory Hierarchy

This example illustrates the construction of a Directory Hierarchy with Multimedia objects. Unlike traditional file systems, each Directory can hold many different types of objects, including Files, Streams, other Directories, and objects which inherit multiple interfaces. The object's interfaces define not only the exported attributes and programming interfaces, but in the case of DSM_Streams, the method of delivery (e.g., continuous MPEG-2 Transport over the Network to the client).

Build a Multimedia Directory:

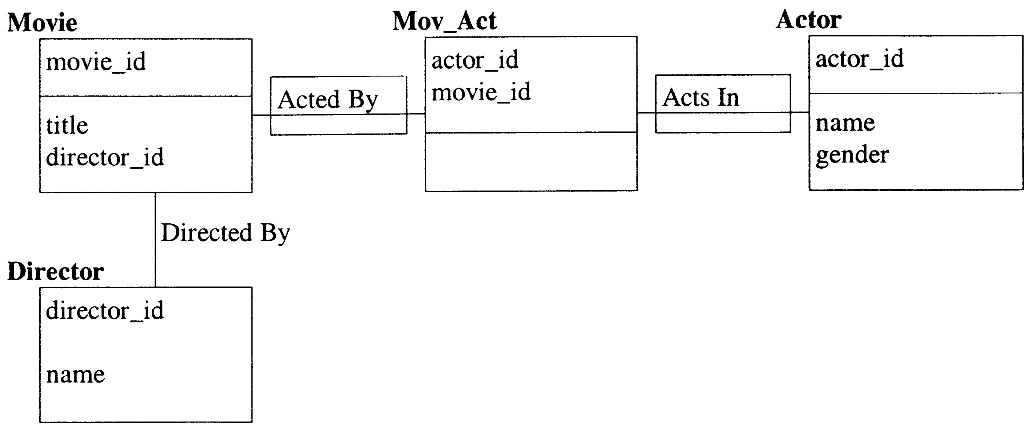
```
/* prerequisite:
 * resolve gatewayref through DSM_Session_attach()
 */

void *bindMovieDirectory(DSM_ObjRef *gatewayref,
                         DSM_Directory *vodsvc,
                         DSM_Stream *str1,
                         DSM_Stream *str2,
                         DSM_File *file1)
  {
  char *dirkind = "dir";
  char *strkind = "str";
  char *filekind = "fil";
  char *theatre = "VoD";
  char *movie1 = "Batman";
  char *movie2 = "Raiders";
  char *button1 = "vcr";
  CORBA_Environment *ev;
  CosNaming_Name *theatrename;
  CosNaming_Name *movie1name;
  CosNaming_Name *movie2name;
  CosNaming_Name *buttonname;

  /* here alloc memory as needed for types above */

  theatrename = makeCosSimpleName(dirkind, theatre);
  movie1name = makeCosSimpleName(strkind, movie1);
  movie1name = makeCosSimpleName(strkind, movie2);
  buttonname = makeCosSimpleName(filekind, button1);

  DSM_Directory_bind_context(gatewayref, ev,
                             vodsvc, theatrename);
  /* here insert code for error checking on ev */

  DSM_Directory_bind(vodsvc, ev, str1, movie1name);
  /* here insert code for error checking on ev */

  DSM_Directory_bind(vodsvc, ev, str2, movie2name);
  /* here insert code for error checking on ev */

  DSM_Directory_bind(vodsvc, ev, file1, button1name);
  /* here insert code for error checking on ev */
}
```

B.4 Movie Information Database

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 the Service, View and Directory Interfaces in its IDL definition:
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 **Security_authenticate** followed by **DSM_Session_attach()**. This causes the **Security_authenticate()** request body to be placed in the ServiceContextList of the **DSM_Session_attach().**
6. The OWNER opens the database service, then creates and populates the database tables with movie information:



Create a Movie Database (Note ev error checking not shown):

```
/* prereq
 * create a View Interoperable Object Ref using DSM_Lifecycle_create
 * resolve a ServiceGateway Ref using DSM_Session_attach
 */
void createMovieDatabase(DSM_ObjRef *gatewayref, IOR_IOR *viewref)
{
  char *viewnamekind = "viw";
  char *viewnameid = "MovieInfo";
  CosNaming_Name *viewname;
  char *statement;
  CORBA_Environment *ev;

  /* here alloc memory as needed for types above */

  /* Bind the Movie View to the Service Gateway */
  viewname = makeSimpleCosName(viewnamekind, viewnameid);
  DSM_Directory_bind(gatewayref, ev, viewname, viewref);

  /* execute SQL statements to create and populate a movie table */
  statement =
  "CREATE TABLE movie(movie_id char(5), title char(30), director_id
   char(5))";
  DSM_View_exec(viewref,ev,statement);

  statement =
    "INSERT INTO movie VALUES ('M3', 'Raiders of the Lost Ark', 'D2')";
  DSM_View_exec(viewref,ev,statement);
  statement = "INSERT INTO movie VALUES ('M2', 'BeetleJuice', 'D1')";
  DSM_View_exec(viewref,ev,statement);
  statement = "INSERT INTO movie VALUES ('M1', 'Batman', 'D1')";
  DSM_View_exec(viewref,ev,statement);
```

Execute SQL statements to create and populate an actor table:

```
statement =
"CREATE TABLE actor(actor_id char(5), name char(30), gender char(6))";
DSM_View_exec(viewref,ev,statement);

statement =
"INSERT INTO actor VALUES ('A7', 'Harrison Ford', 'male')";
DSM_View_exec(viewref,ev,statement);
statement =
"INSERT INTO actor VALUES ('A6', 'Kim Bassinger', 'female')";
DSM_View_exec(viewref,ev,statement);
statement =
"INSERT INTO actor VALUES ('A5', 'Michael Keaton', 'male')";
DSM_View_exec(viewref,ev,statement);
statement =
"INSERT INTO actor VALUES ('A4', 'Alec Baldwin', 'male')";
DSM_View_exec(viewref,ev,statement);
statement =
"INSERT INTO actor VALUES ('A3', 'Geena Davis', 'female')";
DSM_View_exec(viewref,ev,statement);
statement =
"INSERT INTO actor VALUES ('A2', 'Karen Allen', 'female')";
DSM_View_exec(viewref,ev,statement);
statement =
"INSERT INTO actor VALUES ('A1', 'Jack Nicholson', 'male')";
DSM_View_exec(viewref,ev,statement);
```

Execute SQL statements to create and populate a director table:

```
statement =
 "CREATE TABLE director(director_id char(5), name char(20))";
DSM_View_exec(viewref,ev,statement);

statement = "INSERT INTO director VALUES ('D2', 'Steven Spielberg')";
DSM_View_exec(viewref,ev,statement);
statement ="INSERT INTO director VALUES ('D1', 'Tim Burton')";
DSM_View_exec(viewref,ev,statement);
```

Execute SQL statements to create and populate a mov_act table:

```
  statement = "CREATE TABLE mov_act(movie_id(5), actor_id char(5))";
  DSM_View_exec(viewref,ev,statement);

  statement = "INSERT INTO mov_act VALUES ('M2', 'A4')";
  DSM_View_exec(viewref,ev,statement);
  statement = "INSERT INTO mov_act VALUES ('M2', 'A3')";
  DSM_View_exec(viewref,ev,statement);
  statement = "INSERT INTO mov_act VALUES ('M3', 'A7')";
  DSM_View_exec(viewref,ev,statement);
  statement = "INSERT INTO mov_act VALUES ('M3', 'A2')";
  DSM_View_exec(viewref,ev,statement);
  statement = "INSERT INTO mov_act VALUES ('M1', 'A5')";
  DSM_View_exec(viewref,ev,statement);
  statement = "INSERT INTO mov_act VALUES ('M1', 'A1')";
  DSM_View_exec(viewref,ev,statement);
  statement = "INSERT INTO mov_act VALUES ('M1', 'A6')";
  DSM_View_exec(viewref,ev,statement);
}
```

READER Procedure:

The READER in this example is the client (Set top) application. The READER follows a similar procedure (to Owner Procedure) for opening the View Service. A client with READER privileges only shall not perform write operations such as **DSM_View_exec()**, but may perform read operations such as **DSM_View_query()** and **DSM_View_read()**.

Retrieve the name of the director of the movie Batman:

```
/* prereq: resolve the view with Directory_resolve */

DSM_ObjRef *viewref, *newviewref;
CORBA_Environment *ev;
DSM_u_long numrows = 40;
char *statement;
DSM_View_ResultDescribe *describe;
DSM_View_Result *result;
DSM_u_short cursor;

/* here alloc memory as needed for types above */

statement =
"SELECT name FROM director d, movie m WHERE d.director_id =
 m.director_id AND m.title = 'Batman'";

DSM_View_query(viewref, ev, statement, numrows, describe, result, newviewref);

/* describe will now contain a sequence of one FieldDescribe, indicating one
 * result column.
 * The FieldDescribe will have these member values:
 *   fieldName = "name"
 *   aType = VTC__CHAR
 *   typeParameters = struct InfoChar
 *        e.g., InfoChar has length = 15 and nullTerminated = TRUE
 */
cursor = 0;

DSM_View_read(newviewref, ev, cursor, numrows, result);

/* result will now point to a char that holds
 * the string name of the director
 */
```

Retrieve the list of actors in Beetlejuice:

```
statement =
"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'";
```

Retrieve the list of female actors in Beetlejuice:

```
statement =
"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 the actors from the View.

B.5 View as a Personalized Directory

The View interface can be included with a Directory interface to produce a Directory that can sort and filter. This is useful for personalized Directories, or to front-end file systems that have sort and filter capabilities. In this case the View is not a database, but rather its syntax is used as a simplified method for producing different 'views' of the Directory binding list. With View, relational queries can be performed using the exported attributes of the objects in a Directory.

Consider a Service "VIP3::Index" and "VIP3::Game" defined by the following IDL:

```
module VIP3{
    interface Index : DSM::Directory, DSM::View, {};
    interface Game : DSM::File {
        unsigned short attribute Age;
    };
};
```

The Index is a Directory with View interface. It is not a Database. The Game is a File with an attribute that identifies the recommended player's age. The Game File can be downloaded to the client using **DSM_File_read().** But there will be so many games, it is desirable to filter them when listing the names of the games present in the Directory.

Bind and access a Directory with View:

```
/* prereq
 * create an Index Interoperable Object Ref with DSM_Lifecycle_create
 * resolve a ServiceGateway Ref using DSM_Session_attach
 */
void testDirectoryView(DSM_ObjRef *gatewayref, IOR_IOR *indexref)
{
  char *indexnamekind = "VIP3::Index";
  char *indexnameid = "Games";
  Cos_NameComponent *indexname;

  DSM_ObjRef *newidxref;
  CORBA_Environment *ev;
  DSM_u_long numrows = 40;
  char *statement;
  DSM_View_ResultDescribe *describe;
  DSM_View_Result *result;
  DSM_u_short cursor;

  /* here alloc memory as needed for types above */

  /* Bind the Games Index to the Service Gateway */
  indexname = makeSimpleCosName(indexnamekind, indexnameid);
  DSM_Directory_bind(gatewayref, ev, indexname, indexref);
  /* here insert code for error checking on ev */

  /* here insert code to populate the Directory with many game files
   * using DSM_Directory_bind
   */

  /* Produce a Directory of games for children 5 years old */
  statement =
    "SELECT Game FROM * WHERE Age = 5";

  DSM_View_query(indexref, ev, statement, numrows, describe, result, newidxref);
  /* here insert code for error checking on ev */

  /* list the contents of the new Directory View, which will
   * produce a bindings list of only those games for Age 5
   */
  DSM_Directory_list(newidxref, ev, count, bindings);
  /* here insert code for error checking on ev */

  /* here print the bindings for those Games with Age = 5
   * which are the results of the Directory_list
   */
}
```

Annex C
(informative)
ONC RPC XDR Mappings

C.1 Overview

The DSM-CC User-to-User protocol offers a choice of RPC and network stack. The preferred RPC is Universal Networked Objects (UNO). Some of the more popular alternate industry standard RPCs are Distributed Computing Environment (DCE) and Open Network Computing (ONC). DCE has defined a protocol within the UNO Interoperable Object Reference, called DCE-CIOP. DCE is defining its own On-the-Wire formats that are UNO compatible. UNO offers dynamic interoperability, using a generic client stub that can dynamically assemble request and reply message body from individual arguments. This is called dynamic marshaling. ONC requires pre-compiled request and reply message body structures (although parameters can be variable length or opaque). ONC compilers are universally available and public domain, and the ONC specification has been stable for many years. All of the DSM-CC core and extended operation message bodies can be pre-compiled as ONC structures.

This informative annex describes the DSM-CC On-the-Wire formats for both UNO and ONC. ONC request and reply message formats are defined in such a way as to be compatible with UNO. In particular, RPC Request and Reply Headers are defined for ONC which carry **service_context**, **object_key**, **requesting_principal**, and **reply_status**. A Tagged Protocol Profile is defined for ONC that allows ONC program and version information to be carried in the Interoperable Object Reference (IOR).

C.2 General RPC Message Formats

Please refer to Informative Annex E for the UNO RPC message definitions.

Please refer to Internet RFC 1057 and 1014 for ONC RPC and XDR definitions, respectively.

The UNO RPC request message has a three-part format:

struct RequestMessage		

	struct GIOP::MessageHeader	char magic[4];
		Version GIOP_version;
		boolean byte_order; /* CDR boolean */
		octet message_type;
		unsigned long message_size;

	struct GIOP::RequestHeader	IOP::ServiceContextList service_context;
		unsigned_long request_id;
		boolean response_expected;
		CORBA_sequenceOctet object_key;
		string operation;
		Principal requesting_principal:

	Request Body	*<in and inout parameters>*

The UNO RPC reply message has a three-part format:

struct ReplyMessage	

struct GIOP::MessageHeader	char magic[4];
	Version GIOP_version;
	boolean byte_order; /* CDR boolean */
	octeet message_type;
	unsigned long message_size;

struct GIOP::ReplyHeader	IOP::ServiceContextList service_context;
	unsigned_long request_id;
	replyStatusType reply_status;

Reply Body	*<out and inout parameters>*

The DSM ONC RPC request message has the format:

struct rpc_msg			unsigned int xid;

	struct call_body		unsigned int rpcvers;
			unsigned int prog;
			unsigned int vers;
			unsigned int proc;
			opaque_auth cred;
			opaque_auth verf;

	<struct opRequest>	struct DSM_XDR_ReqHeader	IOP_ServiceContextList *service_context;
			opaque *object_key;
			opaque *requesting_principal;

			<operation ins and inout parameters >;

The DSM-CC ONC RPC successful reply has the format:

struct rpc_msg			unsigned int xid;
	struct reply_body		opaque_auth verf;

	<struct opReply>	struct DSM_XDR_ReplyHeader	IOP_ServiceContextList *service_context;
			ReplyStatusType *reply_status;
			DSM_XDR_Exception *exception;

			<operation out and inout parameters>

Note: an unsuccessful reply has a different format. Please refer to Internet RFC 1057.

The pre-compiled rpc stub will insert rpcvers for the ONC version and proc for the operation id.

The client DSM-CC Library has to supply prog, vers, <struct opRequest>, and <struct OpReply> for the ONC rpc call. <struct opRequest> and <struct opReply> are structure definitions in .h file, generated by ONC rpcgen from an External Data Representation (XDR) .x file.

The ONC prog and vers are obtained from the object reference with the DSM ONC Tagged Protocol Profile, defined below.

Key parameters needed for DSM-CC / ONC operation:

1. In the request message, **service_context**, **object_key** and **requesting_principal**. These are place in the DSM_XDR_ReqHeader.
2. In the reply message, service_context, **reply_status** and **exception**. These are place in the DSM_XDR_ReplyHeader.

Authentication structures are carried in the ServiceContextList of request or reply, and not in the ONC header. Therefore, the ONC request cred, verf and reply verf are all set to AUTH_NULL.

With the above structure a DSM-CC library can have a common application layer API, common structure and routines for handling authentication, download, network resource bindings, and exception environment. The structures can then be readily sent/received in either UNO or ONC messages.

C.3 CORBA IDL C to XDR Mapping

This subclause defines mappings between CORBA IDL C Mappings and XDR input format (.X file format).

C.3.1 Mapping for Integer Data Types

```
/*  machine-independent types  */

#if SIZEOF_LONG == 64
typedef int CORBA_long;                          /* 32 bits */
typedef unsigned int CORBA_unsigned_long;        /* unsigned 32 bits */
typedef unsigned int CORBA_Status;
typedef unsigned int CORBA_Flags;
typedef long CORBA_longlong;
typedef unsigned long CORBA_unsigned_longlong;
#else
typedef long CORBA_long;
typedef unsigned long CORBA_unsigned_long;
typedef unsigned long CORBA_Status;
typedef unsigned long CORBA_Flags;
struct CORBA_longlong {long v0; unsigned long v1;};
struct CORBA_unsigned_longlong {unsigned long v0; unsigned long v1;};
#endif
```

IDL C Mapping **long int** must be represented by XDR **long**.

C.3.2 Mapping for void

IDL C mapping type **void** is represent by type **char** in XDR, except in unions.

C.3.3 Mapping for Constants

Constants are represented by %#define statement with encapsulating ifdef RPC_HDR:

```
#ifdef RPC_HDR
%#define <rest of statement>
%#define <rest of statement>
                    ...
#endif
```

For example, DSM::AccessRole types are defined as follows in XDR:

```
typedef CORBA_char DSM_AccessRole;
#ifdef RPC_HDR
%#define DSM_MANAGER 'M'
%#define DSM_OWNER 'O'
%#define DSM_BROKER 'B'
%#define DSM_WRITER 'W'
%#define DSM_READER 'R'
#endif
```

C.3.4 Mapping for octet

CORBA::octet is represented by **unsigned char**.

C.3.5 Mapping for Fixed-length Constructed Types

Fixed length constructed types struct and enum are identical to C struct and enum.

C.3.5.1 Mapping for struct

typedef struct in CORBA IDL or C mapping must be represented by **struct** in XDR. For example, the correct XDR form for DSM_version is:

```
struct DSM_Version {
    CORBA_char aMajor;
    CORBA_char aMinor;
    };
```

C.3.6 Mapping for sequences

Variable-length structs require special handling to enable proper encoding and decoding. The XDR array angle bracket <> syntax must not be used. Using XDR array <> syntax will result in non-Inter-operable code.

The XDR mapping for sequences is similar to the CORBA C mapping for sequences, with the following exceptions:

- An enclosing **#ifndef RPC_XDR ... #endif** is used to instruct rpcgen not to produce a function for the type during the XDR phase.
- struct is used instead of typedef struct, as described above under 'Mapping for struct'
- _maximum and _length members shall be either XDR type u_int or CORBA_unsigned_long.

In addition, an XDR function shall be explicitly defined for the type. This function will cause XDR to directly encode from the CORBA C mapping and decode to the CORBA C mapping.

C.3.6.1 Example: Mapping for opaque

The XDR DSM_opaque type is equivalent to DSM::opaque and CORBA::sequence<octet>. An enclosing #ifndef RPC_XDR is used to instruct rpcgen not to produce a function for the type during the XDR phase. Instead, an XDR function is separately defined for the type, causing XDR to encode from the CORBA C mapping and decode to the CORBA C mapping.

For example, DSM::opaque is defined in IDL as:

```
module DSM{
    typedef sequence<octet> opaque;
};
```

DSM::opaque CORBA C mapping is:

```
typedef struct {
      CORBA_unsigned_long _maximum;
      CORBA_unsigned_long _length;
      CORBA_octet  * _buffer;
      } CORBA_sequence_octet;
typedef CORBA_sequence_octet DSM_opaque;
```

DSM::opaque mapping in XDR is:

```
#ifndef RPC_XDR

struct DSM_opaque{
      u_int  _maximum;
      u_int  _length;
      CORBA_octet  * _buffer;
};

#endif
```

DSM_opaque XDR encode/decode function is:

```
bool_t xdr_DSM_opaque(xdrs, objp)
    XDR *xdrs;
    DSM_opaque *objp;
{
    if (!xdr_int(xdrs, (u_int *)&objp->_maximum)) {
                return (FALSE);
        }
    if (!xdr_u_int(xdrs, &objp->_length)) {
                return (FALSE);
        }
    if (!xdr_array(xdrs, (char **)&objp->_buffer,
        (u_int *)&objp->_length, ~0, sizeof(CORBA_octet), xdr_CORBA_octet)) {
                return (FALSE);
        }
        return (TRUE);
}
```

C.3.6.2 Example: Mapping for PathSpec

For example, DSM::PathSpec is defined in IDL as:

```
module DSM {
    typedef sequence<Step> PathSpec;
};
```

DSM::PathSpec mapping in XDR is:

```
#ifndef RPC_XDR

struct DSM_PathSpec {
    u_int _maximum;
    u_int  _length;
    DSM_Step  * _buffer;
};

#endif
```

DSM_PathSpec XDR encode/decode function is:

```
bool_t xdr_DSM_PathSpec(xdrs, objp)
    XDR *xdrs;
    DSM_PathSpec *objp;
{
    if (!xdr_int(xdrs, (u_int *)&objp->_maximum)) {
                return (FALSE);
        }
    if (!xdr_u_int(xdrs, &objp->_length)) {
                return (FALSE);
        }
    if (!xdr_array(xdrs, (char **)&objp->_buffer,
        (u_int *)&objp->_length, ~0, sizeof(DSM_Step), xdr_DSM_Step)) {
                return (FALSE);
        }
        return (TRUE);
}
```

C.3.7 Mapping for string

The XDR syntax is to place angle brackets after the identifier. Maximums within the angle brackets are carried forward.

For example, the following IDL illustrates two strings:

```
typedef string password;
typedef string<10> id;
```

The corresponding XDR mapping is:

```
string password<>;
string id<10>;
```

C.4 DSM-CC ONC Protocol Profile for the Interoperable Object Reference

The DSM-CC ONC Protocol Profile is composed of at least the following tagged components:

- DSM::AddrComponent structure. This contains the IP address and port of the Object Implementation.
- DSM::ObjectKey. This identifies the unique client/service instance within the context of the IP address and port.
- DSM_ONC::IntfComponent. This identifies the program and version for the most derived interface supported.

Additionally it may contain:

- DSM::IntfCode. This identifies the interfaces supported by the object.
- DSM_ONC::IntfComponent. Additional IntfComponents can be included to provide program and version for inherited interfaces.

The Object Reference follows the IDL/XDR mapping rules described above, and is big-endian.

C.5 Exceptions

DSM-CC Exceptions are defined as a union to be carried in the reply header. XDR form:

```
enum DSM_XDR_ExceptionKind {
        EXSYS = 0,
        EXSTRING = 1,
        EXAUTH = 2,
        EXNOTFOUND = 3,
        EXCANNOTPROCEED = 4,
        EXVOID = 5
};

union DSM_XDR_ExceptionValue switch (DSM_XDR_ExceptionKind kind){
        case EXSYS: CORBA_exception_body ex_sys;
        case EXSTRING: string ex_string<>;
        case EXAUTH: DSM_NO_AUTH ex_auth;
        case EXNOTFOUND: DSM_NOT_FOUND ex_notFound;
        case EXCANNOTPROCEED: DSM_CANNOT_PROCEED ex_cannotProceed;
        case EXVOID: void;
};

struct DSM_XDR_Exception {
  string id<>;
  DSM_XDR_ExceptionValue value;
};
```

If there is no exception value, use the EXVOID case. The resulting DSM_XDR_Exception can be assigned to/from CORBA_Exception.

C.6 Request and Reply Header Structures

For each RPC, DSM-CC request and reply headers are pre-pended to the message body. XDR Form:

```
/* DSM Request Header. CORBA Parameters not found in ONC header
 * This is placed in the RPC request structure of DSM-CC ONC messages
 */

struct DSM_XDR_ReqHeader {
 IIOP_ServiceContextList service_context;
 DSM_opaque object_key;          /* handle of the target object within */
                                 /* host and port scope*/
 DSM_Principal requesting_principal; /* end user(human) is a sequence of octet*/
};

/* DSM_XDR_ReplyHeader. CORBA Parameters not found in ONC header
 * This is placed in the RPC reply structure of DSM-CC ONC messages
 */

struct DSM_XDR_ReplyHeader {
 IIOP_ServiceContextList service_context;
 CORBA_completion_status reply_status;
 DSM_XDR_Exception exception;
};
```

IDL **in** or **inout** parameters are considered as **in** parameters. IDL **out** or **inout** parameters are considered as **out** parameters.

An operation with no **in** parameters shall send the **DSM_XDR_ReqHeader** as the request structure.

An operation with one or more **in** parameters shall send the **DSM_XDR_ReqHeader** as the first member of the request structure. The remaining members of the request structure shall be the **in** parameters, passed as a pointer if they are constructed types(i.e., cannot be passed as an argument to a function), and passed as a value if they are basic or enumerated types.

An operation with no **out** parameters shall send the **DSM_XDR_ReplyHeader** as the reply structure.

An operation with one or more **out** parameters shall send the **DSM_XDR_ReplyHeader** as the first member of the reply structure. The remaining members of the request structure shall be the **out** parameters, passed as a pointer if they are constructed types, and passed as a value if they are basic or enumerated types.

For example, the XDR for File read() and File write() is

```
struct DSM_XDR_File_readReq {
    DSM_XDR_ReqHeader reqHeader;
    DSM_u_longlong  *aOffset;
    DSM_u_long  aSize;
    CORBA_boolean aReliable;
};

struct DSM_XDR_File_readReply {
    DSM_XDR_ReplyHeader replyHeader;
    DSM_opaque *rData;
}
```

C.7 DSM-CC RPC Program Numbers

The following DSM-CC program numbers are registered with ONC:

ISO/IEC JTC1/SC29/WG11: 399900 - 399919 [20 numbers] DSM

399900 DSM::Base
399901 DSM::Access
399902 CosNaming::NamingConect
399903 CosNaming::BindingIterator
399904 DSM::Directory
399905 DSM::Stream
399906 DSM::File
399907 DSM::ServiceGatewaySI
399908 DSM::SessionGateway
399909 DSM::Service
399910 DSM::Interfaces
399911 DSM::Session
399912 DSM::State
399913 DSM::View
399914 DSM::Composite
399915 DSM::DownloadSI
399916 DSM::Event
399917 - 39919 <reserved>

C.7.1 RPC Program Dispatch Tables Mapping

The DSM-CC ONC functions are named according to the IDL C mapping conventions. The functions are numbered in the program dispatch table as follows:

1. First: the operations in the order they appear in the IDL.
2. Second: the attribute functions, get followed by set for each, in the order they appear in the IDL. A readonly attribute shall generate only a get operation.

For example, The XDR for the Base interface is

```
#ifdef RPC_HDR
%#ifndef DSMBASE_P
%#define DSMBASE_P
#endif

#define DSM_BASE_PROGRAM 399900
#define DSM_BASE_VERSION 1


program DSM_BASE {
    version DSM_BASE_VERS {
        DSM_XDR_ReplyHeader DSM_Base_close(DSM_XDR_ReqHeader) = 1;
        DSM_XDR_ReplyHeader DSM_Base_destroy(DSM_XDR_ReqHeader) = 2;
    } = DSM_BASE_VERSION;
} = DSM_BASE_PROGRAM;

#ifdef RPC_HDR
%#endif
#endif
```

As another example, the XDR for the File interface is

```
#ifdef RPC_HDR
%#ifndef DSMFILE_P
%#define DSMFILE_P
#endif

#define DSM_FILE_PROGRAM 399906
#define DSM_FILE_VERSION 1

struct DSM_XDR_File_readReq {
  DSM_XDR_ReqHeader reqHeader;
  DSM_u_longlong  *aOffset;
  DSM_u_long  aSize;
  CORBA_boolean aReliable;
};

struct DSM_XDR_File_readReply {
  DSM_XDR_ReplyHeader replyHeader;
  DSM_opaque *rData;
};

struct DSM_XDR_File_writeReq {
  DSM_XDR_ReqHeader reqHeader;
  DSM_u_longlong  *aOffset;
  DSM_u_long  aSize;
  DSM_opaque *rData;
};

struct DSM_XDR_File__get_ContentReply {
  DSM_XDR_ReplyHeader replyHeader;
  DSM_opaque *Content;
};

struct DSM_XDR_File__set_ContentReq {
 DSM_XDR_ReqHeader reqHeader;
 DSM_opaque *Content;
};
struct DSM_XDR_File__get_ContentSizeReply {
 DSM_XDR_ReplyHeader replyHeader;
 DSM_u_longlong *contentSize;
};

program DSM_FILE{
 version DSM_FILE_VERS {
  DSM_XDR_File_readReply
       DSM_File_read(DSM_XDR_File_readReq) = 1;
  DSM_XDR_ReplyHeader
       DSM_File_write(DSM_XDR_File_writeReq) = 2;
  DSM_XDR_File__get_ContentReply
       DSM_File__get_Content(DSM_XDR_ReqHeader) = 3;
  DSM_XDR_ReplyHeader
       DSM_File__set_Content(DSM_XDR_File__set_ContentReq) = 4;
  DSM_XDR_File__get_ContentSizeReply
       DSM_File__get_ContentSize(DSM_XDR_ReqHeader) = 5;
  } = DSM_FILE_VERSION;
} = DSM_FILE_PROGRAM;

