
**Information technology — Coding of
audio-visual objects —**

Part 6:
**Delivery Multimedia Integration Framework
(DMIF)**

*Technologies de l'information — Codage des objets audiovisuels —
Partie 6: Charpente d'intégration de livraison multimédia (DMIF)*

Contents

1 Scope	1
2 Conformance.....	1
2.1 Requirements for conformance	1
2.2 DMIF-Application Interface	1
2.3 DMIF signalling messages.....	2
3 Normative references	2
4 Definitions	2
5 Symbols and abbreviations	3
6 Organization of this part of ISO/IEC 14496.....	3
7 Compliance terms.....	4
8 DMIF overview.....	5
8.1 ISO/IEC 14496 terminal architecture	5
8.2 The objectives of DMIF.....	5
8.3 The DMIF communications architecture	6
8.4 DMIF computational model.....	8
8.5 DMIF QoS model	9
8.6 The User Plane in the Delivery Layer.....	9
9 Remote interactive scenario.....	10
9.1 Additional requirements	10
9.1.1 Introduction.....	10
9.1.2 Billing policies.....	10
9.1.3 Possible evolution to Heterogeneous Networks	11
9.2 DMIF within the context of the ISO/OSI layers.....	11
10 The DMIF-Application Interface.....	12

© ISO/IEC 1999

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland
Printed in Switzerland

10.1 The DMIF-Application Interface key concepts.....	12
10.2 Common semantic elements.....	13
10.2.1 QoS Descriptor	13
10.3 DMIF-Application Interface primitives	13
10.4 DMIF-Application Interface semantics	15
10.4.1 DA_ServiceAttach ()	15
10.4.2 DA_ServiceAttachCallback ()	15
10.4.3 DA_ServiceDetach ()	16
10.4.4 DA_ServiceDetachCallback ().....	16
10.4.5 DA_ChannelAdd ()	16
10.4.6 DA_ChannelAddCallback ()	16
10.4.7 DA_ChannelDelete ()	16
10.4.8 DA_ChannelDeleteCallback ()	17
10.4.9 DA_UserCommand ().....	17
10.4.10 DA_UserCommandCallback ().....	17
10.4.11 DA_Data ()	17
10.4.12 DA_DataCallback ().....	17
11 The DMIF-Network Interface	17
11.1 The DMIF-Network Interface key concepts	17
11.2 Common syntax elements.....	17
11.2.1 DMIF Descriptors	17
11.2.2 DMIF to DMIF data	20
11.2.3 Resource Descriptors	20
11.2.4 QoS Descriptors	22
11.2.5 Direction parameter.....	23
11.2.6 Reason parameter	24
11.2.7 Response parameter	24
11.3 DMIF-Network Interface primitives.....	24
11.4 DMIF-Network Interface semantics	26
11.4.1 DN_SessionSetup ()	26

11.4.2 DN_SessionRelease ()	26
11.4.3 DN_ServiceAttach ()	26
11.4.4 DN_ServiceDetach ().....	27
11.4.5 DN_TransMuxSetup ()	27
11.4.6 DN_TransMuxRelease ().....	27
11.4.7 DN_ChannelAdd()	28
11.4.8 DN_ChannelAdded().....	28
11.4.9 DN_ChannelDelete ()	29
11.4.10 DN_TransMuxConfig ().....	29
11.4.11 DN_UserCommand ().....	29
12 Control Plane mappings	30
12.1 Default syntax	30
12.1.1 Syntax elements.....	30
12.1.2 DNI mapping to DMIF signalling messages	35
12.2 Syntax for IP networks with (or without) RSVP signalling, using TCP for DMIF signalling	42
12.2.1 Approach overview.....	42
12.2.2 DSM-CC Resource Descriptors used.....	42
12.2.3 Usage of networkSessionIds	43
12.2.4 Usage of transactionIds	43
12.2.5 DNI mapping to socket actions and RSVP signalling	43
12.3 Syntax for IP networks with (or without) RSVP Signalling, using UDP for DMIF signalling	45
12.3.1 Approach overview.....	46
12.3.2 DSM-CC Resource Descriptors used.....	46
12.3.3 Usage of networkSessionIds	46
12.3.4 Usage of transactionIds	46
12.3.5 DNI mapping to socket actions and RSVP signalling	46
12.4 Syntax for ATM networks with Q.2931 signalling.....	47
12.4.1 Approach overview.....	47
12.4.2 DSM-CC Resource Descriptors used.....	47
12.4.3 Usage of networkSessionIds	47