#ifdef RPC_HDR
%#endif
#endif
```

Annex D
(informative)
Using DSM-CC U-N Session Messages with ATM

D.1 Methods of using DSM-CC over ATM

This annex presents several example methods of using DSM-CC with ATM, each of which conforms to the DSM-CC functional model.

From the basic principle of ATM SVC connectivity under the context of a multimedia service session, four methods that tie the Call Reference (which represents the Q.2931 call concept) and the sessionId + resourceNum (which represents the DSM-CC session concept) are discussed. See also Table D-1 Methods of using DSM-CC over ATM.

1. Session Method
2. Network Method with AddResource messages between the Server and the SRM
3. Network Method with NO AddResource messages between the Server and the SRM
4. Integrated Method

Physical scenarios using different Access Networks that correspond to each method, are listed in Table D-1. These different Access Networks that can part of physical scenarios for DSM-CC over ATM are described in DAVIC 1.3 Part 4 [7].

The term Server used in this annex refers to a complex of servers. A server may or may not have a signaling unit front-end.

Note that Q.2931 is used generically in this annex to indicate both ATM UNI and ITU signaling. No difference is made between the two.

D.1.1 Session Method

The Session Method requires that any Server that needs an ATM SVC connectivity must first inform the SRM (using the ServerAddResourceRequest message). The SRM then initiates a network level signaling procedure (typically Q.2931 in an ATM network) to establish an ATM SVC connection between the Server and a Client. The network level signaling message initiated by the SRM to the ATM network includes the resourceId (sessionId+resourceNum) to allow the association of the DSM-CC resources with ATM connections.

D.1.2 Network Method with AddResource messages between the Server and the SRM

The Network Method with AddResource messages requires that resources be allocated by the Server prior to sending the AddResource message. The AddResource messages are thus used to inform the SRM providing it with the opportunity to request the release of the connection. The Q.2931 signaling initiated by the Server includes the resourceId (sessionId+resourceNum). In the case of an IWU, the AddResource messages can be used to inform the IWU of MPEG characteristics (PID, Pmt) necessary to deliver the MPEG program stream to the appropriate Client.

D.1.3 Network Method with NO AddResource messages between the Server and the SRM

The Network Method allows any Server that needs an ATM SVC connectivity to initiate network level signaling (typically Q.2931) to the ATM network. The Q.2931 SETUP message that the Server sends to the ATM UNI, or the CONNECT message from the SRM, is required to contain a resourceId (sessionId+resourceNum) and the resourceGroupTag. The resourceGroupTag is required in the signaling messages with this method as it is the only way to pass it when AddResource messages are not used. The SRM is in the signaling path to allow it to do resource management within a session. Any additional non-ATM resources must be signaled in DSM-CC messages.

D.1.4 Integrated Method

With the Integrated Method, the DSM-CC messages are mapped into Q.2931 signaling messages. The SRM "sees" the ATM connection resources via Q.2931 signaling and thus manages both sessions and resources. Any additional non-ATM resources must be signaled in DSM-CC messages.

Table D-1 Methods of using DSM-CC over ATM

METHOD OF USING DSM-CC WITH ATM	PHYSICAL SCENARIOS
1) SESSION METHOD SRM initiates/releases connection based on exchange of AddResource messages	• Server Direct: a) Proxy Signaling for Client and SRM in Access Network b) Non-ATM Access Network to Core ATM Network Inter-Working Unit (IWU) and SRM in Access Network c) Network Signaling by SRM with "join" capabilities within the network (potential future) • Server Proxy with scenarios 1a, 1b, 1c above (note that signaling flows remain the same if a Server Proxy exists)
2) NETWORK METHOD with AddResource messages between the Server and the SRM Server initiates/releases connection and notifies SRM of connection using AddResource message	• Server Direct: a) Client Direct (Signaling in Client, SRM in Core Network) (= **DAVIC scenario #2** [7]) b) Proxy Signaling Agent (PSA) for Client in Access Network, SRM in Core Network (= **DAVIC scenario #3** [7]) c) Signaling for Client in Access Network, Non-ATM Access Network to Core ATM Network IWU, and SRM in Core Network (= **DAVIC scenario #3** [7]) d) Signaling for Client in Access Network, Non-ATM Access Network to Core ATM Network IWU, and SRM in Access Network (= **DAVIC scenario #1** [7]) • Server Proxy with 2a, 2b, 2c, 2d above (note that signaling flows remain the same if a Server Proxy exists)
3) NETWORK METHOD with NO AddResource messages between Server and SRM Server initiates/releases connection, SRM learns of ATM connection resources via Q.2931 signaling since it is in the signaling path (no AddResource messages required between Server and SRM), Client learns of connection use at the U-U level	• Server Direct: a) Proxy Signaling Agent (PSA) for Client in Access Network, SRM in Access Network (= **DAVIC scenario #1** [7]) b) Signaling for Client in Access Network, Non-ATM Access Network to Core ATM Network IWU, and SRM in Access Network (B-HLI field used to hold any resource information required by the IWU (see H.310 [8])) (= **DAVIC scenario #1** [7]) c) Network Signaling by SRM with "join" capabilities within the network (Core and Access Networks are ATM) (potential future)

	• Server Proxy with 3a, 3b, 3c above (note that signaling flows remain the same if a Server Proxy exists)
4) INTEGRATED METHOD Client or Server initiates/releases connection, SRM "sees" ATM connection resources via Q.2931 signaling and thus manages both sessions and resources	a) Client Direct and Server Direct, SRM anywhere in the Network (potential future)

D.2 Association of DSM-CC connection resources to ATM SVCs

D.2.1 DSM-CC resourceId Mapping into Q.2931

To allow DSM-CC Users to associate each ATM SVC connection with a DSM-CC resource and session, a resourceId needs to be carried along the ATM SVC connection signaling. The current view is that the Generic Identifier Transport (GIT) Information Element in Q.2931 [1] is the proper field to carry the resourceId. For this purpose a specific GIT type for MPEG-2 DSM-CC is assigned in Q.2931.

Table D-2 defines Generic Identifier fields for DSM-CC.

Table D-2 Generic Identifier fields defined for DSM-CC

Generic Identifier Type	DSM-CC information	Maximum field length (in octets)
1	sessionId	20
2	resourceNum associationTag	2 2

INTERIM SOLUTION NOTE: Interim BHL-I solution to be used prior to the availability of the Generic Identifier Transport:

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. The reduced deviceId will be assigned by the SRM in the UNConfig command sequence, or will be pre-provisioned in the Client/Server at installation.

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 Table D-2. The reduced sessionId will be contained in the 5 least significant bytes of the 10-byte sessionId. The 5-byte reduced sessionId value follows a specific format and is adopted as shown below:

MSb 1,2	= 13818-6 reserved (00)

a) Network assigned

MSb 1-2	= Network assigned (11)
MSb 3-40	= 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	= Client assigned (01)
MSb 3-32	= reduced deviceId value of network domain significance.
MSb 33-40	= reduced session number value

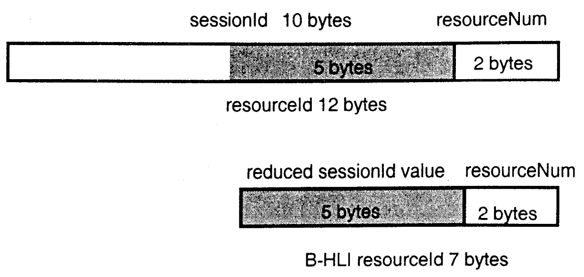


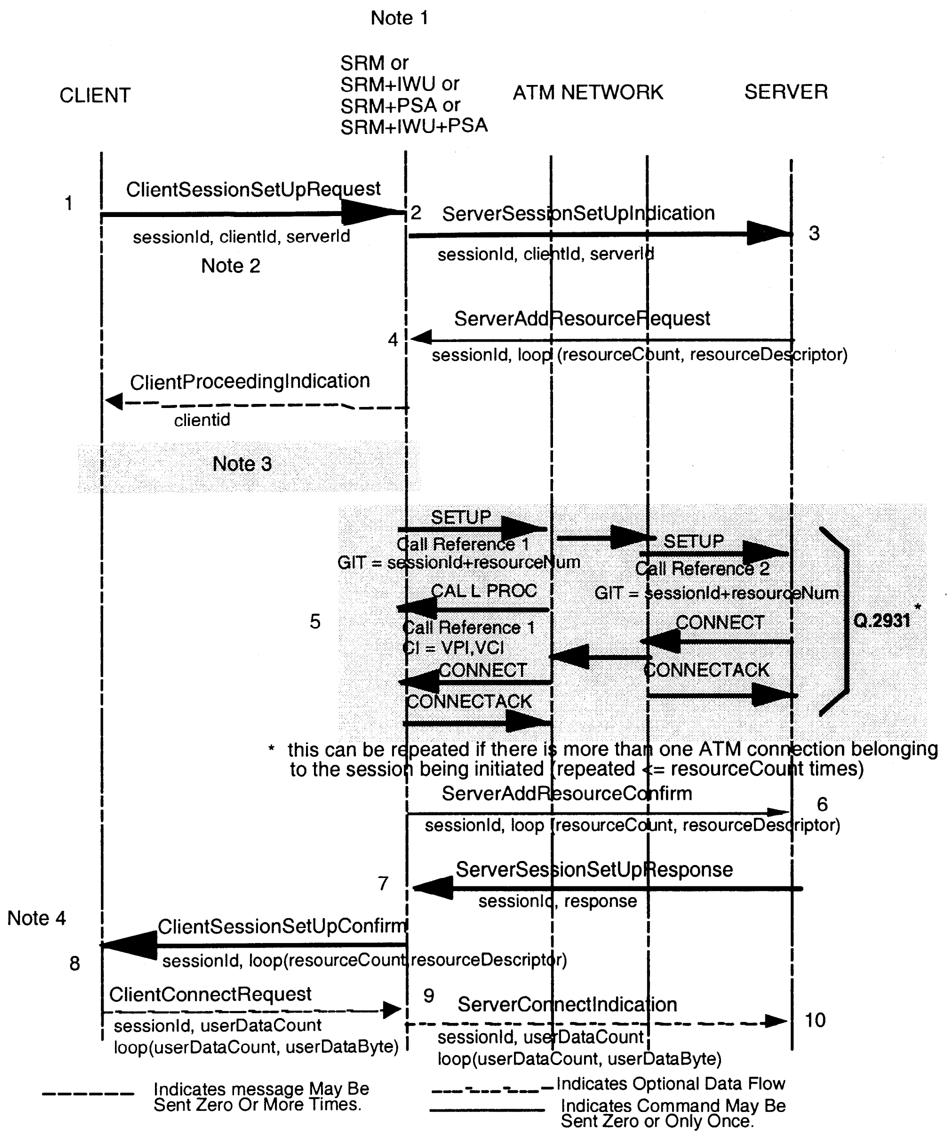
Figure D-1 Composition of compressed resourceId for interim B-HLI solution

D.3 Session Method Command Sequences

D.3.1 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).

D.3.1.1 Client Session Set-Up


Note 1
SRM or SRM+IWU or SRM+PSA or SRM+IWU+PSA
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
GIT = sessionId+resourceNum
SETUP
Call Reference 2
GIT = sessionId+resourceNum
CALL PROC
5
Call Reference 1
CI = VPI,VCI
CONNECT
CONNECT
CONNECTACK
CONNECTACK
Q.2931*
* this can be repeated if there is more than one ATM connection belonging to the session being initiated (repeated <= resourceCount times)
ServerAddResourceConfirm
6
sessionId, loop (resourceCount, resourceDescriptor)
7
ServerSessionSetUpResponse
sessionId, response
Note 4
ClientSessionSetUpConfirm
8
sessionId, loop(resourceCount, resourceDescriptor)
ClientConnectRequest
sessionId, userDataCount
loop(userDataCount, userDataByte)
9 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.

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 SRM and Server and SRM are assumed.

Figure D-2 Session Method: Client Session Set-Up

DSM-CC protocol:

Step 1

The Client sends a ClientSessionSetUpRequest message to the SRM.

Step 2

The SRM 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 valid, accepts the request. After the Server has accepted the session, the Server can either:

a) immediately send the SessionSetUpResponse, accepting the session, and then add the necessary resources using ServerAddResourceRequest, or
b) send one ServerAddResourceRequest message to the SRM to request resources, and wait for the ServerAddResourceConfirm, before sending the ServerSessionSetUpResponse to accept the session.

The Server gathers all resources required from the Network to support the service. If the service needs one or more ATM SVC connections, the Server embeds the proper number of ATM Connection resource descriptors and a resourceGroup descriptor in the ServerAddResourceRequest message. These resource descriptors contain sufficient information for the SRM to later establish a connection between the following, depending on the network architecture (refer to Table D-1):

- the Server and the Client (scenario 1a in Table D-1),
- the Server and the IWU (scenario 1b), or
- the Server and Client using "join" functionality (scenario 1c), provided the DSS2 Supplementary signaling required is available.

> Note: The capability to "join" the Client side and the Server side is not available in ITU DSS2 capability set 2 step 1. The scenario1c in method 1 is included in this part of ISO/IEC 13818 in order to promote discussion on the use of potential ATM signaling standards with DSM-CC.

Step 4

Upon receipt of the ServerAddResourceRequest message, the SRM can process the ATM Connection and resourceGroup resource descriptors included in the message. For an "ATM SVC" resource, the SRM 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 BUT NEGOTIABLE and the available value is within the requested range, then it can be satisfied and will be assigned by the SRM to the session. The resourceGroupTag value from the resourceGroup descriptor is used to link the non-ATM access Client view with the ATM SVC Server view.

For scenario 1c (see Table D-1), the SRM shall then establish the Q.2931 connection with the Client. Refer to Figure D-3.

Step 5

For any "ATM SVC" resource not rejected, the SRM shall initiate a connection matching an available access resource and containing the sessionId+resourceNum in the SETUP message.

Q.2931 protocol:

The SRM shall initiate 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 SRM
- Calling Party Number = ATM address of the Client (scenario 1a & 1c in Table D-1) or
  = ATM address of the IWU (scenario 1b in Table D-1).
- Called Party Number = ATM address of the Server as derived from the serverId field.
- ATM Adaptation Layer Parameters, ATM Traffic Descriptor and Quality-of-Service parameter = values imported from the "ATM SVC" resource request descriptor.
- Generic Identifier Transport (GIT) = sessionId+resourceNum corresponding to this "ATM SVC" resource.

After the requested connections between the Client and Server are established the SRM shall send a ServerAddResourceConfirm message indicating the resources which were successfully allocated.

The exact procedure to connect the Client to the ATM SVC being set up depends on the network architecture. For scenario 1b in Table D-1, a pass-band architecture, this procedure may involve the SRM 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. For scenarios 1a and 1c in Table D-1, no special procedure is required, however, in the case of scenario 1c, a "join" signaling must be sent to the network by the SRM.

When the Server is informed of the SETUP on its ATM UNI, it accesses the sessionId+resourceNum from the GIT information element, and associates the Call Reference with this session information.

Both the SRM and the Server maintain a Call Reference and sessionId+resourceNum association so that one can be retrieved from the other until the ATM SVC connection is released.

DSM-CC protocol:

Step 6

The ServerSessionSetUpResponse message shall signal the SRM of the Server's readiness to begin using the connections.

Step 7

After receipt of the ServerSessionSetUpResponse, the SRM shall inform the Client through the ClientSessionSetUpConfirm message.

Step 8

Both Client and Server are now ready to exchange User-to-User messages.

If the Client has userDataBytes to be delivered to the Server, it shall send a 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 shall indicate the number of userDataBytes present.

Step 9

On receipt of the ClientConnectRequest with a valid sessionId, the Network shall send a ServerConnectIndication to the Server. After sending the message, there is no change of state for the session at the Network.

Step 10

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

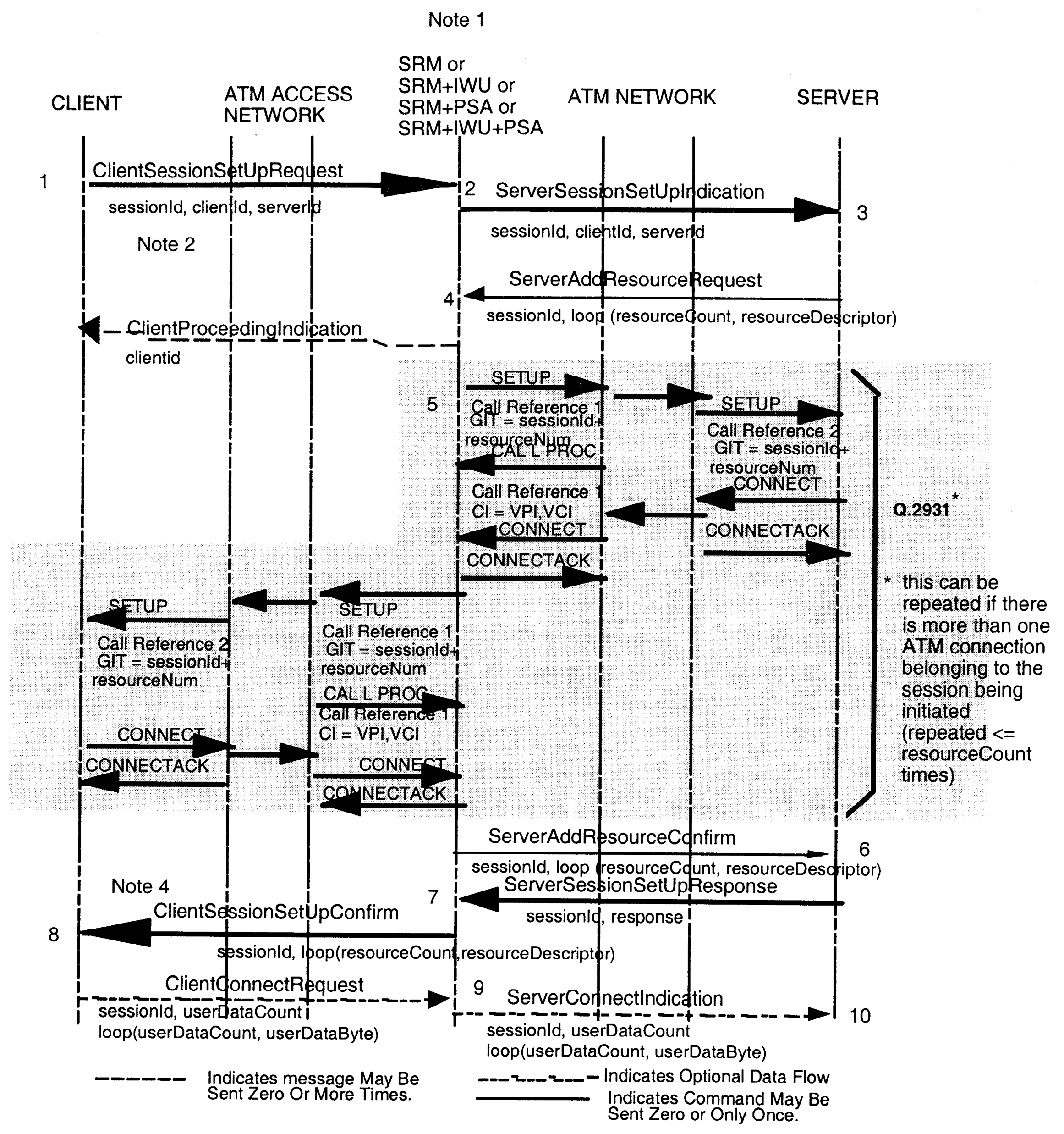

Note 1
CLIENT
ATM ACCESS NETWORK
SRM or SRM+IWU or SRM+PSA or SRM+IWU+PSA
ATM NETWORK
SERVER
1 ClientSessionSetUpRequest
sessionId, clientId, serverId
Note 2
2 ServerSessionSetUpIndication
sessionId, clientId, serverId
3
ServerAddResourceRequest
4
sessionId, loop (resourceCount, resourceDescriptor)
ClientProceedingIndication
clientid
SETUP
5 Call Reference 1 GIT = sessionId+ resourceNum
CALL PROC
SETUP
Call Reference 2 GIT = sessionId+ resourceNum
CONNECT
Call Reference 1 CI = VPI,VCI
CONNECT
CONNECTACK
CONNECTACK
Q.2931*
* this can be repeated if there is more than one ATM connection belonging to the session being initiated (repeated <= resourceCount times)
SETUP
Call Reference 2 GIT = sessionId+ resourceNum
SETUP
Call Reference 1 GIT = sessionId+ resourceNum
CALL PROC
Call Reference 1 CI = VPI,VCI
CONNECT
CONNECTACK
CONNECT
CONNECTACK
ServerAddResourceConfirm
6
sessionId, loop (resourceCount, resourceDescriptor)
Note 4
ServerSessionSetUpResponse
7
sessionId, response
ClientSessionSetUpConfirm
8
sessionId, loop(resourceCount,resourceDescriptor)
ClientConnectRequest
sessionId, userDataCount loop(userDataCount, userDataByte)
9 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.

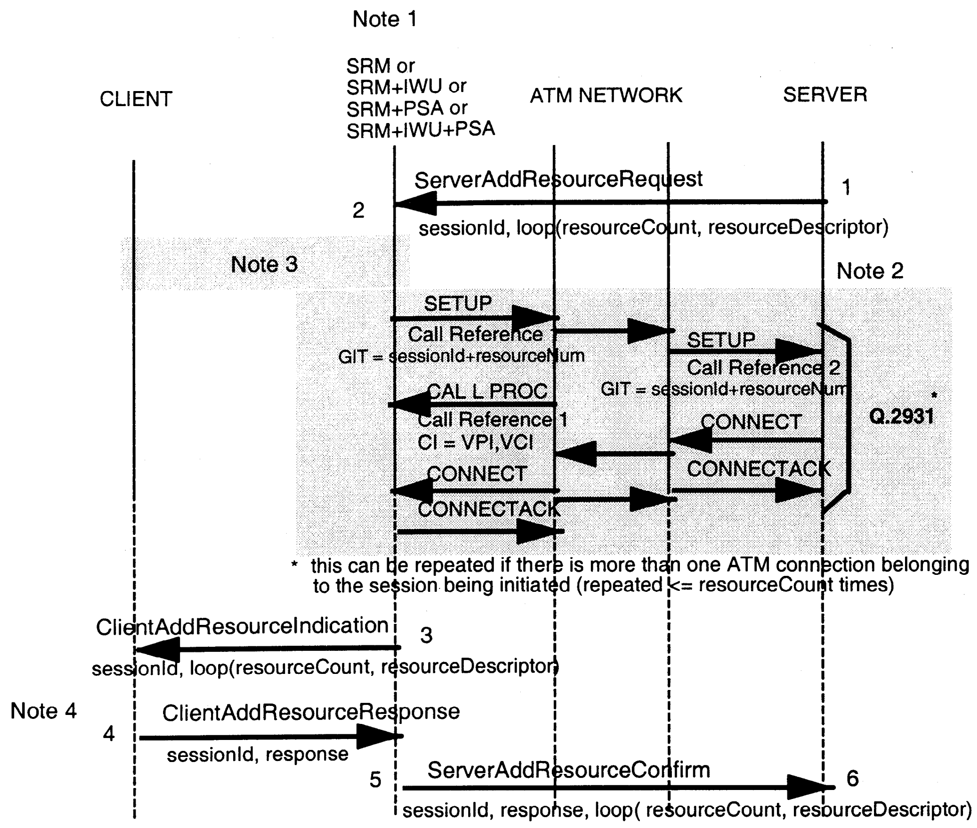
Figure D-3 Session Method: Client Session Set-Up for scenario 1(c)

Network Signaling in SRM, with "join" capabilities in SRM (Notes and Step descriptions are the same as for previous figure (Figure D-2).

D.3.2 Add 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 SRM to request new resources within the context of the established session.

D.3.2.1 Add Resource Request by the Server


Note 1
SRM or SRM+IWU or SRM+PSA or SRM+IWU+PSA
CLIENT
ATM NETWORK
SERVER
ServerAddResourceRequest
1
2
sessionId, loop(resourceCount, resourceDescriptor)
Note 3
Note 2
SETUP
Call Reference
GIT = sessionId+resourceNum
SETUP
Call Reference 2
GIT = sessionId+resourceNum
CALL PROC
Call Reference 1
CI = VPI,VCI
CONNECT
CONNECT
CONNECTACK
CONNECTACK
Q.2931
*
* this can be repeated if there is more than one ATM connection belonging to the session being initiated (repeated <= resourceCount times)
ClientAddResourceIndication
3
sessionId, loop(resourceCount, resourceDescriptor)
Note 4
ClientAddResourceResponse
4
sessionId, response
5
ServerAddResourceConfirm
6
sessionId, response, loop(resourceCount, resourceDescriptor)

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. See next figure for the message flow for scenario 1(c).
Note 4: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Figure D-4 Session Method: ServerAddResourceRequest, SRM initiates connection

DSM-CC Protocol:

Step 1

The Server requests the needed connection resources by sending the ServerAddResourceRequest message to SRM with a list of resourceDescriptors.

Step 2

If the SRM does not reject the requested "ATM SVC" resources, it proceeds by securing a connection on the access network portion and initiates an ATM core network connection matching the available access resources through Q.2931 signaling messages, containing the sessionId + resourceNum in the SETUP message.

Step 3

SRM 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 so, sends ClientAddResourceResponse to the network with the response field set to rspOK. At this point the Client shall consider the additional resources as being committed to the session.

Step 5

After all requested connections are established and a positive ClientAddResourceResponse is received, the SRM sends a ServerAddResourceConfirm message indicating the additional resources were successfully allocated. After sending the message, the SRM 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.

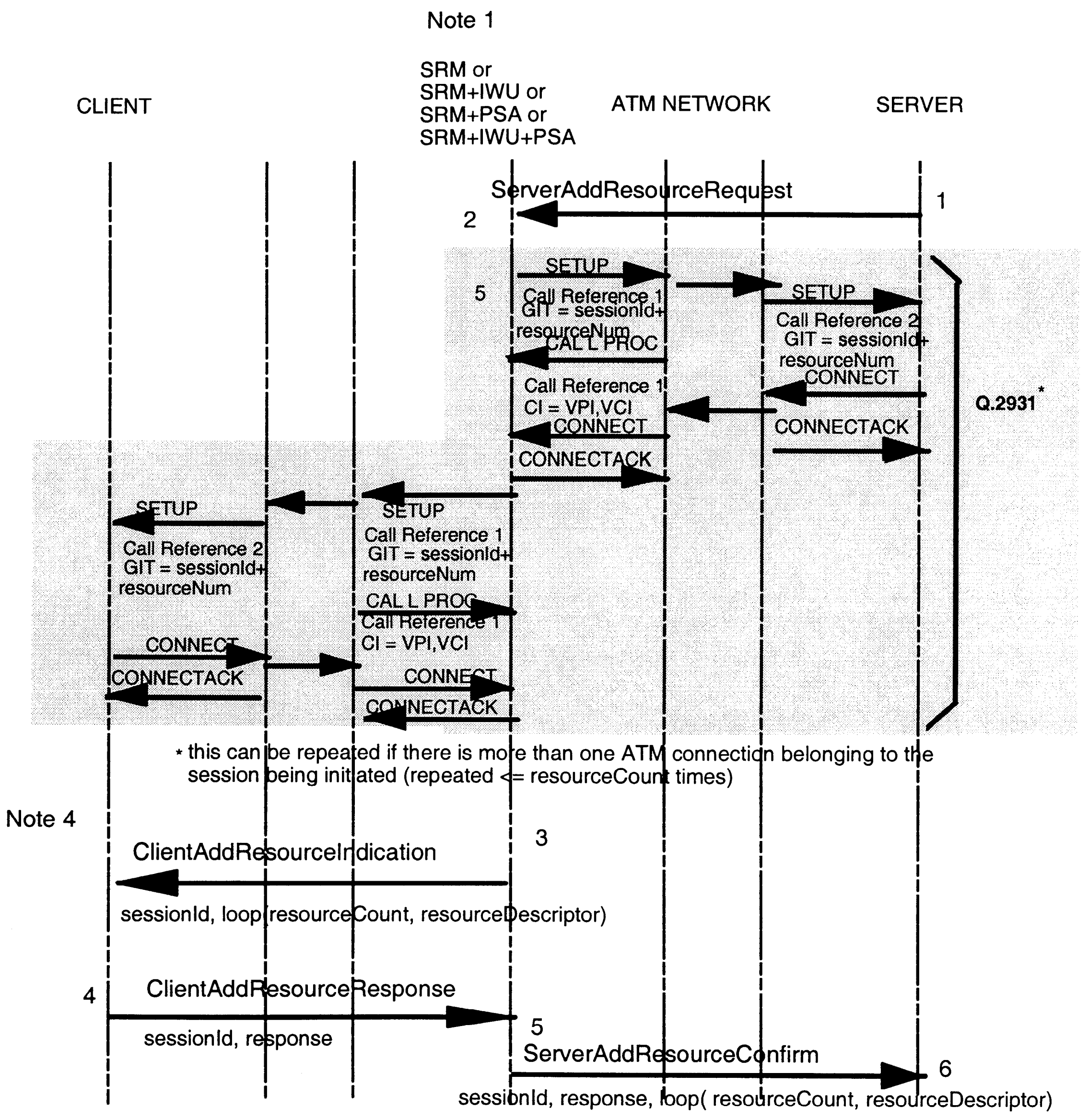


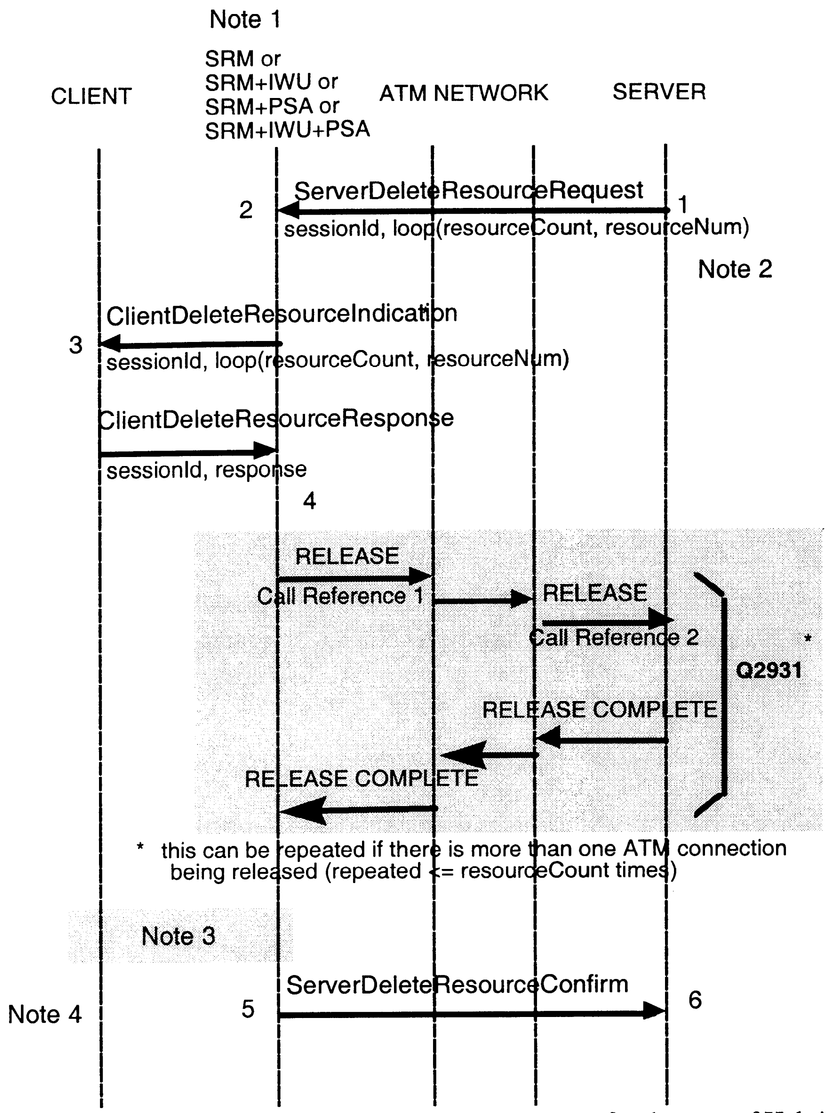
Figure D-5 Session Method: ServerAddResourceRequest for scenario 1(c)

Network Signaling in SRM, with "join" capabilities in SRM (Notes and Step descriptions are the same as for previous figure).

D.3.3 Resource Deletion

After a resource has been successfully requested, either through the Session Set-Up scenario or the Session Add Resource Request scenario, the resource can later be requested to be deleted.

D.3.3.1 Resource Deletion by the Server


Note 1
SRM or SRM+IWU or SRM+PSA or SRM+IWU+PSA
CLIENT
ATM NETWORK
SERVER
ServerDeleteResourceRequest
sessionId, loop(resourceCount, resourceNum)
Note 2
ClientDeleteResourceIndication
sessionId, loop(resourceCount, resourceNum)
ClientDeleteResourceResponse
sessionId, response
RELEASE
Call Reference 1
RELEASE
Call Reference 2
Q2931
RELEASE COMPLETE
RELEASE COMPLETE
* this can be repeated if there is more than one ATM connection being released (repeated <= resourceCount times)
Note 3
ServerDeleteResourceConfirm
Note 4
1
2
3
4
5
6

Note 1: Optional role of the IWU is to manage the access network resources for the 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. See next figure for the message flow for scenario 1(c).

Note 4: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Figure D-6 Session Method: Resource Deletion Request by Server, ATM SVC connection released by SRM

DSM-CC protocol:

Step 1

The Server shall inform the SRM of its request for deletion of one or multiple assigned resources via the ServerDeleteResourceRequest message. The resources are identified by their corresponding **resourceId**

Step 2

Upon receipt of a ServerDeleteResourceRequest, the SRM shall verify that the session exists and is associated with the Server, and that the resourceIds are valid for the session. If so, the SRM shall send a ClientDeleteResourceIndication message to the Client.

Step 3

Upon receipt of a ClientDeleteResourceIndication, the Client shall verify that the session exists and that the resourceIds are valid for the session. The Client responds by sending a ClientDeleteResourceResponse message to the SRM. 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 SRM will process the deletion of the identified resources on the Client Access Network from the established session. For scenario 1c in Table D-1, this requires Q.2931 release procedures.

If the SRM detects a **resourceId** identifying an "ATM SVC" resource, it shall retrieve the corresponding Call Reference and initiate a Q.2931 Call/Connection Clearing procedure at its ATM UNI.

Q.2931 protocol for Server Side Connection Clear:

The SRM sends a Q.2931 RELEASE message to the UNI, with the following information elements:

Call Reference = Call Reference retrieved from the **resourceId**.

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 **resourceId** which is then given to the session in the Server for housekeeping chores.

Q.2931 protocol for Client Side Connection Clear:

For scenario 1c, the Q.2931 protocol for Call/Connection Clearing is repeated here for the Client side.

DSM-CC protocol:

Step 5

After all the resources have been deleted (or as soon as the deletion process has been initiated - it does not have to be completed), the SRM shall send the Server a ServerDeleteResourceConfirm message. At this point the SRM shall consider the resource deletion process completed.

Step 6

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

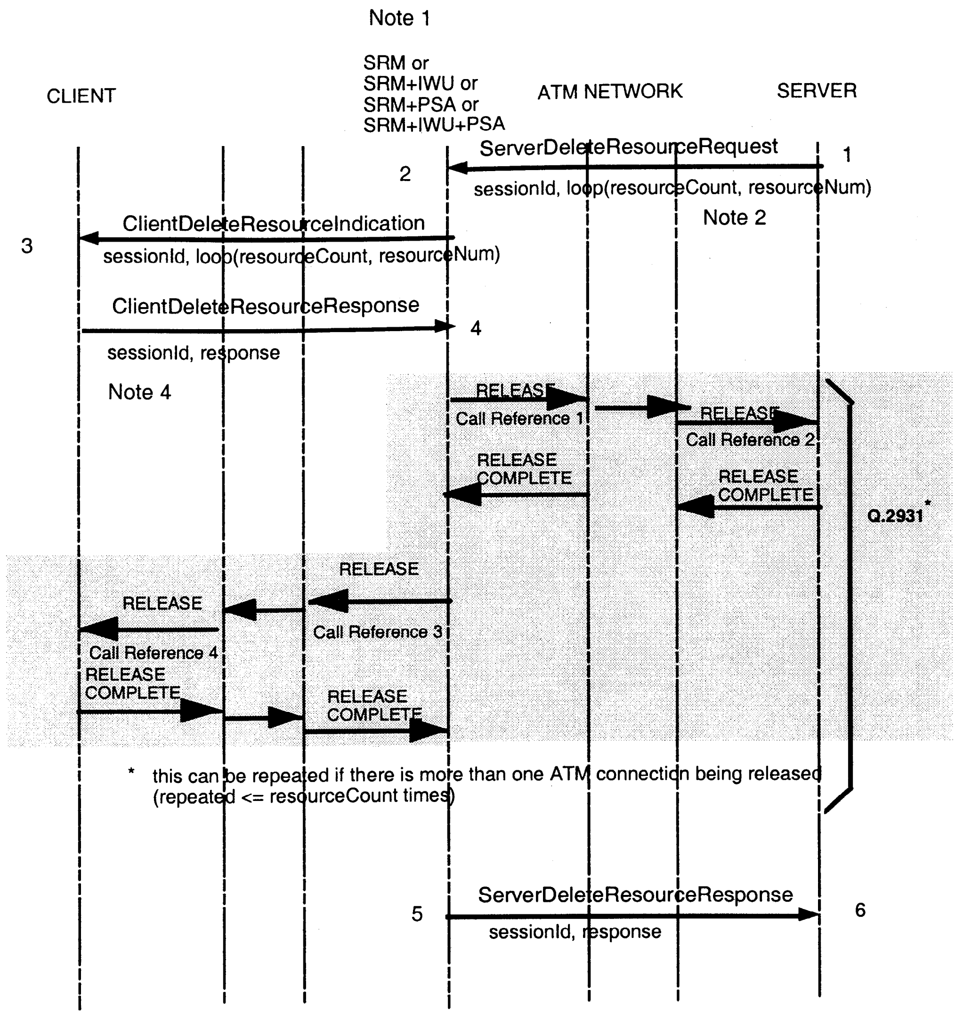

Note 1
SRM or
SRM+IWU or
SRM+PSA or
SRM+IWU+PSA
CLIENT
ATM NETWORK
SERVER
ServerDeleteResourceRequest
1
2
sessionId, loop(resourceCount, resourceNum)
Note 2
ClientDeleteResourceIndication
3
sessionId, loop(resourceCount, resourceNum)
ClientDeleteResourceResponse
4
sessionId, response
Note 4
RELEASE
Call Reference 1
RELEASE
Call Reference 2
RELEASE COMPLETE
RELEASE COMPLETE
Q.2931*
RELEASE
RELEASE
Call Reference 3
Call Reference 4
RELEASE COMPLETE
RELEASE COMPLETE
* this can be repeated if there is more than one ATM connection being released (repeated <= resourceCount times)
ServerDeleteResourceResponse
5
6
sessionId, response

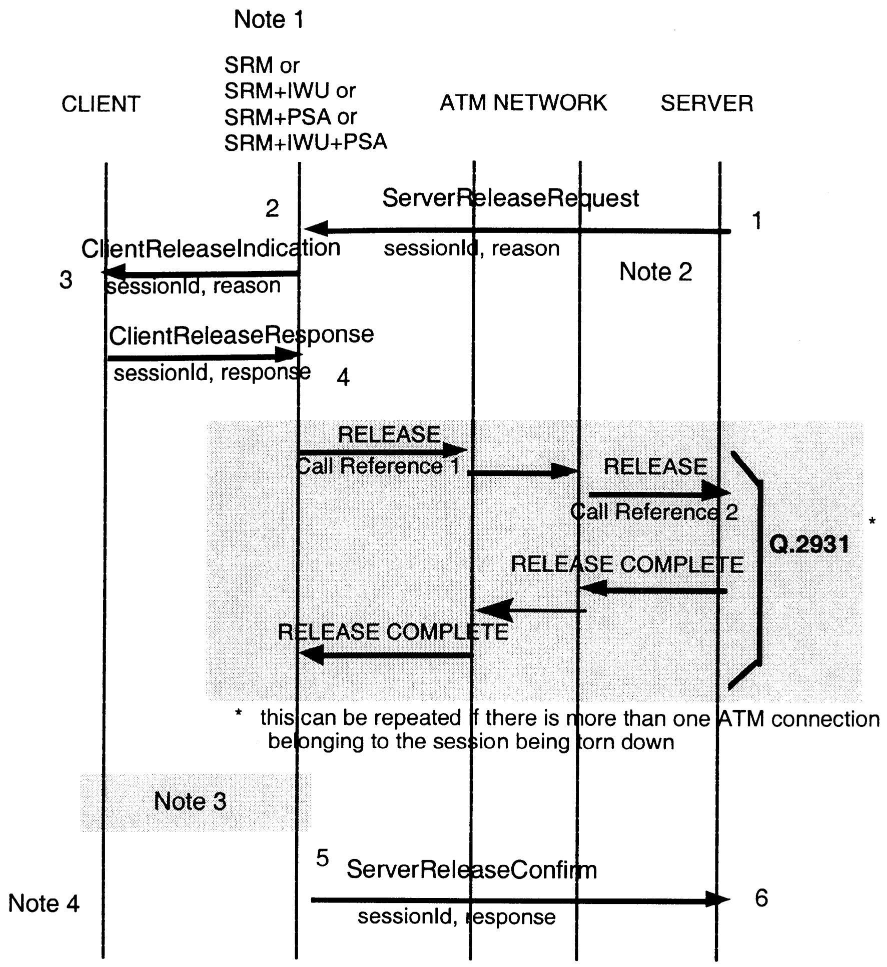
Figure D-7 Session Method: ServerDeleteResourceRequest for scenario 1(c)

Network Signaling in SRM, with "join" capabilities in SRM (Notes and Step descriptions are the same as for previous figure).

D.3.4 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 SRM to request that session is to be torn down. For a session containing an "ATM SVC" resource, the scenarios follow below.

D.3.4.1 Session Tear-Down 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. See next figure for the message flow for scenario 1(c).

Note 4: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Figure D-8 Session Method: Server Session Tear Down

The Steps in Figure D-8 are similar to those for Figure D-10, with Client and Server interchanged.

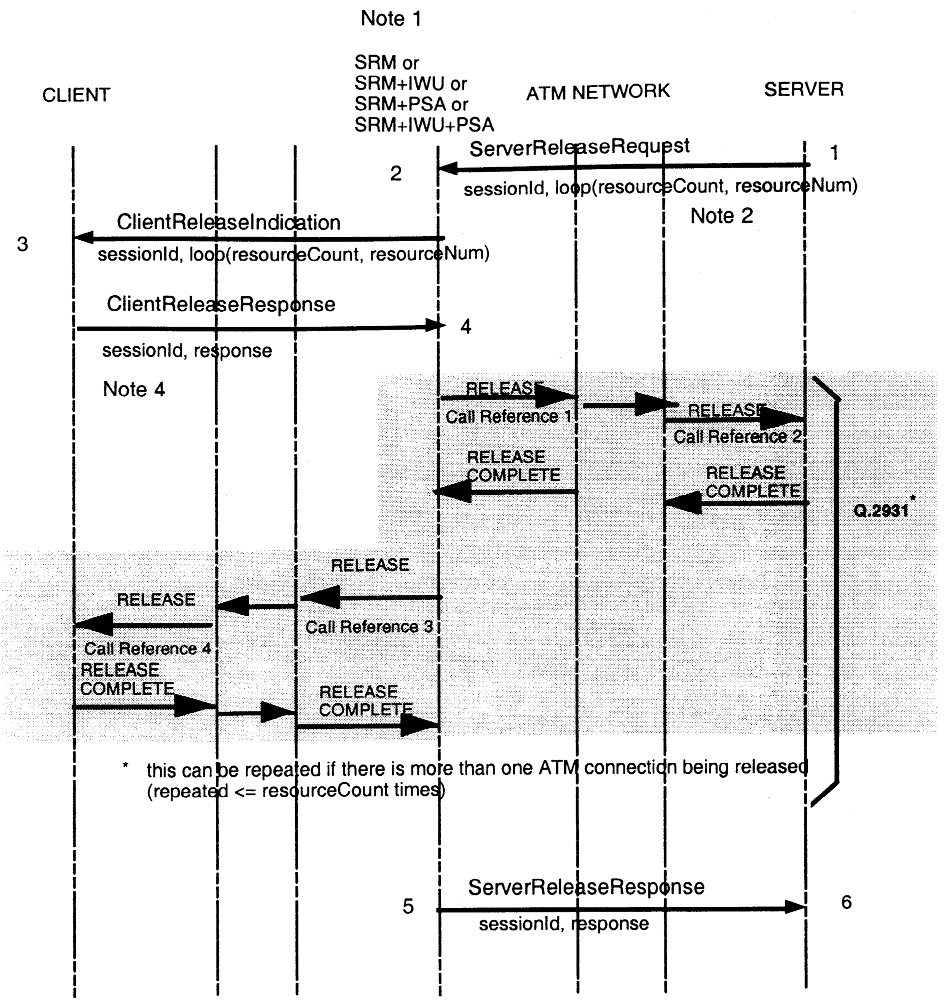
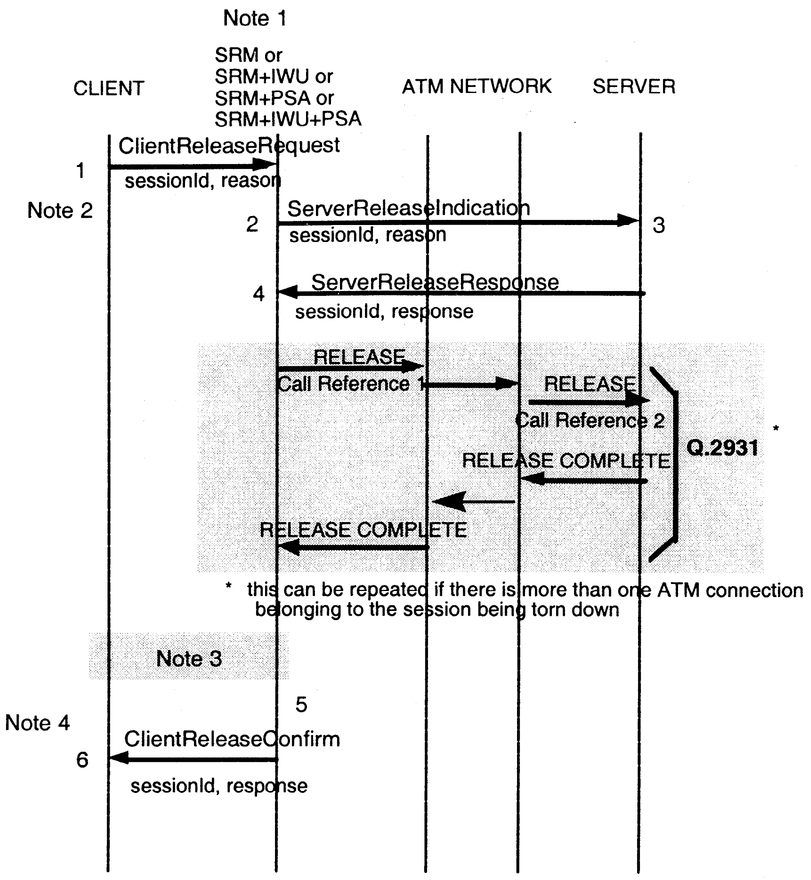


Figure D-9 Session Method: ServerDeleteResourceRequest for scenario 1(c)

Network Signaling in SRM, with "join" capabilities in SRM (Notes are the same as for Figure D-10). The Steps in Figure D-9 are similar to those for Figure D-10, with Client and Server interchanged.

D.3.4.2 Session Tear-Down by Client



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 message flow for scenario 1(c) is constructed similarly to that shown in Figure D-9.

Note 4: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Figure D-10 Session Method: Client Session Tear Down

DSM-CC protocol:

Step 1

The Client informs the SRM of its request for session tear-down via the ClientReleaseRequest message.

Step 2

The SRM 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 SRM shall retrieve all resources allocated to the identified session and proceed to delete those resources from the Client Access Network and the ATM Core Network.

If the SRM detects an "ATM SVC" resource belonging to the session, it shall retrieve the corresponding Call Reference from the resourceId and initiate a Q.2931 Call Connection Clearing at its ATM UNI.

Q.2931 protocol for Server Side Connection Clear:

The description of this Call/Connection Clearing procedure is similar to the one in the Resource Deletion scenario.

Q.2931 protocol for Client Side Connection Clear:

The Q.2931 protocol for Call/Connection Clearing is repeated here for the Client side for scenario 1c.

DSM-CC protocol:

Step 5

After all the resources have been deleted (or as soon as the deletion process has been initiated - it does not have to be completed), the SRM shall send the Client a ClientReleaseConfirm message. At this point the SRM shall consider the resource deletion process completed.

Step 6

On receipt of the ClientReleaseConfirm the Client shall consider the session terminated.

D.4 Network Method with DSM-CC AddResource messages between the Server and SRM

The Network Method with DSM-CC AddResource messages between the Server and SRM requires that resources be allocated by the Server prior to sending the AddResource message. The AddResource messages are thus used to inform the SRM, providing it with the opportunity to request the release of the connection.

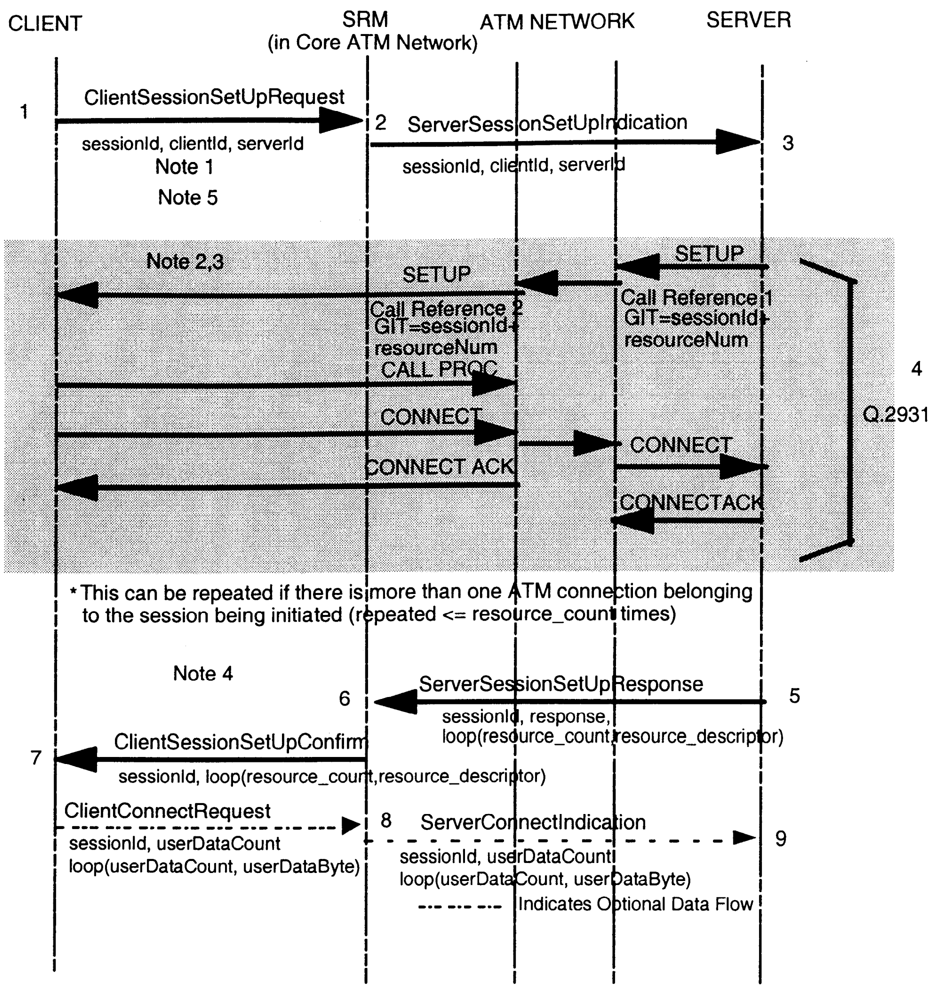
Note: The command sequences in 13.4 cover the case of scenario 2a in Table D-1. The other scenario command sequences are similar and are obtained by substituting the ATM signaling termination to the Client by a termination to either the PSA (scenario 3b) or the IWU (scenarios 3c & d)

D.4.1 Session Set-Up

The following subclauses show Client Session Set-Up when the Server sets up the ATM connection.

D.4.1.1 Client Session Set-Up, Server ATM Connection Set-Up

In this command sequence, ATM 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 SRM through sessionId+resourceNum: when the Server has finished allocating resources, it sends a ServerSessionSetUpResponse to the SRM.


CLIENT
SRM (in Core ATM Network)
ATM NETWORK
SERVER
1
ClientSessionSetUpRequest
sessionId, clientId, serverId
Note 1
Note 5
2 ServerSessionSetUpIndication
sessionId, clientId, serverId
3
Note 2,3
SETUP
Call Reference 2
GIT=sessionId+resourceNum
CALL PROC
CONNECT
CONNECT ACK
SETUP
Call Reference 1
GIT=sessionId+resourceNum
CONNECT
CONNECTACK
4
Q.2931
*This can be repeated if there is more than one ATM connection belonging to the session being initiated (repeated <= resource_count times)
Note 4
6
ServerSessionSetUpResponse
sessionId, response, loop(resource_count,resource_descriptor)
5
7
ClientSessionSetUpConfirm
sessionId, loop(resource_count,resource_descriptor)
ClientConnectRequest
sessionId, userDataCount
loop(userDataCount, userDataByte)
8 ServerConnectIndication
sessionId, userDataCount
loop(userDataCount, userDataByte)
9
Indicates Optional Data Flow

Note 1: Only relevant parameters in each message are shown.

Note 2: End-to-end information needs to be transmitted with the SETUP message associating the connection to the session resource.

Note 3: Nothing precludes Q.2962 from being used for resource negotiation.

Note 4: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Note 5: sessionId is assigned by the Client in this figure. If the sessionId is NOT assigned by the Client, ATM connection set-up, and AddResource messages must occur AFTER the ClientSessionSetUpConfirm.

Figure D-11 Network Method with AddResource message information: Client Session Set-Up, Server initiates ATM connection

DSM-CC Protocol:

Step 1

The Client shall send a ClientSessionSetUpRequest message to the SRM.

Step 2

The SRM 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

The Server shall take note of the sessionId, gather required resources, assign **resourceNums** to each resource, and initiate Q.2931 signaling.

Step 4

Q.2931 Protocol (initiated by Server):

The Server shall initiate 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.
- Generic Identifier Transport (GIT) = **sessionId+resourceNum**.

Both the SRM and the Server shall maintain a Call Reference and sessionId+resourceNum association so that one can be retrieved from the other. This association shall be kept until the ATM SVC connection is released.

DSM-CC Protocol:

Step 5

The Server notes the successful establishment of the connections(s), and generates a ServerSessionSetUpResponse which includes the resource descriptors for the session. At this point the session is considered established by the Server.

Step 6

The SRM takes note, and generates a ClientSessionSetUpConfirm. At this point the session is considered established by the SRM.

Step 7

Both Client and Server are now ready to exchange User-to-User messages.