12.4.4 Usage of transactionIds	47
12.4.5 DNI mapping to Q.2931 signalling messages	47
13 Terminal Capability Matching.....	51
13.1 DMIF Default signalling with Compatibility Descriptors.....	51
Annex A (informative) Overview of DAI and DNI parameters	52
Annex B (informative) Information flows for DMIF	54
Annex C (informative) Use of URLs in DMIF.....	75
Annex D (informative) Protocol error recovery	78
Annex E (informative) Subset of DSM-CC resource descriptors from DSM-CC ISO/IEC 13818-6.....	82
Annex F (informative) ISO/IEC 14496 content carried over an ETS 300 401 system.....	88
Annex G (informative) Patent statement.....	100

IECNORM.COM : Click to view the full PDF of ISO/IEC 14496-6:1999

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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

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 14496-6 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 14496 consists of the following parts, under the general title *Information technology — Coding of audio-visual objects*:

- *Part 1: Systems*
- *Part 2: Visual*
- *Part 3: Audio*
- *Part 4: Conformance testing*
- *Part 5: Reference software*
- *Part 6: Delivery Multimedia Integration Framework (DMIF)*

Annexes A to G of this part of ISO/IEC 14496 are for information only.

Introduction

The digital technology has transformed the generation, storage and communication of information. With this, the information age was borne allowing humankind to generate multimedia contents, communicate them and store them in meaningful ways. Digitization has also meant a rich variety of delivery and storage technologies, each bringing its own unique characteristics, over which multimedia material is communicated, broadcast or stored. The MPEG Delivery Multimedia Integration Framework (DMIF) allows each delivery technology to be used for its unique characteristics in a way transparent to application developers. DMIF specifies the semantics for the DMIF-Application Interface (DAI) in a way that it satisfies the requirements for both broadcast, local storage and remote interactive scenarios while maintaining uniformity across all cases. By including the ability to bundle connections into sessions DMIF enables network operators to apply appropriate billing policies in the provision of multimedia services. By adopting QoS metrics which relate to the media and not to the transport mechanism, DMIF hides the delivery technology details to applications. These unique features of DMIF give multimedia application developers what they need in terms of permanence and richness beyond what is possible with each individual delivery technology. With DMIF, application developers can begin to invest in commercial multimedia applications with the assurance that their investment will not be made obsolete by new delivery technologies. In order to fully reach its goal, DMIF needs "real" instantiation of its DMIF-Application Interface, and well defined, specific mappings of DMIF concepts and parameters into existing signalling infrastructures.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14496-6:1999

IECNORM.COM : Click to view the full PDF of ISO/IEC 14496-6:1999

Information technology — Coding of audio-visual objects —

Part 6:

Delivery Multimedia Integration Framework (DMIF)

1 Scope

This part of ISO/IEC 14496 specifies the Delivery Layer of ISO/IEC 14496, which allows applications to transparently access and view multimedia streams whether the source of the streams is located on an interactive remote end-system, the streams are available on broadcast media or they are on storage media.

In this part of ISO/IEC 14496, the following aspects are covered:

- DMIF communication architecture
- DMIF-Application Interface (DAI) definition
- URL semantic to locate and make available the multimedia streams
- DMIF Default Signalling Protocol (DDSP) for remote interactive scenarios, and its related variations using existing native network signaling protocols
- Information flows for player access to streams on remote interactive end-systems, from broadcast media or from storage media

2 Conformance

2.1 Requirements for conformance

DMIF conformance is only requested at the boundary of the equipment implementing a DMIF Instance. This includes the syntax and semantics of the particular Control protocol selected for a particular network (e.g., DMIF signalling messages), as well as the compliance to the mappings defined for broadcast and local storage systems.

DMIF conformance specifications is provided in ISO/IEC 14496-4.

2.2 DMIF-Application Interface

The DAI is a reference point. The exact syntax of the DAI is not defined by this part of ISO/IEC 14496, and does not represent a conformance point.

The DAI does not impose any programming language, nor syntax (e.g., the exact format for specifying a particular parameter — within the bounds of its semantic definition — or the definition of reserved values). Moreover the DAI provides only the minimal semantics for defining the behaviour of DMIF.

A real interface needs more than what is specified in the DAI (e.g., methods to initialise, reset, reconfigure, destroy, query the status or register services). Most of these aspects, as well as the detailed syntax, deeply depend on the implementation and language bindings. Moreover such details have no impact on the DMIF model and on the conformance issues, and therefore they are out of the scope of DMIF.

2.3 DMIF signalling messages

This part of ISO/IEC 14496 specifies the syntax and semantics of a few instances of protocols for DMIF peer to DMIF peer interactivity, designed for specific network environments. A DMIF Instance making use of one of the specified protocols shall fully comply to it. A DMIF Instance implementing a protocol not specified in this part of ISO/IEC 14496 cannot be tested for conformance, but may still be evaluated at the DAI Reference Point.

3 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 14496. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 14496 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 13818-6:1998, *Information technology — Generic coding of moving pictures and associated audio information — Part 6: Extensions for DSM-CC*.

ISO/IEC 14496-1:1999, *Information technology — Coding of audio-visual objects — Part 1: Systems*.

ETS 300 401:1997, *Radio broadcasting systems: Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers*.

IEEE 802:1990, *IEEE Organization Unique Identifier*.

ITU-T Recommendation Q.2931:1995, *B-ISDN User Network Interface Layer 3 Specification for Basic Call/Bearer Control*.

ITU-T Recommendation E.164:1991, *Telephone Network and ISDN Operation, Numbering, Routing and Mobile Services: Numbering plan for the ISDN Era*.

RFC1738:1994, *Uniform Resource Locators (URL)*.

4 Definitions

For the purposes of this part of ISO/IEC 14496, the following definitions apply.

4.1 Association Tag: In the context of this specification an Association Tag is used to identify elements within a Network Session with unique end-to-end significant values.

4.2 Channel: Is the entity over which a DMIF User sends or receives data.

4.3 Delivery layer: The layer of an ISO/IEC 14496 terminal that abstracts the delivery mechanism used to store or transmit streaming data and associated control information.

4.4 DMIF-Application Interface: Is the interface between an application (DMIF User) and the Delivery layer.

4.5 DMIF Default Signalling Protocol (DDSP): A session level protocol for the management of multimedia streaming over generic delivery technologies defined in this part of ISO/IEC 14496. The protocol comprises primitives to setup and tear down sessions as well as individual data channels.

4.6 DMIF Instance: Is an implementation of the Delivery layer for a specific delivery technology.

4.7 DMIF-Network Interface: Is a semantic API that abstracts the signalling between DMIF peers irrespectively of the delivery support.

4.8 DMIF peer: a shorthand for "peer DMIF Instance".

4.9 DMIF User: Is the application that exploits the functions offered by the Delivery Layer through the DAI.

4.10 Heterogeneous Network: A Network composed of different transport technologies which are connected in tandem through InterWorking Units.

4.11 Homogeneous Network: A Network composed of one transport technology only.

4.12 Network Session: An association between two DMIF peers providing the capability to group together the resources needed for an instance of a service. The Network Session is identified by a network-wide unique ID. A Network Session could group one or more Service Sessions.

4.13 Service: Is an entity identified by a Service Name (opaque to DMIF) which responds to DAI primitives.

4.14 Service Session: A local association between the local DMIF Instance and a particular service.

4.15 Transmux Channel: A native network transport channel: it includes the transport protocol stack that is directly supported by the signalling means of the network. It provides the basic transport stack element that DMIF may complement with additional multiplexing or protection tools.

5 Symbols and abbreviations

CAT: Channel Association Tag

DAI: DMIF-Application Interface

DDSP: DMIF Default Signalling Protocol

DS: DMIF Signaling

DNI: DMIF-Network Interface

QoS: Quality of Service

TAT: Transmux Association Tag

URL: Universal Resource Locator

6 Organization of this part of ISO/IEC 14496

This part of ISO/IEC 14496 contains the following technical clauses:

Clause	Definition
8	DMIF Overview - This clause explains how DMIF reaches its two-fold goal of hiding the delivery technology details from the DMIF User, and ensuring interoperability between end-systems. It describes the DMIF communications architecture and explains how this architecture can support current and future delivery technologies, preserving application from continuous updates to follow evolving technologies.
9	The remote interactive scenario - In this clause the additional requirements due to a remote interactive scenario are considered.
10	The DMIF-Application Interface - This clause defines a (semantic) API, called the DAI, that allows for the development of applications involving media content delivery, irrespective of delivery technology.

The DAI provides primitives for contacting remote services, for establishing data communication channels with these services, and also exchanging data over these channels. It should be noted that the DAI provides a minimum semantics, which means that a DMIF User would have to turn to a specific implementation of the DAI for a syntax; moreover this implementation would probably provide additional functionality. The only elements whose syntax is specified at the DAI are the URL, for locating services.