If the Client has userDataBytes to be delivered to the Server, it shall send a 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 shall indicate the number of userDataBytes present.

Step 8

On receipt of the ClientConnectRequest with a valid sessionId, the Network shall send a ServerConnectIndication to the Server. After sending the message, there is no change of state for the session at the Network.

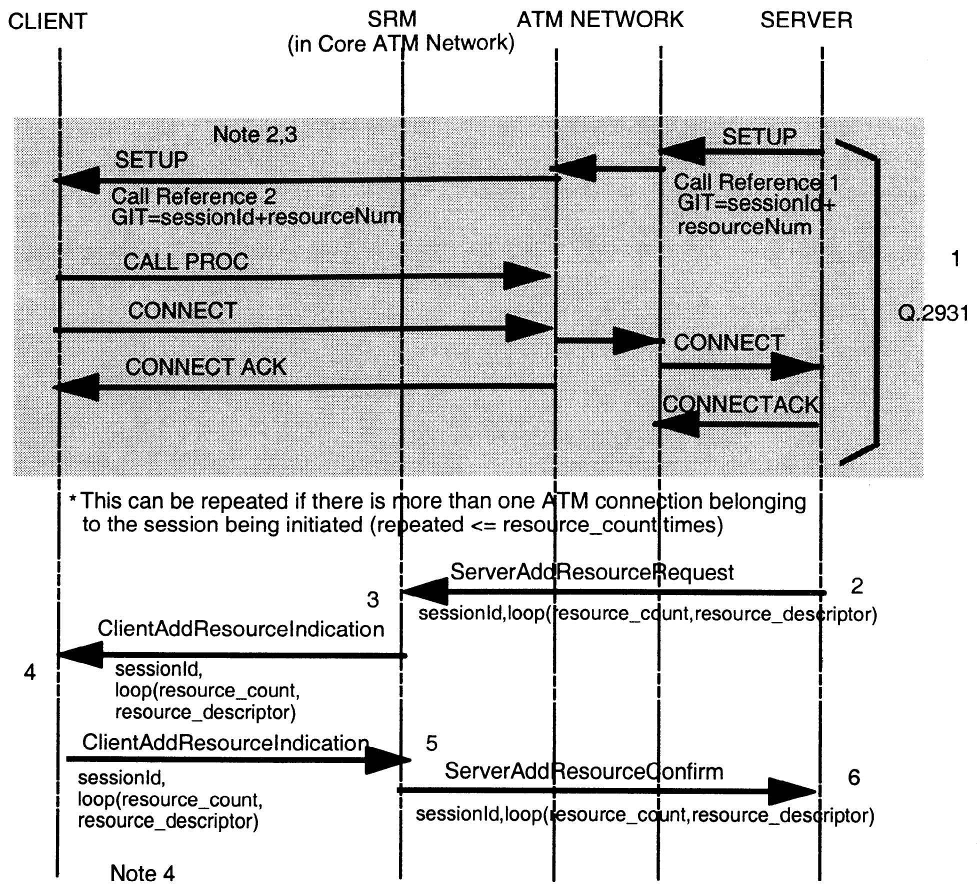
Step 9

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

The Server can now begin to exchange the User-to-User messages with the Client over the established connections.

D.4.2 Add Resource Request

D.4.2.1 Add Resource Request by Server and ATM SVC Connection Set-Up by Server


CLIENT
SRM (in Core ATM Network)
ATM NETWORK
SERVER
Note 2,3
SETUP
Call Reference 2
GIT=sessionId+resourceNum
CALL PROC
CONNECT
CONNECT ACK
SETUP
Call Reference 1
GIT=sessionId+resourceNum
CONNECT
CONNECTACK
1
Q.2931
*This can be repeated if there is more than one ATM connection belonging to the session being initiated (repeated <= resource_count times)
ServerAddResourceRequest
2
sessionId,loop(resource_count,resource_descriptor)
3
ClientAddResourceIndication
4
sessionId, loop(resource_count, resource_descriptor)
ClientAddResourceIndication
sessionId, loop(resource_count, resource_descriptor)
5
ServerAddResourceConfirm
6
sessionId,loop(resource_count,resource_descriptor)
Note 4

Note 1: Only relevant parameters in each message are shown.
Note 2: End-to-end information needs to be tran*sm*itted with the SETUP message associating the connection to the session resource.
Note 3: Nothing precludes Q.2962 from being used for resource negotiation.
Note 4: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Figure D-12 Network Method with AddResource messages: Server AddResourceRequest, Server initiates ATM connection

Step 1

Q.2931 Protocol (initiated by Server):

The Server shall initiate 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.
- 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.
- Generic Identifier Transport (GIT) = **sessionId+resourceNum**.

Both the SRM and the Server shall maintain a Call Reference and sessionId+resourceNum association so that one can be retrieved from the other. This association shall be kept until the ATM SVC connection is released.

DSM-CC Protocol:

Step 2

The Server notes the successful establishment of the connection(s), and generates a ServerAddResourceRequest with a list of the resources actually allocated for the session. Each descriptor contains a field indicating that the connections have already been established

Step 3

If the SRM accepts the addition of this resource, it shall generate a ClientAddResourceIndication to inform the Client of the new resource(s) now available. If the SRM decides the resource request is invalid (e.g., QoS exceeds network guidelines), the SRM generates a ServerClearIndication to both the Client and the Server to clear the resource connection just established and Steps 4-6 are skipped.

Step 4

The Client shall generate a positive ClientAddResourceResponse if it accepts the new resource(s). At this point the Client shall consider the new resource(s) set up and ready for use.

Step 5

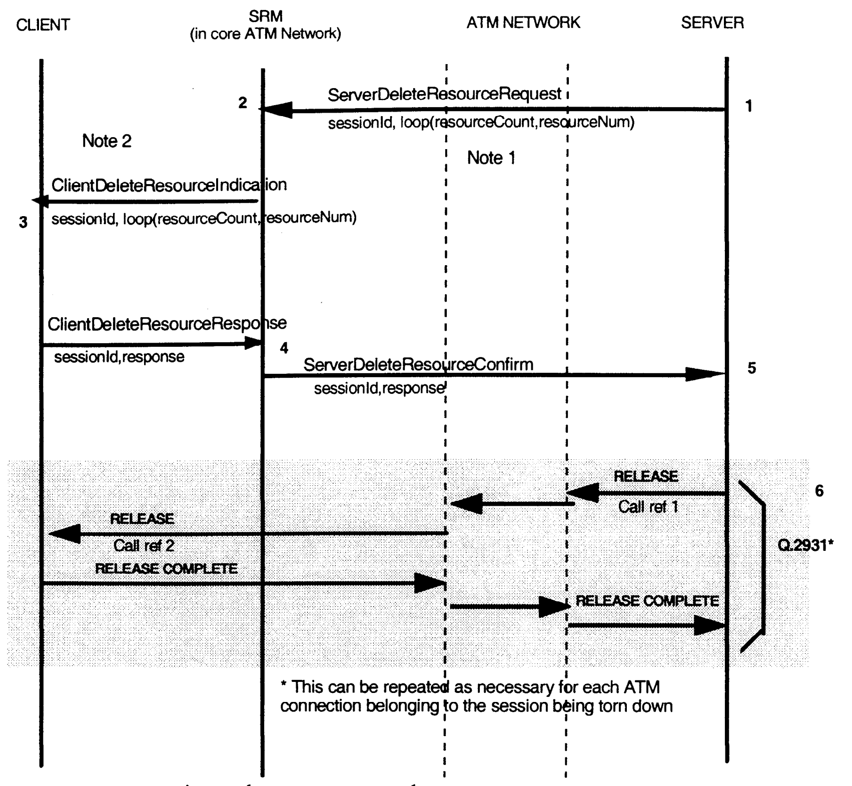
The SRM shall then generate a ServerAddResourceConfirm. At this point the SRM shall consider the new resource(s) set up and ready for use.

Step 6

At this point the Server shall consider the new resource(s) set up and ready for use.

D.4.3 Resource Deletion

D.4.3.1 Resource Deletion Request by Server and ATM SVC Connection Release by Server


CLIENT
SRM (in core ATM Network)
ATM NETWORK
SERVER
ServerDeleteResourceRequest
sessionId, loop(resourceCount,resourceNum)
1
2
Note 2
Note 1
ClientDeleteResourceIndication
sessionId, loop(resourceCount,resourceNum)
3
ClientDeleteResourceResponse
sessionId,response
4
ServerDeleteResourceConfirm
sessionId,response
5
RELEASE
Call ref 1
6
RELEASE
Call ref 2
RELEASE COMPLETE
Q.2931*
RELEASE COMPLETE
* This can be repeated as necessary for each ATM connection belonging to the session being torn down

Note 1: Only relevant parameters in each message are shown

Note 2: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Figure D-13 Server Delete Resource, ATM SVC connections released by the Server

DSM-CC protocol:

Step 1

The Server informs the SRM of its request for resource deletion via the ServerDeleteResourceRequest message.

Step 2

The SRM takes note of the request, and then informs the Client via the ClientDeleteResourceIndication message.

Step 3

The Client generates a ClientDeleteResourceResponse for the SRM At this point the Server shall consider the resource deleted.

Step 4

The SRM passes a ServerDeleteResourceConfirm to the Server. At this point the SRM shall consider the resource deleted.

Step 5

At this point the Server shall consider the resource deleted.

Step 6

Q.2931 protocol:

The Server sends a Q.2931 RELEASE message to the UNI, with the following information elements:

Call Reference = Call Reference retrieved from the sessionId+resourceNum.

When the Client receives the corresponding RELEASE message at its UNI, it retrieves the Call Reference from the message and from the Call Reference, the associated sessionId+resourceNum which is then given to the session function in the Client for housekeeping chores.

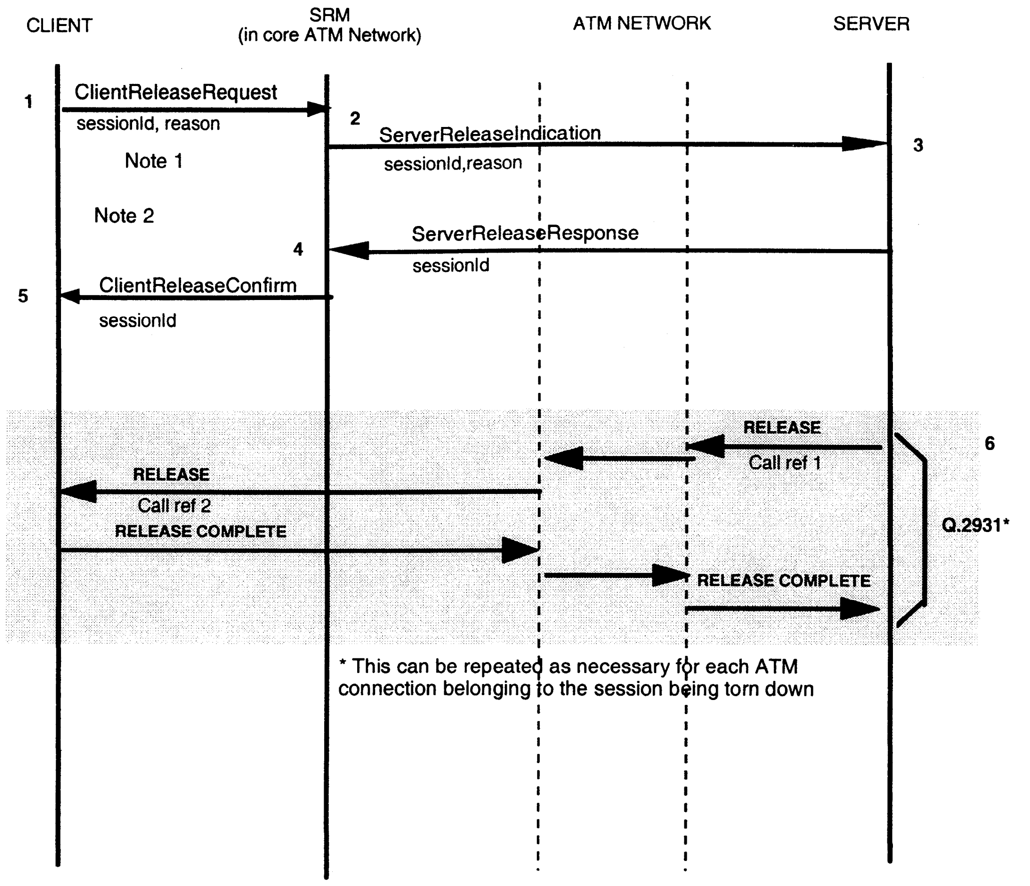
D.4.4 Session Tear-Down

After a session has been established through the Session Set-Up, either the Client or Server can later request that session is to be torn down.

D.4.4.1 Session Tear-Down Request by Server and ATM SVC Connection Release by Client

This case is symmetrical to the following command sequence Figure D-14 when the session tear-down request is by the Client and the ATM SVC connection Release is by the Server.

D.4.4.2 Session Tear-Down Request by Client and ATM SVC Connection Release by Server



Note 1: Only relevant parameters in each message are shown

Note 2: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Figure D-14 Client Session Tear-Down, ATM SVC connections released by the Server

DSM-CC protocol:

Step 1

The Client informs the SRM of its request for session tear-down via the ClientReleaseRequest message.

Step 2

The SRM takes note of the request, and then informs the Server via the ServerReleaseIndication message.

Step 3

The Server generates a ServerReleaseResponse for the SRM. At this point the Server shall consider the session released.

Step 4

The SRM passes a ClientReleaseConfirm to the Client. At this point the SRM shall consider the session released.

Step 5

At this point the Client shall consider the session released.

Step 6

If the Server detects an "ATM SVC" resource belonging to the session, it can retrieve the corresponding Call Reference from the sessionId+resourceNum and initiate a Q.2931 Call Connection Clearing at its ATM UNI.

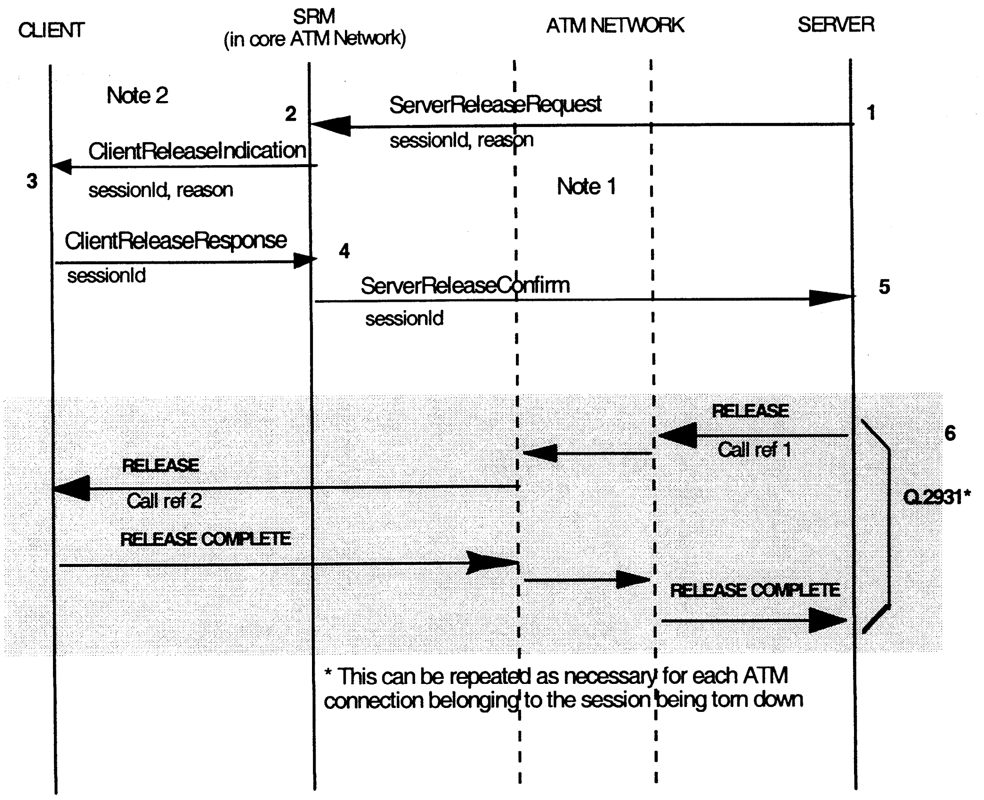
Q.2931 protocol:

The Server sends a Q.2931 RELEASE message to the UNI, with the following information elements:

Call Reference = Call Reference retrieved from the sessionId+resourceNum.

When the Client receives the corresponding RELEASE message at its UNI, it retrieves the Call Reference from the message and from the Call Reference, the associated sessionId+resourceNum which is then given to the session function in the Client for housekeeping chores.

D.4.4.3 Session Tear-Down Request by Server and ATM SVC Connection Release by Server



Note 1: Only relevant parameters in each message are shown
Note 2: Connections for the exchange of User-Network messages between Client and SRM and Server and SRM are assumed.

Figure D-15 Server Session Tear-Down, ATM SVC connections released by the Server

DSM-CC protocol:

Step 1

The Server informs the SRM of its request for session tear-down via the ServerReleaseRequest message.

Step 2

The SRM takes note of the request, and then informs the Client via the ClientReleaseIndication message.

Step 3

The Client generates a ClientReleaseResponse for the SRM At this point the Server shall consider the session released.

Step 4

The SRM passes a ServerReleaseConfirm to the Server. At this point the SRM shall consider the session released.

Step 5

At this point the Server shall consider the session released.

Step 6

Q.2931 protocol:

The Server sends a Q.2931 RELEASE message to the UNI, with the following information elements:

Call Reference = Call Reference retrieved from the sessionId+resourceNum.

When the Client receives the corresponding RELEASE message at its UNI, it retrieves the Call Reference from the message and from the Call Reference, the associated sessionId+resourceNum which is then given to the session function in the Client for housekeeping chores.

D.5 Network Method with NO DSM-CC AddResource messages between the Server and SRM

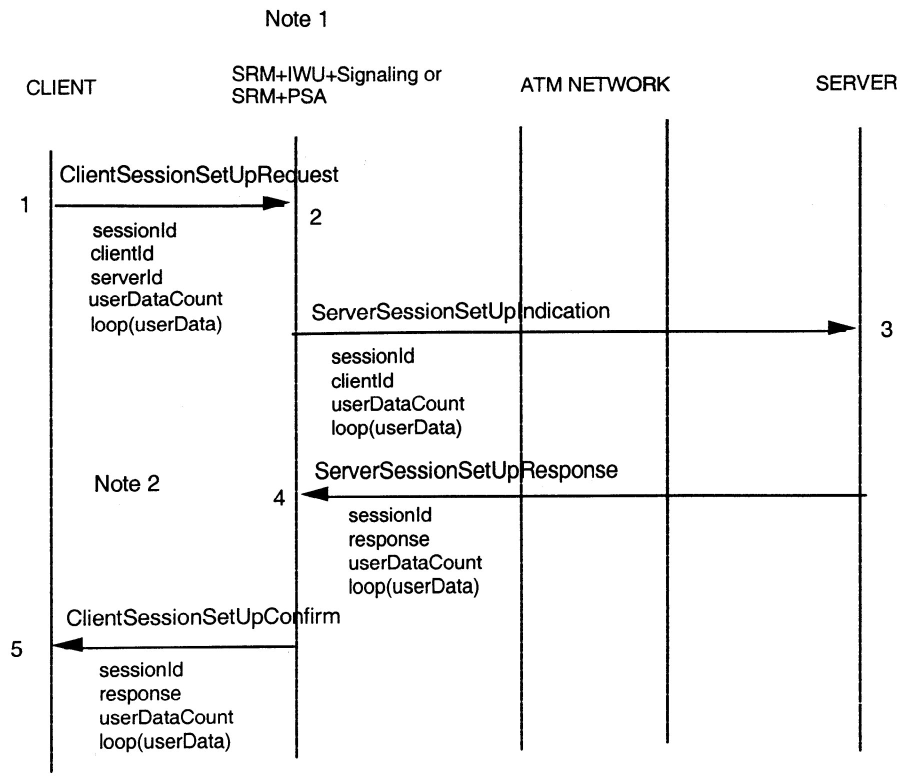
To optimize the number of DSM-CC message exchanges, this method allows the DSM-CC AddResource messages to be skipped. The SRM learns of connections since it is in the signaling path.

Any additional non-ATM resources shall be signaled using DSM-CC AddResource messages.

D.5.1 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.

D.5.1.1 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 SRM and Server and SRM are assumed.

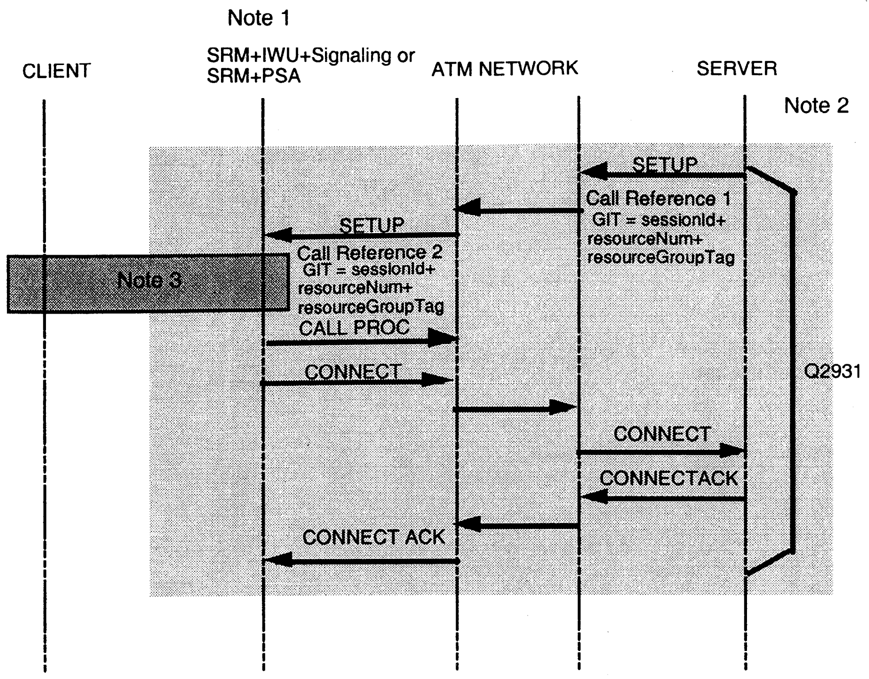
Figure D-16 Network Method with no AddResource messages: Client Session Set-Up

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 SRM assigns a sessionId as part of the session establishment sequence. The sessionId+resourceNum+resourceGroupTag must be included in all subsequent Q.2931 signaling messages. The Steps for this message flow are the same as corresponding ones in subclause D.3.1.1 Client Session Set-Up.

D.5.2 Add Resource Request

D.5.2.1 Add 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.



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 and will notify the Client of the sessionId, resourceNum and resourceGroupTag. See next figure for the message flow for scenario 3(c).

Figure D-17 Network Method with No AddResource messages: Server-initiated ATM Resource Request

The Server sends a Q2931 SETUP message to 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 SRM
- Generic Identifier Transport (GIT) = sessionId+resourceNum

When the SRM 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 SRM determines who the Client is, and later establishes a connection, depending on the scenario (refer to Table D-1):

- the Server and the Client using PSA (scenario 3a),
- the Server and the IWU (scenario 3b), proceeds to establish a path from the IWU to the Client and connect that path with the ATM connection that the Server just established or
- the Server and Client using "join" functionality (scenario 3c), provided the DSS2 Supplementary signaling required is available.

Note: The capability to "join" the Client side and the Server side is not available in ITU-T DSS2 capability set 2 step 1. The scenario1(c) in method 1 is included in this part of ISO/IEC 13818 in order to promote discussion on the use of potential ATM signaling standards with DSM-CC.

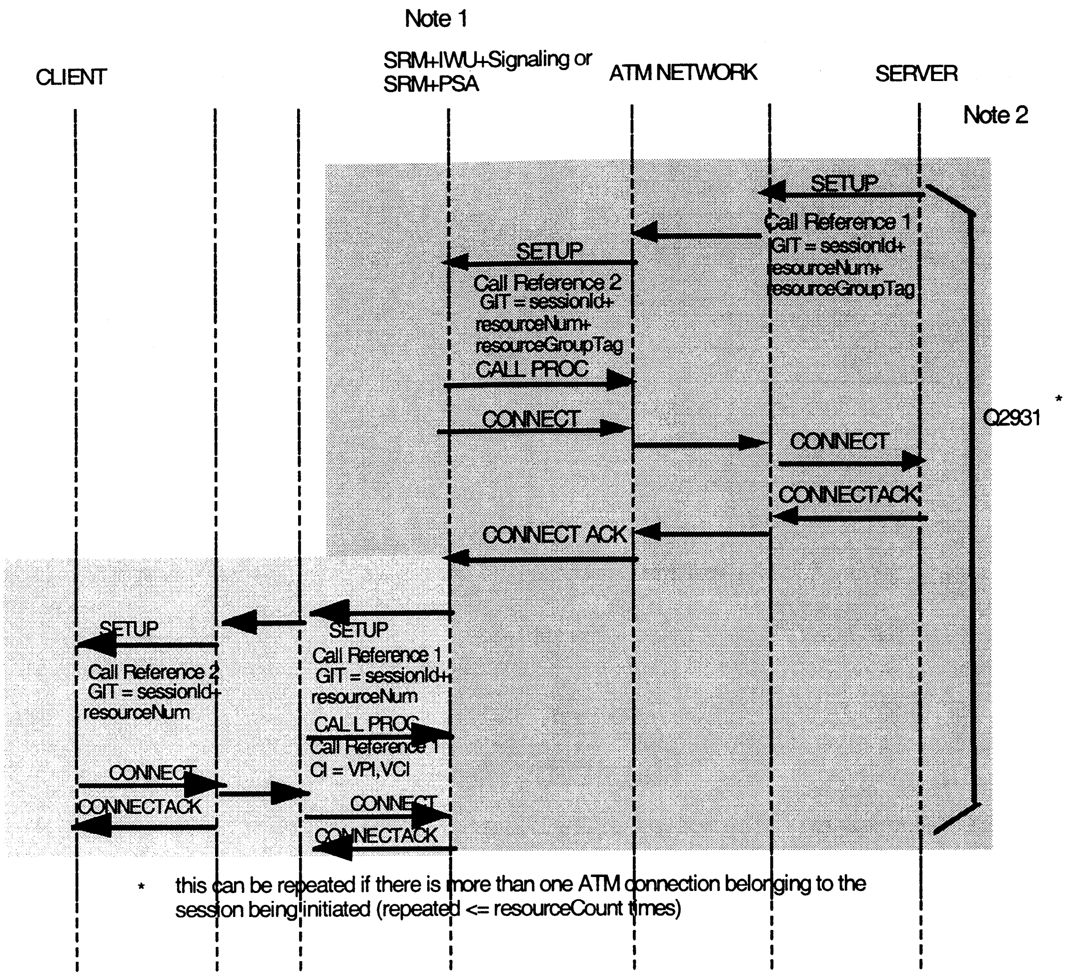

Note 1
CLIENT
SRM+IWU+Signaling or SRM+PSA
ATM NETWORK
SERVER
Note 2
SETUP
Call Reference 1
GIT = sessionId+ resourceNum+ resourceGroupTag
SETUP
Call Reference 2
GIT = sessionId+ resourceNum+ resourceGroupTag
CALL PROC
CONNECT
CONNECT
CONNECTACK
CONNECT ACK
Q2931 *
SETUP
Call Reference 2
GIT = sessionId+ resourceNum
SETUP
Call Reference 1
GIT = sessionId+ resourceNum
CALL PROC
Call Reference 1
CI = VPI,VCI
CONNECT
CONNECTACK
CONNECT
CONNECTACK
* this can be repeated if there is more than one ATM connection belonging to the session being initiated (repeated <= resourceCount times)

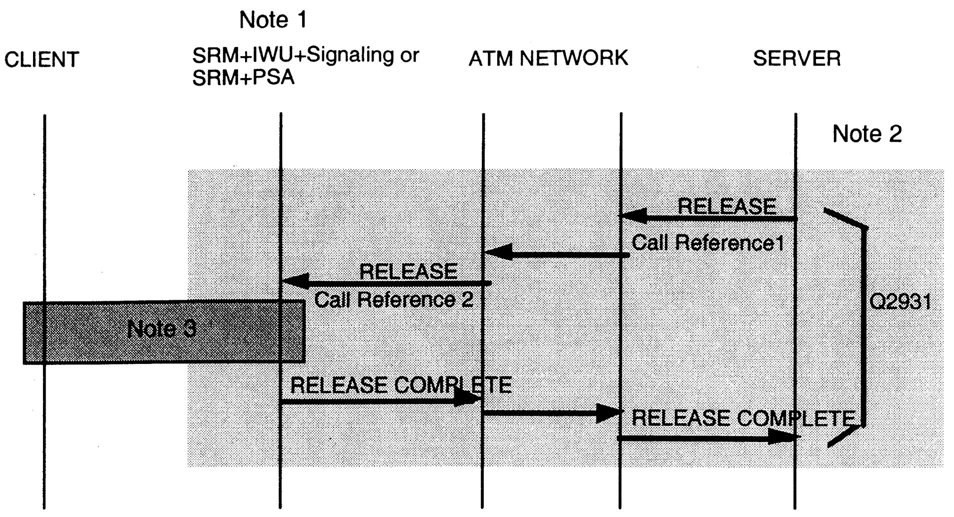
Figure D-18 Network Method with no AddResource messages: Server-initiated ATM Resource Request for scenario 3(c)

Network Signaling in SRM, with "join" capabilities in SRM (Notes and Step descriptions same as for previous figure).

D.5.3 Connection Clearing

D.5.3.1 Connection Clearing by the Server

The Server can delete an established ATM SVC connection by initiating the Q.2931 Connection Clearing procedure:



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. See next figure for message flow for scenario 3(c).

Figure D-19 Network Method with no AddResource messages: Resource Deletion via Server-initiated Call/Connection Release

When the SRM receives the RELEASE message on its UNI, it keys off the Call Reference to retrieve the sessionId.

The SRM 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.

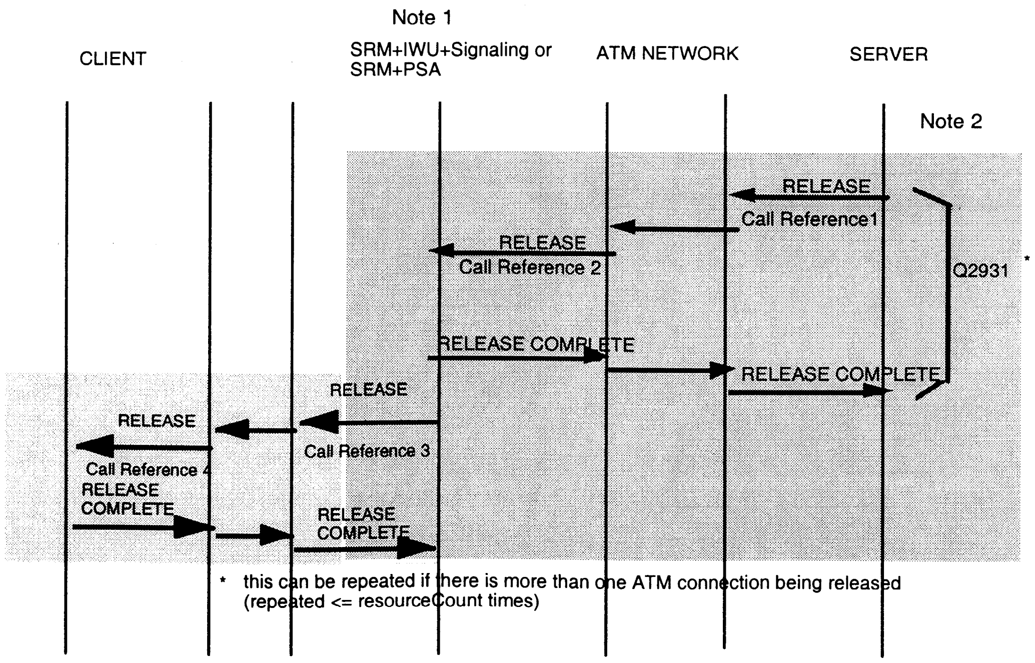
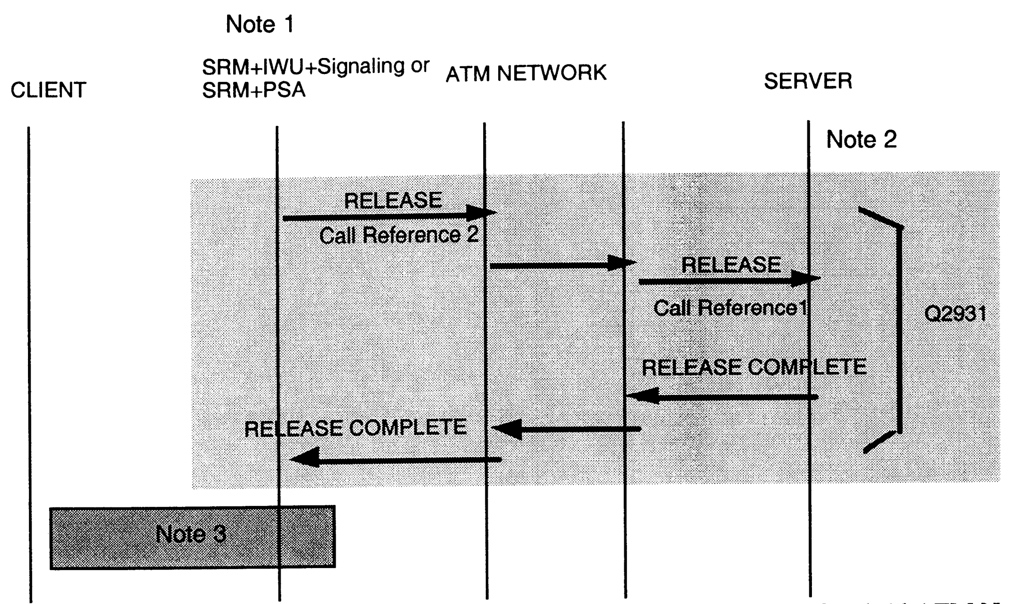


Figure D-20 Network Method with no AddResource messages: Server-initiated Resource Deletion for scenario 3(c)

Network Signaling in SRM, with "join" capabilities in SRM (Notes and Step descriptions same as for previous figure).

D.5.3.2 Connection Clearing by the Client

The Client can delete an established ATM SVC connection by initiating the Q.2931 Connection Clearing procedure through SRM:



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. See signaling part of Figure D-6 for message flow for scenario 3(c).

Figure D-21 Network Method: Resource Deletion via Client-initiated Call/Connection Release

When the Server receives the RELEASE message on its UNI, it keys off the Call Reference to retrieve the sessionId.

When the SRM receives the RELEASE COMPLETE message on its UNI, it keys off the Call Reference to retrieve the sessionId.

The SRM then closes the usage record of the resource associated with this Call Reference.

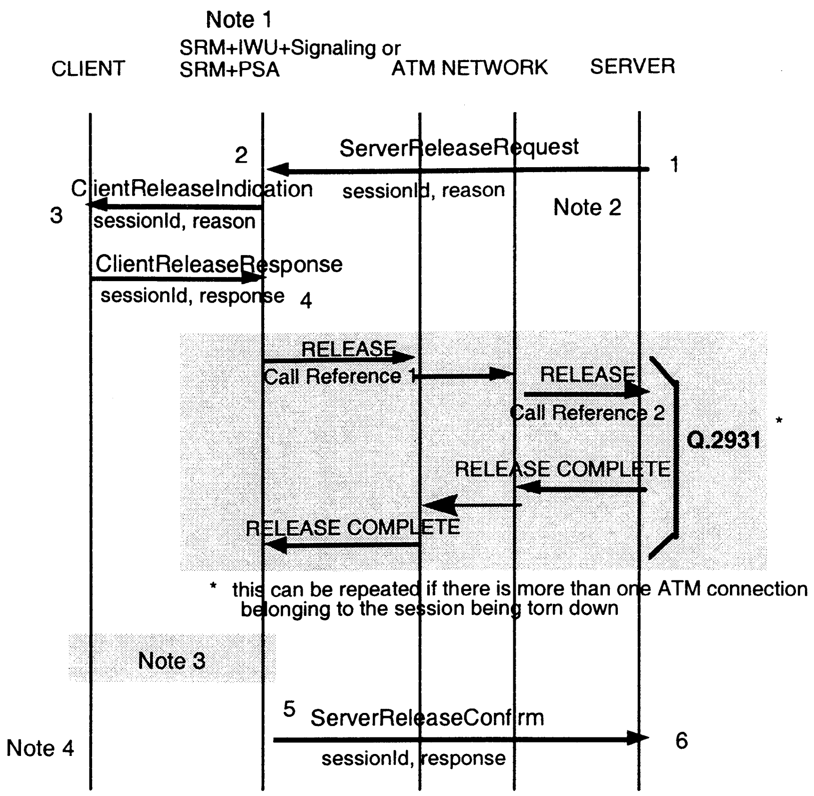
D.5.4 Session Tear-Down

A session can only be torn down through the DSM-CC Session Tear-Down scenario, initiated either by the Client, Server or SRM. 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 SRM shall delete all the resources associated with the session.

D.5.4.1 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 subclause D.3.4.1 Session Tear-Down by Server.



Note 1: Optional role of the IWU is to manage the access network resources in the case of Hybrid ATM Network - MPEG TS

Note 2: Only relevant parameters in each message are shown.

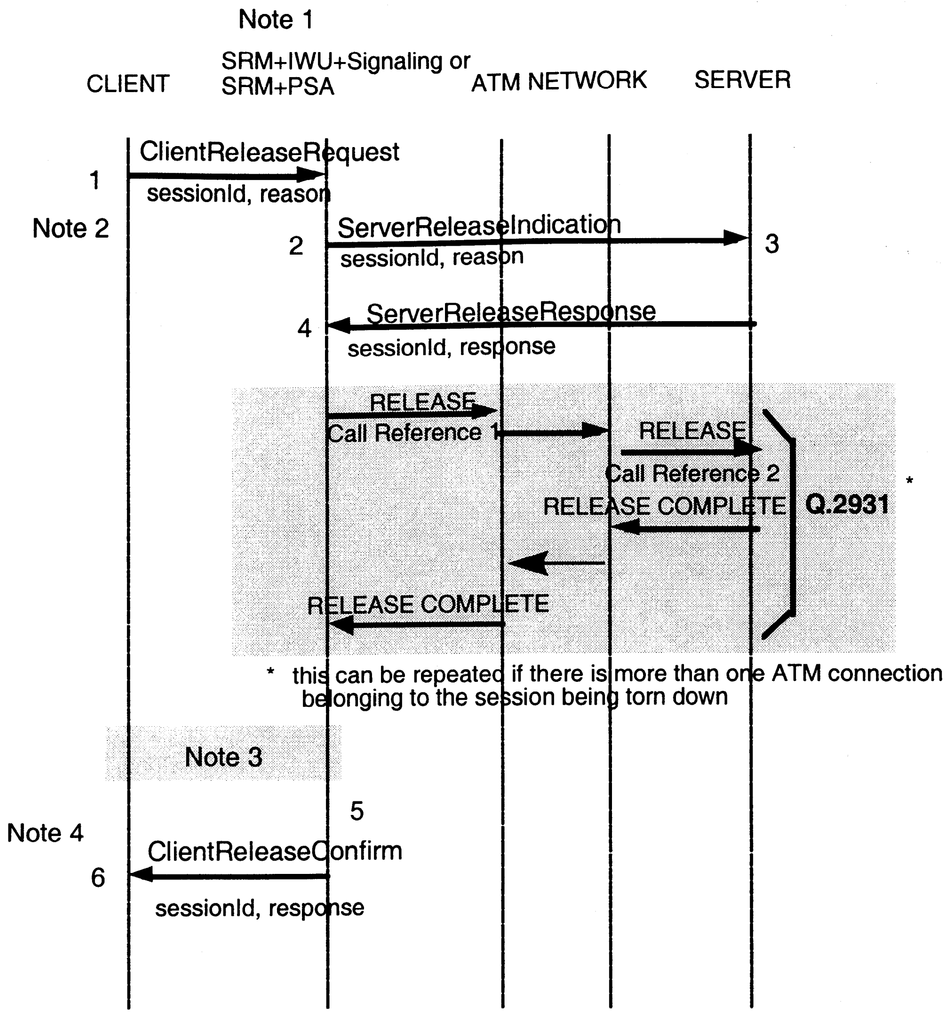
Note 3: Connection Control messages with Client will be specific to the type of access network. See signaling part of Figure D-8 for message flow for scenario 3(c).

Figure D-22 Network Method: Server Session Tear-Down using Session Method

Steps are as in subclause D.3.4.1 Session Tear-Down by Server.

D.5.4.2 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 subclause D.3.4.2 Session Tear-Down by Client.



Note 1: Optional role of the IWU is to manage the access network resources in the 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. See signaling part of Figure D-9 for message flow for scenario 3(c).

Figure D-23 Network Method: Client Session Tear-Down using Session Method

Steps are as in subclause D.3.4.2 Session Tear-Down by Client.

D.6 Integrated Method Command Sequences

Note: Not all the signaling elements to satisfy all scenarios of Integrated DSM-CC operation with ATM are presently available. The complete set of scenarios will be described however in order to promote discussion and potential adoption of the appropriate mechanisms in the B-ISDN signaling standards.

With the Integrated Method, the DSM-CC messages are mapped into Q.2931 signaling messages. The SRM “sees” the signaling, and thus manages both the DSM-CC sessions and resources.

Other DSM-CC U-N messages such as ClientStatusRequest that are not naturally coupled with Q.2931 connection signals can be handled in Q.2932 facility messages. [2]

Additional Extensions required to Q.2931 that will allow DSM-CC integration with the ATM network:

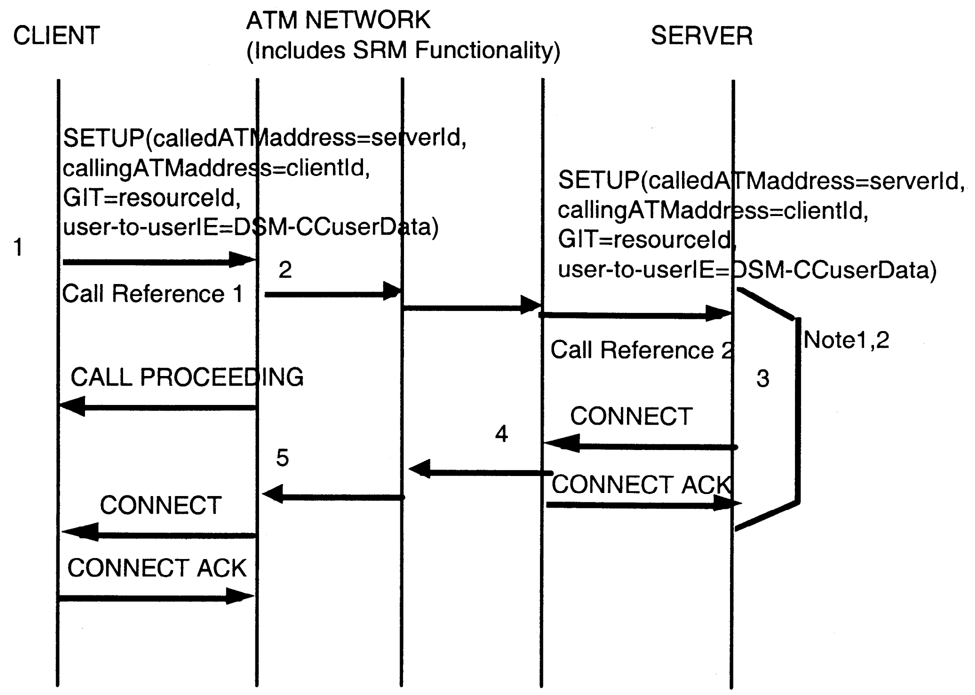
1. Use user-to-user Information Element (Q.2957 [5]) in the Q.2931 SETUP message if userData is to be carried in the message. Optionally the userData field can be conveyed in the ATM connection being set up by the SETUP message, similar to ITU-T Recommendation H.245 [6].

2. Create a DSM-CC Information Element as part of the FACILITY message in draft recommendation Q.2932 [2] for DSM-CC status messages and for session release messages.

D.6.1 Session Set-Up

The message flows for integrated Session Set-Up are the same those for the Session Method (see subclause D.3.1.1 Client Session Set-Up), with the enhancement that the Q.2931 signaling messages here contain all information that previously had to be carried separately, in the DSM-CC SessionSetUp and AddResource messages).

D.6.1.1 Client Session Set-Up


CLIENT
ATM NETWORK (Includes SRM Functionality)
SERVER
SETUP(calledATMaddress=serverId, callingATMaddress=clientId, GIT=resourceId, user-to-userIE=DSM-CCuserData)
1
2
Call Reference 1
SETUP(calledATMaddress=serverId, callingATMaddress=clientId, GIT=resourceId, user-to-userIE=DSM-CCuserData)
Call Reference 2
Note1,2
3
CALL PROCEEDING
CONNECT
4
5
CONNECT ACK
CONNECT
CONNECT ACK

Note 1: The Client uses a standard default QoS for the initial connection set up for the session.
Note 2: SETUP messages carry DSM-CC sessionId+resourceNum in Generic Identifier Transport Information Element field.

Figure D-24 Integrated Method: Client Session Set-Up

Step 1

The Client allocates a **sessionId** and requests a session SETUP with one connection. This connection may become the general end-to-end data connection over which DSM-CC non-Network resources can be managed,. similar to ITU-T Recommendation H.245 [6]. The **serverId** is the Q.2931 Called Party Number and the **clientId** is the Q.2931 Calling Party Number.

Step 2

The Network verifies the **clientId** from the Network's point of view, and if positive, allows the SETUP message to be sent to the Server identified by **serverId**, which must be in the form of an ATM network address.

Step 3

Upon receipt of the session SETUP the Server verifies the **clientId** from the Server's point of view, and if positive accepts the session and the one connection. By sending the CONNECT signal the Server shall indicate its acceptance of the session.

Step 4

Through the CONNECT signal the Server will indicate acceptance of the session.

Step 5

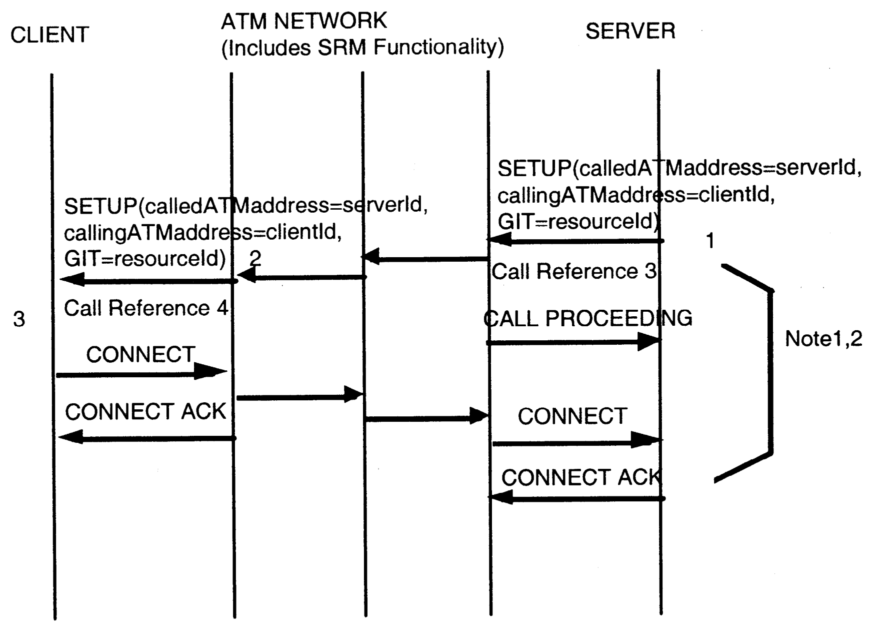
The Network notes the start of the session and by passing the CONNECT to the Client it informs the Client of the start of the session.

D.6.1.2 Server Session Set-Up

The Server Session Set-Up is symmetrical to the Client Session Set-Up described in subclause D.6.1.1 Client Session Set-Up.

D.6.2 Integrated Method for Adding Resources

The message flows for integrated addition of resources are the same those for the Add Resource Session Method (see subclause D.3.2 Add Resource Request, with the enhancement that the Q.2931 signaling messages here contain all information which previously had to be carried separately, in the DSM-CC Add Resource messages. The figure below shows the example of an integrated Add Resource request by the Server.


CLIENT
ATM NETWORK
(Includes SRM Functionality)
SERVER
SETUP(calledATMaddress=serverId,
callingATMaddress=clientId,
GIT=resourceId)
1
Call Reference 3
CALL PROCEEDING
Note1,2
SETUP(calledATMaddress=serverId,
callingATMaddress=clientId,
GIT=resourceId)
2
3
Call Reference 4
CONNECT
CONNECT ACK
CONNECT
CONNECT ACK

Note 1: Q.2962/3 [3, 4] may be used for resource negotiation/re-negotiation
Note 2: SETUP messages carry DSM-CC sessionId+resourceNum in Generic Identifier Transport Information Element field.

Figure D-25 Integrated Method: ServerAddResource

Step 1

When the Server wants to add a resource that requires an ATM connection it issues a SETUP message that includes the sessionId + resourceNum in the Generic Transport Identifier (GIT).

Step 2

The Network notes the additional connections requested for the Session and proceeds by sending the SETUP message to the Client.

Step 3

The Client accepts the resource with a CONNECT.

D.6.3 Connection Clearing

The message flows for integrated Connection Clearing are the same those for the Session Method Delete Resource (see subclause D.3.3 Resource Deletion), with the enhancement that the Q.2931 signaling messages here contain all information that previously had to be carried separately, in the DSM-CC Delete Resource messages. The figure below shows the example of an Connection Clearing request by the Server.

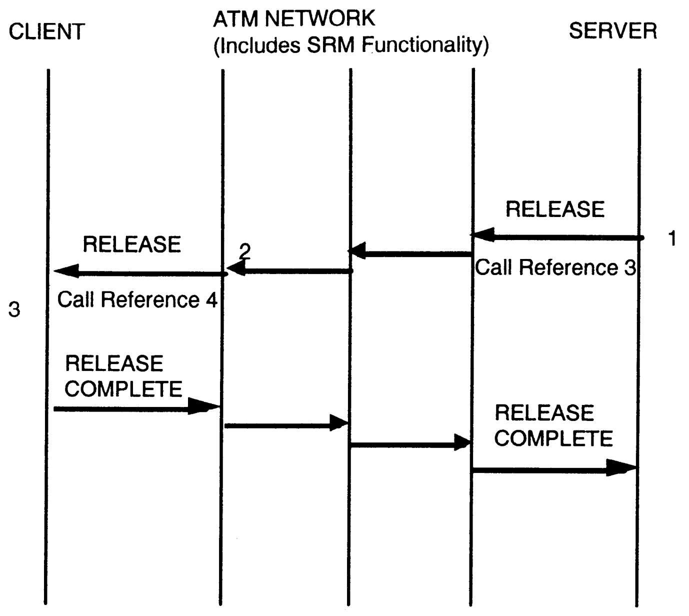


Figure D-26 Integrated Method: Connection Clearing

Step 1

RELEASE clears a connection for the Server.

Step 2

The Network notes the connection clearing and proceeds by sending the RELEASE message to the Client.

Step 3

The Client accepts the connection clearing with a RELEASE COMPLETE message.

D.6.4 Session Tear-Down

All connections can be released while a session may still be maintained. In this case, the message flows to tear down the session are the same as those for the Session Method, in subclause D.3.4 Session Tear-Down, but will be encapsulated in Q.2932 Facility messages for the Integrated Method.

D.6.4.1 Server Session Tear-Down

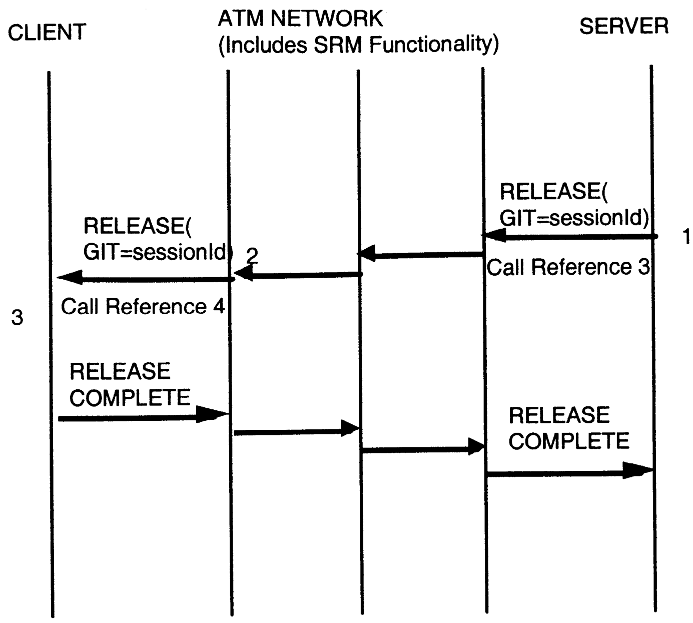


Figure D-27 Integrated Method: Server Initiated Session Tear-Down

Step 1

RELEASE clears a connection for the Server. To indicate that this RELEASE releases the session, the sessionId is sent in the GIT of the RELEASE message. At this point the Server considers the session terminated.

Step 2

The Network notes the session clearing and proceeds by sending the RELEASE message to the Client. At this point the Network considers the session terminated.

Step 3

The Client accepts the session clearing with a RELEASE COMPLETE message. At this point the Client considers the session terminated.

D.6.4.2 Client Session Tear Down

Client Session tear-down is symmetrical to Server Session tear-down.

D.7 References

[1] ITU-T Recommendation Q.2931 (02/95), *Digital Subscriber Signalling System No. 2 – User-network interface (UNI) layer 3 specification for basic call/connection control.*

[2] ITU-T Recommendation Q.2932.1 (7/96), *Digital Subscriber Signalling System No. 2 – Generic functional protocol: Core functions.*

[3] ITU-T Recommendation Q.2962 (07/96), *Digital subscriber signalling system No. 2 – Connection characteristics negotiation during call/connection establishment phase.*

[4] ITU-T Recommendation Q.2963.1 (07/96), *Digital Subscriber Signalling System No. 2 – Connection modification: Peak cell rate modification by connection owner;* and
ITU-T Recommendation Q.2963.2 (09/97), *Digital Subscriber Signalling System No. 2 – Connection modification: Modification procedures for sustainable cell rate parameters.*

[5] ITU-T Recommendation Q.2957 (02/95), *Stage 3 description for additional transfer supplementary services using B-ISDN digital subscriber Signalling System No. 2 (DSS 2) – Basic Call.*

[6] ITU-T Recommendation H.245 (07/97), *Control protocol for multimedia communication.*

[7] Digital Audio-Visual Council, *DAVIC Specification 1.3, Part 4: Delivery System Architecture and Interfaces and Part 12: System Dynamics, Scenarios and Protocol Requirements,* December 1997.

[8] ITU-T Recommendation H.310 (11/96), *Broadband audiovisual communications systems and terminals.*

Annex E
(informative)
UNO INTER-OPERABLE RPC PROTOCOL STACK

E.1 Abstract

The annex recommends an inter-operable protocol stack for the presentation, session, transport, and network layers of a the protocol stack. 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 layers.

E.2 Motivation

If a client and a service distribute across a network, issues regarding inter-operation arise. A concept which frames these issues 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 inter-operation. While multiple distinctions between domains are valid, the issues addressed within this annex are:

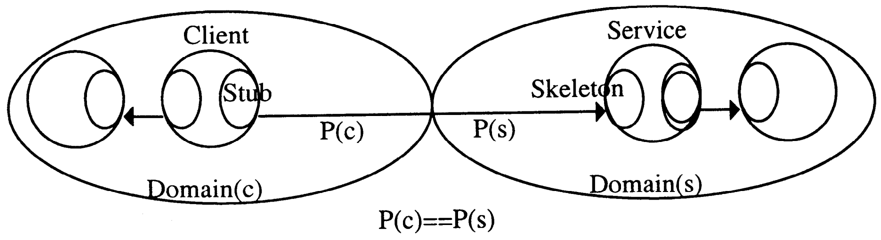
Type Domain: The objective is to encode invocation signatures which are intelligible to multiple domains. The implication is that a domain which wishes to inter-operate with another domain must provide typecode space which the domain understands. A typecode is a Interface Definition Language construct which encodes an interface name into a concrete value. A domain which wishes to inter-operate with another domain must either adopt the same typecode, or there must be mechanism which can translate the typecode values.

Protocol Domain: Since protocol stacks may be diverse, inter-operation may be difficult. For inter-operation to be feasible, either a domain must share the same native protocol stack with another domain, or there must be mechanism to translate the protocol stack. The solution, in the second case, constitutes a protocol gateway.

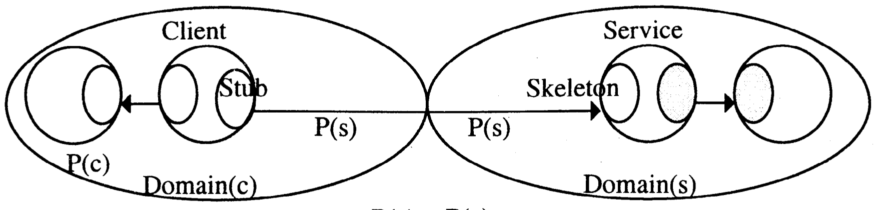
Before a solution to these issues is discussed, subclause E.3 first explores the solution space. The framework allows nodes to detect the situation where domains share a common native protocol stack, but still converge if a more subtle solution exists.

E.3 Solution Space

The spectrum of design options is extensive and is discussed below. Subclause E.4 then describes a framework which, while it anticipates diverse design centers, also provides the mechanism to converge to an inter-operable solution.

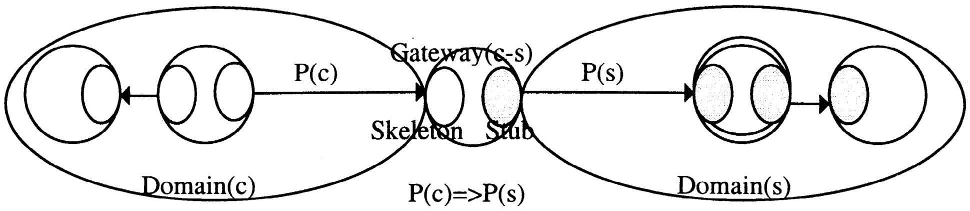


In the example above, a client native protocol, P(c), matches the service native protocol, P(s). The solution should allow a node to detect that, although the node and another node reside in distinct domains, the nodes share the native protocol, and can retain the native protocol to inter-operate.



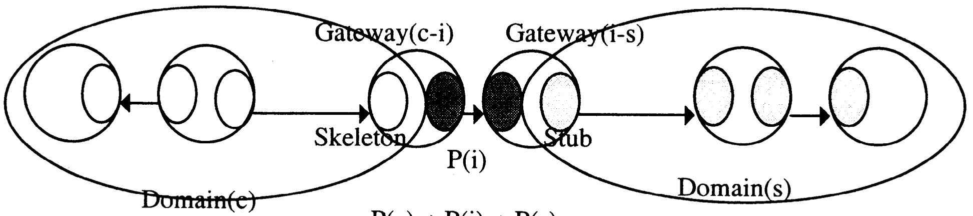
P(c)==P(c)

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) is 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.



P(c)=>P(s)

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 visible to neither the client at the application level nor to the service at the application level.



P(c)=>P(i)=>P(s)

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 to a protocol which a domain supports for inter-domain inter-operation, it is feasible to cascade a gateway which translates into the canonical protocol to the gateway which translates from the canonical protocol in order to achieve inter-operation. The solution does not mandate that the protocol within either domain adopts the canonical protocol.

E.4 Inter-operation Framework

To address the solution for the issues raised above, the use of the Universal Network Object solution of the Object Management Group is recommended. Subclause E.5, Protocol Selection, describes the framework which allows the design options noted above and provides the interface to converge to a solution. Subclause E.6, Common Data Representation, describes conventions at the presentation level that encode data structures, which result from the compilation of the Interface Definition Language into a concrete message payload. If the client side and a service adopt the conventions, the client operation signature and the service operation signature are consistent. Subclause E.7, UNO Session Protocol, describes the message set of the session level. It realizes remote procedure call semantics. Subclause E.8, Transport and Network Semantics, describes the semantics at the transport and service levels.

E.5 Protocol Selection

The inter-operation 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. The profile describes the protocol decisions which the native domain must adopt to inter-operate with the remote domain. The interface on which the solution builds is:

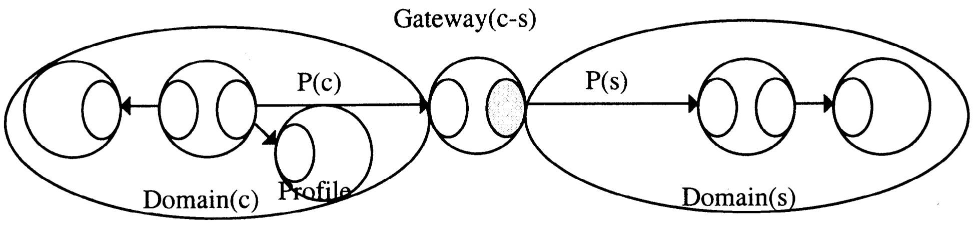
```
typedef unsigned long  ProfileId;
const ProfileId                     TAG_INTERNET_IOP = 0;
struct TaggedProfile {
      ProfileId          tag;
      sequence<octet>  profile_data;
};
```

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         iiop_version;
      string          host;
      unsigned short   port;
      sequence<octet>  object_key;
};
```