- 11 The DMIF-Network Interface (informative) - The DMIF-Network Interface (DNI) is a semantic API that abstracts the signalling between DMIF peers irrespectively of the delivery support. The DNI is informative, and is introduced in support to the description of the DMIF Signalling Protocol.
- 12 Control Plane mappings - This clause describes the mappings of the DNI primitives into DMIF Signalling Protocol messages. A Default DMIF Signalling Protocol is specified as well as a set of mappings into native network signalling protocols (extended through additional DMIF signalling messages, usually carried opaque to native signalling in a dedicated channel). These specifications ensure that different implementations of a DMIF Instance interoperate.
- 13 Terminal Capability Matching – Specifies how DMIF determines compatibility between DMIF terminals.
- Annex A Overview of DAI and DNI parameters (informative) - This Annex provides the glue to link concepts and parameters as defined in DAI and DNI.
- Annex B Information flows for DMIF (informative) - This Annex defines the scenarios for remote interactive end-system, broadcast and local storage DMIF implementations.
- Annex C Use of URLs in DMIF (informative) - This Annex defines the URL schemes currently supported.
- Annex D Protocol error recovery (informative) – This Annex specifies the state machine for the transactions in the DMIF Signalling Protocol.
- Annex E Subset of DSM-CC resource descriptors from DSM-CC ISO/IEC 13818-6 (informative) – This Annex reports selected dsm-cc resource descriptors from ISO/IEC 13818-6 that are used by this part of ISO/IEC 14496.
- Annex F ISO/IEC 14496 content carried over an ETS 300 401 system (informative) – This Annex defines how ISO/IEC 14496 content can be carried over an ETS 300 401 System (DAB), and describes how to access it through the DMIF Application Interface.
- Annex G Patent statement (informative).

7 Compliance terms

Table 1 defines the meaning of terms used in this part of ISO/IEC 14496.

Table 1 — Definition of terms used in this part of ISO/IEC 14496

Term	Definition
shall	Means that an item is required. For example, the phrase “The widget shall be blue.” means that no color other than blue is acceptable for the widget.
shall not	Means that an item is not allowed. For example, the phrase “The widget shall not be blue.” means that the color blue is not acceptable for the widget.
must	Not used. Use “shall” instead.
must not	Not used. Use “shall not” instead.
should	Means that, if possible, an item shall be used. For example, the phrase “The widget should be blue.” Means that blue is the preferred color for the widget but, if blue is not possible, other colors are acceptable.

should not	Means that, if possible, an item shall not be used. For example, the phrase “The widget should not be blue.” Means that it is preferred that the color of the widget not be blue, however it is allowable to use that color if necessary.
may	Means that an item is optional. For example, the phrase “The widget may be blue.” Means that it is allowable to use blue as the widget color.
may not	Not used. Use “shall not” instead.

8 DMIF overview

8.1 ISO/IEC 14496 terminal architecture

The generic ISO/IEC 14496 terminal architecture is depicted in Figure 1. It comprises three layers: the Compression Layer, the Sync Layer and the Delivery Layer. The Compression Layer is media aware and delivery unaware; the Sync Layer is media unaware and delivery unaware; the Delivery Layer is media unaware and delivery aware.

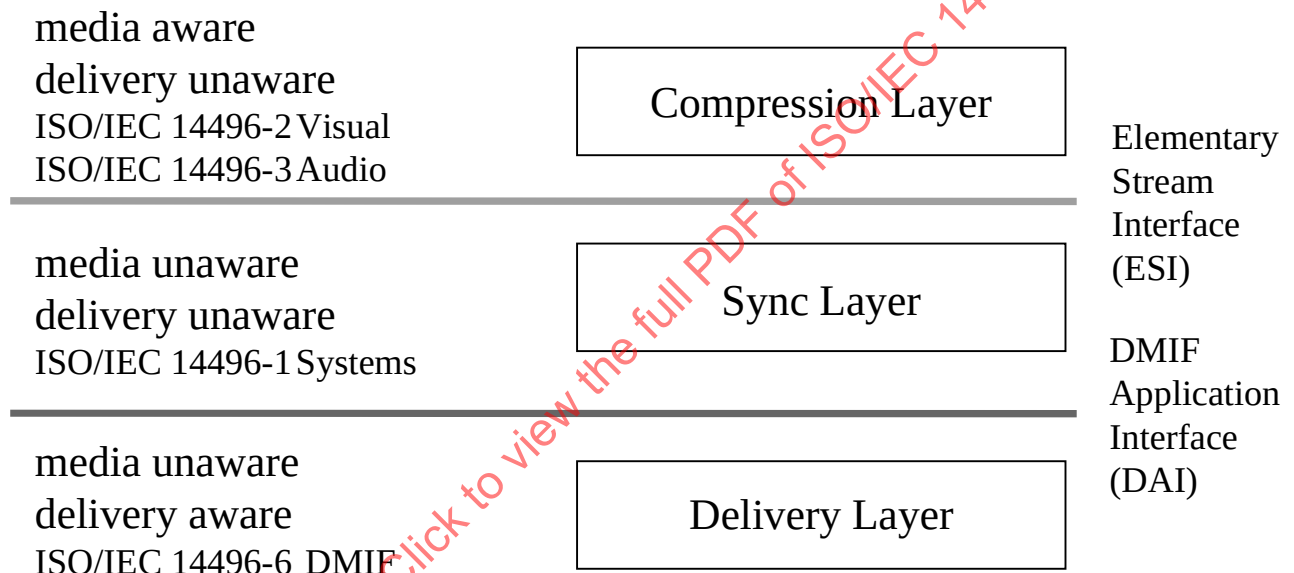


Figure 1 — ISO/IEC 14496 terminal architecture

The Compression Layer performs media encoding and decoding of Elementary Streams and is specified in ISO/IEC 14496-2 and ISO/IEC 14496-3; the Sync Layer manages Elementary Streams and their synchronisation and hierarchical relations and is specified in ISO/IEC 14496-1; the Delivery Layer ensures transparent access to content irrespective of delivery technology and is specified in this part of ISO/IEC 14496.

The boundary between the Compression Layer and the Sync Layer is named Elementary Stream Interface (ESI) and its minimum semantic is specified in ISO/IEC 14496-1.

The boundary between the Sync Layer and the Delivery Layer is named DMIF-Application Interface (DAI) and its minimum semantic is specified in this part of ISO/IEC 14496.

8.2 The objectives of DMIF

The Delivery Multimedia Integration Framework has a number of objectives:

- to hide the delivery technology details from the DMIF User

- to manage real time, QoS sensitive channels
- to allow service providers to log resources per session for usage accounting
- to ensure interoperability between end-systems

DMIF defines a communication architecture that hides the details of the delivery technologies below an interface that is exposed to the application, called the DMIF-Application Interface (DAI). Delivery technologies include transport network technologies (e.g., the Internet, or an ATM infrastructure), as well as broadcast technologies and local storage technologies.

The DAI separates the delivery aware and delivery unaware layers of the ISO/IEC 14496 terminal architecture. In the case of transport network technologies the DAI fulfils the specific needs of ISO/IEC 14496 based applications, however, its design is generic enough to allow the exploitation of DMIF in contexts other than ISO/IEC 14496 based applications.

The DAI is a semantic API that allows the development of applications irrespectively of the delivery support. No syntax is specified for this Interface, since this would be programming language and operating system dependent; moreover only the minimum functionality is specified.

DMIF takes into account the QoS management aspects and the requirement of allowing service providers to log resources per session for usage accounting, in order to facilitate the implementation of appropriate billing policies. In the case of interactive networks it derives a generic protocol to fulfil the requirements (the DMIF Default Signalling Protocol – DDSP).

By adapting the generic protocol to the native network signalling mechanisms, DMIF fully specifies the behaviour of the DMIF Instances operating on particular network environments, thus ensuring interoperability for those cases. Other network environments and protocol implementations are not prevented, as far as the DAI functionality is preserved, but are out of the scope of this specification.

8.3 The DMIF communications architecture

The integration framework of DMIF covers three major technologies, interactive network technology, broadcast technology and the storage technology; this is shown in Figure 2.

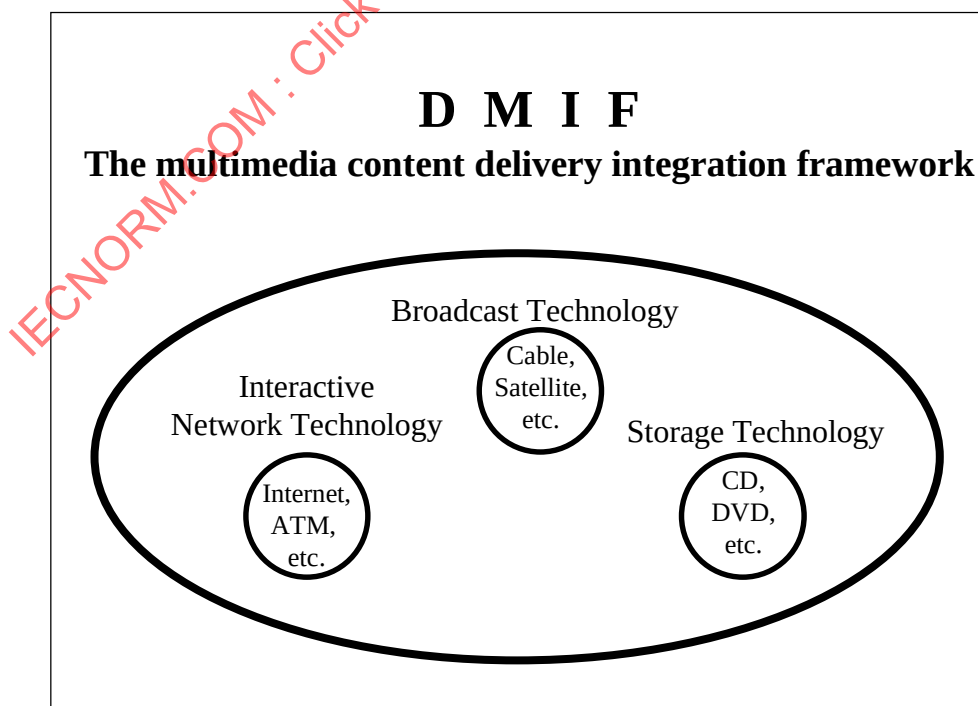


Figure 2 — DMIF addresses the delivery integration of three major technologies

Figure 3 represents the DMIF communication architecture. The shaded boxes clarify what are the boundaries of a DMIF implementation: this part of ISO/IEC 14496 normatively specifies the behaviour of a DMIF implementation at those boundaries, while the additional modelling and definitions which apply to elements internal to the shaded boxes have only informative value.