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 behind the network port.



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 inter-operate without a protocol gateway. If the client native protocol and the service native protocol do not match, the solution requires the interposition of a protocol gateway. The client can find the gateway through whatever is the convention for the client domain. In the case of DSM-CC, 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.

E.6 Common Data Representation

The inter-operation 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 inter-operation solution are known as the Common Data Representation. This 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.

E.6.1 Encapsulation

The inter-operation 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 the 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.

E.6.2 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 within another encapsulation.

E.6.3 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

E.6.4 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 ordered 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 by 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 (ISO 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.

E.6.5 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)}

TCKIND	VALUE	TYPE	PARAMETER
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-	0xFFFFFFFF	simple	long(indirection)

A complete description of the TypeCode conventions is beyond the scope of this annex. For details, refer to the Universal Network Object specification of the Object Management Group.

E.7 UNO Session Protocol

The inter-operation solution requires certain semantics for the UNO session protocol. The premise is that the session protocol realizes a remote procedure call. The discussion of this subclause will focus on the message set which a client and a service exchange.

E.7.1 Message Set

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 Version below, differ to aid comprehension).

```
struct Version {
      char              major;
      char              minor;
};
```

```
struct MessageHeader {
      char            magic[4];
      Version          GIOP_version;
      boolean         byte_order;
      octet           message_type;
      unsigned long  message_size;
};
```

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.

E.7.1.1 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 {
      IOP::ServiceContextList  service_contest;
      unsigned long            request_id;
```

```
        boolean                     response_expected;
        sequence<octet>             object_key;
        string                      operation;
        Principal                   requesting_principal;
};
```

The interpretation of each field is:

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                   context_id;
        sequence<octet>             context_data;
};
typedef sequence<ServiceContext> ServiceContextList;
```

RequestId: The field allows the remote procedure software on the client side to scoreboard results.

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.

ObjectKey: The field identifies the target of the invocation. The description of the Internet Inter-Operation Protocol describes the data found in the sequence<octet> field.

Operation: The field specifies the operation name.

Principle: The field relates to the Principle concept. A principal is a trusted object on which authentication techniques build.

E.7.1.2 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 {
        IOP::ServiceContextList     service_context;
        unsigned long               request_id;
        ReplyStatusType             reply_status;
};
```

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 inter-operable 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.

E.7.1.3 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 CancelRequestHeader {
      unsigned long     request_id;
};
```

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.

E.7.1.4 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     request_id;
      sequence<octet> object_key;
}:
```

E.7.1.5 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           request_id;
      LocateStatusType        locate_status;
};
```

E.7.1.6 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.

E.7.1.7 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.

E.7.2 Session Semantics

The session layer, in combination with the transport layer and the network layer, 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.

E.8 Transport and Network Semantics

The objective of the solution is to be implementable on a wide range of transport protocols. The inter-operation protocol, however, does expect certain semantics at the transport layer and at the network layer as described below.

- 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, or alignment.
- Reliable Transport: The transport is to be reliable. The transport is to guarantee that the target of the message receives the byte stream in the order in which it was sent, at most once, and that the source of the message receives a 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 not specified) and Internet Protocol (Version 4.0 to Version 6.0).

Annex F
(informative)
Use of U-U Object Carousel

F.1 Introduction

Normative clause 11 contains the U-U Object Carousel specification. This informative Annex provides additional information about the use of U-U Object Carousels.

F.2 Purpose of U-U Object Carousels

The U-U Object Carousel specification makes it possible to deliver U-U Objects in a broadcast network in an inter-operable way. The specification has been designed in such an way that implementations can easily offer applications a U-U API to access the objects broadcast. Consequently, implementations can provide a single API to applications to access objects that are delivered to the Client either by interactive networks or by broadcast networks. Figure F-1 illustrates this functionality by showing the protocol stacks for both kinds of networks.

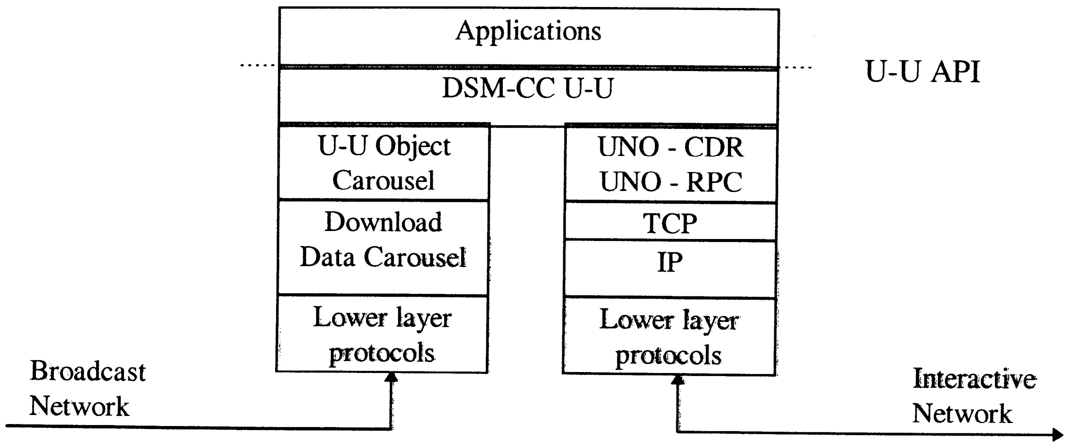


Figure F-1 Protocol stacks for both Broadcast and Interactive Networks

Given the protocol stack for interactive networks, the U-U Object Carousel specification was designed in such a way that a maximum coherency exists with OMG-UNO (GIOP and IIOP). In broadcast networks, one Server may serve many Clients with different architectures. Therefore, a representation protocol is necessary that specifies how the U-U objects are carried on the wire. In Interactive networks, the object data is transported via the Internet Inter ORB Protocol (IIOP) on top of TCP/IP. In IIOP, the bits on-the-wire are defined by Common Data Representation (CDR) to make an exchange of objects between ORBs with different architectures possible. In order to avoid having two different representation protocols in hybrid Clients, U-U Object Carousels should also make use of CDR or related standards such as CDR Lite. CDR Lite has been chosen as the default data encoding standard because of efficiency reasons.

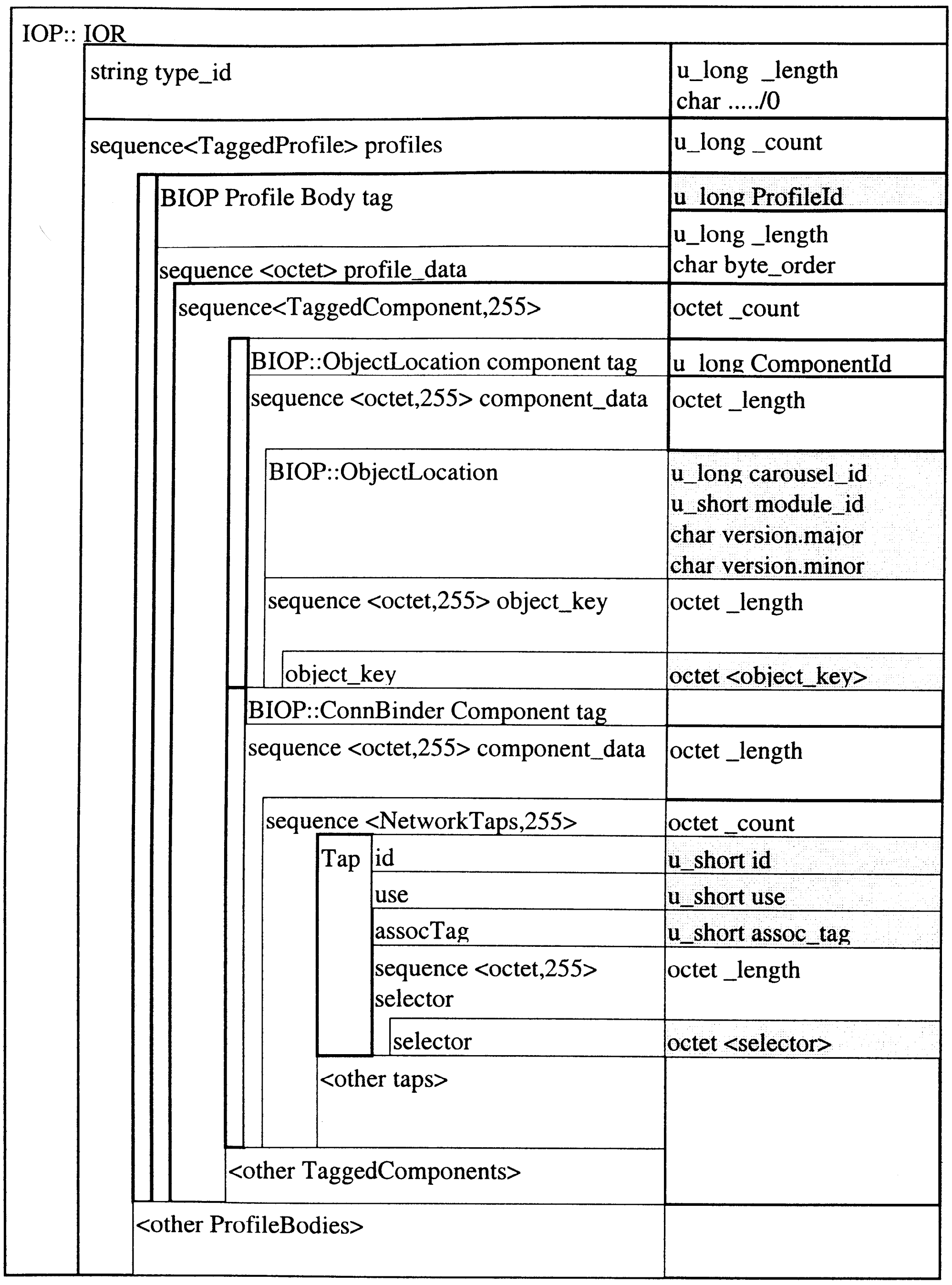
F.3 IDL structures

In the U-U Object Carousel specification, a number of IDL structures are given. In this subclause, the structure of the IOR, the Generic object Message, and the Directory Message are illustrated.

F.3.1 Inter-operable object Reference

BIOP uses the Inter-Operable Reference format (IOR) defined by OMG for object References. An IOR contains Profile Bodies that encapsulate all the basic information that a particular protocol stack needs to identify the object. A single Profile Body holds enough information to drive a complete invocation of the object using that protocol.

The Profile Body defined by BIOP has the DSM::LiteComponentProfile structure. The structure of the IOR with the BIOP ProfileBody inside is shown below. The right column shows the encoding of the fields using CDR-Lite. The shaded fields indicate that the fields are described in clause 11.



IOP:: IOR	
string type_id	u_long _length char ...../0
sequence<TaggedProfile> profiles	u_long _count
BIOP Profile Body tag	u_long ProfileId
sequence <octet> profile_data	u_long _length char byte_order
sequence<TaggedComponent,255>	octet _count
BIOP::ObjectLocation component tag	u_long ComponentId
sequence <octet,255> component_data	octet _length
BIOP::ObjectLocation	u_long carousel_id u_short module_id char version.major char version.minor
sequence <octet,255> object_key	octet _length
object_key	octet <object_key>
BIOP::ConnBinder Component tag	
sequence <octet,255> component_data	octet _length
sequence <NetworkTaps,255>	octet _count
Tap id	u_short id
use	u_short use
assocTag	u_short assoc_tag
sequence <octet,255> selector	octet _length
selector	octet <selector>
<other taps>	
<other TaggedComponents>	
<other ProfileBodies>	

Figure F-2 The structure of the IOR with the BIOP ProfileBody inside

The BIOP Profile Body shall contain at least the LiteComponent BIOP::ObjectLocation and the LiteComponent DSM::ConnBinder. The BIOP::ObjectLocation component uniquely locates the object within the broadcast network. The DSM::ConnBinder component contains a number of Taps that point to DownloadInfoIndication() messages that describe the delivery parameters of the Module in which the object is conveyed. Optionally, Taps may be included that point directly to the connections on which the Modules are delivered. This option facilitates the use of aggressive acquisition procedures that start acquiring Blocks from the network before the actual Module attributes (like size) are known.

F.3.2 Generic object Message

The generic object message format consist of a header, a sub-header and a body. The structure of the Generic object Message is shown below. The right column shows the encoding of the fields using CDR Lite. The shaded fields indicate that the fields are described in clause 11.

BIOP::GenericObjectMessage				
	BIOP::MessageHeader	magic[4]		char magic[4]
		biop_version		char version.major char version.minor
		byte_order		char byte_order
		message_type		octet message_type
		message_size		u_long message_size
	BIOP::MessageSubHeader	objectKey		octet __length
				octet <objectKey>
		objectKind		long _length
				string <objectKind>
		objectInfo		short __length
				octet <objectInfo>
		serviceContext List		octet _length
			Service Context	u_long Service_Id
				u_short _length
				octet <context_data>
	sequence <octet>	messageBody		u_long length
				octet <data >

Figure F-3 The structure of the Generic object message

F.3.3 Directory Message

The BIOP Directory message is instantiated from the generic object format. Hence, the Directory message has the same format but some fields are further detailed. In particular, the messageBody contains the DirectoryMessageBody structure which is shown below. The right column shows the encoding of the fields using CDR Lite. The shaded fields indicate that the fields are described in clause 11.

BIOP::DirectoryMessageBody				
	sequence <Binding,65535>			u_short _length
		sequence <NameComponent,255>		octet __length
		name	NameComponent	octet __length
				string id
				octet _length
				string kind
			<other NameComponent>	
		binding_type		octet binding_type
		object_ref		IOP::IOR object_ref
		sequence <octet,65535> object_info		u_short _length
			object_info	octet <object_info>
	<other Binding>			

Figure F-4 BIOP Directory message

The body of the Directory message consists basically of a loop of 'Bindings'. A binding links an object name to a particular IOR and provides additional information about the Object.

F.4 Support for New Object Representations

The generic object message format can be instantiated for any kind of objects. To instantiate the generic object message format into a dedicated object message, the semantics of the objectInfo and messageBody fields have to be defined. The objectInfo field is intended to carry the attributes of the Object, while the messageBody is intended to carry the data of the Object. As described in clause 11, the instantiated Directory message also has an objectInfo field that may be used to carry the attributes of the bound object. By carrying the proper attributes in that field, quick browsing through Directories and Object attributes is supported. Hence, the instantiation of the generic object message format for a particular objectKind should also specify which attributes are carried in the objectInfo field of the (parent) Directory message.

As a illustration, this subclause defines a new object and illustrates how the object message format is created for this object. The object is a BIOP::StreamEvent which inherits the DSM::Stream interface and the DSM::Event interface. In a similar way, a BIOP::FileEvent object could be defined that inherits the DSM::File inherits and also the DSM::Event interface.

The BIOP::StreamEvent object inherits the standard DSM::Stream interface and the DSM::Event interface.

```
module BIOP {
    interface StreamEvent : DSM::Stream, DSM::Event {
    }
};
```

The resulting BIOP::Stream event message now has the following attributes. From the Stream interface, it has inherited the DSM::Stream::Info_T attribute, while from the Event Interface, it has inherited the DSM::Event::EventList_T attribute. In addition, it has the Access attributes. To instantiate a message format for this object, it is necessary to specify which attributes are transmitted and where. For the StreamEvent object, it is decided that the Access attributes are not transmitted and that the other attributes will be encapsulated in the objectInfo field from the object message, because there is no real advantage in having these attributes available in the (parent) Directory object. The Object representation of the StreamEvent interface consists of the sequence of Taps and the list of eventId's that are associated with the names published in the EventList_T attribute. Hence, the following rules define the instantiation of the BIOP::StreamEvent Object:

1. The objectKind field shall contain the string "DSM::StreamEvent".
2. The Access attributes of the Stream Object are not encapsulated in either the objectInfo field of the File message nor the objectInfo field of the (parent) Directory message. The DSM::Stream::Info_T and EventList_T attributes shall be inserted at the beginning of the objectInfo field of the Object message.
3. The messageBody field shall contain the BIOP::StreamEventMessageBody structure. The syntax and semantics of the BIOP::StreamEventMessageBody are defined below:

```
module BIOP {
    struct StreamEventMessageBody {
        sequence <Tap,255>                  stream;
        sequence <unsigned short,255>       eventIdList;
    };
};
```

The semantics of the stream and eventIdList fields are defined below.

The **stream** field contains one or more taps that are associated with this stream object. Regarding the content of the stream, either one or more taps are present with a TapUse value of BIOP_ES_USE or one tap is present with a TapUse value of BIOP_PROGRAM_USE. In the first case, the stream consists of a number of elementary streams, while in the second case, the stream consists of an MPEG-2 Program.

The **eventIdList** contains the eventId's that are correlated to the event names published in the EvenList_T attribute. The sequence of the eventId's shall be equal to the sequence of the EventName's.

F.5 How to resolve an object from its IOR

BIOP uses the Inter-Operable Object Reference format (IOR) defined by OMG for object References. An IOR contains Profile Bodies that encapsulate all the basic information that a particular protocol stack needs to identify the object. A single Profile Body holds enough information to drive a complete invocation of the object using that protocol.

The BIOP Profile Body shall contain the LiteComponent BIOP::ObjectLocation and the LiteComponent DSM::ConnBinder. The BIOP::ObjectLocation component uniquely locates the object within the broadcast network. The DSM::ConnBinder component contains a number of Taps that point to DownloadInfoIndication() messages that describe the delivery parameters of the Module in which the object is conveyed. Figure F-5 explains in more detail how an object is resolved from its IOR.

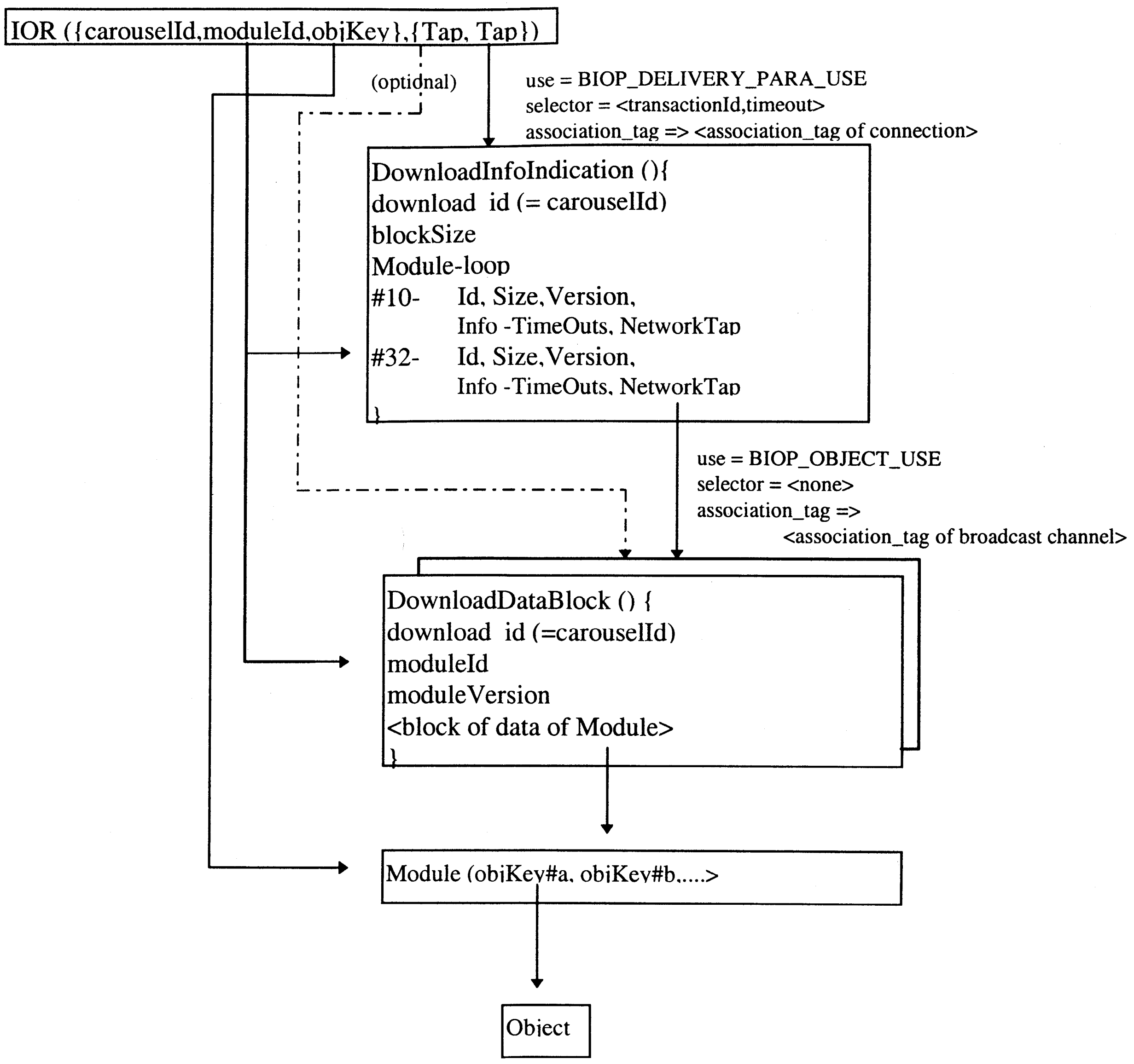

IOR ({carouselId,moduleId,objKey},{Tap, Tap})
(optional)
use = BIOP_DELIVERY_PARA_USE
selector = <transactionId,timeout>
association_tag => <association_tag of connection>
DownloadInfoIndication (){
download id (= carouselId)
blockSize
Module-loop
#10- Id, Size,Version,
Info -TimeOuts, NetworkTap
#32- Id, Size,Version,
Info -TimeOuts, NetworkTap
}
use = BIOP_OBJECT_USE
selector = <none>
association_tag =>
<association_tag of broadcast channel>
DownloadDataBlock () {
download id (=carouselId)
moduleId
moduleVersion
<block of data of Module>
}
Module (objKey#a, objKey#b.....>
Object

Figure F-5 How to resolve an object from its object reference.

The IOR of the object contains the BIOP Profile Body that contains, in turn, the LiteComponent BIOP::ObjectLocation and the LiteComponent BIOP::ConnBinder. The BIOP::ObjectLocation component uniquely locates the object within the broadcast network using the carouselId, moduleId, and objectKey identifiers. The DSM::ConnBinder component contains a number of Taps that point to DownloadInfoIndication() messages that describe the delivery parameters of the Module in which the object is conveyed.

As a first step to resolve the IOR, the Client has to acquire the DownloadInfoIndication() message that conveys the module delivery parameters of the Module. The Tap (or Taps) that points to this message includes a transaction_id field to identify the DownloadInfoIndication() message. The Tap also includes a time out value to time-out the acquisition process.

When the DownloadInfoIndication() message has been acquired, the Client searches in the Module description loop for the module 'moduleId'. The description of that Module includes three time out parameters that characterize the delivery of the module in time. The description also includes a Tap that points to the network channel on which the module is actually delivered. The blockSize of the blocks of the modules is indicated also in the DownloadInfoIndication() message.

Subsequently, the Client starts the acquisition of the Module in which the object is conveyed. In general, the Module will contain multiple objects which are related to each other at an application level. After the complete Module has been acquired, the requested object can be retrieved from the Module by inspecting the different object keys in the message sub-headers.

F.6 Service Gateway and Download support

Each U-U Object Carousel has a Service Gateway. The Service Gateway provides the root Directory of the content that is broadcast by this U-U Object Carousel. The IOR of the ServiceGateway shall contain the BIOP Profile Body. The BIOP Profile Body shall contain the LiteComponents BIOP::ObjectLocation and DSM::ConnBinder.

The IOR of the Service Gateway is broadcast by means of DownloadServerInitiate() messages. The use of the DownloadServerInitiate() messages for the carriage of the IOR of the ServiceGateway is such that DSM-CC Download (non-flow controlled scenario) can be employed as a part of the ServiceGateway attach functionality. Such Download phase could be used to download the required code necessary to access the U-U Object Carousel. For this functionality, the DownloadServerInitiate() message has to convey at least a set of Taps; one with a TapUse value of DOWNLOAD_CTRL_DOWN_USE and one with a TapUse value of DOWNLOAD_DATA_DOWN_USE. The first Tap points to the connection on which the DownloadInfoIndication() messages are delivered (these messages contain the descriptions of the modules that have to be downloaded). An implementation could use multiple Taps to signal the second Tap points to the connection on which the DownloadDataBlock() messages are to be delivered.

Figure F-6 illustrates how the relationship is between the DownloadServerInitiate() message and the DownloadInfoIndication() messages used for download. As illustrated, the Taps that point to the DownloadInfoIndication() message have a TapUse value of DOWNLOAD_CTRL_DOWN_USE. The selector field of the Tap contains the transaction_id of the DownloadInfoIndication() message and a timeout value to time-out the acquisition of the message. If necessary, multiple Taps with TapUse values of DOWNLOAD_CTRL_DOWN_USE may be carried in the DownloadServerInitiate() message for example to support multiple Client architectures. In this scenario, the differentiation between the Client architectures could be done using the compatibilityDescriptor() structure of the DownloadInfoIndication() messages. The other Tap directly points to the connection on which the DownloadDataBlock() messages are being broadcast.

The semantics of the DownloadInfoIndication() messages that are used to support the non-flow controlled Download phase are defined in clause 7. Note that after the Download phase, the resolve operation of the IOR of the ServiceGateway is as described previously.

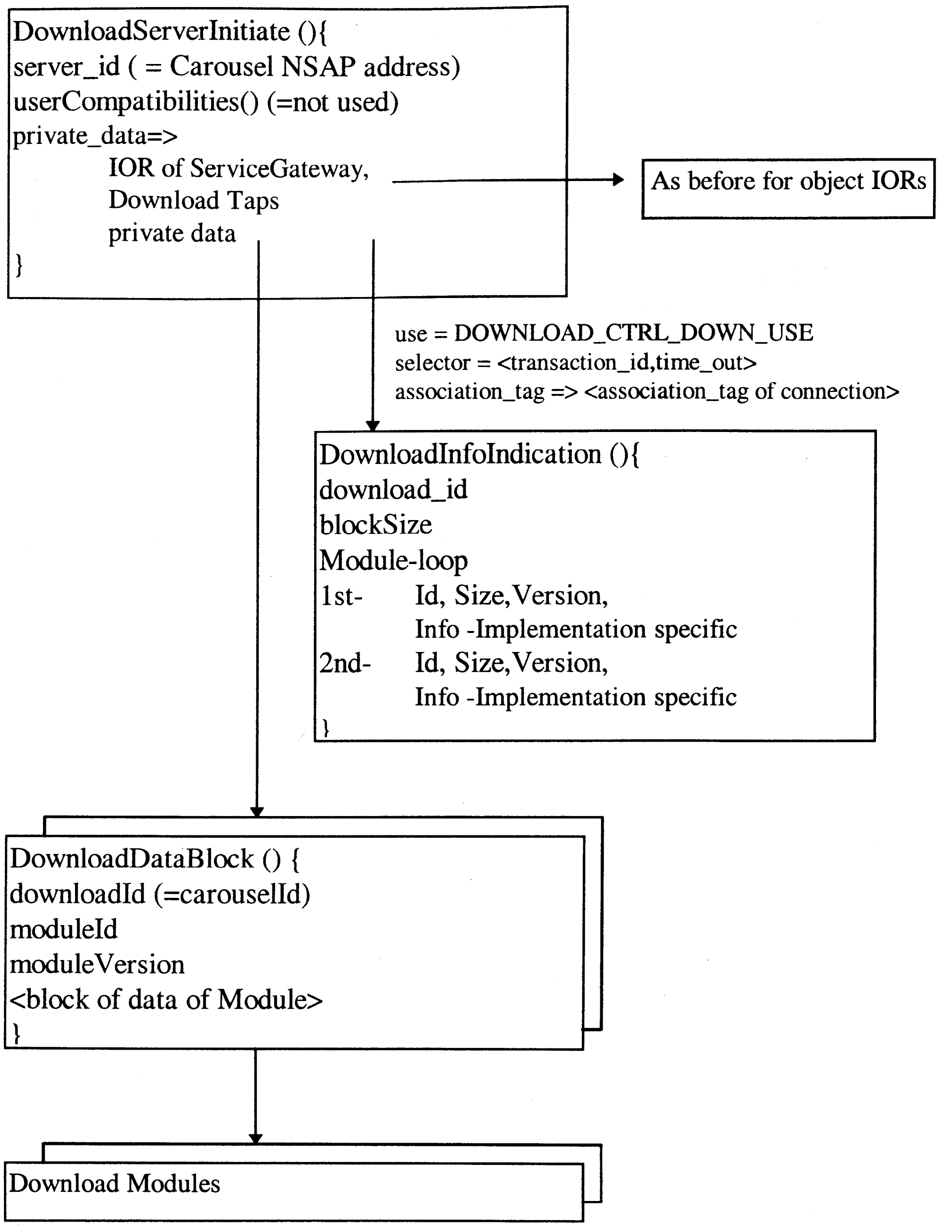

DownloadServerInitiate (){
server_id (= Carousel NSAP address)
userCompatibilities() (=not used)
private_data=>
IOR of ServiceGateway,
Download Taps
private data
}
As before for object IORs
use = DOWNLOAD_CTRL_DOWN_USE
selector = <transaction_id,time_out>
association_tag => <association_tag of connection>
DownloadInfoIndication (){
download_id
blockSize
Module-loop
1st- Id, Size,Version, Info -Implementation specific
2nd- Id, Size,Version, Info -Implementation specific
}
DownloadDataBlock () {
downloadId (=carouselId)
moduleId
moduleVersion
<block of data of Module>
}
Download Modules

Figure F-6 Relationship between DownloadServerInitiate() and Download Taps

F.7 U-U Object Carousels on top of MPEG-2 TS Broadcast Networks

The BIOP specification is network independent and is applicable for any type of broadcast network. Network independence is achieved by using the Tap concept of clause 5. A Tap facilitates a reference to a particular network connection by means of an association tag. Obviously, in the course of resolving an object, Clients have to associate the Taps to the connections of the broadcast network. Clients need, therefore, an association table that makes the associations between the Taps and the connection of the broadcast network.

The implementation of U-U Object Carousels on top of broadcast networks that are based on MPEG-2 Transport Streams is supported by the U-U object Specification by three descriptors. These descriptors facilitate i) the association of a MPEG-2 Program (i.e., PMT) with a U-U Object Carousel, ii) the association of a tap with a PID, iii) the localisation of the bit stream identified by the PID on which the IOR of the Service Gateway is broadcast, and iv) the distributed implementation of a U-U Object Carousel on top of multiple MPEG-2 Programs.

The last feature is extremely useful when one U-U Object Carousel is broadcast using multiple transponders and, thus, multiple Transport Streams. This requires the use of one dedicated MPEG-2 program per Transport Stream. Figure F-7 illustrates how one U-U Object Carousel can be implemented using multiple MPEG-2 programs and shows simultaneously how the different association_tag's are related.

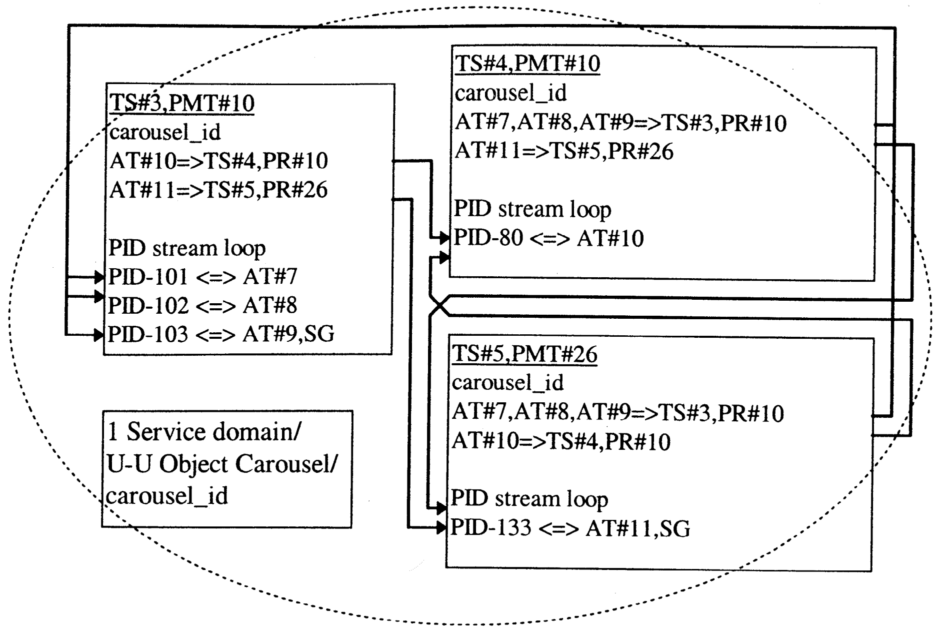


Figure F-7 Distributed implementation of U-U Object Carousel using multiple MPEG-2 Programs.

In Figure F-7, three MPEG-2 programs are used to implement the U-U Object Carousel. Each PMT exists in another Transport Stream and has the carousel_identifier_descriptor() included. This unambiguously associates the PMT with the U-U Object Carousel. Transport Streams are denoted as TS.

Each PID listed in the PMT has an associated assocation_tag_descriptor (denoted as AT). For example, the bit stream that is identified by PID#103 and associated with AT#9 is used to broadcast DownloadServerInitiate() messages that carry the IOR of the ServiceGateway.

The association tags that belong to PIDs that are part of other MPEG-2 Programs are described by deferred_association_tags_descriptors(). These descriptors defer the resolve operation of an association tag to another MPEG-2 program in another Transport Stream.

When a Client has to attach to a Service Gateway of a U-U Object Carousel, it should know at least the NSAP Carousel address that uniquely identifies the U-U Carousel with the broadcast network (see Service Domain in clause 11). Based on that information, the Client can find a PMT that belongs to the carousel and the PID which identifies the bit stream on which the DownloadServerInitiate() messages are broadcast. This PID has an association_tag with the use field set to 0x0000. Subsequently, the Client acquires the DownloadServerInitiate() message, performs a possible download, and subsequently resolves the IOR of the ServiceGateway. Subsequent references to other Taps can now always be resolved because all association tags are known relative to this PMT.

Annex G
(informative)
Shared Resources and the Association Tag

G.1 Introduction

This annex provides further clarification on the associationTag, the sharedResourceNumber and the sharedResourceRequestId in clause 4, U-N Session Messages. Various network types have been used as examples.

G.2 Use of the Association Tag

The association tag is used to indicate a horizontal association and also identify the descriptor ties. Figure G-1 provides an example of an ATM SVC connection which is mapped into a non-ATM HFC Client access network connection. The Server view of this resource has been assigned resourceNum=1 and associationTag=1. In this illustration, the non-ATM access network consists of an MPEG-2 Transport Stream (TS) in the downstream direction and TDMA in the upstream direction. Another associationTag, with the value 50, provides the descriptor association (a "vertical binding") between the TS and its associated MPEG Program descriptor; vertical bindings are not visible to the ServiceGateway interface or the User-to-User Library. The associationTag 1 is, however, visible to both interfaces and is maintained constant across the end-to-end connection.

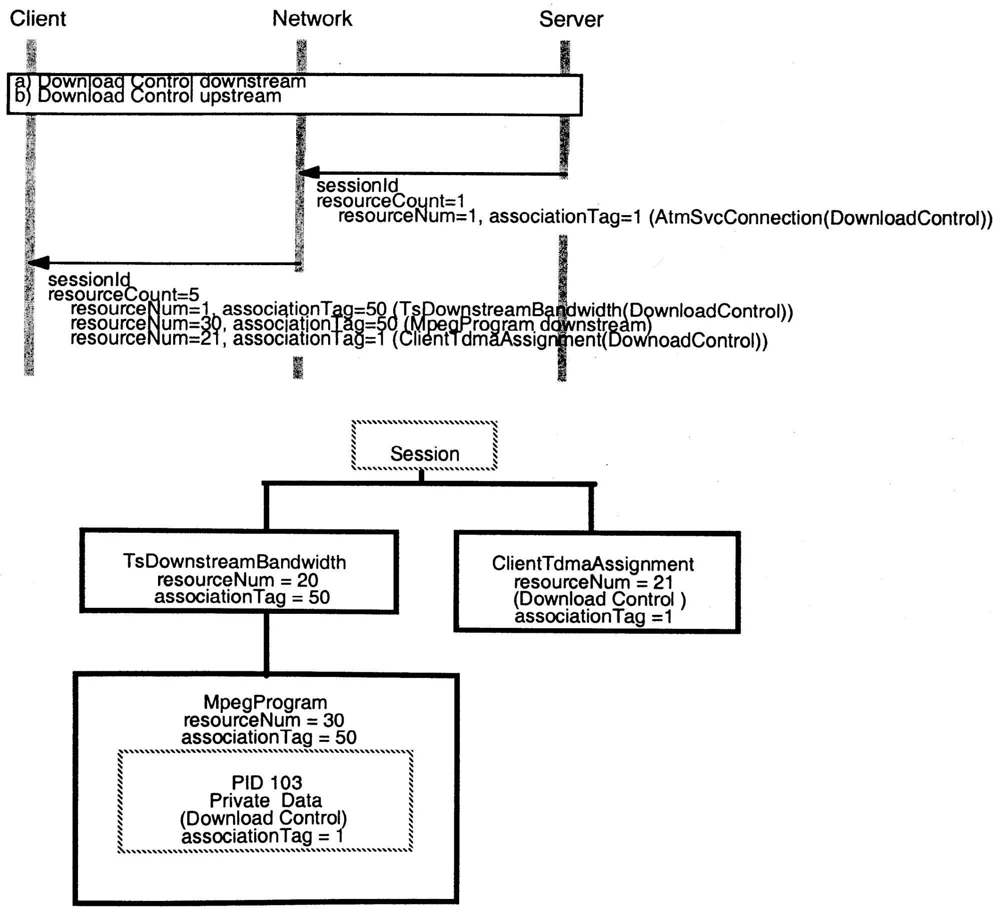

Client
Network
Server
a) Download Control downstream
b) Download Control upstream
sessionId
resourceCount=1
resourceNum=1, associationTag=1 (AtmSvcConnection(DownloadControl))
sessionId
resourceCount=5
resourceNum=1, associationTag=50 (TsDownstreamBandwidth(DownloadControl))
resourceNum=30, associationTag=50 (MpegProgram downstream)
resourceNum=21, associationTag=1 (ClientTdmaAssignment(DownoadControl))
Session
TsDownstreamBandwidth
resourceNum = 20
associationTag = 50
ClientTdmaAssignment
resourceNum = 21
(Download Control)
associationTag =1
MpegProgram
resourceNum = 30
associationTag = 50
PID 103
Private Data
(Download Control)
associationTag = 1

Figure G-1 Use of the associationTag to indicate horizontal association of different resources in the same end-to-end connection and stack nesting

G.3 Use of the SharedResource Descriptor

The sharedResource Descriptor is used when a number of resources are required to share a common resource. In Figure G-2, an AtmSvcConnection (with resourceNum 1) is shared by an RPC IP upstream resource (with resourceNum 12) as well as by an DownloadControl IP upstream resource (resourceNum 14). This occurs in cases where a single ATM Virtual Connection (VC) is assigned to MPEG, RPC and DownloadControl as shown in later examples. The sharedResource Descriptors, which have been assigned resourceNum 13 and 15, provide the linkage to the shared resource which has resourceNum 1. This is shown in Figure G-2 and Figure G-3 (left).

```
sessionId
resourceCount=7
    resourceNum=1, associationTag=0 (AtmSvcConnection)
    resourceNum=13, associationTag=1 (SharedResource(1))
    resourceNum=12, associationTag=1 (RPC upstream)
    resourceNum=15, associationTag=2 (SharedResource(1))
    resourceNum=14, associationTag=2 (DownloadControl upstream)
```

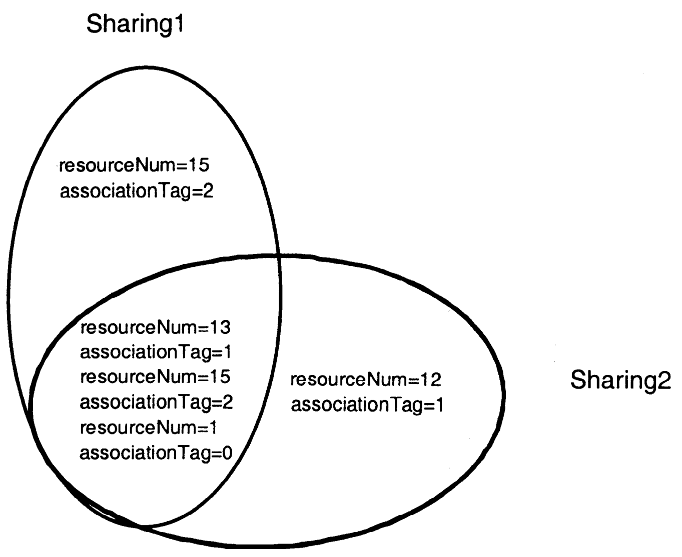


Figure G-2 Use of SharedResource Descriptor

G.4 Use of the SharedRequestId Descriptor

The sharedRequestId descriptor is used in place of the sharedResource descriptor in the case where the assignment of the resource numbers is done by the Network as opposed to being done by the originator of the resource request (a User). This case is shown in Figure G-3. A similar approach to that done in subclause G.3 is followed, except that instead of the sharedResourceNum descriptor being used to refer to a resourceNum, a sharedResourceRequestId descriptor is used to refer back to the resourceRequestId.

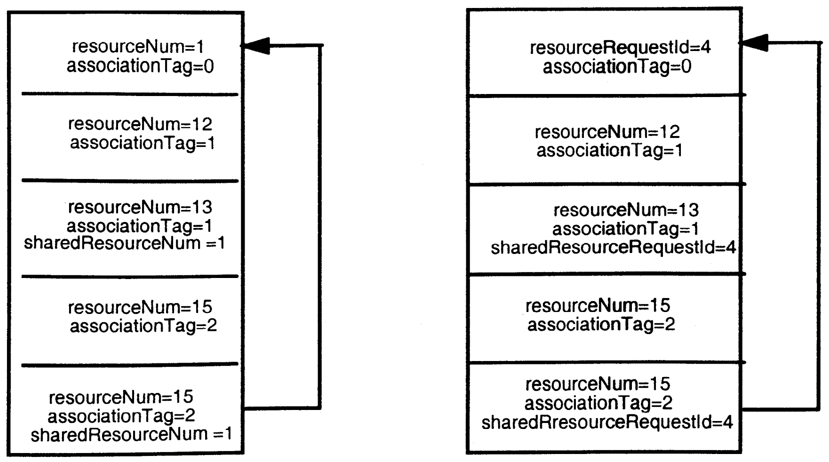


Figure G-3 Use of SharedRequestId Descriptor

G.5 Common Examples of Use

The following examples provide different realistic scenarios for connections indicating how both the Association Tags and the Shared Resource descriptors are used (associationTag, SharedResource Descriptor and SharedRequestId Descriptor):

- Download Phase, Multiple ATM SVCs
- Video Play Phase, Multiple ATM SVCs
- Single Asymmetric ATM SVC

G.5.1 Download Phase, Multiple ATM SVCs

This example uses separate ATM Switched Virtual Connections (SVCs) in order to carry the flows identified below:

1. Download Data downstream
2. Download Data Response upstream
3. Download Control downstream
4. Download Control upstream
5. RPC

Two scenarios are considered:

- End-to-End ATM.
  In this case the ATM SVC is maintained intact between the Server and the Client.

- Core ATM network and non-ATM HFC Client access.
  In this case the flows being carried on the ATM SVC are mapped to non-ATM resources through the non-ATM resource descriptors.

G.5.1.1 End-to-End ATM

The top diagram in Figure G-4 provides the list of resource descriptors communicated from the Server to the Network and the Network to the Client. Since the system has an end-to-end ATM network, the resource numbers and the association tags are unchanged.

In the example shown in the bottom diagram in Figure G-4, the flows corresponding to DownloadControl, DownloadData and RPC are carried on separate ATM SVCs (resourceNums 1, 2 and 3, respectively) and use their respective associationTags 1, 2 and 3. The bindings to the interfaces are done using the associationTags. Through the bindings, the information format at the interfaces is expected to be in IP format.