The DMIF architecture is such that applications that rely on DMIF for communications do not have to be concerned with the underlying communications method. The implementation of DMIF takes care of the delivery technology details presenting a common interface to the application.

Figure 3 depicts the above concept. An application accesses data through the DMIF-Application Interface, irrespective whether such data comes from a broadcast source, from local storage or from a remote server. A Delivery layer implementation allows the concurrent presence of more than one DMIF Instance. When requesting a specific service the Application supplies a URL that allows the Delivery layer to determine the appropriate DMIF Instance to activate. The DMIF Instances will then translate the Originating Application requests into specific actions to be taken with respect to the broadcast network or the local file system, or translate it into messages to be delivered to the Target Application, taking care of the peculiarities of the involved delivery technology. Similarly, data entering the terminal (from remote servers, broadcast networks or local files) is uniformly delivered to the Originating Application through the DAI. The DAI allows the DMIF User to group elementary streams into services and to specify the QoS requirements for the desired elementary streams.

DMIF separates the common features that should be implemented in each DMIF Instance from the details of the delivery technology. In the case of interactive networks DMIF specifies a logical interface (the DMIF-Network Interface – DNI) between a hypothetical module implementing this common features and the network specific modules. The DNI specifies the information flow that should occur between peers of such hypothetical module; the network specific modules specify the exact mapping of the DNI primitives into signalling messages.

DNI primitives need to be adapted to the native network signalling mechanisms in order to provide access to the network resources. DNI primitives represent one of the possible solutions to ensure the DAI functionality when interacting with a remote peer. A straightforward mapping of DNI primitives into signalling messages is defined, and named DMIF Default Signalling Protocol (DDSP). The DMIF Default Signalling Protocol is a session level protocol for the management of multimedia streaming over generic delivery technologies. The protocol comprises primitives to set-up and tear down sessions as well as individual data channels.

DNI primitives can be mapped into different protocols (e.g., in a LAN, or in the domain of a particular telecom operator) to provide the DAI functionality in the presence of different requirements imposed by a particular network environment (e.g., for admission control, billing policies ...).

In the case of the Broadcast and Local Storage scenarios the model is simpler, and no internal interface has been identified. Conceptually, such a DMIF Instance includes the features of a Target DMIF peer, as well as those of the Target Application. This implies that such a DMIF Instance is not agnostic of the application making use of it. Specifically, broadcast or local storage DMIF Instances have to be ISO/IEC 14496 aware if they are used in an ISO/IEC 14496 terminal. All the control messages that are exchanged between peer applications in an interactive scenario are in this case terminated in the DMIF Instance. Stream control commands like PLAY/PAUSE/MUTE are an example.