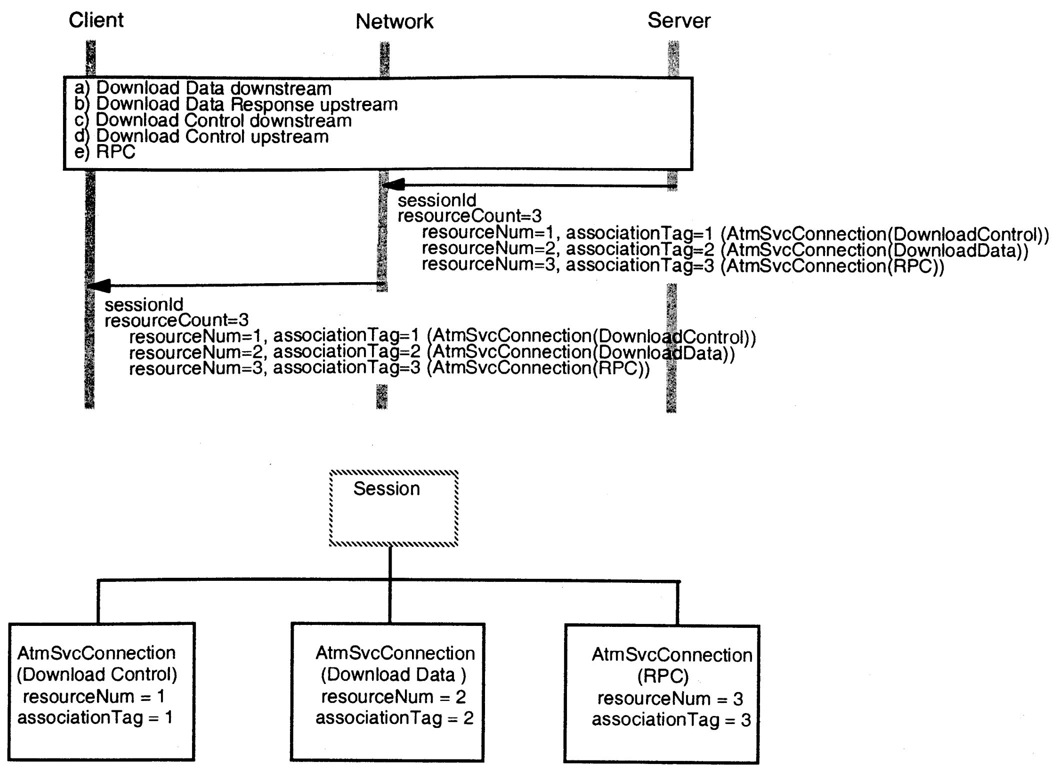

Client
Network
Server
a) Download Data downstream
b) Download Data Response upstream
c) Download Control downstream
d) Download Control upstream
e) RPC
sessionId
resourceCount=3
resourceNum=1, associationTag=1 (AtmSvcConnection(DownloadControl))
resourceNum=2, associationTag=2 (AtmSvcConnection(DownloadData))
resourceNum=3, associationTag=3 (AtmSvcConnection(RPC))
sessionId
resourceCount=3
resourceNum=1, associationTag=1 (AtmSvcConnection(DownloadControl))
resourceNum=2, associationTag=2 (AtmSvcConnection(DownloadData))
resourceNum=3, associationTag=3 (AtmSvcConnection(RPC))
Session
AtmSvcConnection (Download Control) resourceNum = 1 associationTag = 1
AtmSvcConnection (Download Data) resourceNum = 2 associationTag = 2
AtmSvcConnection (RPC) resourceNum = 3 associationTag = 3

Figure G-4 End-to-End ATM, Download Phase, Multiple ATM SVCs Server and Client Views

G.5.1.2 Non-ATM HFC Client View

The top diagram in Figure G-5 provides the list of resource descriptors communicated from the Server to the Network and the Network to the Client. Since the ATM connection terminates in the Network, new resources between the Network and the Client are used over the non-ATM HFC network. The flows on the ATM SVC connection are mapped into non-ATM HFC resources. Although the resource numbers for each flow on the Client side are different, the association tags are kept the same.

The DownloadControl, DownloadData and RPC connections each consist of 2-way flows. The downstream flows are carried on the MPEG TS and the upstream flows are carried on TDMA. The bindings to the corresponding interfaces are done using the associationTags. Through the bindings, the information format at the interfaces is expected to be in IP format.

Since the MPEG TS is carried over TsDownstreamBandwidth (resourceNum 20), the SRM assigns an associationTag (50) to associate it with the MpegProgram (resourceNum 30). The Association Tag in this instance is used to identify a stack and does not appear in interface bindings at the User-to-User level.

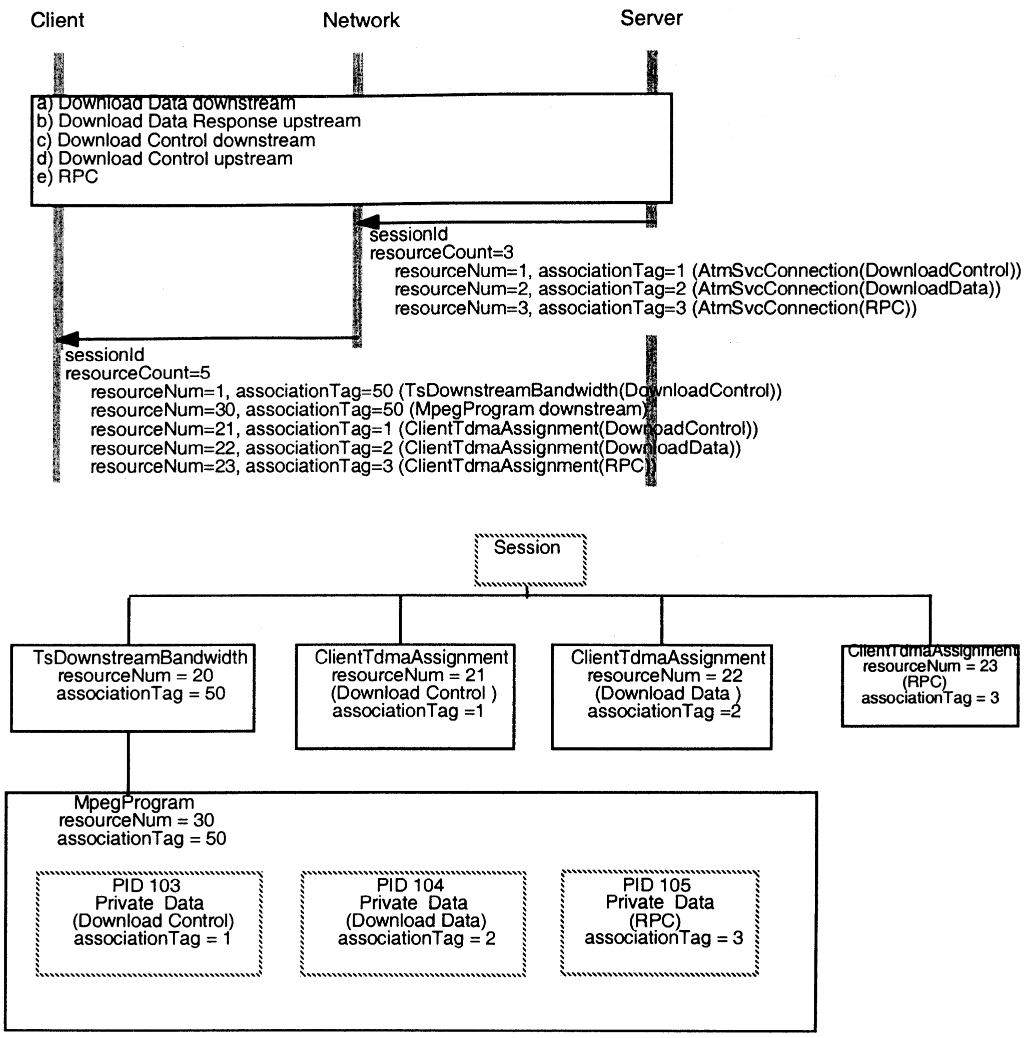

Client
Network
Server
a) Download Data downstream
b) Download Data Response upstream
c) Download Control downstream
d) Download Control upstream
e) RPC
sessionId
resourceCount=3
resourceNum=1, associationTag=1 (AtmSvcConnection(DownloadControl))
resourceNum=2, associationTag=2 (AtmSvcConnection(DownloadData))
resourceNum=3, associationTag=3 (AtmSvcConnection(RPC))
sessionId
resourceCount=5
resourceNum=1, associationTag=50 (TsDownstreamBandwidth(DownloadControl))
resourceNum=30, associationTag=50 (MpegProgram downstream)
resourceNum=21, associationTag=1 (ClientTdmaAssignment(DownloadControl))
resourceNum=22, associationTag=2 (ClientTdmaAssignment(DownloadData))
resourceNum=23, associationTag=3 (ClientTdmaAssignment(RPC))
Session
TsDownstreamBandwidth resourceNum = 20 associationTag = 50
ClientTdmaAssignment resourceNum = 21 (Download Control) associationTag =1
ClientTdmaAssignment resourceNum = 22 (Download Data) associationTag =2
ClientTdmaAssignment resourceNum = 23 (RPC) associationTag = 3
MpegProgram resourceNum = 30 associationTag = 50
PID 103 Private Data (Download Control) associationTag = 1
PID 104 Private Data (Download Data) associationTag = 2
PID 105 Private Data (RPC) associationTag = 3

Figure G-5 Non-ATM HFC Client View Corresponding to Download Phase, Multiple ATM SVCs Server and Client Views

G.5.2 Video Play Phase, Multiple ATM SVCs

This example uses separate ATM SVCs in order to carry the flows identified below:

1. MPEG Audio/Video/Data downstream
2. RPC

Two scenarios are considered:

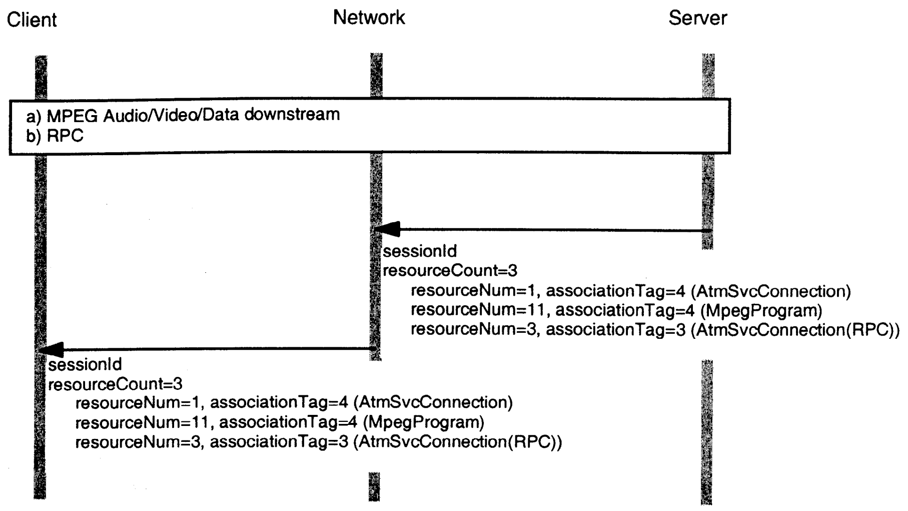
- End-to-End ATM.
  In this case the ATM SVC is maintained intact between the Server and the Client.

- Core ATM network and non-ATM HFC Client access.
  In this case the flows being carried on the ATM SVC are mapped to non-ATM resources through the non-ATM resource descriptors.

G.5.2.1 End-to-End ATM

In Figure G-6 provides the list of resource descriptors communicated from the Server to the Network and the Network to the Client. Since it is end-to-end ATM, the resource numbers and the association tags are unchanged.

In the bottom diagram in Figure G-6, the flows corresponding to MPEG Audio/Video/Data downstream and RPC are carried on separate ATM SVCs (resourceNums 1 and 3, respectively) and use the respective associationTags (1 and 3). The associationTag 4 identifies a stack and is not used in the bindings. The bindings to the corresponding interfaces are done using the associationTags 10, 11, 6 and 3. Through the associationTag 3 binding, the information format at the interface is expected to be in IP format.


Client
Network
Server
a) MPEG Audio/Video/Data downstream
b) RPC
sessionId
resourceCount=3
resourceNum=1, associationTag=4 (AtmSvcConnection)
resourceNum=11, associationTag=4 (MpegProgram)
resourceNum=3, associationTag=3 (AtmSvcConnection(RPC))
sessionId
resourceCount=3
resourceNum=1, associationTag=4 (AtmSvcConnection)
resourceNum=11, associationTag=4 (MpegProgram)
resourceNum=3, associationTag=3 (AtmSvcConnection(RPC))

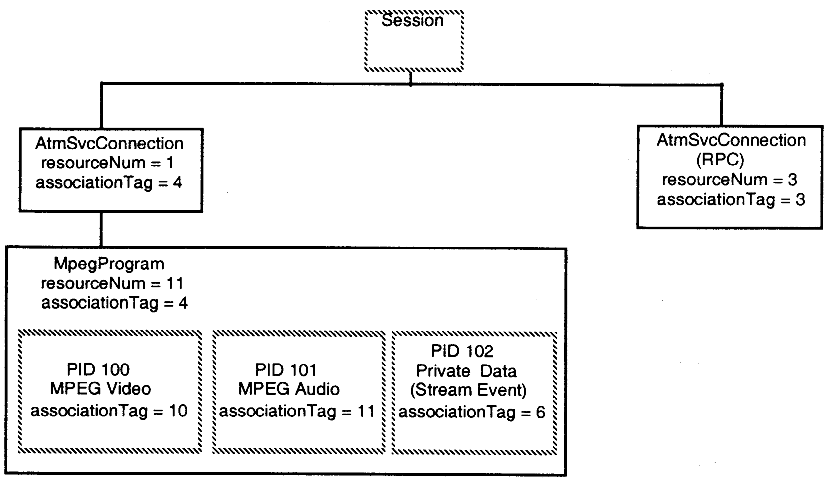


Figure G-6 End-to-End ATM, Video Play Phase, Multiple ATM SVCs Server and Client Views

G.5.2.2 Non-ATM HFC Client View

The top diagram in Figure G-7 provides the list of resource descriptors communicated from the Server to the Network and the Network to the Client. Since the ATM connection terminates in the Network, new resources between the Network and the Client are used over the non-ATM Hybrid Fiber Coax (HFC) network. The flows on the ATM SVC connection are mapped into the non-ATM HFC resources. Although the resource numbers for each flow on the Client side are different, the association tags are kept the same.

In the bottom diagram of Figure G-7, the MPEG Audio/Video/StreamEvent consist of only downstream flows and the RPC consists of 2-way flows. Both MPEG Audio/Video/StreamEvent and RPC downstream flows are carried on the MPEG TS and the RPC upstream flow is carried on a TDMA connection. The bindings to the corresponding interfaces are done using the associationTags.

Since the MPEG TS is carried over TsDownstreamBandwidth (resourceNum 20), the SRM assigns an associationTag (50) to associate it with the MpegProgram resourceNum (11). The Association Tag in this instance is used to identify a stack and does not appear in interface bindings at the User-to-User level.

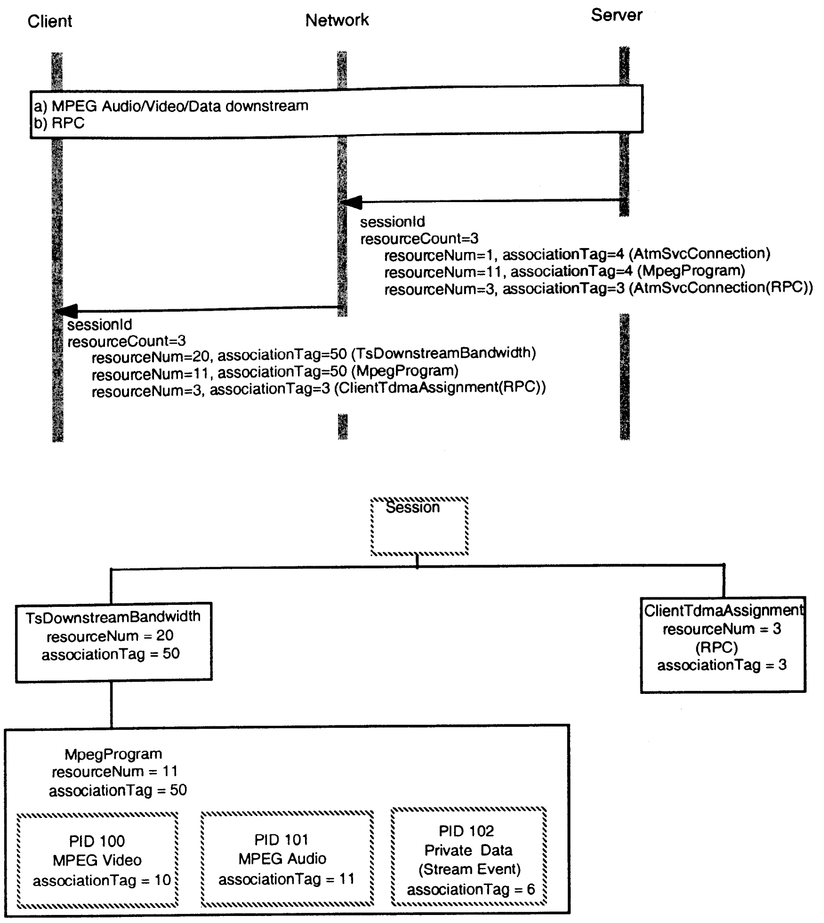


Figure G-7 Non-ATM HFC Client View Corresponding to Video Play Phase, Multiple ATM SVCs Server View

G.5.3 Single Asymmetric ATM SVC

This example uses a single asymmetric ATM SVC in order to carry the flows identified below:

1. MPEG Audio/Video/Data downstream
2. RPC over TCP/IP downstream flow over private data on MPEG TS
3. RPC over TCP/IP upstream flow over ATM
4. Download Control and Data downstream flow in UDP/IP over private data on MPEG TS
5. Download Control and Data Response upstream flow over ATM

Two scenarios are considered:

- End-to-End ATM.
  In this case the ATM SVC is maintained intact between the Server and the Client.

- Core ATM network and non-ATM HFC Client access.
  In this case the flows being carried on the ATM SVC are mapped to non-ATM resources through the non-ATM resource descriptors.

G.5.3.1 End-to-End ATM

The top diagram in Figure G-8 provides the list of resource descriptors communicated from the Server to the Network and the Network to the Client. Since it is an end-to-end ATM network, the resource numbers and the association tags are unchanged.

The bottom diagram in Figure G-8 shows how the SharedResource descriptor is used to share the AtmSvcConnection between the MpegProgram and Ip descriptors.

The MPEG Transport Stream Resource number (resourceNum 10), through the SharedResource descriptor (resourceNum 11) shares the AtmSvcConnection (resourceNum 1) with the other IP connections. The link between the MPEG TS and the SharedResource is maintained through the Association Tag, associationTag 1. The Audio, the Video and the StreamEvent (carried in PIDs 100, 101 and 102, respectively) are assigned the associationTags 10, 11 and 6, respectively.

Since the RPC consists of 2-way flows, the downstream is carried on the MPEG TS (PID 204) and the upstream is carried on the IP connection (resourceNum 12). The associationTag 2 links the two directions to the same interface. The SharedResource (resourceNum 13) also is tagged at associationTag 2. Similarly the PID (resourceNum 14) and resourceNum 15 for Download are tagged at associationTag 3.

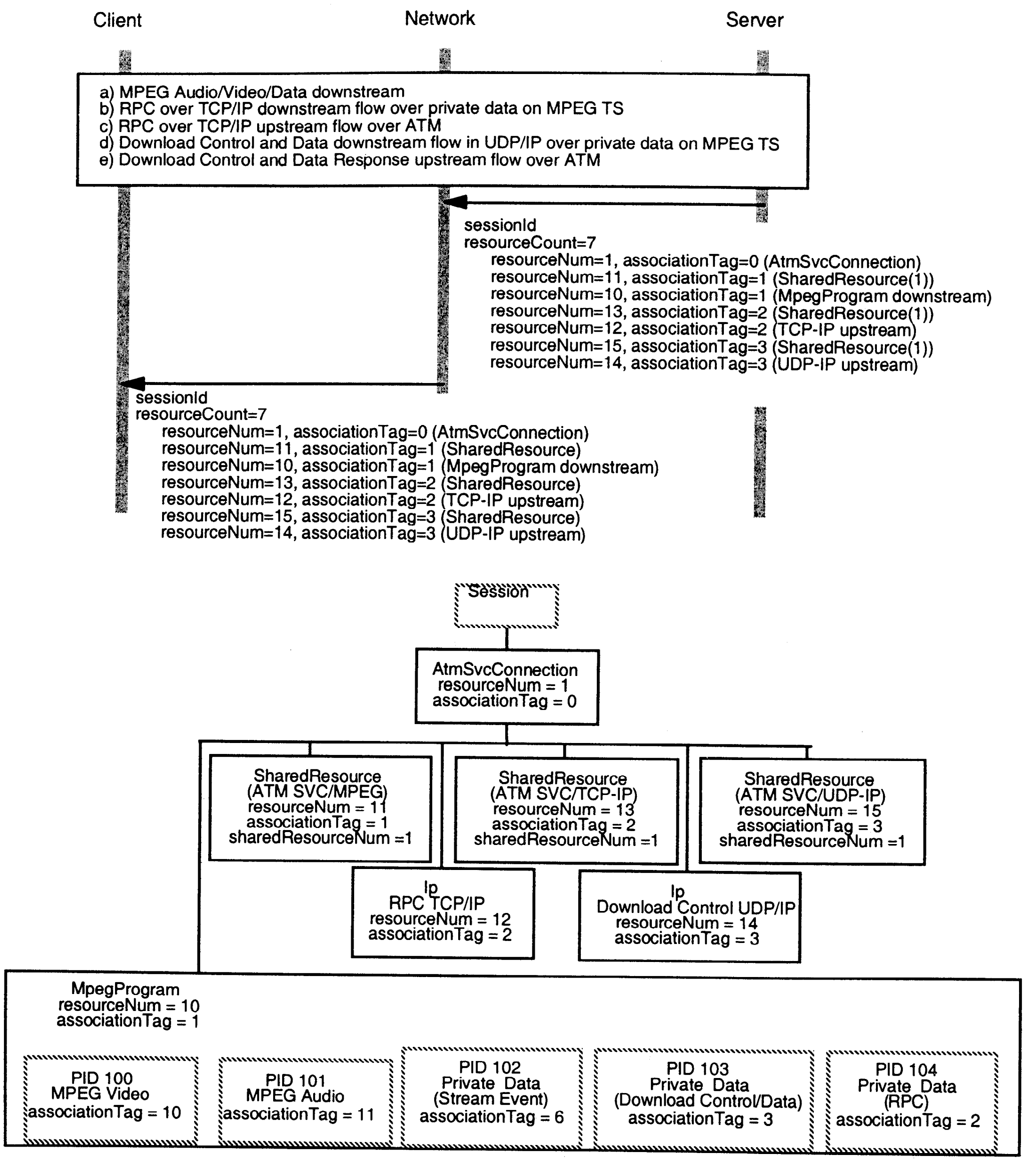

Client
Network
Server
a) MPEG Audio/Video/Data downstream
b) RPC over TCP/IP downstream flow over private data on MPEG TS
c) RPC over TCP/IP upstream flow over ATM
d) Download Control and Data downstream flow in UDP/IP over private data on MPEG TS
e) Download Control and Data Response upstream flow over ATM
sessionId
resourceCount=7
resourceNum=1, associationTag=0 (AtmSvcConnection)
resourceNum=11, associationTag=1 (SharedResource(1))
resourceNum=10, associationTag=1 (MpegProgram downstream)
resourceNum=13, associationTag=2 (SharedResource(1))
resourceNum=12, associationTag=2 (TCP-IP upstream)
resourceNum=15, associationTag=3 (SharedResource(1))
resourceNum=14, associationTag=3 (UDP-IP upstream)
sessionId
resourceCount=7
resourceNum=1, associationTag=0 (AtmSvcConnection)
resourceNum=11, associationTag=1 (SharedResource)
resourceNum=10, associationTag=1 (MpegProgram downstream)
resourceNum=13, associationTag=2 (SharedResource)
resourceNum=12, associationTag=2 (TCP-IP upstream)
resourceNum=15, associationTag=3 (SharedResource)
resourceNum=14, associationTag=3 (UDP-IP upstream)
Session
AtmSvcConnection resourceNum = 1 associationTag = 0
SharedResource (ATM SVC/MPEG) resourceNum = 11 associationTag = 1 sharedResourceNum =1
SharedResource (ATM SVC/TCP-IP) resourceNum = 13 associationTag = 2 sharedResourceNum =1
SharedResource (ATM SVC/UDP-IP) resourceNum = 15 associationTag = 3 sharedResourceNum =1
Ip RPC TCP/IP resourceNum = 12 associationTag = 2
Ip Download Control UDP/IP resourceNum = 14 associationTag = 3
MpegProgram resourceNum = 10 associationTag = 1
PID 100 MPEG Video associationTag = 10
PID 101 MPEG Audio associationTag = 11
PID 102 Private Data (Stream Event) associationTag = 6
PID 103 Private Data (Download Control/Data) associationTag = 3
PID 104 Private Data (RPC) associationTag = 2

Figure G-8 End-to-End ATM, Single Asymmetric ATM SVC Server and Client Views

G.5.3.2 Non-ATM HFC Client View

The top diagram in Figure G-9 provides the list of resource descriptors communicated from the Server to the Network and the Network to the Client. Since the ATM connection terminates in the Network, new resources between the Network and the Client are used over the non-ATM HFC network. The flows on the ATM SVC connection are mapped into non-ATM HFC resources. Although the resource numbers for each flow on the Client side are different, the association tags are kept the same.

The bottom diagram in Figure G-9 shows how the SharedResource descriptor is used to share the ClientTdmaAssignment resource between the Ip descriptors used to carry the DownloadControl/Data and the RPC flows. Since the TsDownstreamBandwidth only carries the MPEG TS, both descriptors share the same associationTag (1).

The MPEG Transport Stream Resource number (resourceNum 30) carries the Audio, the Video and the StreamEvent in PIDs 100, 101 and 102, respectively. These are assigned Association Tag numbers 10, 11 and 6, respectively.

Since the RPC consists of 2-way flows, the downstream is carried on the MPEG TS (PID 204) and the upstream is carried on the IP connection (resourceNum 12). The Association Tag (associationTag 2) links the two directions to the same interface. The SharedResource (resourceNum 31) also shares the associationTag 2. Similarly the PID, (resourceNum 14) and resourceNum 32 for Download share the associationTag 3.

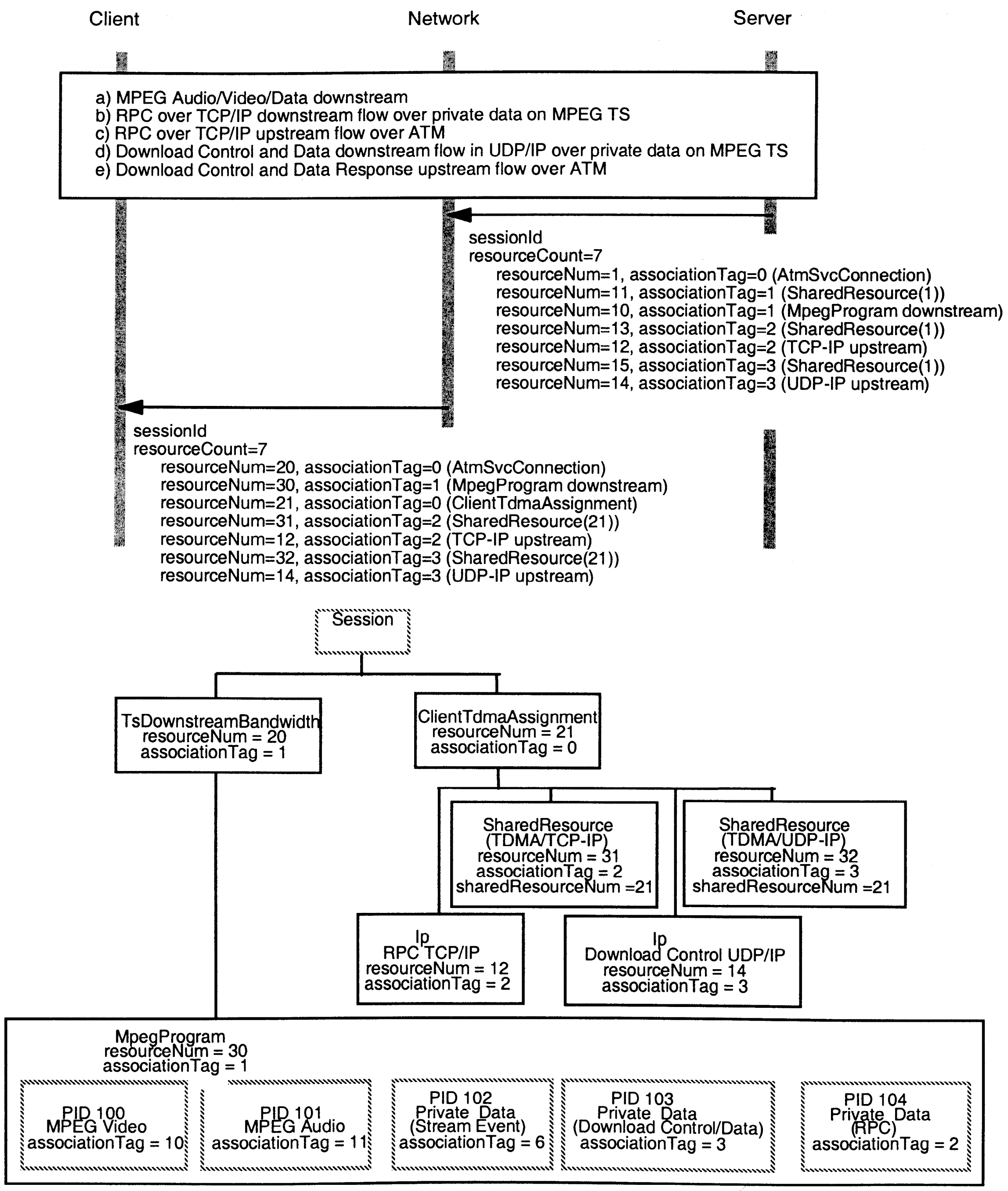

Client
Network
Server
a) MPEG Audio/Video/Data downstream
b) RPC over TCP/IP downstream flow over private data on MPEG TS
c) RPC over TCP/IP upstream flow over ATM
d) Download Control and Data downstream flow in UDP/IP over private data on MPEG TS
e) Download Control and Data Response upstream flow over ATM
sessionId
resourceCount=7
resourceNum=1, associationTag=0 (AtmSvcConnection)
resourceNum=11, associationTag=1 (SharedResource(1))
resourceNum=10, associationTag=1 (MpegProgram downstream)
resourceNum=13, associationTag=2 (SharedResource(1))
resourceNum=12, associationTag=2 (TCP-IP upstream)
resourceNum=15, associationTag=3 (SharedResource(1))
resourceNum=14, associationTag=3 (UDP-IP upstream)
sessionId
resourceCount=7
resourceNum=20, associationTag=0 (AtmSvcConnection)
resourceNum=30, associationTag=1 (MpegProgram downstream)
resourceNum=21, associationTag=0 (ClientTdmaAssignment)
resourceNum=31, associationTag=2 (SharedResource(21))
resourceNum=12, associationTag=2 (TCP-IP upstream)
resourceNum=32, associationTag=3 (SharedResource(21))
resourceNum=14, associationTag=3 (UDP-IP upstream)
Session
TsDownstreamBandwidth resourceNum = 20 associationTag = 1
ClientTdmaAssignment resourceNum = 21 associationTag = 0
SharedResource (TDMA/TCP-IP) resourceNum = 31 associationTag = 2 sharedResourceNum =21
SharedResource (TDMA/UDP-IP) resourceNum = 32 associationTag = 3 sharedResourceNum =21
Ip RPC TCP/IP resourceNum = 12 associationTag = 2
Ip Download Control UDP/IP resourceNum = 14 associationTag = 3
MpegProgram resourceNum = 30 associationTag = 1
PID 100 MPEG Video associationTag = 10
PID 101 MPEG Audio associationTag = 11
PID 102 Private Data (Stream Event) associationTag = 6
PID 103 Private Data (Download Control/Data) associationTag = 3
PID 104 Private Data (RPC) associationTag = 2

Figure G-9 Non-ATM HFC Client View Corresponding to Single Asymmetric ATM SVC Server Views

G.5.4 Single Asymmetric ATM PVC

The example with ATM Permanent Virtual Connections (PVC) is similar to the example in 1 "Single Asymmetric ATM SVC", except AtmSvcConnection is replaced by AtmConnection and SVC is replaced by PVC in the text.

G.5.5 Download Phase, Multiple ATM PVCs

The example with PVC is similar to the example in 2 "End-to-End ATM, Download Phase, Multiple ATM SVCs", except AtmSvcConnection is replaced by AtmConnection and SVC is replaced by PVC in the text.

G.5.6 Video Play Phase, Multiple ATM PVCs

The example with PVC is similar to the example in 3 "End-to-End ATM, Video Play Phase, Multiple ATM SVCs", except AtmSvcConnection is replaced by AtmConnection and SVC is replaced by PVC in the text.

G.5.7 Use of sharedResourceRequest Descriptors

In examples 1 through 6 above, the resource numbers are assigned by the originator of the request. If the resource numbers are assigned by the Network, then the sharedResourceRequest descriptor replaces the SharedResource descriptor at every instance of its occurrence. Correspondingly, the sharedResourceNum is replaced by sharedResourceRequestId. In the request message, from the originator to the Network, the ResourceNum fields are filled with 0's. In the confirm message from the Network to the originator, the ResourceNum fields take their actual values assigned by the Network.

Annex H
(informative)
Switched Digital Broadcast Service

H.1 Introduction

In some network architectures, such as Hybrid Fiber Coax (HFC) and Fiber to the Curb (FTTC), the economics of network design favor the delivery of digital broadcast programs to an Inter-Working Unit (IWU) within a delivery system, and from there, multicasting the programs to Clients. Although an entire set of broadcast programs is available at the IWU, it may not be practical for some system architectures to simultaneously deliver all digital broadcast programs to each Client. In this informative annex, a system assumption is that in order for a Client to switch from channel to channel, the IWU must be signaled for the channel connection to change.

The Switched Digital Broadcast (SDB) Channel Change Protocol (CCP), defined in normative clause 10, specifies a protocol specifically for this application. The SDB Channel Change Protocol is exchanged between a Client and an SDB Server (part of an IWU) in the Network. The SDB Channel Change Protocol forms a separate functional group within DSM-CC U-N protocols. So, although this protocol has been designed to work harmoniously with other clauses of ISO/IEC 13818-6, it has also been designed to used independently from the rest of DSM-CC.

The broadcastProgramId is a number that is used to uniquely identify each of the different broadcast programs that are available to a Client from the Network. How broadcastProgramId's are administered is outside the scope of this part of ISO/IEC 13818 (one possibility is to have the SRM administer these numbers).

A mapping in the Client is needed between the Client's view of the broadcast program (e.g., a name such as "PBS", or a channel number such as "10"), and the broadcastProgramId. The format of this mapping (commonly contained in an electronic program guide) and the means of its transport to the Client are outside the scope of this part of ISO/IEC 13818.

Three resource descriptors are defined to support Switched Digital Broadcast Service: The SDBContinuousFeed resource descriptor is defined to enable the addition or deletion of broadcastProgramId's to the entire set of programs available at the IWU; the SDBEntitlement resource descriptor is defined to identify the set of the broadcast programs from which a particular client can select within an established SDB session; and the SDBAssociations resource descriptor is defined to point to the individual connections at the Client used for program viewing and for the CCP control channel.

A mapping in the SDB Server is needed between the broadcastProgramId and the connection resources that transport the broadcast program. Although this information could be represented in a proprietary way, DSM-CC resource descriptors have been defined for this purpose and may be used. Descriptors of note are the AtmConnection resource, which contains port/VPI/VCI fields, and the MpegProgram resource, which indicates the MPEG program number and PIDs of the broadcast program as it arrives at the IWU from the distribution network.

H.2 Switched Digital Broadcast Service

The main architectural elements that illustrate the Switched Digital Broadcast Service are illustrated in Figure H-1.

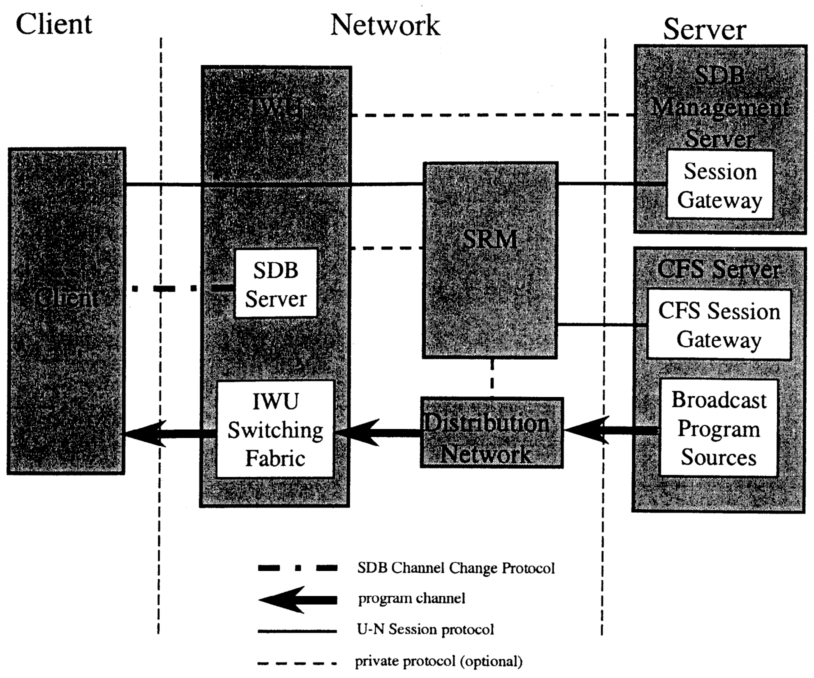


Figure H-1 Digital Broadcast Service Architecture

As an example, the Switched Digital Broadcast Service may be provided over a Fiber-to-the-Curb (FTTC) system. Two connections for signaling are required from the Client, one for the Channel Change Protocol which terminates on the SDB Server in the IWU, and one for the DSM-CC U-N which terminates in the SRM. In this example, the Channel Change Protocol is carried directly over ATM AAL5.

Other implementations are possible. For example, both SDB Channel Change and User-Network messages could be carried over a single connection: the IWU could terminate SDB Channel Change messages and route User-Network messages to the SRM. Alternately, it is possible to implement an SDB Server and an SRM in a single platform, and terminate both sets of messages in a single place.

H.3 Functional Flows

This annex describes several functional flows. Although many of the flows given below are recognized to exist, they are implementation specific and are provided for information only. The following functional flows are considered:

- Broadcast Program Configuration
- Client Service Profile Transfer to the SDB Server
- Broadcast Program Guide Transfer To Client
- Switched Digital Broadcast Service Session Establishment
- Client Initiated Channel Changes
- Network Initiated Channel Changes
- Switched Digital Broadcast Service Session Release

H.3.1 Broadcast Program Configuration

The Broadcast Program sources must be delivered to the broadcast distribution network, and from there to the IWU switching fabric. An association must be established between connection resources used by a Broadcast Program, and its broadcastProgramId. This association, which describes the Broadcast Programs available to the SDB Server, may be performed by private means. Alternately, DSM-CC Continuous Feed Sessions may be used to facilitate the configuration of this information in the SDB Server.

The following discussion describes how Broadcast Programs may be configured by the Continuous Feed Session mechanism.

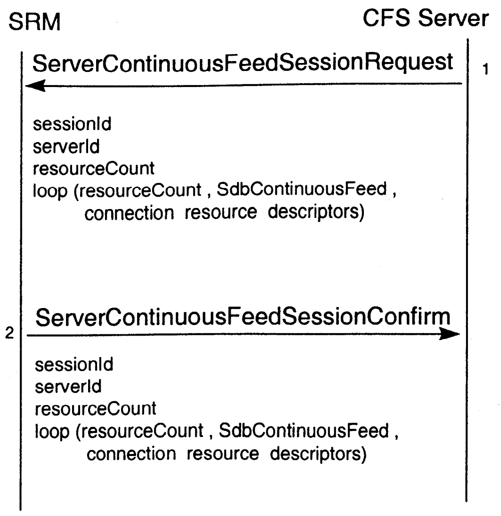

SRM
CFS Server
ServerContinuousFeedSessionRequest
1
sessionId
serverId
resourceCount
loop (resourceCount , SdbContinuousFeed ,
connection resource descriptors)
2
ServerContinuousFeedSessionConfirm
sessionId
serverId
resourceCount
loop (resourceCount , SdbContinuousFeed ,
connection resource descriptors)

Figure H-2 Broadcast program configuration using Continuous Feed Session establishment

Step 1:

To initiate a continuous feed session, the Continuous Feed Session Server (CFS Server) sends a ServerContinuousFeedSessionRequest to the SRM to establish Broadcast Program feeds on the Distribution Network. The ServerContinuousFeedSessionRequest includes a SDBContinuousFeed resource descriptor and connection resource descriptors. The SDBContinuousFeed resource descriptor contains the SDB Service ID (sdbId), which provides the list of broadcast programs within the continuous feed session: for each broadcast program, the CFS Server provides a broadcastProgramId (a globally unique identifier for a Broadcast Program) and the associationTag of its associated connection resource. The sdbId must be assigned beforehand and its management is outside the scope of this part of ISO/IEC 13818. The assignment of broadcastProgramId's can be done by the CFS Server or by the SRM. If the CFS Server assigns the broadcastProgramId's, it provides the assigned values in the request; if the SRM assigns them, the broadcastProgramId's are provided in the ServerContinuousFeedSessionConfirm message.

The format of the connection resource descriptors depends upon the connection establishment procedures in use.

Note that when the ServerContinuousFeedSessionRequest message is received by the SRM, it may need to set up connections on the distribution network to the IWUs. For each Broadcast Program, the SRM must then provide an SDB Server with the association between broadcastProgramId and the SDB Server view of the connection resource that carries the broadcast program. The protocol used to transfer this information between the SRM and SDB Server(s) is outside the scope of this part of ISO/IEC 13818.

Step 2:

The SRM replies with a ServerContinuousFeedSessionConfirm message to the CFS Server. The broadcastProgramId's allocated by the SRM are provided back to the CFS Server within a SDBContinuousFeed resource descriptor. The CFS Server will request connection resources to be allocated by the SRM, if needed, via appropriate resource descriptors.

H.3.2 Client Service Profile Transfer to the SDB Server

One method of authentication is to perform entitlement checking in the SDB Server. Requests for broadcast programs are validated at the SDB Server using the list of the broadcastProgramId's entitled to a Client within a session. If the Client is not entitled to the requested program, the SDBProgramSelectConfirm message, which is used by the SDB Server to respond to the Client, will fail with either a response code of rspEntitlementFailure or rspRedirect. In the latter case, the Client will be directed to an alternate channel.

The Client Service Profile information is maintained by the SDB Management server. The SDBEntitlement resource descriptor has been defined for use between the SDB Management Server and the SRM to communicate Client entitlements within a session. This information may be sent as part of the Switched Digital Broadcast Service session establishment, as described in subclause H.3.4. However, other means of communicating the entitlements from the SDB Management server to the SDB Server through private means are not precluded. The communication of entitlements from SRM to the SDB Server, and its checking, are outside the scope of this part of ISO/IEC 13818.

H.3.3 Broadcast Program Guide Transfer to Client

A list of the available broadcast programs (e.g., in an electronic program guide) may be required by the Client in order to request a Broadcast Program. Through this, the video user's notion of a Broadcast Program (e.g., the channel number, and perhaps a printable string, such as "CNN") is related to the broadcastProgramId.

The electronic program guide may include additional application level information about the broadcast program content (such as language / parental advisory, event cost, start time, etc.), and may include additional connection information. In the case of ATM, if the VPI/VCI values of Broadcast Programs are static, they could also be included in the electronic program guide for use by the Client to improve channel "surfing" response time. With this information, the Client need not wait for a SDBProgramSelectConfirm message to determine the VPI/VCI to which to "tune".

The format of the electronic program guide and the method by which it is sent to the Client is out of the scope of this part of ISO/IEC 13818. One method by which the electronic program guide may be sent to the Client is via a broadcast carousel mechanism (see clauses 7 and 11). The electronic program guide may be managed in its totality by a network service provider and passed as a single entity to the Client, or pieces may be managed by individual SDB service providers, and transferred to the Client separately.

H.3.4 Switched Digital Broadcast Service Session Establishment

The purpose of a SDB Session is to manage the resources needed to provide a SDB service. Typically, a minimum of two resources are needed in order to provide a SDB service: a bi-directional control channel between the Client and a SDB Server, and a "downstream" data channel to carry a broadcast program from the IWU to the Client.

The DSM-CC session protocol can be used to establish SDB sessions; an example on how this is done is given below in Figure H-3. However, in simple implementations, dynamic session establishment using the SRM may not be needed and the sessionNumber (the least significant 4 bytes of the sessionId) used in the SDB Channel Change Protocol messages could be a number assigned, following a rule, to distinguish between different SDB service requests sharing the control channel.

The SDB Channel Change Protocol supports simultaneous SDB sessions to a Client. Additional sessions may be used to receive multiple video streams (e.g., in order to support "picture in picture"), or for other purposes, such as to gain background access to a broadcast carousel.

The session gateway in the SDB Management Server terminates Client SDB session establishment requests. The SDB Management Server is a logical entity only; in practice, this function may be implemented by a network provider together with the SRM. Alternately, this function may be provided by a service provider as a standalone implementation, or implemented together with the Continuous Feed Server.

Figure H-3 illustrates a normal SDB session establishment.

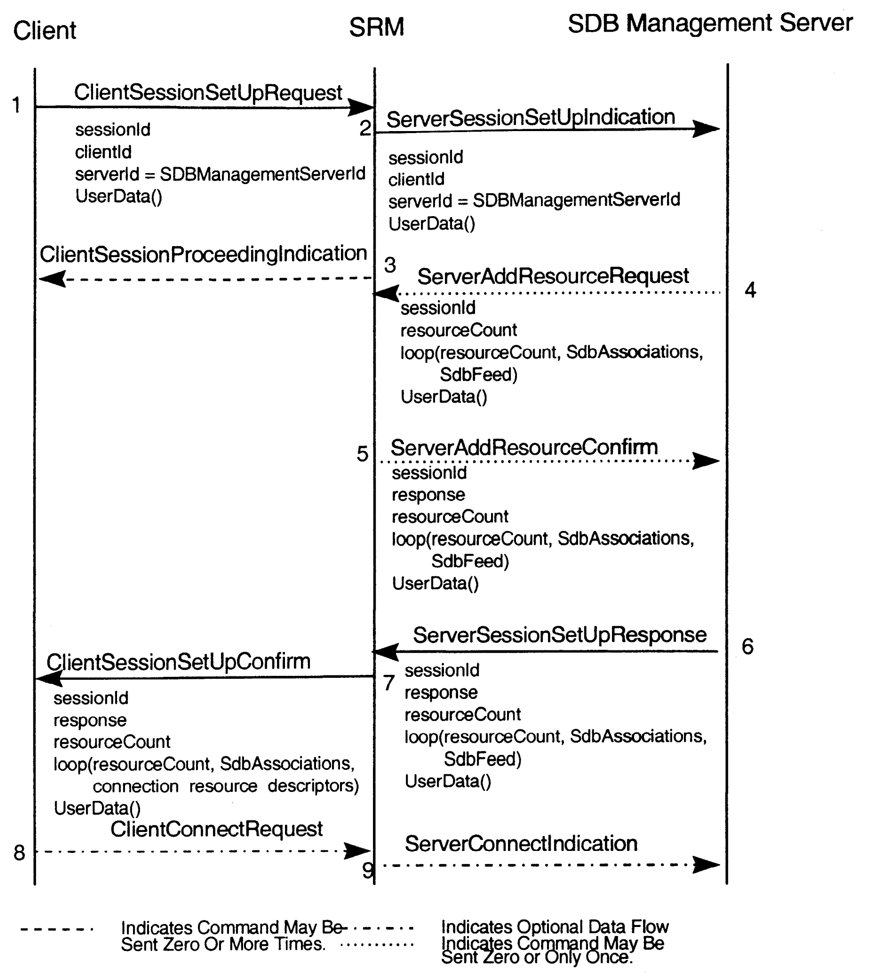

Client
SRM
SDB Management Server
1 ClientSessionSetUpRequest
sessionId
clientId
serverId = SDBManagementServerId
UserData()
2 ServerSessionSetUpIndication
sessionId
clientId
serverId = SDBManagementServerId
UserData()
ClientSessionProceedingIndication
3 ServerAddResourceRequest
4
sessionId
resourceCount
loop(resourceCount, SdbAssociations, SdbFeed)
UserData()
5 ServerAddResourceConfirm
sessionId
response
resourceCount
loop(resourceCount, SdbAssociations, SdbFeed)
UserData()
ServerSessionSetUpResponse
6
ClientSessionSetUpConfirm
7
sessionId
response
resourceCount
loop(resourceCount, SdbAssociations, connection resource descriptors)
UserData()
sessionId
response
resourceCount
loop(resourceCount, SdbAssociations, SdbFeed)
UserData()
8 ClientConnectRequest
9 ServerConnectIndication
Indicates Command May Be Sent Zero Or More Times.
Indicates Optional Data Flow
Indicates Command May Be Sent Zero or Only Once.

Figure H-3 SDB Session Establishment

Step 1 (Client):

The Client sends a ClientSessionSetUpRequest to the SRM, indicating the SDB Management Server's serverId. If the DSM-CC U-U interface is used (see clause 5), the Client may include a SessionUU attach() request within the UserData of the ClientSessionSetupRequest message in order to identify the requested SDB service.

Step 2 (SRM):

The SRM sends a ServerSessionSetUpIndication to the SDB Management Server.

Step 3 (SRM):

The SRM may optionally send a ClientSessionProceedingIndication message to the Client.

Step 4 (SDB Management Server)

In this example, the SDB Management Server concludes by the serverId, or through the information provided in the SessionUU attach(), that the Client wishes to access an SDB service. Accordingly, it sends a ServerAddResourceRequest message to the SRM. In other cases, the Client may access a directory at the SDB Management Server from which SDB service(s) may be selected. In this example, the SDB Management Server through the SDBAssociations resource descriptor, requests two resources -- one for program viewing and one for the CCP control channel. An additional resource descriptor, the SDBEntitlement, may be used to provide a list of broadcastProgramId's for the Client's entitlements. Provision is made in the SDBEntitlement descriptor to either authorize additional programs over those to which the Client is already entitled or deny authorization to programs to which the Client was entitled.

Step 5 (SRM):

The SRM allocates the resource needed to support the control channel and a nominal resource for initial program viewing based on the Client entitlements for the SDB Session in the SDBEntitlement. Note that the SRM may need to request the resources from the IWU at this point, and may communicate Client entitlement information to the IWU. How this is done is outside the scope of this part of ISO/IEC 13818.

The SRM sends a ServerAddResourceConfirm message to the SDB Management Server. The response code is set to rspNeResourceOk if all resources could be successfully allocated; otherwise, it is set to rspNeResourceFailed.

Step 6 (SDB Management Server):

Once the SDB Management Server has determined that the resources required for the new SDB session have been allocated, it accepts the session, and sends a ServerSessionSetUpResponse to the SRM. The assigned tag values are reused by the SRM for the connection resources established to the Client. If the U-U Interface is used (see clause 5), the associationTag's are also carried in the SessionUU attach() response in order to bind the connection resources at the Client with the MPEG_DOWN_USE and SDB_CTRL_USE taps. Private data in the ServerSessionSetUpResponse may be used to communicate the electronic program guide.

Step 7 (SRM):

The SRM sends the ClientSessionSetUpConfirm message to the Client. The list of resource descriptors included with this message comprise, from the Client's view, the connection resource (for example the AtmConnection) corresponding to the sdbControlAssociationTag (the associationTag of the connection resource will correspond to that of the sdbControlAssociationTag) and the connection resource (e.g., TsDownstreamBandwidth) for program viewing, corresponding to the sdbProgramAssociationTag.

In the case where the Client establishes multiple SDB sessions, a desired optimization would be to share the SDB Control Channel among them. Sharing of the SDB Control Channel may be accomplished by private means between the SRM and the IWU: an AtmConnection may be dedicated to serve as the SDB Control Channel for all the broadcast sessions at a Client. The SRM then provides the Client with the identical AtmConnection resource in the ClientSessionSetupConfirm messages for all its broadcast sessions. This also implies that the SRM will not tear down the shared AtmConnection when a particular session is released, as long as there is at least one active broadcast session at that Client.

H.3.5 Client Initiated Channel Changes

In order to request a Broadcast Program, the following is assumed: a SDB Session has been established, the SDB Server has received the Client entitlements for the established SDB session (see subclause H.3.4) and the Client has knowledge of broadcast program Id's (e.g., via an electronic program guide).

In Figure H-4, the Client requests a Broadcast Program from the SDB Server. The first parameter in the SDBProgramSelectRequest message body is the sessionId. This is used to distinguish between different SDB sessions that may be active at a single Client. The second field is broadcastProgramId, which is used to identify the Broadcast Program the Client desires.

Parameter checking is performed and appropriate error response code values returned to the Client upon error. Typical errors at this step may be rspNoSession (invalid session number) or rspFormatError (badly formatted message).

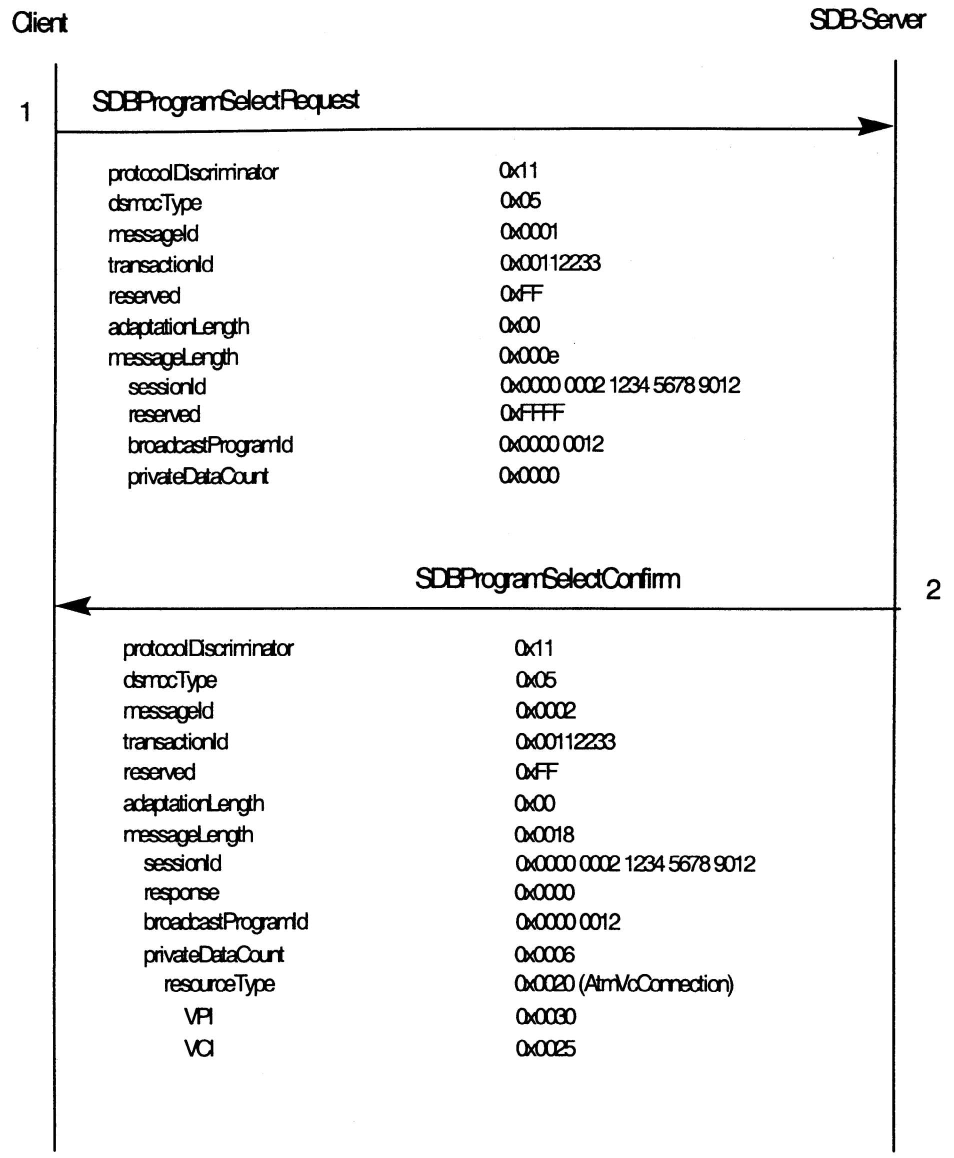
The Client's entitlements can be used at this step to validate the request. The response code, rspEntitlementFailure, has been defined for this purpose. Depending upon implementation, the SDB Server may either block the delivery of the requested program or provide an alternate program in the case of entitlement failure. Such a feature might be used when the Client requests an un-subscribed channel, and instead of completely rejecting the request, the Client is directed to a broadcast program that contains an advertisement. In the case where an alternate channel is to be provided, the response code rspRedirect is sent to the Client, and the broadcastProgramId in the SDBProgramSelectConfirm message is set to the broadcastProgramId of the alternate broadcast program.

The SDB Server checks the ability of the network to provide the desired Broadcast Program. This checking includes the applicability of the available resources and the operational state of the Broadcast Program. If the Broadcast Program cannot be delivered due to facility or network failure, the rspBcProgramOutOfService response code is returned.

The SDB Server also checks the availability of resources necessary to provide the desired program. If insufficient resource exist at the SDB Server, then the rspNoServerResources response code is returned. If insufficient resources exist in the network, the rspNoNetworkResources response code is returned.

If the request can be successfully satisfied, then the connection is made that allows the Client to receive the requested Broadcast Program, and a SDBProgramSelectConfirm message is sent with a response value of rspOk, and indicates the necessary connection information needed by the Client to receive the Broadcast Program. The format of this information is unspecified, and may be transported in privateData, if needed. In the case of FTTC, the connection information might include ATM VPI/VCI information; in the case of HFC, it might include PhysicalChannel and MpegProgram resource descriptors.

An example of a successful channel change message sequence follows:



protocolDiscriminator	0x11
dsmccType	0x05
messageId	0x0001
transactionId	0x00112233
reserved	0xFF
adaptationLength	0x00
messageLength	0x000e
sessionId	0x0000 0002 1234 5678 9012
reserved	0xFFFF
broadcastProgramId	0x0000 0012
privateDataCount	0x0000

protocolDiscriminator	0x11
dsmccType	0x05
messageId	0x0002
transactionId	0x00112233
reserved	0xFF
adaptationLength	0x00
messageLength	0x0018
sessionId	0x0000 0002 1234 5678 9012
response	0x0000
broadcastProgramId	0x0000 0012
privateDataCount	0x0006
resourceType	0x0020 (AtmVcConnection)
VPI	0x0030
VCI	0x0025

Figure H-4 Example Client Initiated Channel Change

H.3.6 Network Initiated Channel Changes

Network initiated channel changes can be used to "tune" a Client to a channel (for example, just before a Pay-Per-View event begins), or away from a channel (for example, when a Pay-Per-View event ends, or when a Client is unsubscribed).

The SDBProgramSelectIndication message contains a sessionId, which is comprised of the sessionNumber and deviceId of the Client. As with Client initiated channel changes, a session must be explicitly or administratively established before the SDBProgramSelectIndication message can be sent. The U-N Pass-Thru Receipt message protocol (defined in clause 12) is one method by which the SDB Management Server may alert a Client to initiate a session. The Pass-Thru Receipt message protocol also allows the SDB Management Server to notify the Client of the appropriate parameters to establish the session and initiate the service.

A reason field is provided to indicate why the SDB Server has decided to initiate the SDBProgramSelectIndication. If the reason is rsnNormal, then the SDB Server has decided to provide a channel. If the reason is rsnEntitlementFailure, then the Client is no longer entitled to the channel, and will no longer receive it. In this case, the broadcastProgramId is set to all 0's to indicate that no program is available or is set to the broadcastProgramId of an alternate program as appropriate (e.g., a broadcast program that contains an advertisement). It is expected that the Client application software will recover from this situation appropriately, but how this is done is implementation dependent.

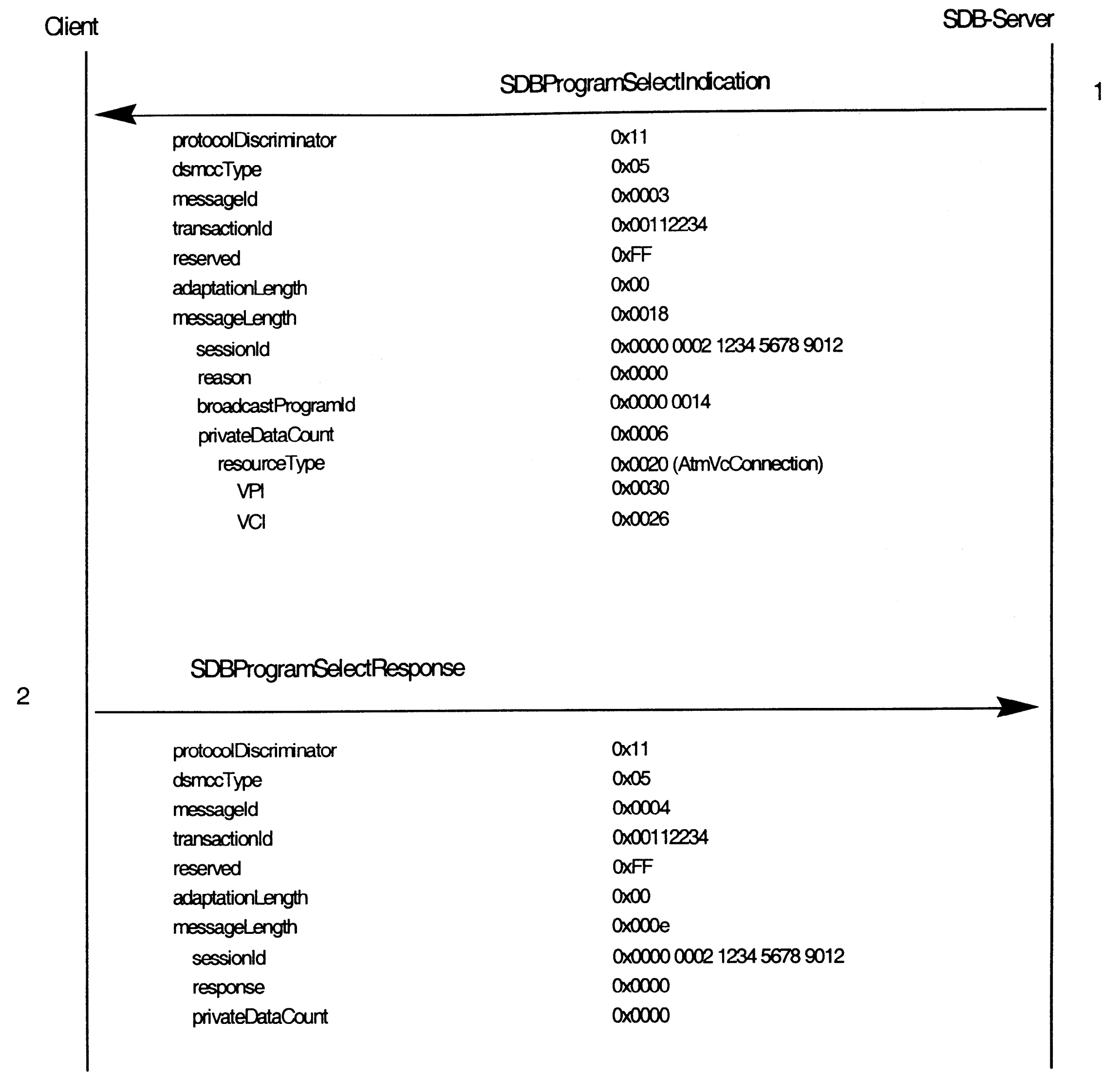
If supplied, the broadcastProgramId indicates the new Broadcast Program to be received, and privateData may be used to transport appropriate connection information. These fields have the same syntax and semantics as in the SDBProgramSelectConfirm.

When received by the Client, parameter checking is performed and appropriate error response code values returned to the SDB Server upon error. Typical errors at this step may be rspNoSession (invalid session number), and rspFormatError (badly formatted message).

If the indication is successfully processed in the Client, it responds with a SDBProgramSelectResponse message value set to rspOk.

In the case of a response time-out waiting for the SDBProgramSelectConfirm message, the SDB Server may retry (up to a maximum number of retries), and if unsuccessful, discontinue the program. In this case, it is the responsibility of the Client to recover appropriately. How this is done is implementation dependent.

An example successful network initiated channel change message sequence follows:



protocolDiscriminator	0x11
dsmccType	0x05
messageId	0x0003
transactionId	0x00112234
reserved	0xFF
adaptationLength	0x00
messageLength	0x0018
sessionId	0x0000 0002 1234 5678 9012
reason	0x0000
broadcastProgramId	0x0000 0014
privateDataCount	0x0006
resourceType	0x0020 (AtmVcConnection)
VPI	0x0030
VCI	0x0026

protocolDiscriminator	0x11
dsmccType	0x05
messageId	0x0004
transactionId	0x00112234
reserved	0xFF
adaptationLength	0x00
messageLength	0x000e
sessionId	0x0000 0002 1234 5678 9012
response	0x0000
privateDataCount	0x0000

Figure H-5 Example Network Initiated Channel Change

H.3.7 Digital Broadcast Session Release

SDB sessions are released identically to other DSM-CC sessions. Figure H-6 illustrates the message flow for a SDB session release.

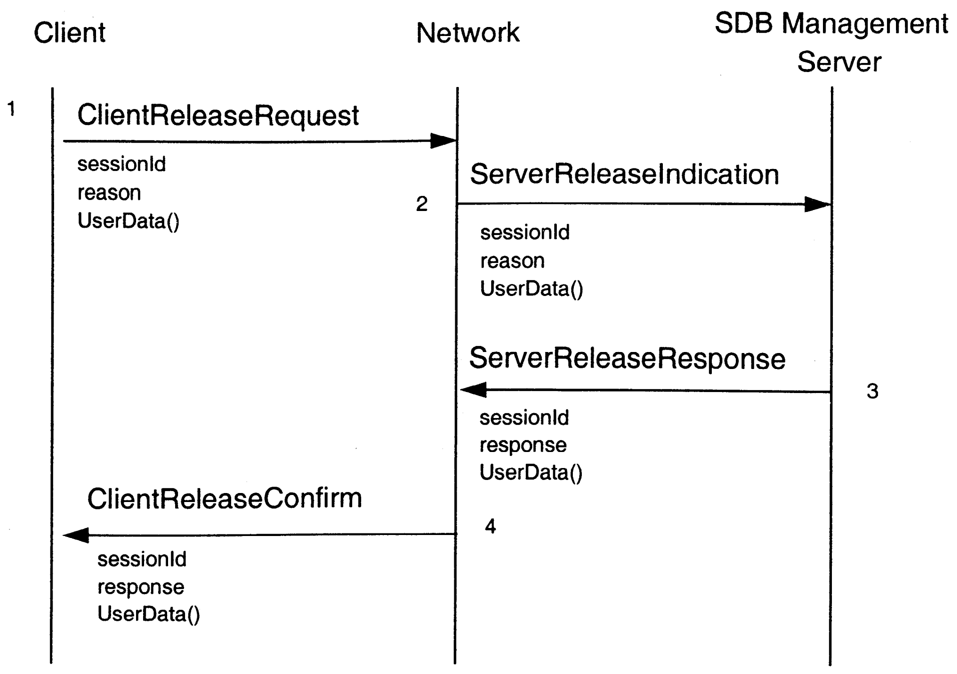


Figure H-6 SDB Session Release

Annex I
(informative)
Example U-N Life Cycle Walk Through

I.1 Introduction

This appendix provides an example for a scenario and message flows for a DSM-CC session, including the phases of User-Network Configuration, Session Establishment, and Download. User-to-User information may be found in clause 5. There are many possible scenarios for the 3 phases listed. This is but a small set of examples.

I.2 General Flow

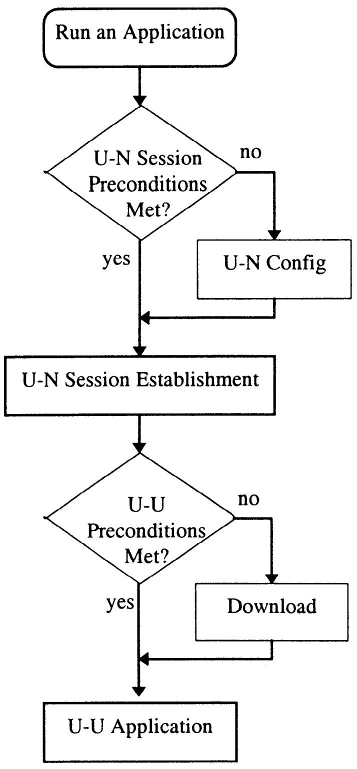


Figure I-1 Flow for Running an Application

Each step, whether it be U-N Config, U-N Session Establishment, U-N Download or running a U-U Application, has certain preconditions and post-conditions. In general, if the pre-conditions are met for a step, the flow proceeds to the step is processed. Otherwise, the preconditions must be satisfied, possibly causing the flow to proceed to other prerequisite steps.

Figure I-1 illustrates the general flow for running an application. As can be seen, if the pre-conditions for Session Establishment and User-to-User application have been met, the steps performed are U-N Session Establishment and the running of the application.

It is possible that other processes outside of DSM-CC can be used to satisfy a pre-condition. For example, a bootp request on a LAN might be used by the Client to obtain its client id (network address).

I.3 U-N-Configuration

The U-N-Config procedures serve to automatically register a Client's device, which is connected to the DSM-CC Network. In addition, network specific parameters (e.g., actual timer values) can be made known to the Client's device.

I.3.1 Pre Conditions

Before the U-N-Config procedure starts, the Client's device has to know (from means outside the scope of DSM-CC):

- the path to the U-N configuration entity: (network address)
- how to approach the U-N configuration entity: UNConfigRequest message
- its own unique hardware identity: deviceId
- its own hardware and software capabilities: deviceType, deviceMajorRev, deviceMinorRev
- the timer value for receiving the UNConfigResponse
- how to interpret the UNConfigConfirm message.

I.3.2 Procedure

The Client device generates the UNConfigRequest message containing

- deviceId
- deviceType
- deviceMajorRev
- deviceMinorRev

and sends it to the UNConfig manager.

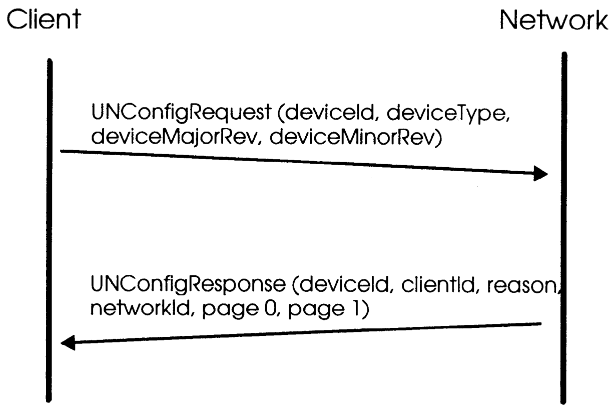

Client
Network
UNConfigRequest (deviceId, deviceType, deviceMajorRev, deviceMinorRev)
UNConfigResponse (deviceId, clientId, reason, networkId, page 0, page 1)

Figure I-2 U-N Configuration Scenario

It has to be verified that the requirements for the underlying signaling network are satisfied (see clause 9).

The ConfigManager responds with UNConfigConfirm containing

1. reason: (various codes)
2. deviceId: as in UNConfigRequest message
3. clientId: a globally defined 20 byte network service access point (NSAP) address identifying the Client
4. networkId: using format of a globally defined 20 byte NSAP address, which will be used for subsequent U-N messages (address of session manager)
5. DSM-CC specific information (e.g., timer values, see clause 3)
6. network dependent parameters (e.g., private to network operator; outside of the scope of DSM-CC).

I.3.3 Post Conditions

The post condition is that the Client Device now

- knows clientId and networkId
- knows timer values for U-N Session protocol
- knows network specific parameters
- knows DSM-CC client specific data
- knows what to do next and how to do it

I.4 U-N Session Setup

I.4.1 Pre Conditions

The Client device knows (from U-N Config, U-N Download, or other means)

- how to approach the session manager.
- how to initiate U-N Session messages (i.e., code available in Set Top Box, messages exist).

For the user-to-network part of SessionSetup message (see clause 4).

- the clientId: 20 byte NSAP address format.
- the serverId: 20 byte NSAP address format.
- who is assigning sessionId (and resourceId).

For the user-to-user data in SessionSetup message (see clause 5).

- rPathSpec: symbolic name identifying path to the first service gateway and (optional) in addition the service selected. The default initiation service may be resolved from "null".
- rClientRef: a unique system-wide address/identification of the Client node (may be redundant).
- rClientProfile: (user capabilities, see clause 6) used by the STB to identify its characteristics to the Server.
- aEndUser: a parameter identifying the consumer.

I.4.2 Procedure

The Client generates a ClientSessionSetupRequest message containing

- the clientId: 20 byte NSAP address format.
- the serverId: 20 byte NSAP address format.
- optional the sessionId: 10 byte field.
- rPathSpec: symbolic name identifying path to the first service gateway and (optional) in addition the service selected. The default initiation service may be resolved from "null".
- rClientRef: a unique system-wide address/identification of the Client node (may be redundant).
- rClientProfile: (user capabilities, see clause 6) used by the STB to identify its characteristics to the Server.
- aEndUser: a parameter identifying the consumer.

and sends this message to the SRM.

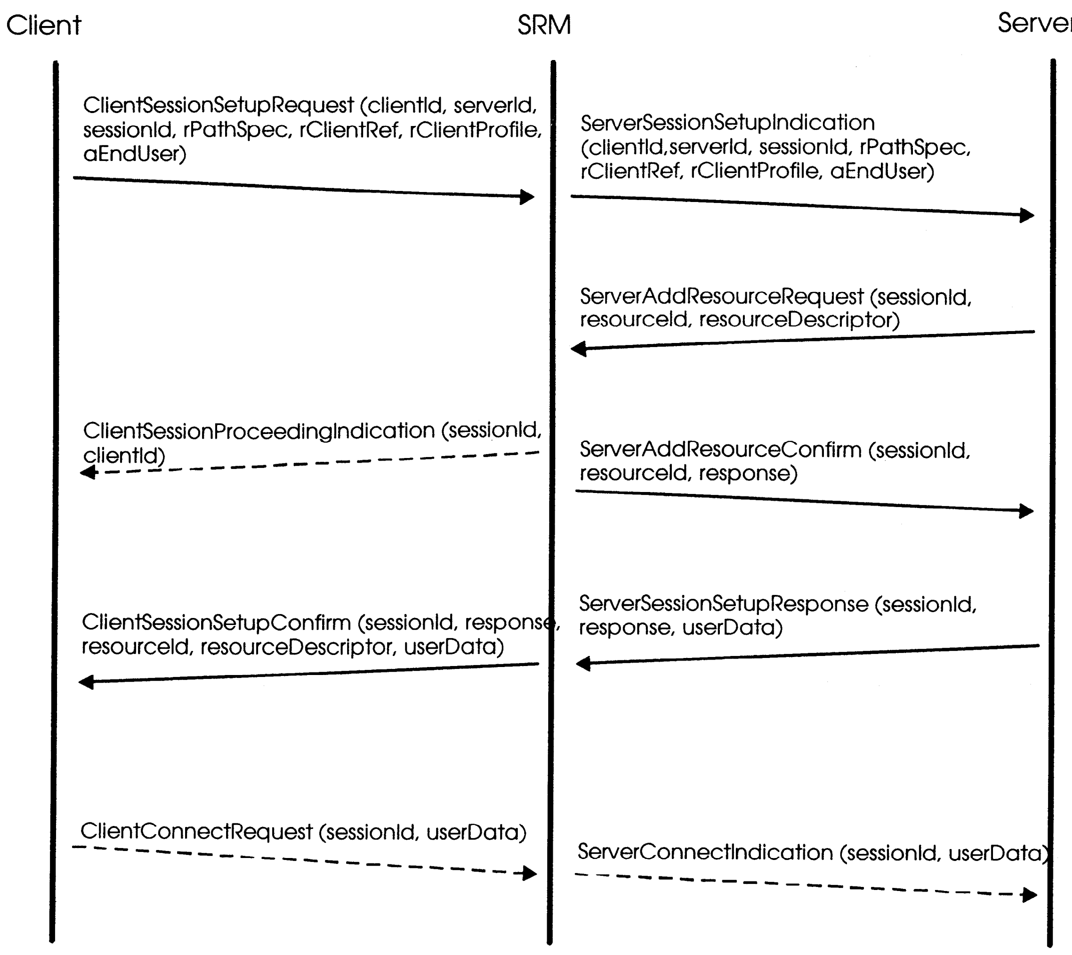

Client
SRM
Server
ClientSessionSetupRequest (clientId, serverId, sessionId, rPathSpec, rClientRef, rClientProfile, aEndUser)
ServerSessionSetupIndication (clientId,serverId, sessionId, rPathSpec, rClientRef, rClientProfile, aEndUser)
ServerAddResourceRequest (sessionId, resourceId, resourceDescriptor)
ClientSessionProceedingIndication (sessionId, clientId)
ServerAddResourceConfirm (sessionId, resourceId, response)
ServerSessionSetupResponse (sessionId, response, userData)
ClientSessionSetupConfirm (sessionId, response, resourceId, resourceDescriptor, userData)
ClientConnectRequest (sessionId, userData)
ServerConnectIndication (sessionId, userData)

Figure I-3 Session Setup Scenario (with nested Resource Request sequence)

The session and resource manager (SRM) receives the SetupRequest and resolves the serverId to the network address (i.e., communication path) of the Server. The SRM provides a consistency check to the other fields contained in the message. If there is no sessionId setup yet, the SRM will assign it. Then the ServerSessionSetupIndication is forwarded to the Server.

There are two possible continuations of the scenario:

- either the session establishment is acknowledged by the Server immediately without adding resources
- or the first request/negotiating of resources is performed before the session establishment is acknowledged by the Server.

The following describes the scenario with **resources being requested immediately** (see Figure I-3).

The Server receives the SessionSetupIndication. The ClientCapablities and the Service Name are evaluated by some higher layer entity at the Server side and is used to request/negotiate those resources which are assigned by the SRM. For that purpose, a ServerAddResourceRequest, which contains

- sessionId: as in ServerSessionSetupIndication
- resourceDescriptors for each resource being requested,

is generated by the Server and sent to the SRM.

If the Network has successfully reserved the resources (i.e., established connections where applicable), the SRM will acknowledge the resources using a ServerAddResourceConfirm message which contains:

- sessionId
- response code
- resourceDescriptors specifying the status of each resource which was requested.

If all resources were granted by the Network as requested/negotiated by the Server, the Server will send a ServerSessionSetupResponse message containing

- sessionId
- response code
- userData

The SRM will forward the message as a ClientSessionSetupConfirm, which contains:

- sessionId
- response code
- list of resources which were added during the resource request/negotiating phase
- the userData will be transported as received.

In order to account for variances in implementation architectures, optional Connect messages are provided in the case where it is necessary that a U-N layer Client acknowledgment be sent back to the Server (e.g., to overcome a possible race condition). In this case, the Client may inform the Server that all resources are available according to its point of view by sending a ClientConnectRequest message to the SRM. This message contains:

- sessionId
- userData

These parameters in term are passed to the Server using ServerConnectIndication.

Note that SessionSetupProceeding messages are also optional.

For an embedded "download" scenario, the ClientSessionSetupRequest / ServerSessionSetupIndication userData may contain the DownloadInfoRequest message. In this case, the ServerSessionSetupResponse / ClientSessionSetupConfirm then will contain a DownloadInfoResponse message. The U-N Download protocol is described in subclause I.I.5.

Although further text is not provided here, Figure I-4 shows the scenario where **resources are requested using AddResource messages after session establishment**.

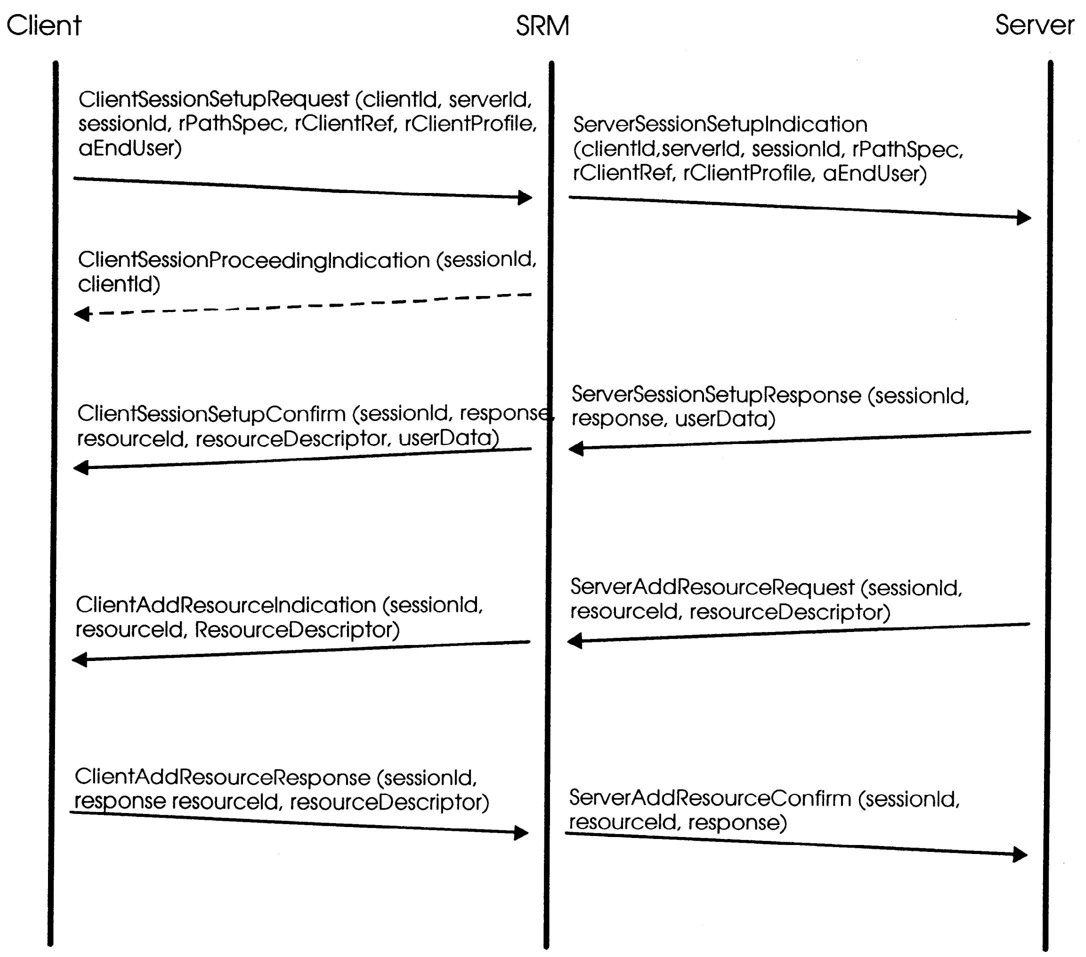

Client
SRM
Server
ClientSessionSetupRequest (clientId, serverId, sessionId, rPathSpec, rClientRef, rClientProfile, aEndUser)
ServerSessionSetupIndication (clientId,serverId, sessionId, rPathSpec, rClientRef, rClientProfile, aEndUser)
ClientSessionProceedingIndication (sessionId, clientId)
ServerSessionSetupResponse (sessionId, response, userData)
ClientSessionSetupConfirm (sessionId, response, resourceId, resourceDescriptor, userData)
ServerAddResourceRequest (sessionId, resourceId, resourceDescriptor)
ClientAddResourceIndication (sessionId, resourceId, ResourceDescriptor)
ClientAddResourceResponse (sessionId, response resourceId, resourceDescriptor)
ServerAddResourceConfirm (sessionId, resourceId, response)

Figure I-4 Session Setup Scenario (with Add Resource sequence after session establishment)

I.4.3 Post Conditions

The Client knows

- sessionId
- the list of resources to be used for the session including the states
- the userData

Optionally, the Client may know in addition:

- information contained in the DownloadInfoResponse message

The Client session state is active (CSActive).

The SRM knows

- the list of resources visible to Client or Server
- sessionId

The SRM session state is active (NSActive).

The Server knows

- sessionId
- the list of resources visible to Server

The Server session state is active (SSActive).

The state associated with functions outside the session protocol itself, but required to transition to "object land", is (launch).

I.5 U-N Download

I.5.1 Pre Conditions

There are several possibilities for the transport connections:

1. There are two connections: one downstream control and data, one upstream control
2. There are three connections: one upstream control, one downstream control, one broadband downstream data
3. There are two connections: one upstream control, one broadband downstream control and data
4. There is one connection: one download data (data carousel).

The flow-controlled scenario is described below. For information about all download scenarios, see clause 7.

The Client device

- knows how to approach the download manager
- has a path to the download Server
  (up to 3 connections may be needed: control down, control up, data down)
- knows the download transport connection topology (by convention)
- knows whether DownloadInfoRequest and DownloadInfoResponse/Indication messages have to be contained in SessionSetup messages
- knows how to interpret private data fields contained in DownloadInfoResponse/Indication.

The Server

- knows what information has to be downloaded depending on the information provided in a DownloadInfoRequest message.

I.5.2 Procedure

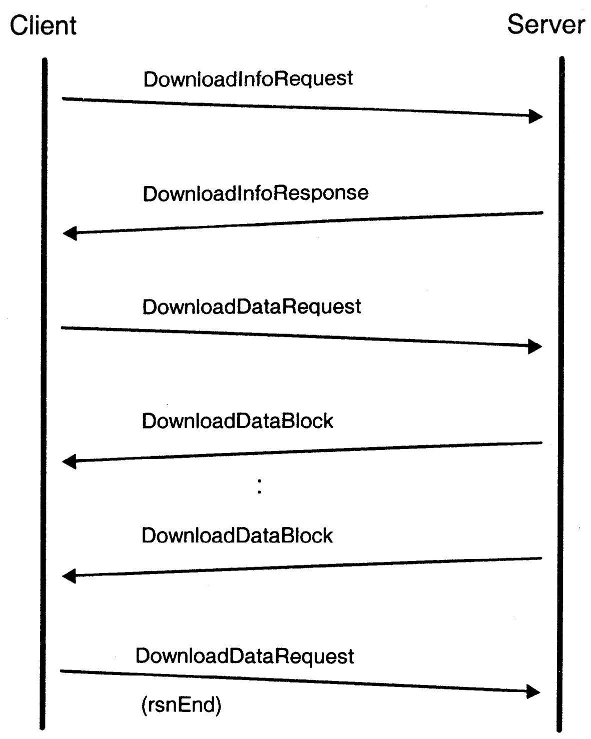


Figure I-5 Flow Controlled Download Scenario

The Client requests the Download procedure by sending a DownloadInfoRequest message to the Download Server.

The Server will respond using a DownloadInfoResponse message to inform the Client about the parameters used for the download.

The Client then starts the download by sending a DownloadStartRequest message.

The Server will download the requested information using a sequence of DownloadDataBlock messages each containing a part of the whole download information.

The Client acknowledges the receipt of the data blocks by sending a DownloadDataResponse message. In the flow controlled download procedure, the DownloadDataResponse message triggers the Download Server to continue sending DownloadDataBlock messages.

I.5.3 Post Conditions

The information has been downloaded.

The Client knows what to do next and how to do this.

Annex J
(informative)
Example of an OSI NSAP Address Format

J.1 Purpose

The purpose of this annex is to describe a possible implementation for clientId and serverId, two User-to-Network Session message data fields used in many of the User-to-Network Session messages. The normative clauses of this part of ISO/IEC 13818 define these fields to be in the format of an OSI NSAP (Open Systems Interconnection Network Service Access Point) address. For more details on clientId and serverId, refer to clause 4, User-to-Network Session Messages.

J.2 Introduction

ISO 8348 1987/A2 (Information processing systems -- Data communications -- Network service definition -- Addendum 2: Network layer addressing) defines the OSI NSAP address formats. The ATM Forum industry consortium uses a subset of these address formats -- named AESA (ATM End System Address) -- in the User-Network Interface (UNI) Specification. The example selected below is the E.164 NSAP.

Note: Any discrepancies between the OSI and ATM Forum specifications, and the address format described below represents an error in the format presented here.

J.3 E.164 NSAP

While any NSAP is acceptable for DSM-CC (i.e., inter-networking is beyond the scope), an example is shown using the E.164 version of the AESA address format. The characteristics of this format are as follows:

The generic OSI NSAP address consists of two domains:

- Initial Domain Part (IDP), which consists of two sub-parts:
  - 1-byte Authority and Format Identifier (AFI)
  - A variable-length Initial Domain Identifier (IDI), which depends on the value of the AFI.
- Domain Specific Part (DSP) which depends on the value of the IDI.

The E.164 NSAP version is a fixed 20-byte OSI NSAP address and is formatted as follows:

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

where

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 accommodated into the OSI NSAP. This work is outside the scope of DSM-CC.

Annex K
(informative)
Stream PlayList

K.1 Overview

It is desirable to implement an interface that enables client applications to generate and run playlists. A playlist is a sequence of QStreams that are delivered one after the other. A robust playlist interface is proposed that makes it easy for the client to generate repeats of a QStream in the playlist, and loops.

The StreamLink is defined as a building block with which to construct playlist queues. The StreamLink will have two pointers, one to an open QStream, and the other to the next StreamLink in the queue. The end of the queue will have null as its next StreamLink pointer.

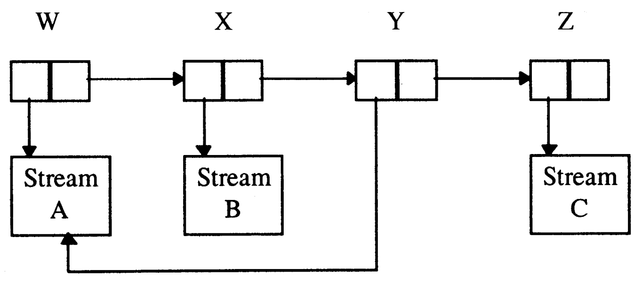
The operations

- QStream next () returns a StreamLink.
- StreamLink queue() connects one StreamLink to a next StreamLink and sets start, stop, scale values.
- StreamLink unqueue() disconnects one StreamLink from its next StreamLink.
- StreamLink destroy() deletes a StreamLink.
- StreamLink go() will cause the StreamLink start, stop scale values to be sent to the associated stream state machine as if a play() operation had been invoked.

The example above is implemented as follows:

1. Send QStream next() to A, get StreamLink W
2. Send QStream next() to B, get StreamLink X
3. Send QStream next() to A, get StreamLink Y
4. Send QStream next() to C, get StreamLink Z
5. Send StreamLink queue() to W, point to X, setting start, stop and scale
6. Send StreamLink queue() to X, point to Y, setting start, stop and scale
7. Send StreamLink queue() to Y, point to Z, setting start, stop and scale
8. Send StreamLink queue() to Z, no next link, setting start, stop and scale
9. Start the first state machine for QStream A, by sending a start() to W.

The resulting queue looks like:



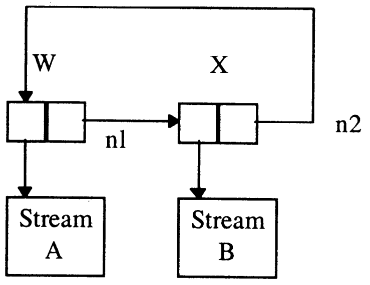
The function go() determines the first StreamLink to activate, or the first focus. When AppNPT rStop is reached, the focus shifts to the next StreamLink. Each time the focus shifts to a new StreamLink, the start, stop and scale values of the StreamLink are passed into the bound QStream and its state machine is initialized as if a **Stream play** had just been issued.

Here is the IDL for the StreamLink:

```
module DSM {
    interface QStream : Stream {                    // Inherit from Stream
        // Add
        void next (out StreamLink rNewLink);
        };
    interface StreamLink {
        attribute u_long count;             // the number of iterations to perform
        attribute QStream thisStr;          // pointer to an associated stream
        attribute StreamLink nextLink;      // pointer to the next StreamLink
        //
        // Cause one StreamLink to point to another with startup parameters
        void queue(in StreamLink nextLink, in NPT rStart, in NPT rStop, in Scale rScale)
                raises (QUE_LIMIT);
        //
        // Remove the pointer to the next  StreamLink
        void unqueue();
        //
        // Initialize and activate the associated stream state machine.
        // If the playlist is a loop, decrement count each time the Streamlink is passed.
        // Stop the play when count = 0
        void go(in u_long count);
        //
        // When done delete the StreamLink
        void destroy();
        //

        };
}
```

A client could browse the queue to find its current path by looking at the thisStr and nextLink of the StreamLinks.

Consider a second example which is a loop of A->B->A->B-> ...



If while QStream A is playing, a queue() is sent to StreamLink W to null the nextLink value, the QStream A will play to completion and stop. If while QStream A is playing, a queue() is sent to W that points to a third Qstream (e.g., QStream C) when QStream A completes (reaches rStop), the QStream C state machine will begin. Since the pointers are always forward, the pointer to the next StreamLink can be changed at any time.

There is only one active QStream state machine at a given time for a playlist queue.

The interfaces between StreamLink and QStream are implementation-specific and beyond the scope of DSM-CC.

K.2 DSM QStream next

DSM QStream next	Create a StreamLink for a QStream.

Client-Service IDL Syntax

```
module DSM {
    interface QStream : Stream {
        void  next (
                out StreamLink rNewLink);
    };
};
```

Semantics

The **QStream next** will associate a QStream with a new StreamLink. The StreamLink can then be used as an entry in a Stream playlist.

Privileges Required
READER

Parameters

StreamLink rNewLink	output	A unique StreamLink which can be used for constructing a playlist.

Annex L
(informative)
Service Transfer Message Flows

L.1 Introduction

Service transfer describes a mechanism for transferring a Service from one service domain to another. It may be used in the normal course of a service or in emergency cases. Each use is briefly described below:

L.1.1 Use of service transfer in the normal course of service

In the normal course of service, **DSM SessionUU attach()** and **DSM SessionUU detach()** (see clause 5, U-U Interfaces) can be used to perform Service Transfer. Most applications will have some nesting of navigation and will require a suspension of one level of nesting when proceeding to another. For example, a top level navigator can be located in one service domain and a movie browser in another. When going from the top navigator to the movie browser, the Client can suspend the navigator and attach to the movie browser. Then, when done with the movie browser, the Client can pop back to the top navigator by resuming it at the previous state. This is very common in applications and natural for human behavior.

Note: a Service Transfer does not affect a Session established using the User-Network Session protocol (see clause 4). If the Session context needs to be changed (i.e., changing of a SessionGateway), then User-Network Session operations need to be performed (either by performing a Session Tear-down followed by a new Session Setup, or by performing a Session Transfer).

Two methods are available for Service Transfer: a) Basic application level Service Transfer and b) Enhanced application level Service Transfer with DSM State suspend and DSM State resume.

In the Basic application level Service Transfer method, the parameters required in the next DSM Session attach (i.e., for the destinationServer) are sent to the Client from the Server at the application level. The Client uses this information to do one of the following:

1. Release the Session with the sourceServer and establish a new session with the destinationServer (using the appropriate U-N Session command sequences).
2. Maintain the service with the sourceServer and establish a new session with the destinationServer (using an appropriate U-N Session command sequence).

When the Session with the sourceServer is released, the Client cannot resume the context at a later time.

In the Enhanced application level Service Transfer, the parameters required in the next DSM Session attach may be sent to the Client from the Server at the application level. The Client uses detach, suspend and resume User-to-User Session Messages to do one of the following:

1. Release the Session with the sourceServer (using **detach()** with **aSuspend** true)
2. Suspend a service with forced release of its connection resources (using **suspend()** with **aRelease** true
3. Suspend a service without forced release of its connection resources; i.e., the resources can be reassigned by the SessionGateway to another service within a time-out period (using **suspend()** with **aRelease** false)
4. Maintain the service with the sourceServer

In this method, for the first three items above, the Client can resume the full context at a later time (using **attach()** for item 1, and using **resume()** for items 2 and 3). Message flows for both the Basic and Enhanced Service Transfer are given below.