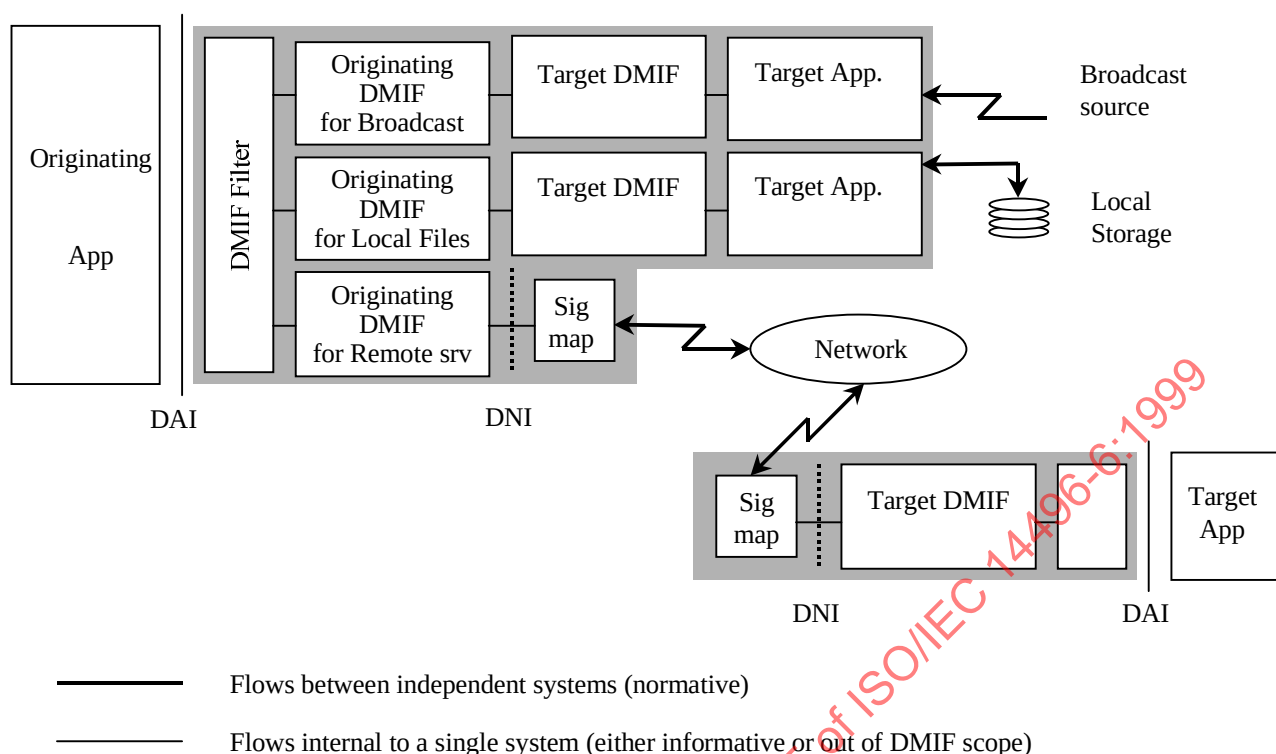


Figure 3 — DMIF communication architecture

DMIF allows the concurrent presence of one or more DMIF Instances, each one targeted for a particular delivery technology, in order to support in the same terminal multiple delivery technologies and even multiple scenarios (broadcast, local storage, remote interactive). Multiple delivery technologies may be activated by the same application, which could therefore seamlessly manage data sent by broadcast networks, local file systems and remote interactive peers.

8.4 DMIF computational model

When an application requests the activation of a service, it uses the Service primitives of the DAI (see clause 10), and creates a Service Session. In the case of a local storage or broadcast scenario, the DMIF Instance locates the content which is part of the indicated service; in case of interactive scenarios the DMIF Instance contacts its corresponding peer and creates a Network Session with it. The peer DMIF Instance in turn identifies the peer Application that runs the service, and establishes a Service Session with it. Network Sessions have network-wide significance, Service Sessions have instead local meaning. The Delivery layer maintains the association between them. Each DMIF Instance uses the native signalling mechanism for the respective network to create and then manage the Network Session (e.g., DMIF Default Signalling Protocol integrated with ATM signalling). The application peers then use this session to create connections that are used to transport application data (e.g., ISO/IEC 14496 Elementary Streams).

When an application needs a Channel, it uses the Channel primitives of the DAI (see clause 10), indicating the Service they belong to. In the case of a local storage or broadcast scenario, the DMIF Instance locates the requested content, which is scoped by the indicated service, and prepares itself to read it and pass it in a channel to the application; in case of interactive scenarios the DMIF Instance contacts its corresponding peer to get access to the content, reserves the network resources (e.g., connections) to stream the content, and prepares itself to read it and pass it in a channel to the application; in addition, the remote application locates the requested content, which is scoped by the indicated service. DMIF uses the native signalling mechanism for the respective network to reserve the network resources. The remote application then uses these resources to deliver the content.

Figure 4 provides a high level view of a service activation and of the beginning of data exchange in the case of interactive scenarios; the high level walk-through consists of the following steps:

The Originating Application requests the activation of a service to its local DMIF Instance: a communication path between the Originating Application and its local DMIF peer is established in the control plane (1)

The Originating DMIF peer establishes a Network Session with the Target DMIF peer: a communication path between the Originating DMIF peer and the Target DMIF Peer is established in the control plane (2)

The Target DMIF peer identifies the Target Application and forwards the service activation request: a communication path between the Target DMIF peer and the Target Application is established in the control plane (3)

The peer Applications create channels (requests flowing through communication paths 1, 2 and 3). The resulting channels in the user plane (4) will carry the actual data exchanged by the Applications.

DMIF is involved in all four steps above.