L.1.2 Use of Service Transfer in emergency cases

The U-N Session protocol allows a service domain Server to tear down a session and redirect the Client U-U Library to reestablish a session to a substitute service domain. This service transfer takes place without the Client's application being required to be aware of the change of the session.

L.2 Basic application level Service Transfer

L.2.1 Service Transfer: sourceServer to destinationServer with sourceServer Session Release

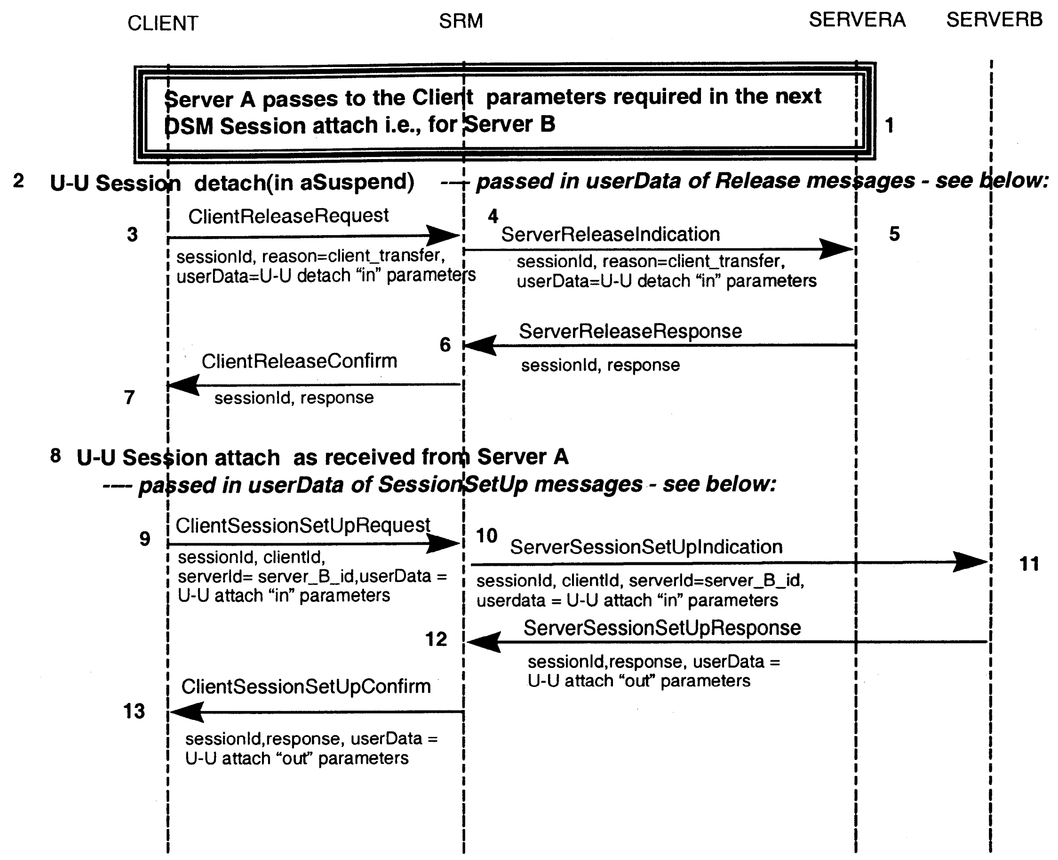

CLIENT
SRM
SERVERA
SERVERB
Server A passes to the Client parameters required in the next DSM Session attach i.e., for Server B
1
2 U-U Session detach(in aSuspend) — passed in userData of Release messages - see below:
3 ClientReleaseRequest
sessionId, reason=client_transfer, userData=U-U detach "in" parameters
4 ServerReleaseIndication
sessionId, reason=client_transfer, userData=U-U detach "in" parameters
5
6 ServerReleaseResponse
sessionId, response
ClientReleaseConfirm
7 sessionId, response
8 U-U Session attach as received from Server A — passed in userData of SessionSetUp messages - see below:
9 ClientSessionSetUpRequest
sessionId, clientId, serverId= server_B_id,userData = U-U attach "in" parameters
10 ServerSessionSetUpIndication
sessionId, clientId, serverId=server_B_id, userdata = U-U attach "in" parameters
11
12 ServerSessionSetUpResponse
sessionId,response, userData = U-U attach "out" parameters
ClientSessionSetUpConfirm
13 sessionId,response, userData = U-U attach "out" parameters

Figure L-1 Service Transfer from the sourceServer to the destinationServer -- sourceServer's (Server A) Session is released

Step 1

Server A (called the Source Server) passes to the Client all the next Session attach parameters required in the transfer including the serverId of the destinationServer.

Step 2

In this message flow, since the Client wants to release the session, the Client issues a detach with aSuspend set to false.

Steps 3-7 will result in a complete release of the session between Server A and the Client.

Step 8

The Client uses information obtained in Step 1 above to form the new DSM SessionUU attach().

Steps 9-13 complete the session set-up with Server B.

L.2.2 Service Transfer: sourceServer to destinationServer, Service maintained on sourceServer

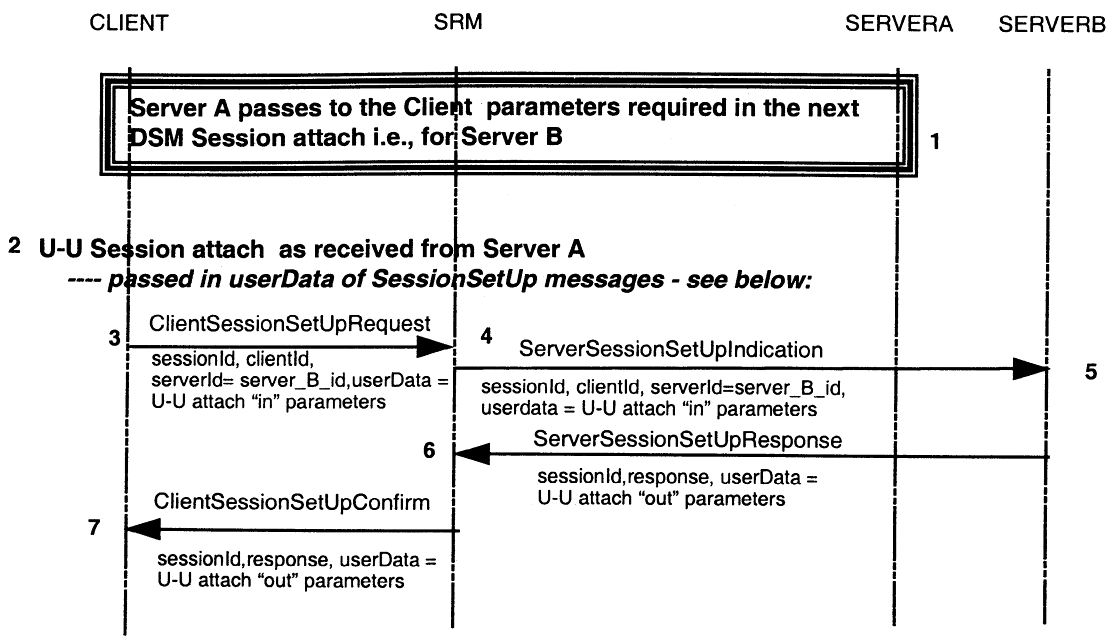

CLIENT
SRM
SERVERA
SERVERB
Server A passes to the Client parameters required in the next DSM Session attach i.e., for Server B
1
2 U-U Session attach as received from Server A
---- passed in userData of SessionSetUp messages - see below:
3
ClientSessionSetUpRequest
sessionId, clientId, serverId= server_B_id,userData = U-U attach "in" parameters
4
ServerSessionSetUpIndication
sessionId, clientId, serverId=server_B_id, userdata = U-U attach "in" parameters
5
6
ServerSessionSetUpResponse
sessionId,response, userData = U-U attach "out" parameters
7
ClientSessionSetUpConfirm
sessionId,response, userData = U-U attach "out" parameters

Figure L-2 Service Transfer from the sourceServer to the destinationServer -- Service maintained on the sourceServer

Step 1

Server A (called the Source Server) passes to the Client all the next Session attach parameters required in the transfer including the serverId of the destinationServer.

Step 2

The Client uses information obtained in Step 1 above to form the new DSM SessionUU attach().

Steps 3-7 complete the session set-up with Server B.

L.3 Enhanced application level Service Transfer

L.3.1 Release the Session with the sourceServer

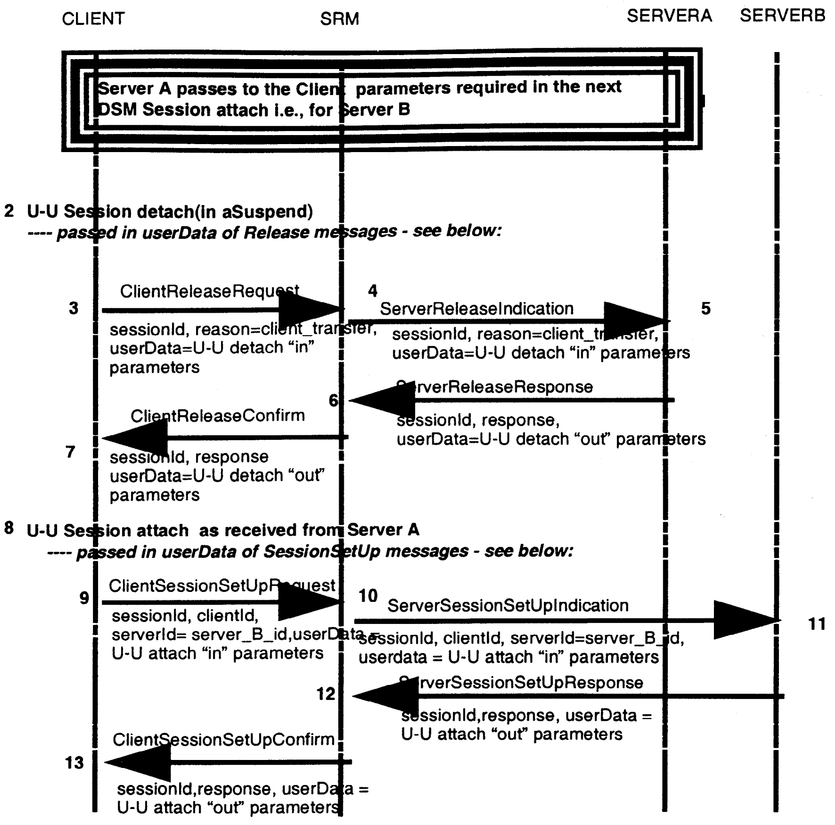

CLIENT
SRM
SERVERA
SERVERB
Server A passes to the Client parameters required in the next DSM Session attach i.e., for Server B
2 U-U Session detach(in aSuspend)
---- passed in userData of Release messages - see below:
3 ClientReleaseRequest
sessionId, reason=client_transfer, userData=U-U detach "in" parameters
4 ServerReleaseIndication
sessionId, reason=client_transfer, userData=U-U detach "in" parameters
5
6 ServerReleaseResponse
sessionId, response, userData=U-U detach "out" parameters
7 ClientReleaseConfirm
sessionId, response userData=U-U detach "out" parameters
8 U-U Session attach as received from Server A
---- passed in userData of SessionSetUp messages - see below:
9 ClientSessionSetUpRequest
sessionId, clientId, serverId= server_B_id,userData= U-U attach "in" parameters
10 ServerSessionSetUpIndication
sessionId, clientId, serverId=server_B_id, userdata = U-U attach "in" parameters
11
12 ServerSessionSetUpResponse
sessionId,response, userData = U-U attach "out" parameters
13 ClientSessionSetUpConfirm
sessionId,response, userData = U-U attach "out" parameters

Figure L-3 Service Transfer from the sourceServer to the destinationServer -- sourceServer's (Server A) Session is released

Step 1

Server A (called the Source Server) passes to the Client all the next Session attach parameters required in the transfer including the serverId of the destinationServer.

Step 2

Since the Client wants to suspend the context and release the session, the Client issues a detach with aSuspend set to true.

Steps 3-7 will result in a complete release of the session between Server A and the Client. The savedContext for the session is returned to the Client in the detach "out" parameters.

Step 8

The Client uses information obtained in Step 1 above to form the new DSM SessionUU attach().

Steps 9-13 complete the session set-up with Server B.

L.3.2 Maintain minimum resources with the sourceServer

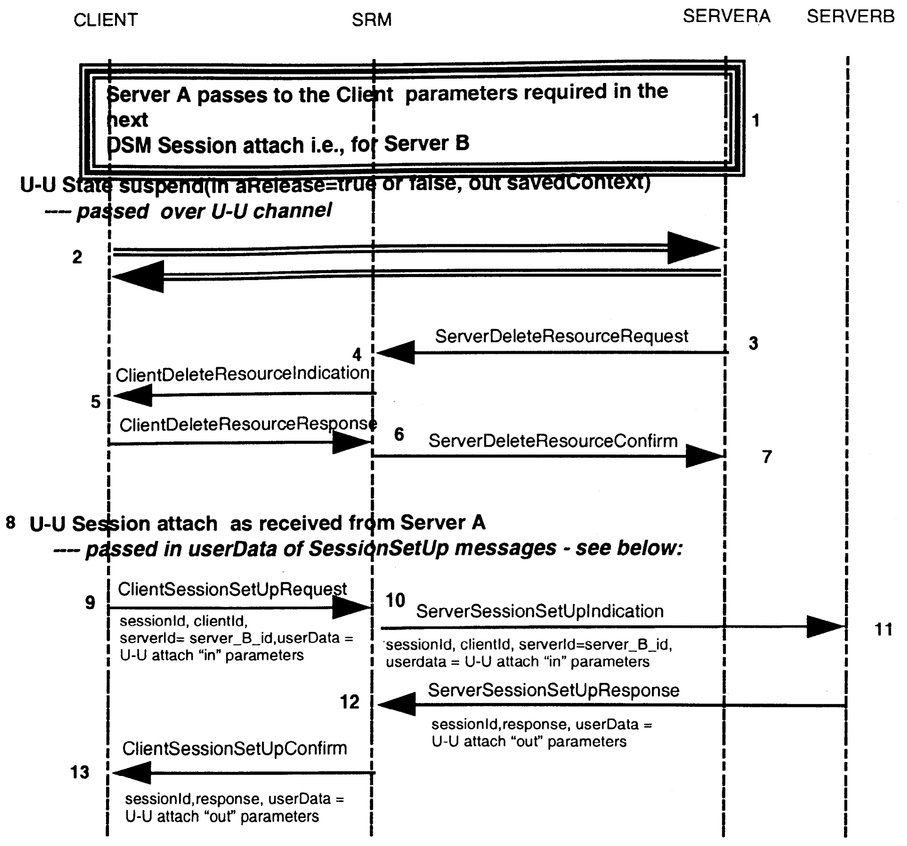

CLIENT
SRM
SERVERA
SERVERB
Server A passes to the Client parameters required in the next
DSM Session attach i.e., for Server B
1
U-U State suspend(in aRelease=true or false, out savedContext)
— passed over U-U channel
2
ServerDeleteResourceRequest
3
4
ClientDeleteResourceIndication
5
ClientDeleteResourceResponse
6
ServerDeleteResourceConfirm
7
8 U-U Session attach as received from Server A
— passed in userData of SessionSetUp messages - see below:
ClientSessionSetUpRequest
9
sessionId, clientId, serverId= server_B_id,userData = U-U attach "in" parameters
10
ServerSessionSetUpIndication
sessionId, clientId, serverId=server_B_id, userdata = U-U attach "in" parameters
11
ServerSessionSetUpResponse
12
sessionId,response, userData = U-U attach "out" parameters
ClientSessionSetUpConfirm
13
sessionId,response, userData = U-U attach "out" parameters

Figure L-4 Service Transfer from the sourceServer to the destinationServer -- Maintain minimum resources with the sourceServer

Step 1

Server A (called the Source Server) passes to the Client all the next Session attach parameters required in the transfer including the serverId of the destinationServer.

Step 2

The Client uses the State suspend message to indicate that the context be suspended while the resources which support the session are reduced. Association Tags for resources that are released remain valid. The Client then receives the UserContext in the savedContext.

Setting aRelease to true forces Server A (the SessionGateway) to release resources immediately. This would be the case if the Service Transfer was to a service on a different Server. Setting aRelease to false allows Server A (the SessionGateway) to reassign or release the resources after a local time-out. This would be the case if the Service Transfer were to a service within the same Server.

Steps 3-7

A minimal session is kept with Server A. DeleteResource messages are exchanged to delete resources which the suspended service no longer requires.

Step 8

The Client uses information obtained in Step 1 above to form the new DSM SessionUU attach()

Steps 9-13 complete the session set-up with Server B.

L.3.3 Maintain the service with the sourceServer

This scenario is identical to the Basic application level Service Transfer method Service Transfer from the sourceServer to the destinationServer with Service maintained on the sourceServer in subclause L.2.2.

L.3.4 Fall back to Server A after Session release with the sourceServer

This scenario relates to the situation where the Client resumes a suspended service. In other words, a Service Transfer using a Session detach with aSuspend true has occurred.

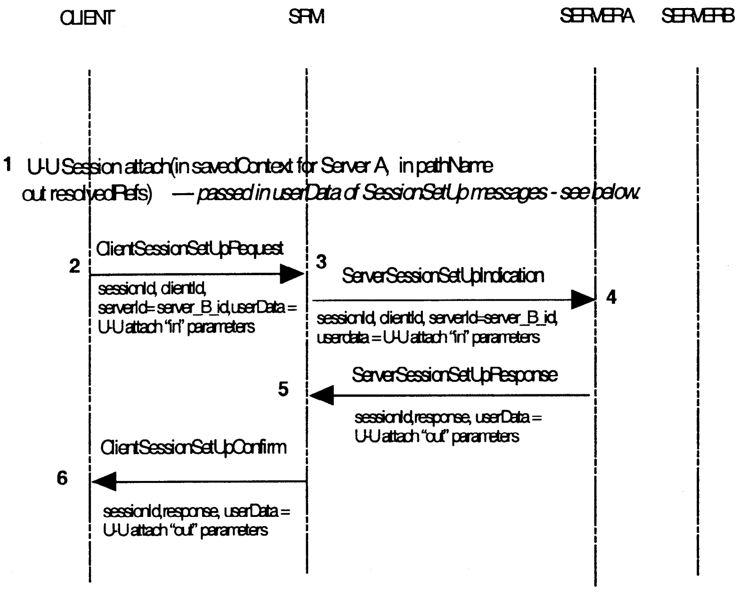

CLIENT
SRM
SERVERA
SERVERB
1 U-U Session attach(in savedContext for Server A, in pathName out resolvedRefs) — passed in userData of SessionSetUp messages - see below.
ClientSessionSetUpRequest
2
sessionId, clientId, serverId= server_B_id,userData = U-U attach "in" parameters
3
ServerSessionSetUpIndication
4
sessionId, clientId, serverId=server_B_id, userdata = U-U attach "in" parameters
ServerSessionSetUpResponse
5
sessionId,response, userData = U-U attach "out" parameters
ClientSessionSetUpConfirm
6
sessionId,response, userData = U-U attach "out" parameters

Figure L-5 Client fall-back to Server A after a sourceServerSession Release

Step 1

The Client retrieves the UserContext received from sourceServer at suspension time and prepares a Session attach with the user context placed in the savedContext.

Steps 2-6

A normal session set-up with Server A is performed with the userData carrying the Session attach.

L.3.5 Resumption of the full context on Server A after reduced Session

This scenario relates to the situation where the Client resumes a suspended service which has undergone a reduction of resources. In other words, it assumes that the Service Transfer with a sourceServer session reduction has occurred.

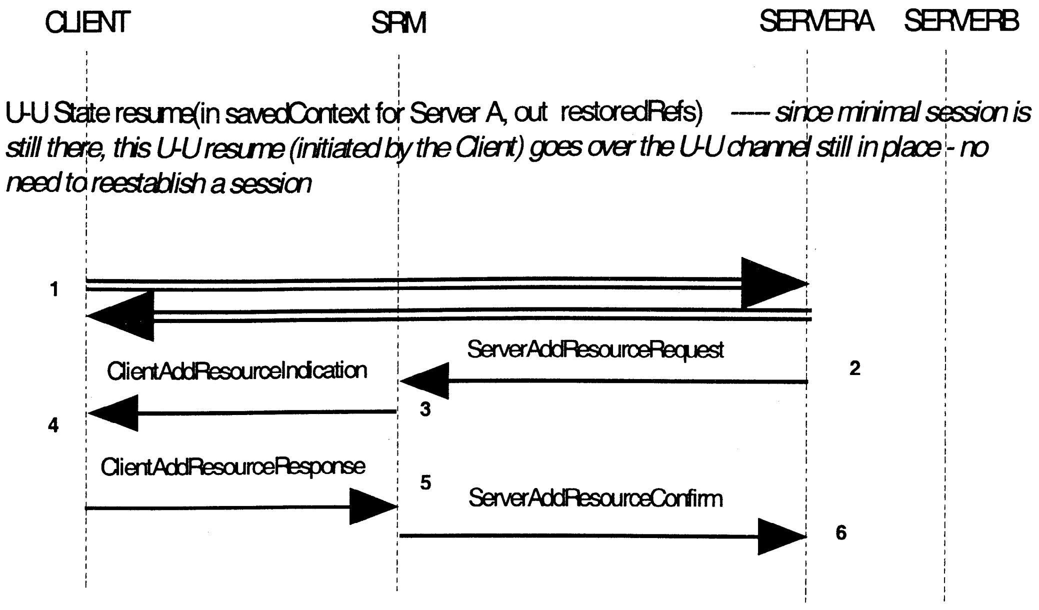

CLIENT
SRM
SERVERA
SERVERB
U-U State resume(in savedContext for Server A, out restoredRefs) —— since minimal session is still there, this U-U resume (initiated by the Client) goes over the U-U channel still in place - no need to reestablish a session
1
ServerAddResourceRequest
2
ClientAddResourceIndication
3
4
ClientAddResourceResponse
5
ServerAddResourceConfirm
6

Figure L-6 Client fall-back to Server A after a sourceServer Session reduction

Step 1

The Client retrieves the UserContext received from the sourceServer at suspension time and prepares a State resume with the user context placed in the savedContext.

Steps 2-6

An Add Resource by Server A is performed to acquire the resources for full context resumption.

L.3.6 Emergency Service Transfer

This scenario assumes that a Server, which is already serving a Client session, has reached a state that it requires to shed off some of its load by transferring the Client's service to another Server with which it has made a prior agreement.

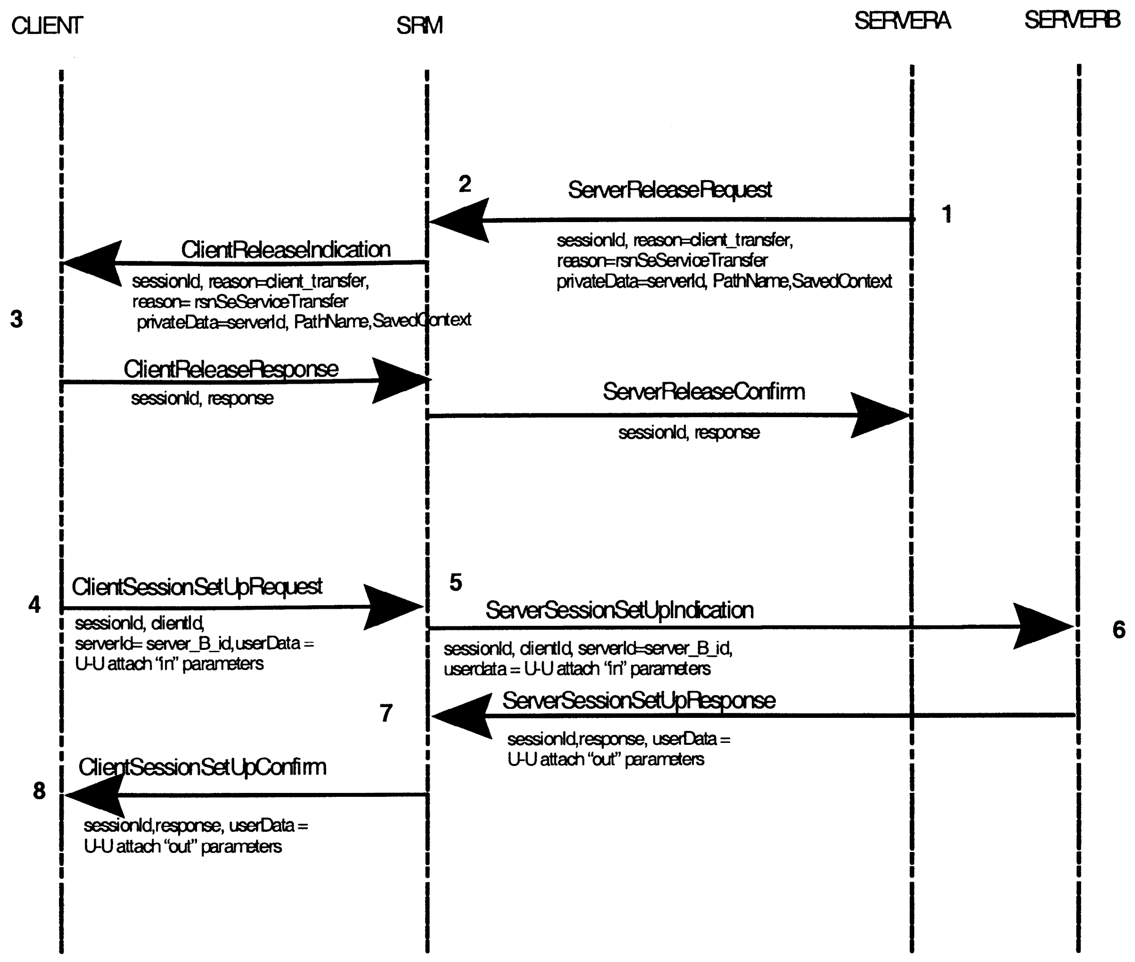

CLIENT
SRM
SERVERA
SERVERB
2
ServerReleaseRequest
1
sessionId, reason=client_transfer,
reason=rsnSeServiceTransfer
privateData=serverId, PathName,SavedContext
ClientReleaseIndication
sessionId, reason=client_transfer,
reason= rsnSeServiceTransfer
privateData=serverId, PathName,SavedContext
3
ClientReleaseResponse
sessionId, response
ServerReleaseConfirm
sessionId, response
4
ClientSessionSetUpRequest
sessionId, clientId,
serverId= server_B_id,userData =
U-U attach "in" parameters
5
ServerSessionSetUpIndication
sessionId, clientId, serverId=server_B_id,
userdata = U-U attach "in" parameters
6
7
ServerSessionSetUpResponse
sessionId,response, userData =
U-U attach "out" parameters
ClientSessionSetUpConfirm
8
sessionId,response, userData =
U-U attach "out" parameters

Figure L-7 Server Emergency Service Transfer

Step 1

For some emergency reason, such as an overload, the Server may want to shed some of the Clients. As a result, it provides ServerId information for a substitute Server, a PathName and a SavedContext in the privateData field of the ServerReleaseRequest, and the reason indicating the rsnSeServiceTransfer.

Step 2

Upon receipt of the ServerReleaseRequest, SRM sends ClientReleaseIndication to the Client with the reason and private data fields unchanged.

Step 3

The Client, when it receives the ClientReleaseIndication, proceeds with the release of the session and passes the private data content, including the reason for the release, to the U-U Library.

Step 4

The Session Object in the Client's U-U Library, when it receives the contents of the private data and the associated reason, requests the establishment of a session to the substitute server using information from the private data to access to the service.

Steps 5-8

A session is established to the substitute server and the Client accesses the service in the substitute service domain. The semantics of the Session Object are such that, to the application, the session would continue. The application does not detach() from the first service domain and attach() to the substitute service domain. Instead, it retains the previous Session object and is unaware of the substitute session domain.

Annex M
(informative)
T.120 Inter-working

M.1 Introduction

This annex provides informative guidelines to ITU-T Recommendations T.120-series and DSM-CC to allow each to benefit from the other standard, and suggests how inter-operation to its counterpart is feasible. These guidelines present: (a) a reference model, (b) a subclause defining features, functions and services of each standard, and a common vocabulary for the two standards, (c) a subclause which lists each standard's components and their relation to the other standard, and (d) the specifics for how to inter-operate from one standard to the other.

M.2 Reference Model for side-by-side integrated DSM-CC/T.120

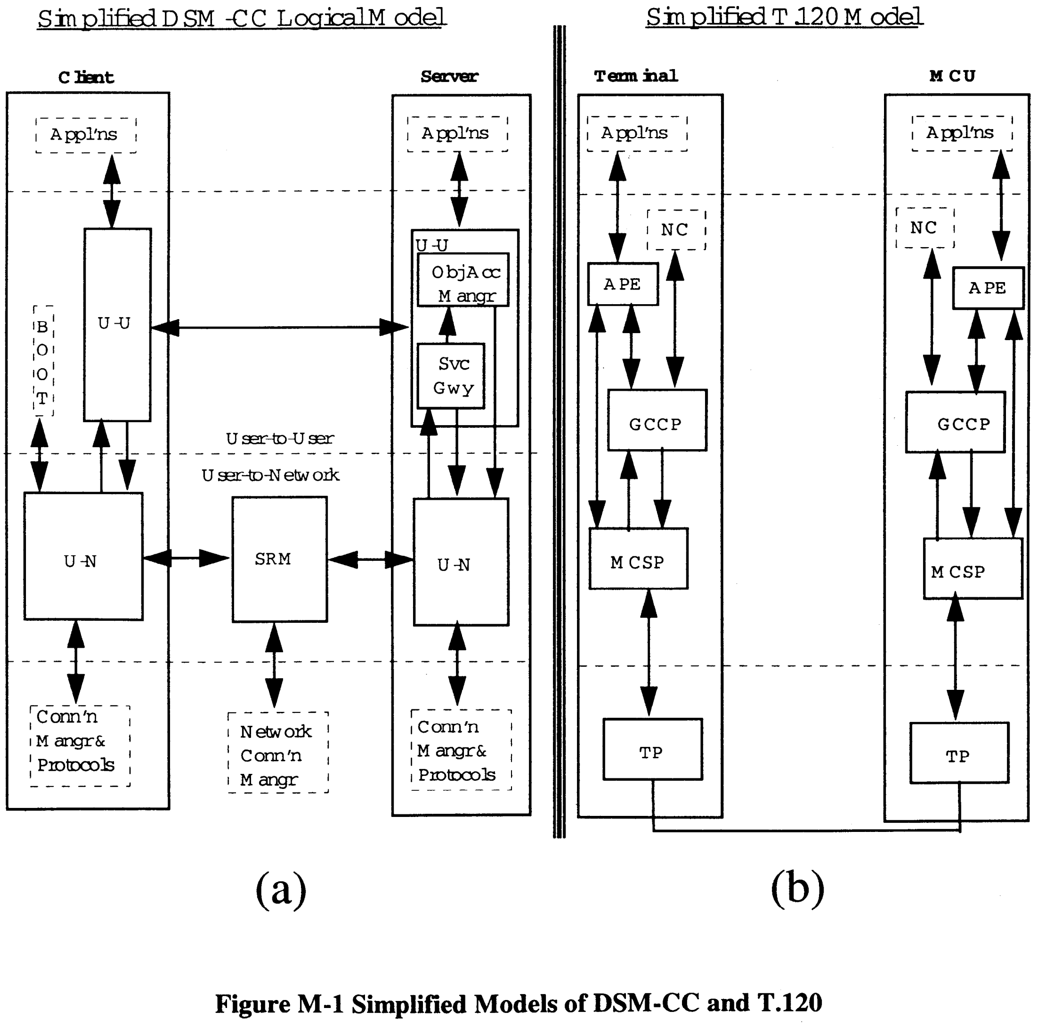


Figure M-1 Simplified Models of DSM-CC and T.120

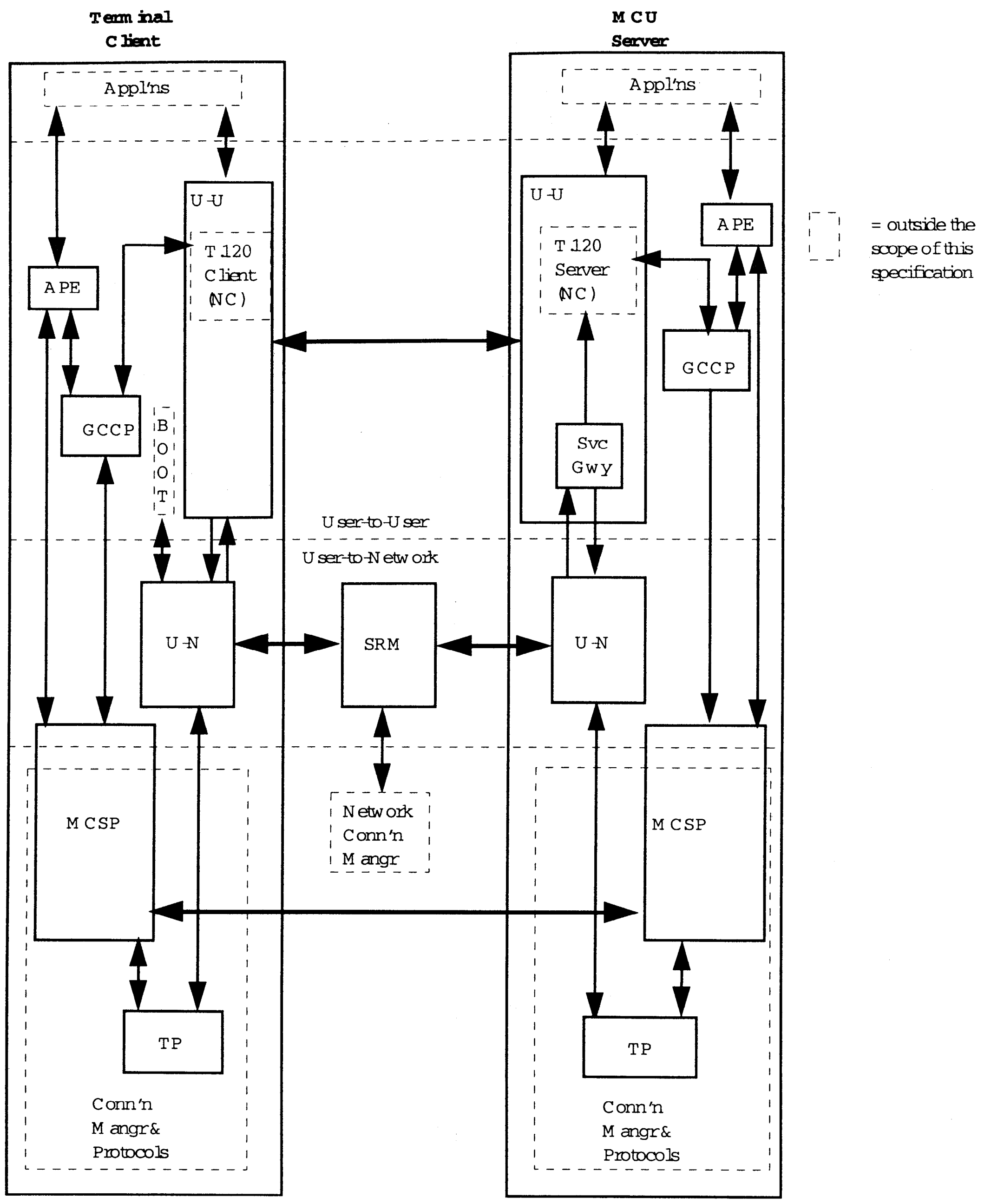


Figure M-2 Outline Reference Model for T.120/DSM-CC inter-working

While it can be argued successfully that the T.120 Node Controller (NC), Application Protocol Entity (APE), and Generic Conference Control Provider (GCCP) fall into the DSM-CC U-U domain, it is harder to place the Multi-point Communication Service Provider (MCSP) within the DSM-CC model.

MCSP has multiple functions. These cover setting up resources (i.e., attach, detach), management of the resources (e.g., channel join, channel convene, etc.) and also utilization of the resources (send data, uniform send data, etc.). The resources include connections (both point-to-point and multi-cast) and non-connection resources (tokens). On the one hand the resource management aspect of MCSP matches the User-to-Network (Session management and resource management) and on the other hand the control of multi-point (broadcast/multi-cast) connections falls under the connection manager.

With side-by-side integration of DSM-CC and T.120, the applications see both DSM-CC U-U API and T.120 APE. The TP is the common point where both DSM-CC U-N and the MCSP interact. The integrity of the MCSP is therefore preserved. The resulting model is shown in Figure M-2.

The characteristics of this model are:

- Hybrid DSM-CC and T.120 applications can use both the DSM-CC U-U portability interface and the T.120 application level (APE) primitives.
- Operation starts with the "boot process" initiating a session to a "conferencing server" (MCU). The client code download may include GCCP, APE etc.,
- Can use all of the DSM-CC features, functions and services in the application if desired.

M.3 Features, Functions and Services of the DSM-CC and the T.120 specifications

M.3.1 Features, Functions and Services of DSM-CC

1. Network independent uniquely identified sessions
   This feature allows a Client to initiate a session with a Service Provider. Sessions allow grouping and management of resources. Both connection resources and non-connection resources are permitted. DSM-CC allows Clients and Servers to use different connection media if desired e.g., at the Client side non-ATM HFC could be used while at the Server side an ATM SVC connection could be used to carry the same video stream or control data. Connections can be point-to-point (for Video on Demand) and multi-cast (for Digital Broadcast). An example of a non-connection resources is Digital Broadcast Service Continuous Feed.
2. Download
   DSM-CC allows a Client to begin a session without carrying application programs. These are downloaded to the Client as the need may arise based on the choice of services by the Client.
3. Defined interfaces for services
   Interfaces are defined for Directory, video stream control and file services.
4. Channel Change
   Client TV channel controls are defined for the Digital Broadcast Service to allow channel change to be carried out in the network.
5. Normal Play Time
   The MPEG Transport Stream is augmented with codes that allow VCR like control of the MPEG video stream.

M.3.2 Features, Functions and Services of T.120

1. Network independent
   Different networks could be used without impacting on the Multi-point Communication Service protocols.

2. Multi-point Communication Service
   Sets up individual and broadcast channels for conferencing, multi-cast private channels, allows control of tokens, and if requested carries data in a uniform sequence to multiple destinations.

3. Conference management
   Allows the setting of conference parameters by the convenor, permits different types of conference scenarios such as meet-me, call-out, call-through and point-to-point, prepares and distributes a conference roster of participants and of active applications.

4. Defined Interfaces for Applications
   Standard multi-point aware applications are specified and guidelines are defined for the incorporation of non-standard applications. The standard applications include simultaneous multi-point file transfer, still image viewing and annotation, shared white-board and facsimile. Work is presently being done on multi-point video.

M.3.3 Inter-working of DSM-CC and T.120 Features, Functions and Services

Table M-1 DSM-CC Session and Resources and their use by T.120

DSM-CC terminology	use in T.120

DSM-CC connection resource for data	- transport pipe for multi-point data. The transport connection MCS uses between 2 nodes can be mapped to this DSM-CC connection resource
DSM-CC connection resource for video/audio data	- transport pipe for video/audio data. T.130 could make use of the DSM-CC resource descriptor that describes this pipe when it ensures QOS between 2 points in the conference. This is relevant to T.131 Network Specific Mapping for DSM-CC.
DSM-CC session	- a DSM-CC session groups together all resources being used between a Client and a Server. This is a useful concept for the new T.130 series on control of multi-point audio/video/real-time data.

M.4 DSM-CC and T.120 Components Harmonized

Each DSM-CC component in Figure M-2 are explained below in relation to T.120 (Table M-2) and each T.120 component in relation to DSM-CC (Table M-3).

Table M-2 DSM-CC Components in relation to T.120

DSM-CC Component	**Relation to T.120**
Applications	A T.120 application that wishes to make use of DSM-CC would interact with DSM-CC using the U-U primitives defined in DSM-CC
BOOT (outside the scope of DSM-CC)	outside the scope of T.120
Client	terminal
Connection Manager and Protocols (outside the scope of DSM-CC)	MCSP connection setup and tear down
Network Connection Manager (outside the scope of DSM-CC)	outside the scope of T.120 - no T.120 counterpart
Server	MCU
Service Gateway	outside the scope of T.120
SRM - Session and Resource Manager	outside the scope of T.120 - no T.120 counterpart
T.120 Client/Server (NC)	Builds the link between DSM-CC and T.120 - it encompasses the T.120 Node Controller (NC) functionality. The T.120 NC drives the GCCP.
U-N box	DSM-CC U-N messages are used for U-N configuration and to set up a DSM-CC session and resources - can be used to set up a T.120 resource, whose handle can then be passed to the T.120 code (through the T.120 Client/NC)
U-U box	link between DSM-CC and T.120 is contained in this box - the T.120 Client/Server

Table M-3 T.120 components in relation to DSM-CC

T.120 Component	Relation to DSM-CC
Applications	A DSM-CC application that wishes to make use of the multi-point capabilities of T.120 would interact with T.120 through the T.120 Client/Server using T.121 Generic Application Template as a guideline
APE - Application Protocol Entity	outside the scope of DSM-CC. The T.120 APE provides services to applications at the same level as the DSM-CC U-U
GCCP - Generic Conference Control Provider	The T.120 Client/Server (NC) will interact with the GCCP to set up and control a T.120 conference. The DSM-CC U-N messages will have already set up the T.120 resource for multi-point data. The handle to this resource is passed to GCCP through the T.120 Client/Server (NC).
MCSP - Multi-point Communication Service Provider	The handle for the T.120 connection resource set up using DSM-CC U-N messages is passed to the MCSP through GCCP.
MCU - Multi-point Communication Unit	Server
NC - Node Controller (outside the scope of T.120)	encompassed in the T.120 Client/Server - this component becomes the link between DSM-CC U-U primitives and T.120 primitives.
Network Specific Mapping (T.131) to DSM-CC	outside the scope of DSM-CC
Terminal	Client
TP - Transport Provider (MCS Transport connections)	outside the scope of DSM-CC

M.5 Specifics for inter-operation between DSM-CC and T.120

From the perspective of ease of integration while maximizing the services to the user the side by side integration model appears to be the best suited for further work at this time. In order to evaluate this model an example of meet-me-conferencing is chosen between two terminals using one MCU server as shown in Figure M-3. In a meet-me conference, a conference is established at an MCU, and terminal nodes (as well as other MCUs if necessary) call into the MCU and join the conference.

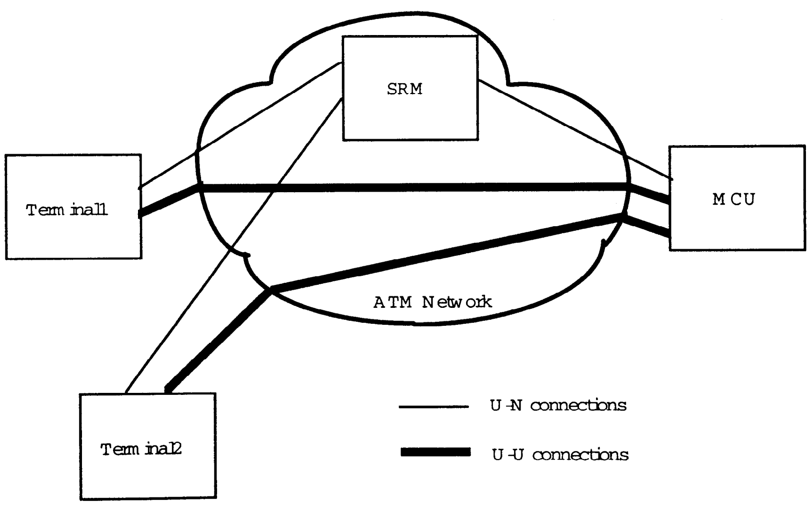


Figure M-3 Integrated DSM-CC/T.120 meet-me example with two terminals and one MCU

The scenario covers the following steps:

1. Terminal 1 creates a conference;
2. Terminal 2 queries for the conference name;
3. Terminal 2 then joins the conference.

Each of the above steps is covered below. The T.120 command flows are covered first, and DSM-CC with T.120 command flows are covered next.

M.5.1 Terminal 1 creates a conference

Figure M-4 shows the case of terminal 1 issuing a conference create using T.120.

Figure M-5 replicates the flow using the side by side integrated DSM-CC and T.120 model.

The boot code sends a ServiceGateway attach to the client U-N. The client U-N generates a ClientSessionSetUpRequest which includes the compatibility descriptors of the terminal. Upon receipt of the ServerSessionSetupIndication, the MCU U-N attaches and launches the T.120 server. The MCU server generates a SessionGateway addResource request with a connection resource descriptor that includes the QOS required for a T.120 data pipe and a connection resource if required for the download of the T.120 code to the Client. This causes the MCU U-N to issue a ServerAddResourceRequest (or, if the server is allowed to directly allocate resources, it goes ahead and allocates them using Q.2931 SETUP messages). Upon receipt of a ServerAddResourceConfirm (or upon a confirm that the requested Q.2931 resources were set up), the server issues a SessionGateway addResource reply to the T.120 server. The T.120 server then issues a ServiceGateway attach reply which is mapped to a ServerSessionSetupResponse, and arrives at the client as a ClientSessionSetupConfirm, including the connection bindings (T.120 use and download use) of the resource descriptors that have been allocated.

At this point, the handles to these resources are passed to the download code and later to the T.120 code to be used. The terminal 1 GCC Provider then proceeds with the GCC-Conference-Create.request, and then in the MCSConnectProvider.request. The steps followed are identical to the straight T.120 case.

M.5.2 Terminal 2 queries a conference

Figure M-6 shows the case of terminal 1 issuing a conference query using T.120.

Figure M-7 replicates the flow using the side by side integrated DSM-CC and T.120 model.

The approach used for the integration is similar to the case of terminal create.

M.5.3 Terminal 2 joins the conference

Figure M-8 shows the case of terminal 1 issuing a conference join using T.120.

Figure M-9 replicates the flow using the side by side integrated DSM-CC and T.120 model.

The approach used for the integration is similar to the case of terminal create.

It is likely that terminal 2, after performing conference query, will proceed directly with conference join. This scenario can therefore trail the query scenario above without the need for releasing and reestablishing a session to the same MCU.

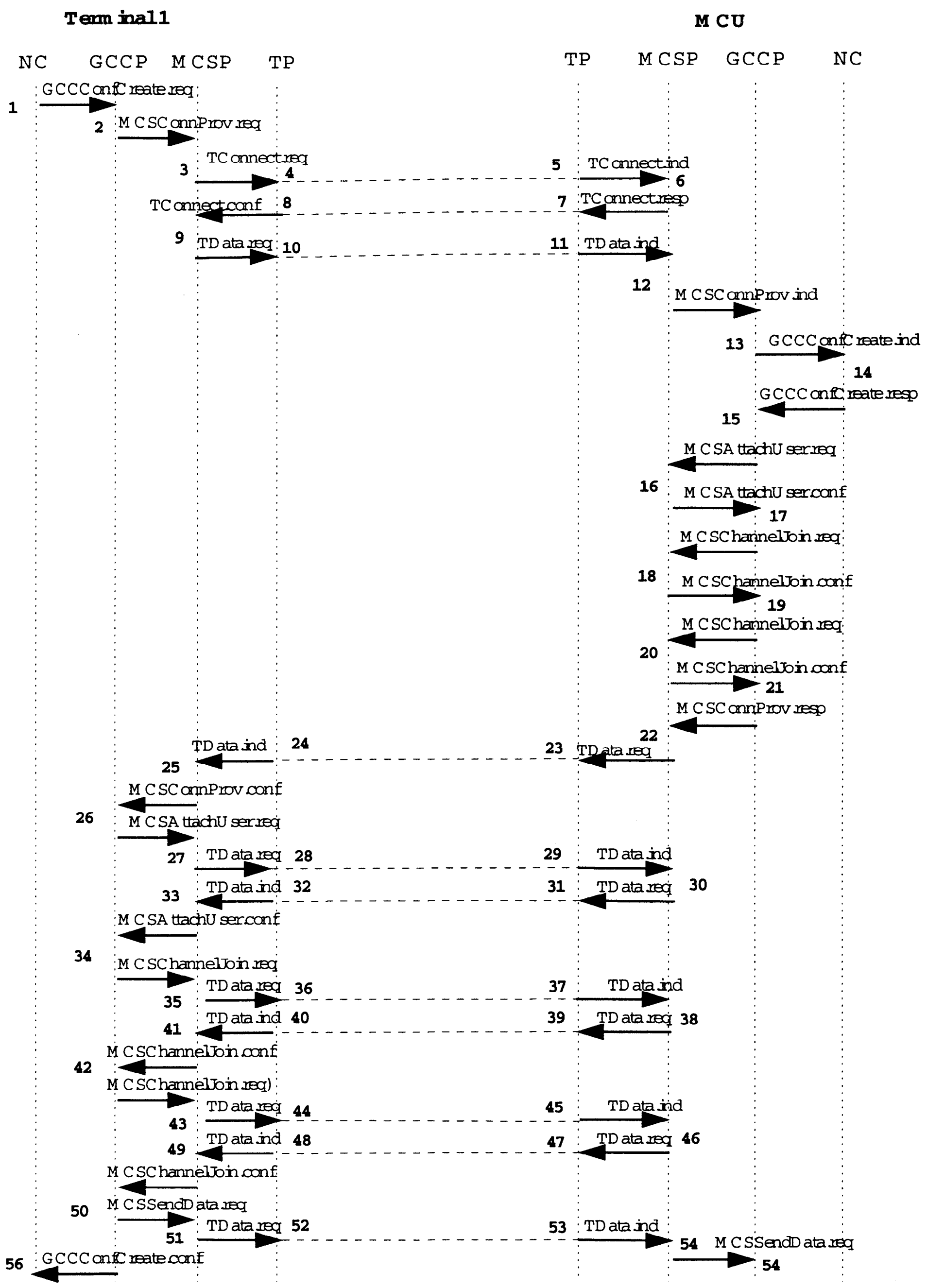


Figure M-4 Conference Create using T.120

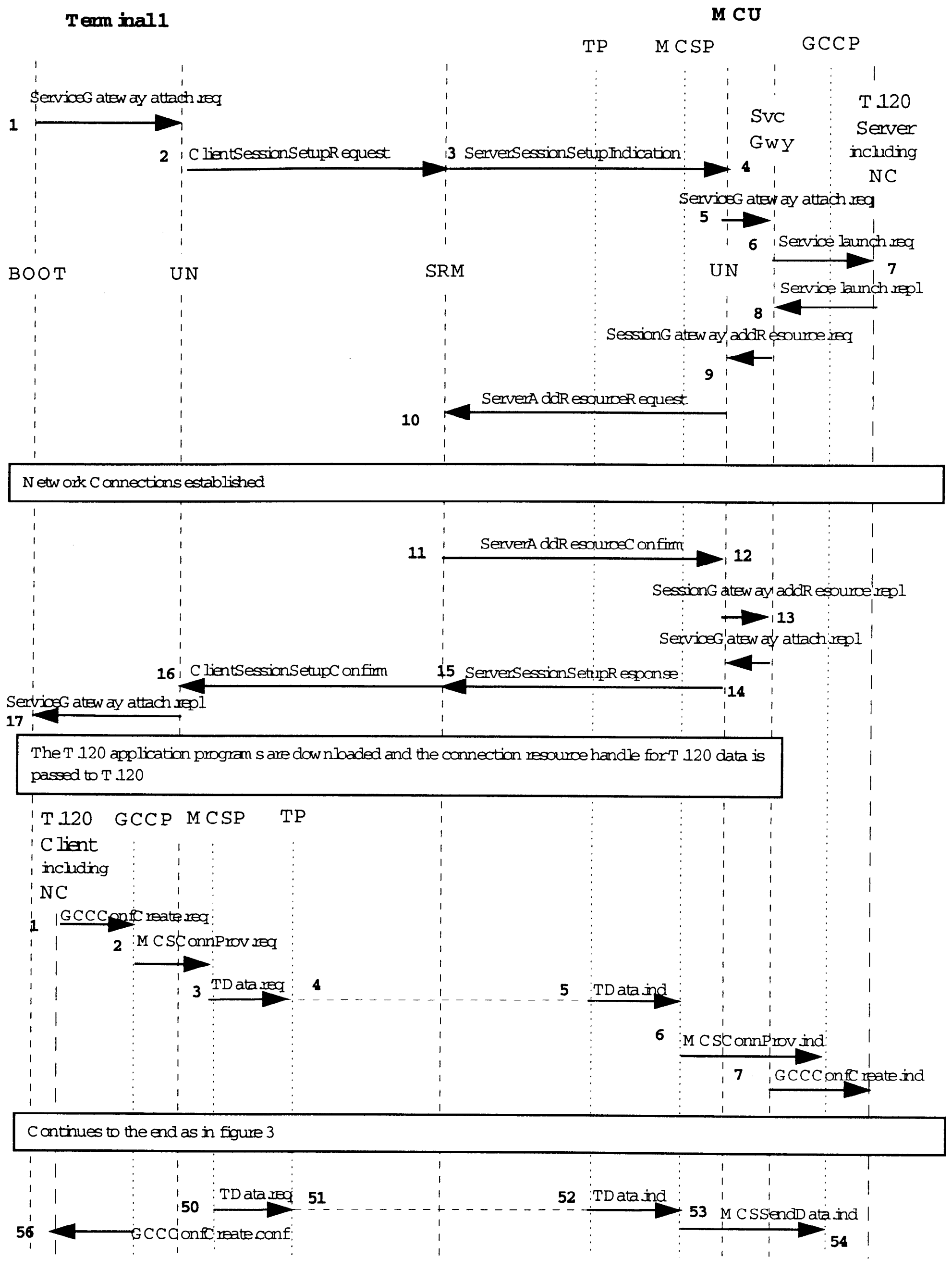

Terminal1
MCU
TP
MCSP
GCCP
1 ServiceGateway attach.req
2 ClientSessionSetupRequest
3 ServerSessionSetupIndication
4
Svc Gwy
T.120 Server including NC
5 ServiceGateway attach.req
6 Service launch.req
7 Service launch.repl
8
BOOT
UN
SRM
UN
SessionGateway addResource.req
9
10 ServerAddResourceRequest
Network Connections established
11 ServerAddResourceConfirm
12
SessionGateway addResource.repl
13
ServiceGateway attach.repl
14
15 ServerSessionSetupResponse
16 ClientSessionSetupConfirm
17 ServiceGateway attach.repl
The T.120 application programs are downloaded and the connection resource handle for T.120 data is passed to T.120
T.120 Client including NC
GCCP
MCSP
TP
1 GCCConfCreate.req
2 MCSConnProv.req
3 TData.req
4
5 TData.ind
6 MCSConnProv.ind
7 GCCConfCreate.ind
Continues to the end as in figure 3
50 TData.req
51
52 TData.ind
53 MCSSendData.ind
54
56 GCCConfCreate.conf

Figure M-5 Conference Create using DSM-CC with T.120

To highlight the interoperability issues the following table describes the flows and parameters for Figure M-5 "Conference Create using DSM-CC with T.120". No similar tables are provided for Conference Query and Conference Join because the issues are similar.

Table M-4 DSM-CC and T.120 - Conference Create - Table of Flows and Parameters

Flow	*Parameters*	*Comment*
1-2 ServiceGateway attach request	in inSC in aPrincipal in savedContext in pathName in downloadInfoReq	- Optional service context information passed between client and service - Identifies a specific user of terminal 1 - Provides context to resume service if session had been suspended - Names the path to DSM-CC U-U service on Server (in this example, the service on the Server knows that the Client will require a T.120 multi-point resource) - Client requests download, compatibility parameters are listed
2-3 ClientSessionSetupRequest	sessionId clientId serverId uuData privateData	- A unique session identifier assigned by terminal 1 or SRM based on terminal 1 User-to-Network configuration - Client identifier in NSAP format could be the E.164 NSAP ATM SVC address of terminal 1 - Server identifier in NSAP format could be the E.164 NSAP ATM SVC address of the MCU - Carries an on the wire encoded format of the ServiceGateway attach in parameters from 1-2 - Not used
3-4 ServerSession SetupIndication	sessionId clientId serverId uuData privateData loop(forwardCount, forwardServerId)	- Same as in 2-3 - Contains the serverIds the ServerSessionSetupIndication was forwarded to - in this case it is NULL
5-6 ServiceGateway attach request	in inSC in aPrincipal in savedContext in pathName in downloadInfoReq	- Same as 1-2
6-7 Service launch request	in aResume in aUserContext in rSessGateway in aPrincipal in rNetResources in downloadInfoReq	- Indicates if this is a resumption of a suspended DSM-CC U-U service - Provides the user context to resume the DSM-CC U-U service - Contains the object reference of the User-to-Network Session Gateway - Specifies a specific user of the terminal 1 from 1-2 - Initial network resources available for the DSM-CC U-U service is NULL - client request for download, including client's compatibility parameters
7-8 Service launch reply	out downloadInfoResp	- T.120 requirements to be downloaded to the client, based on the client's compatibility parameters - Download services will be invoked to actually download the required code.
8-9 SessionGateway addResource request	aUserContext rReqResources	- Provides user context - Provides requested resources - In this case, the DSM-CC U-U service is requesting a T.120 resource with T.120 QOS

Flow	*Parameters*	*Comment*
9-10 ServerAdd ResourceRequest	sessionId loop(resourceCount, resourceDescriptor)	- from 3-4 ServerSessionSetupIndication - contains the resource descriptors to set up required T.120 connections - NOTE: if the Server is to allocate the resource, steps 9-10 & 11-12 can be omitted, and the result of the resource allocation will go in the resource descriptor in the ServerSessionSetUpResponse/ ClientSessionSetupConfirm (14-16)
10-11		- Network Connections are established
11-12 ServerAdd Resource Confirm	sessionId loop(resourceCount, resourceDescriptor) response	- same as 9-10
12-13 SessionGateway addResource reply	rActResources	- Actual Resources allocated
13-14 ServiceGateway attach reply	out outSC out downloadInfoResp out resolvedRefs	- Optional service context information passed between client and service - No download info response - The Interoperable Object References corresponding to resolving of pathName
14-15 ServerSession SetupResponse	session Id serverId nextServerId loop(resourceCount, resourceDescriptor) uuData privateData response	- A unique session identifier from 3-4 - Server identifier of responding server in NSAP format - could be the E.164 NSAP ATM SVC address of the MCU - This is the serverId of the Server to which this session is to be forwarded - Identifies resources allocated to the session - redundant if AddResource messages were exchanged, not redundant if the Server was the resource allocator, which means no AddResource messages were exchanged - Carries an on the wire encoded format of the ServiceGateway attach reply parameters from 13-14 - Not used - Indicates success or failure of the session setup
15-16 ClientSession SetupConfirm	session Id serverId uuData privateData response	- Same as 14-15 (except no nextServerId)
16-17 ServiceGateway attach reply	out outSC out downloadInfoResp out resolvedRefs	- Same as 13-14
17-1		T.120 applications and code are downloaded and the connection resource handle for the T.120 data pipe is passed to T.120

Flow	*Parameters*	*Comment*
1-2 GCC- Conference- Create.req	confName confLocked confConductible termMethod domainParameters calling address called address qOS	- Name by which the conference to be created is identified - When set, prevents anyone from joining this conference unless they are specifically added using GCC-Conference-Add primitive - When set indicates that this conference may be placed in conducted mode using GCC-Conductor-Assign primitive - Indicates manual (using GCC-Conference-Terminate primitive) or automatic when no joined nodes remain - MCS domain parameters - protocol version, max. height of MCS providers, max. size of MCSPDUs - Local handle to the T.120 resources set up earlier using DSM-CC - Local handle to the T.120 resources set up earlier using DSM-CC - not used - required QOS for this connection should have been set in the resource descriptors used to set up the T.120 connections by the Server when it issued the AddResource for the T.120 resources
2-3 MCS-Connect- Provider.req	calling address calling domain selector called address up/down Flag domain parameters qOS userData	- Local handle to the T.120 resources set up earlier using DSM-CC - Conference name from 1-2 in GCCConferenceCreate.req - Local handle to the T.120 resources set up earlier using DSM-CC - Up - Same as 1-2 in GCCConferenceCreate.req - Not needed - DSM-CC T.120 Resources have already been set up - contains the ConferenceCreateRequest GCCPDU
3-4 TData.req	userData	- contains the ConnectInitial MCSPDU
4-5 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
5-6 TData.ind	userData	- same as Tdata.req above
6-7 MCS-Connect- Provider.ind	calling address calling domain selector called address up/down Flag domain parameters qOS userData	- same as MCS-Connect-Provider.req 2-3 - same as 2-3 - Local handle to the T.120 resources set up earlier using DSM-CC - same as 2-3 - May have been modified from 2-3 - Not needed - DSM-CC T.120 Resources have already been set up - Same as 2-3
7-8 GCC- Conference- Create.ind	conferenceName conferenceId domainParameters qOS	- Same as 1-2 - Locally allocated identifier of the newly created conference - May have been modified from 1-2 - Not needed
14-15 GCC- Conference- Create.resp	conferenceName conferenceId tag result	- Same as 7-8 - Same as 7-8 - This parameter is used to identify the returned UserIdIndication GCCPDU sent later - An indication of whether the request was accepted or rejected

Flow	*Parameters*	*Comment*
15-16 MCSAttachUser. req	domainSelector	- Set to conferenceId
16-17 MCSAttachUser. conf	userId result	- unique MCS UserId - An indication of whether the request was accepted or rejected
17-18 MCSChannel Join.req	channelId	- channelId = userId from MCSAttachUser.conf
18-19 MCSChannel Join.conf	channelId result	- channel joined - An indication of whether the request was accepted or rejected
19-20 MCSChannel Join.req	channelId	- identifies channel to join as the GCC Broadcast Channel
20-21 MCSChannel Join.conf	channelId result	- channel joined - An indication of whether the request was accepted or rejected
21-22 MCS-Connect-Provider.resp	domainParameters qOS result userData	- May have been modified - Not needed - An indication of whether the request was accepted or rejected - Contains the ConferenceCreateResponse GCCPDU
22-23 TData.req	userData	- contains the ConnectResponse MCSPDU
23-24 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
24-25 TData.ind	userData	- same as Tdata.req above
25-26 MCS-Connect-Provider.conf	domainParameters qOS result userData	- same as21-22
26-27 MCSAttachUser. req	domainSelector	- Set to conferenceId
27-28 TData.req	userData	- contains the AUrq MCSPDU
28-29 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
29-30 TData.ind	userData	- same as Tdata.req above
30-31 TData.req	userData	- contains the AUcf MCSPDU
31-32 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
32-33 TData.ind	userData	- same as Tdata.req above
33-34 MCSAttach User.conf	userId result	- unique MCS UserId - An indication of whether the request was accepted or rejected
34-35 MCSChannel Join.req	channelId	- channelId = userId from MCSAttachUser.conf

Flow	*Parameters*	*Comment*
35-36 TData.req	userData	- contains the CJrq MCSPDU
36-37 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
37-38 TData.ind	userData	- same as Tdata.req above
38-39 TData.req	userData	- contains the CJcf MCSPDU
39-40 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
40-41 TData.ind	userData	- same as Tdata.req above
41-42 MCSChannel Join.conf	channelId result	- channel joined - An indication of whether the request was accepted or rejected
42-43 MCSChannel Join.req	channelId	- identifies channel to join as the GCC Broadcast Channel
43-44 TData.req	userData	- contains the CJrq MCSPDU
44-45 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
45-46 TData.ind	userData	- same as Tdata.req above
46-47 TData.req	userData	- contains the CJcf MCSPDU
47-48 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
48-49 TData.ind	userData	- same as Tdata.req above
49-50 MCSChannel Join.conf	channelId result	- channel joined - An indication of whether the request was accepted or rejected
50-51 MCS-Send-Data.req	priority channelId userData	- Top - channelId=userId channel of the node directly above this one - Contains the UserIDIndication GCCPDU, containing the tag from earlier
51-52 TData.req	userData	- contains the SDrq MCSPDU
52-53 UserPlane data		- Maps to the appropriate network protocol - e.g. to AAL5 in the case of ATM SVC
53-54 TData.ind	userData	- same as Tdata.req above
54-55 MCS-Send-Data.ind	priority channelId userData sendUserId	- Same as 50-51 MCS-Send-Data.req - Terminal 1 GCCUserId (=NodeId)
50-56 GCC-Conference-Create.conf	conferenceName conferenceId tag result	- Same as 14-15 GCC-Conference-Create.resp

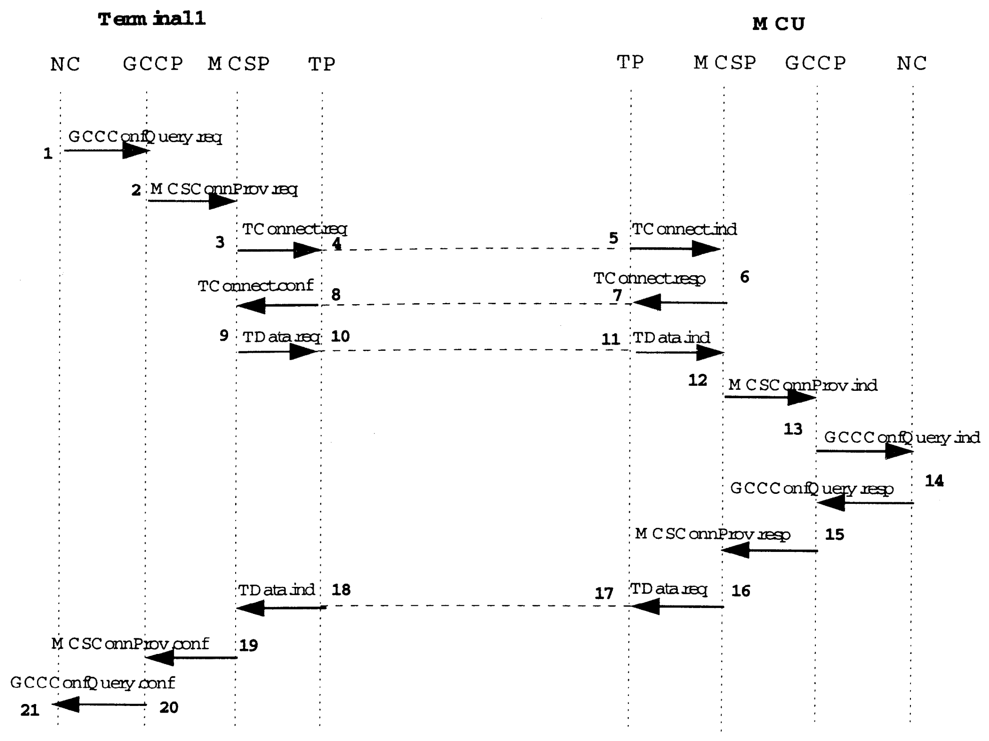


Figure M-6 Conference Query using T.120

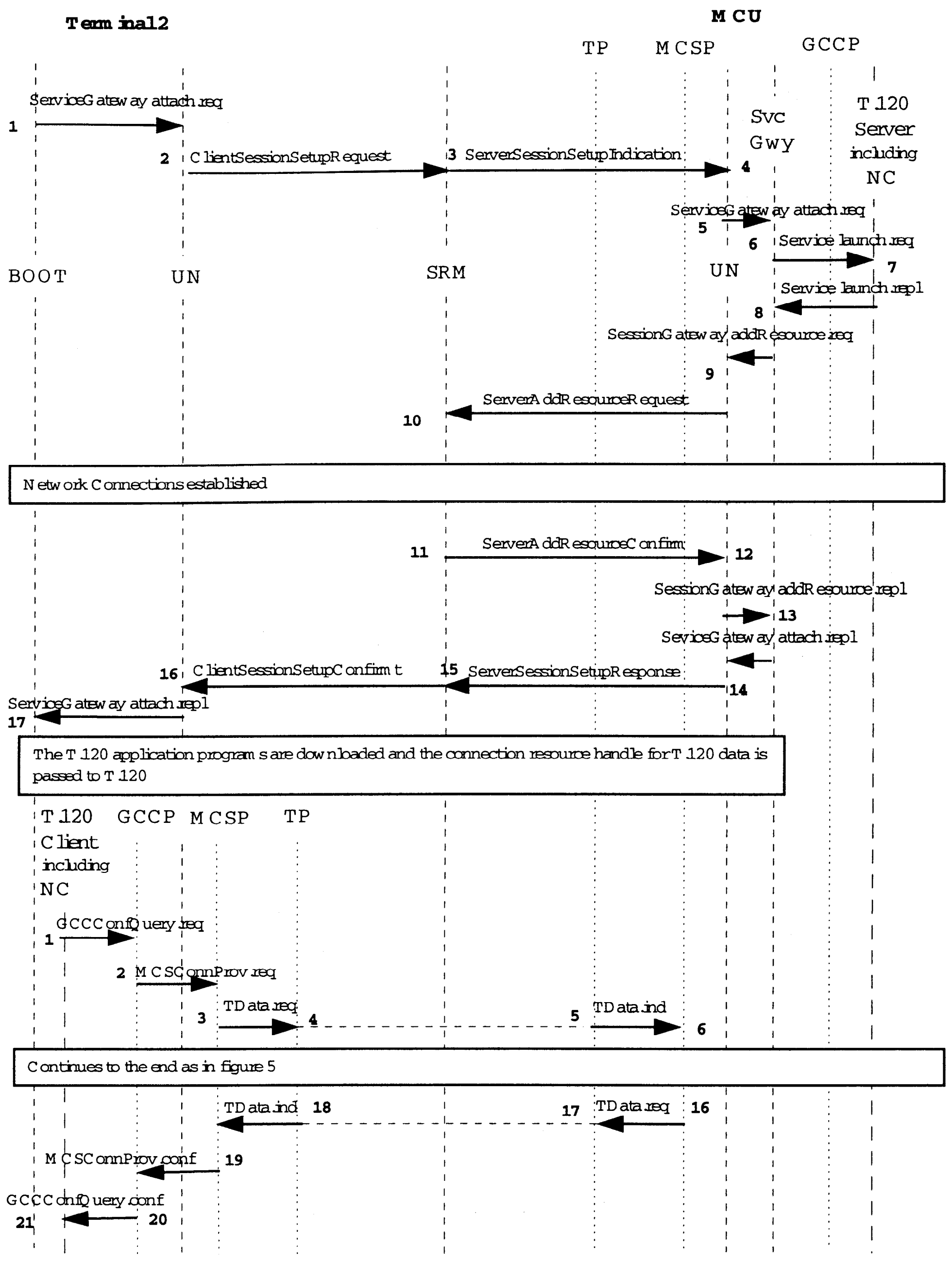

Terminal2
MCU
TP
MCSP
GCCP
Svc Gwy
T.120 Server including NC
ServiceGateway attach.req
1
2 ClientSessionSetupRequest
3 ServerSessionSetupIndication
4
ServiceGateway attach.req
5
6 Service launch.req
7
BOOT
UN
SRM
UN
Service launch.repl
8
SessionGateway addResource.req
9
ServerAddResourceRequest
10
Network Connections established
11 ServerAddResourceConfirm
12
SessionGateway addResource.repl
13
ServiceGateway attach.repl
16 ClientSessionSetupConfirm
15 ServerSessionSetupResponse
14
ServiceGateway attach.repl
17
The T.120 application programs are downloaded and the connection resource handle for T.120 data is passed to T.120
T.120 Client including NC
GCCP
MCSP
TP
GCCConfQuery.req
1
2 MCSConnProv.req
TData.req
3
4
5
TData.ind
6
Continues to the end as in figure 5
TData.ind 18
17 TData.req 16
MCSConnProv.conf
19
GCCConfQuery.conf
21
20

Figure M-7 Conference Query using DSM-CC with T.120

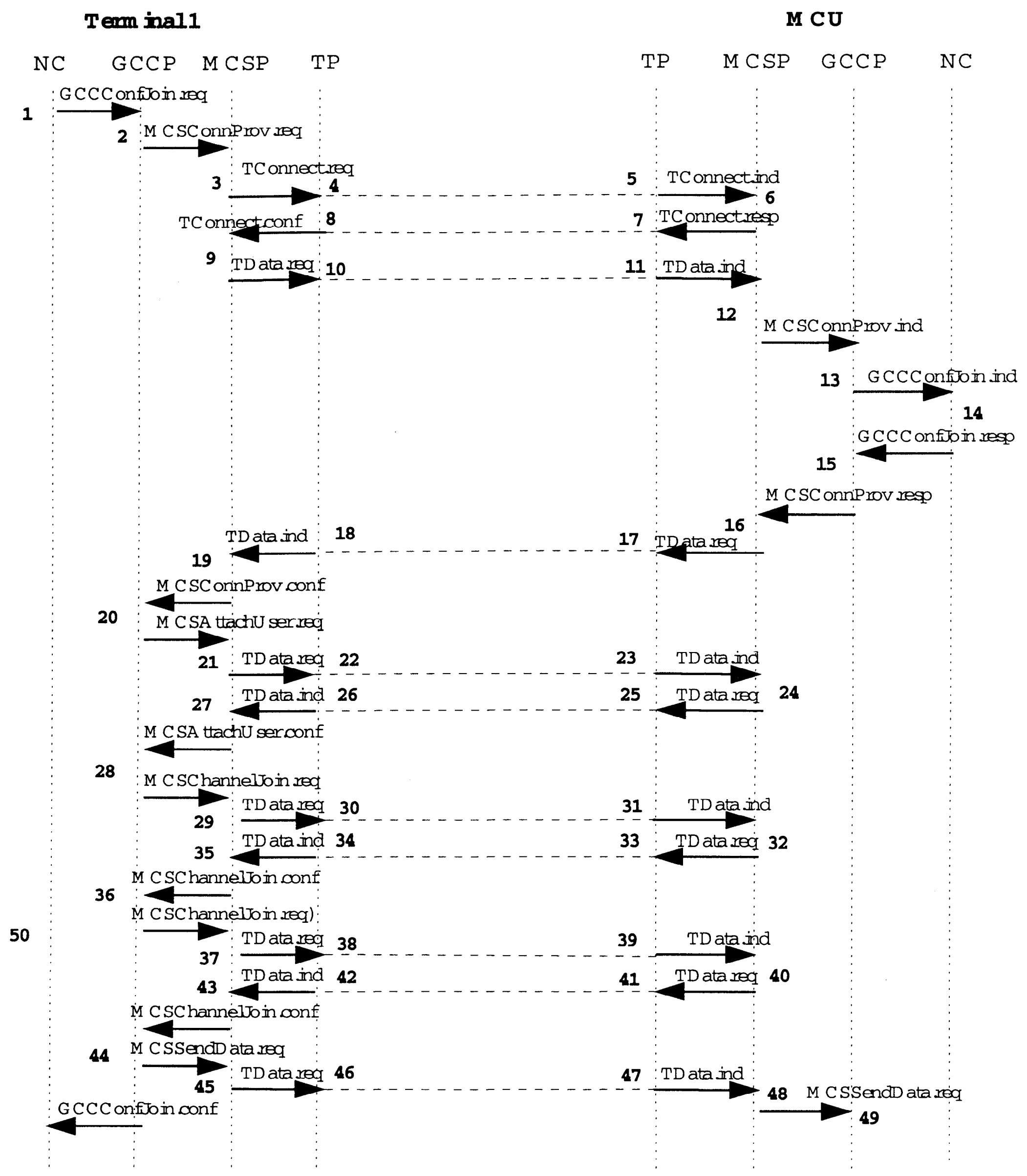


Figure M-8 Conference Join using T.120

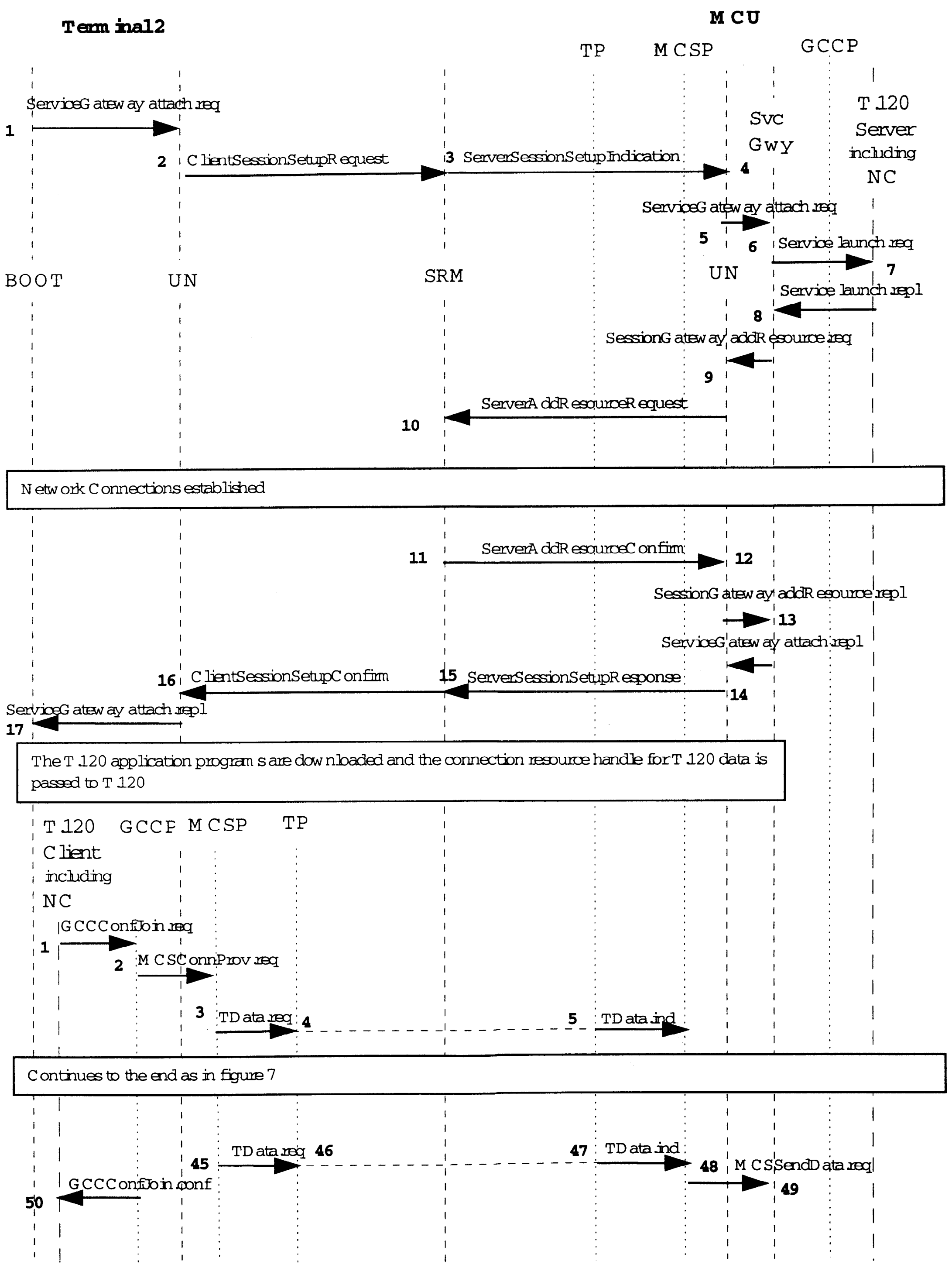

Terminal2
MCU
TP
MCSP
GCCP
T.120 Server including NC
Svc Gwy
1 ServiceGateway attach req
2 ClientSessionSetupRequest
3 ServerSessionSetupIndication
4
5 ServiceGateway attach req
6 Service launch req
7
BOOT
UN
SRM
UN
Service launch repl
8
SessionGateway addResource req
9
ServerAddResourceRequest
10
Network Connections established
11 ServerAddResourceConfirm
12
SessionGateway addResource repl
13
ServiceGateway attach repl
14
15 ServerSessionSetupResponse
16 ClientSessionSetupConfirm
ServiceGateway attach repl
17
The T.120 application programs are downloaded and the connection resource handle for T.120 data is passed to T.120
T.120 Client including NC
GCCP
MCSP
TP
1 GCCConfJoin req
2 MCSConnProv req
3 TData req
4
5 TData ind
Continues to the end as in figure 7
45 TData req
46
47 TData ind
48 MCSSendData req
49
GCCConfJoin conf
50

Figure M-9 Conference Join using DSM-CC with T.120

M.6 T.120 service within DSM-CC

This subclause is provided as a proposal to stimulate potential future work for the extension of DSM-CC to cover T.120 functionality in conjunction with the ITU-T SG8 group.

In this case the T.120 functions are absorbed into the appropriate subsystems within the DSM-CC framework. The characteristics of this model are:

- Applications are built to DSM-CC model
- Define IDL for manipulations of new types of taps for T.120 connections
- A T.120 set of resources (new resource descriptor types) could be provided by the network.
- The DSM-CC services such as directory, video stream control and file can access the multi-point communication service features for perusal by the conference participants.

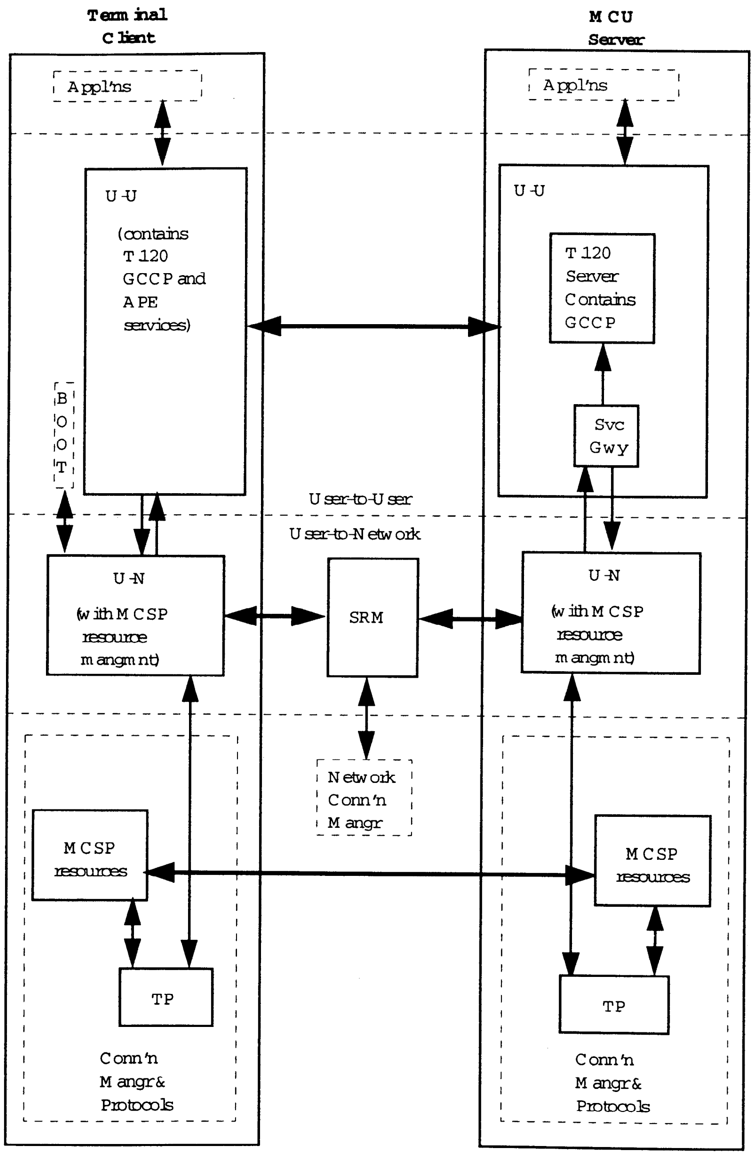


Figure M-10 T.120 service within DSM-CC

M.6.1 An Example of Extending DSM-UU to provide custom interfaces

The example below illustrates how DSM-CC User-to-User can be extended in a custom application. This is provided here as an illustration of a similar use for theT.120.series of protocols.

The objective of this subclause is to present an example which illustrates how such an extension of the interface can take place. The example relates to event distribution. A portion of the Event interface of subclause 5.7.2.4 is shown below:

```
interface Event {
        void subscribe(in string aEventName,
                out u_short aEventId)
                raises(INV_EVENT_NAME);
        void unsubscribe(in u_short aEventId)
                raises(INV_EVENT_ID);
};
```

The standard interface does not specify how the events reach the event target but specifies a mechanism to distribute events through the media stream. The implication of the mechanism is that, to receive the events, the client device must also receive the media stream. The example of this subclause extends the Event interface to support event targets which do not receive a media stream.

The concept is that there is an event source which distributes events to event targets. (Note: The example is not a complete event architecture. A more robust solution is found in the Common Object Services: Event Service of the Object Management Group.) The event source provides the interface below:

```
exception NotConnect {
};
interface EventSource {
        void connect(in EventTarget aEventTarget);
        void disconnect()
                raises(NotConnect);
};
```

The event target provides the interface below:

```
interface EventTarget {
        void push(in any aEvent);
        disconect()
                raises(NotConnect);
};
```

The client of the EventSource interface provides the object reference to the object instance which provides the EventTarget interface. Note: The client of the EventSource interface and the object instance which provides the EventTarget interface could differ. The client, in this case, delegates to another object to field the events. If the client later decides to not receive events, it invokes disconnect().

The EventSource object invokes push() on the EventTarget object to forward the event. The EventSource object can also disconnect() the EventTarget object.

The example packages the standard Stream interface, the standard Event interface, and the above EventSource interface together into one interface which inherits these abstract interfaces:

```
interface StreamEvent : Stream, EventSource, Event {
};
```

Given the above interface definition, it is possible to describe the steps the service side and the client side take to build, install, and activate the StreamEvent object. Starting with the service side.

1) The service compiles the Interface Definition Language into its favorite implementation language. It builds the software which implements the interface. It installs the implementation on device hardware.

2) To publish the object, the service would invoke the bind() function found in subclause 5.7.1.6. The interface declaration for just the bind() function is:

```
bind(in Name aName, in Object aObjRef)

        raises(NotFound, CannotProceed, InvalidName, AlreadBound);
```

The complete declaration is found in subclause 5.7.1.6. The topic to observe here is that the service would select a service name, for example StreamEventService. It would define the leaf name of the name graph into which it installs the service as two fields both of which are strings. The first is the service name; the second is the interface name which here is StreamEvent.

The service could also install a name where the interface field is Stream rather than StreamEvent. The motivation in this case is to install a name which clients not aware of the StreamEvent subclass would recognize. These clients would select the Stream Service and expect to invoke just the old Stream interface.

3) Interfaces within the service domain are not prescribed. To complete the example, however, the discussion illustrates how the service could integrate itself with the service domain. The requirement is to provide an interface which other objects in the service domain invoke to activate the service. A technique which is common is the Object Factory. The Object Factory has one function which is to create a service.

On the client side, the example assumes the client understands the StreamEvent interface. The steps which the client takes are:

1. The client browses the available services. Because the Name it encounters in the name graph provides fields for both the service and the interface, it could invoke the list() function to detect a Stream Service which supports the StreamEvent interface before it activates the service.
2. When the client encounters a Stream Service which exports the StreamEvent interface, it resolve()s the name, which returns the object reference to the Stream Event Service.
3. The client then invokes the StreamEvent interface, either functions for the standard stream control, or functions to subscribe to events which the Stream Service is to distribute to the EventTarget.

Annex N (informative) The Relation of DSM-CC to MHEG-5

N.1 Overview

MHEG-5 (ISO/IEC 13522-5) provides a framework for the distribution of interactive multimedia applications across minimal resource platforms of different types. A MHEG-5 application resides on a server, and as portions of the application are needed, they will be downloaded to the client. In a broadcast environment, this download mechanism could rely, for instance, on cyclic re-broadcasting of all portions of the application.

A minimal MHEG-5 runtime environment has to provide an entity for decoding of the MHEG data structures and an entity called MHEG engine, which parses and interprets the MHEG-5 objects. The engine also communicates with the local presentation environment and the MHEG-5 objects. It responds to the events initiated by the application or the user (for example timer events, or "button pressed") in the application specific way. A MHEG application is always event driven.

An MHEG-5 application consists of Scenes objects and objects that are common to all Scenes within an Application object. At most one Scene is active at any one time. This is the part of the application that has to loaded on the clients' system. Navigation in an application is done by transitioning between Scenes. A Scene contains a group of objects, called Ingredients, that represent information (graphics, sound, video, etc.). The content data is typically not part of the encoded Scene object. Instead, content data can be referenced and stored externally.

A MHEG-5 runtime system can utilize DSM-CC for access to MHEG objects and content data. Such a system is sketched below (the design of the system shown below is not normative by MHEG-5, nor MPEG-2 DSM-CC):

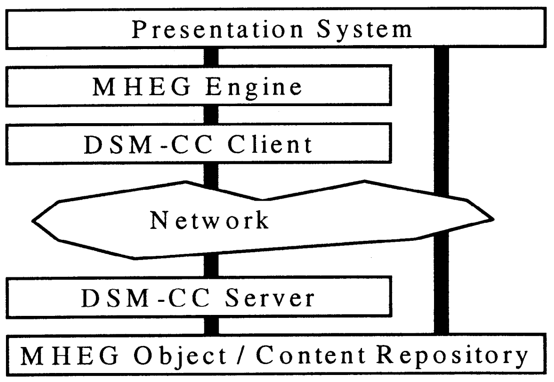


Figure N-1 Example of MHEG-5 runtime system

The following subclauses provide examples and hints, how DSM-CC facilities can be used by an MHEG-5 runtime system. Further information on details of MHEG-5 facilities are available in the MHEG-5 standard.

N.2 Name Space

When an application starts, it is assumed that a service gateway has been located and attached to, so that there is exactly one name space within which the application objects are located. Within that name space, a service has also been located. That service is a DSM-CC directory; within it, there can be other directories, files, and streams.

Furthermore, it is assumed that each object belongs to exactly one application. This assumption is necessary to allow for unambiguous MHEG object references. Three types of retrieved data can be differentiated:

- objects that comply to MHEG-5 (Scenes or Applications)
- the content (such as bitmaps or text) of those objects
- streams (such as video and audio)

For accessing the various objects of an application on the server side, the DSM-CC Directory, File and Stream objects are used. Note that the server, in this context, does not have to be a physical server, but could be implemented, for example, as a broadcast carousel in a pure-broadcast topology.

Each file is either a Scene object, an Application object, or the content data of an Ingredient object, belonging to a Scene or an Application. Each Scene object, Application object and content data is stored in a separate file.

N.2.1 MHEG Object References

MHEG objects and content data can be exchanged in two ways: either the object is exchanged as a DSM-CC file object or within another object. The former method is used for Applications and Scenes, the latter for all other objects contained in a scene or application object. The MHEG objects are identified by an `ObjectReference`, consisting of an optional byte string `GroupIdentifier`, followed by an integer, the `ObjectNumber`. For the `GroupIdentifier`, the following rules may be defined by the application domain:

All `GroupIdentifiers` are ASCII strings. A `GroupIdentifier` reference is mapped on a DSM-CC name within the name space of the service gateway to which the runtime is attached. Within the `GroupIdentifier`, the character '/' (standard ASCII slash) is used to delimit directory references (of the 'depth' type); for instance, if the `GroupIdentifier` is 'apps/otherAppl', it is mapped on the DSM-CC name 'otherAppl' in the directory 'apps' of the service gateway to which the runtime has attached.

Additional rules may be defined (e.g. for shortcut/wildcards of application identifiers or references to the current directory root) if required. For the mapping on DSM-CC, the following additional rules may be used:

- Each Application and Scene object shall have in its `GroupIdentifier` a byte string which maps on the name of the DSM-CC file which contains that object. These objects have their ObjectNumber set to 0 (normative by MHEG-5).
- Each application shall have exactly one Application object. That object shall be contained in a DSM-CC File object with the special name, e.g. 'startup' (normative by the application domain).
- References to content data objects may (as defined by the application domain)
    1. either leave out the `GroupIdentifier`, in which case it is assumed to be a string which maps on the name of a DSM-CC file which contains the object (Application or Scene) of which this object is a part, or
    2. fill in the `GroupIdentifier` with such a string.
- Such objects shall have their ObjectNumber set to a value which is unique within that file (normative by MHEG-5).

N.2.2 Content References

The high-level API has a separate way of referencing the external content of objects belonging to the Ingredient class. This is done by way of a `ContentReference`. The `ContentReference` consists of a byte string. The following rule may be defined by the application domain:

- The exact same mapping shall be used as for the `GroupIdentifier` above;
- the relative name of the content object file is appended to the `GroupIdentifier`, separated with a '/'.

N.3 Stream Events and Normal Play Time

The DSM-CC StreamEvent interface provides the possibility to carry private data in the data structure for the event, in the form of the PrivateDataByte field. These bytes shall be mapped one-to-one on the `StreamEventTag` of the MHEG-5 event `StreamEvent`.
The MHEG-5 internal attribute `CounterPosition` of the Stream class shall also be mapped one-to-one on the value of the DSM-CC Normal Play Time of the corresponding stream.

N.4 Example of DSM-CC file structure for an application

This subclause shows how MHEG-5 objects may be mapped to DSM-CC file structure.

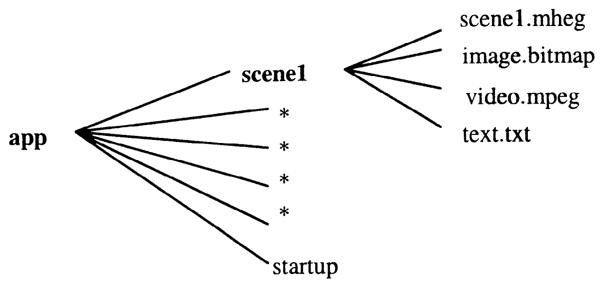


Figure N-2 DSM-CC file structure example

Below is one textual representation of the code for accessing the different objects depicted in the figure above. The first is an Application Object, performing a transition to the first scene. The Scene object identifies the content objects which belong to the scene by referencing the content files.

```
{:application
      :object-identifier ( "app/startup" 0 )
      :on-startup ( :transition-to ( app/scene1/scene1.mheg)
}

{:scene
      :object-identifier ( "app/scene1/scene1.mheg" 0 )
      (:bitmap 1
            (:content-data
       :referenced-content "app/scene1/image.bitmap") ...)
      (:video 2
            (:content-data
       :referenced-content "app/scene1/video.mpeg") ...)
      (:text 3
            (:content-data
       :referenced-content "app/scene1/text.txt") ...))
}
```

N.5 Example of Mapping High-Level API Actions on DSM-CC U-U Primitives

Below is a possible example of a "translation" of MHEG-5 actions to DSM-CC U-U primitives.

Table N-1 Example of MHEG-5 translation to DSM-CC U-U primitives

MHEG-5 Behavior	Object Type	DSM-CC U-U Function
Launch/Spawn	Application	DirectoryOpen*(app.fileid) -> FileObRef* FileRead*(FileObRef)*
Prepare	Scene, content object and stream	DirectoryOpen*(scene.fileid) -> FileObRef* FileRead *(FileObRef)*
Run	Video and Audio	DirectoryOpen*(stream.file) - >StreamObRef* StreamResume(*StreamObRef, starttime, 1/1*)
SetSpeed(0)	Stream	StreamPause(*StreamObRef,* x80000000.x00000000)
Stop	Stream	StreamClose(StreamObRef)
StreamMarker	Stream	StreamSubscribe (*StreamObRef, marker*) StreamNotify (*StreamObRef, marker, call back function*) StreamUnSubscribe (*StreamObRef, marker)*
StreamTimer	Stream	StreamStatus -> Gets normal playtime
Call and Fork	Application	RPC - UNO
OpenConnection	Application	Attach *(ID)*

ICS 35.040

Descriptors: data processing, moving pictures, image processing, video recording, sound recording, video data, audio data, data converting, coding (data conversion), control procedures.

Price based on 542 pages