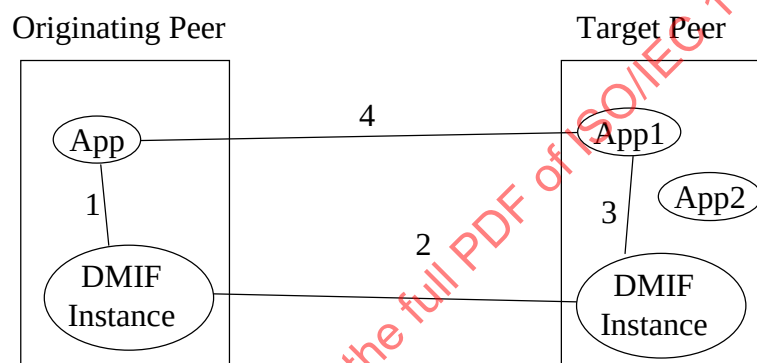


Figure 4 — DMIF computational model

8.5 DMIF QoS model

This part of ISO/IEC 14496 while it specifies the QoS traffic parameters for a given stream at the DAI (e.g., bitrate, maximum access unit size, ...), it does not specify its QoS performance requirements (e.g., delay, loss probability, ...). As a result, the QoS performance should be based only on the transport network and the administrative policy imposed by a given implementation, e.g., choose best-effort for all channels, or guaranteed service for all channels.

8.6 The User Plane in the Delivery Layer

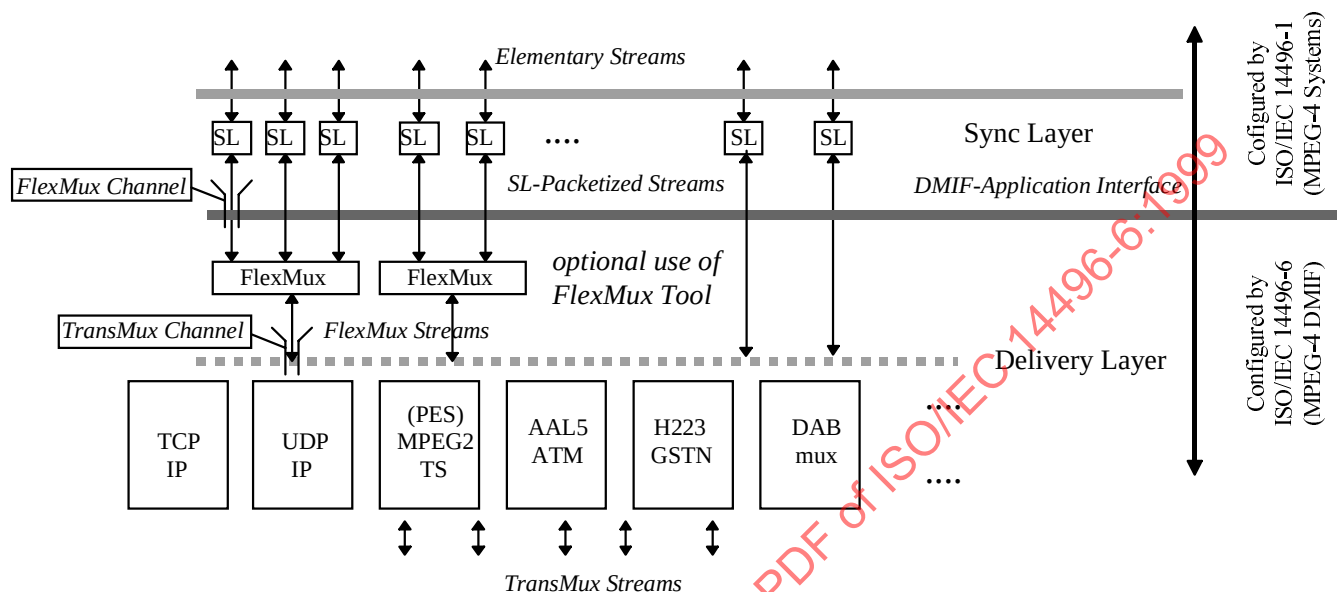
Figure 5 illustrates the User Plane in an ISO/IEC 14496 terminal, and specifically in the Delivery Layer: Elementary Streams cross the DAI in individual channels, and are multiplexed/demultiplexed in the Delivery Layer. The Delivery Layer makes use of the available multiplexing facilities of the native protocol stacks (e.g., IP port numbers, ATM VCs, MPEG-2 PIDs, file names in a file system), which are referred as Transmuxes in this specification. Where necessary it makes use of additional multiplexing or protection tools, and uses DMIF descriptors to appropriately configure the stacks at both ends.

Currently the FlexMux multiplexing tool specified in ISO/IEC 14496-1 is supported.

The Delivery layer is responsible for the configuration of the transport protocol stacks. Each DMIF Instance is in charge of configuring the exact protocol stack for each channel, and of keeping track of the associations of channels and transport resources. The configuration of the Transmux portion of the protocol stack is achieved through the usage of DMIF signalling with Resource Descriptors and/or their execution using native signalling; the

configuration of the remaining portion of the protocol stack is achieved through the usage of DMIF signalling with DMIF Descriptors.

Figure 5 provides a sample of the choice of native transport protocol stacks. It outlines the fact that either an Elementary Stream or a group of streams multiplexed together (e.g., with the FlexMux tool) can be carried over a native transport.



NOTE The figure shows a number of relevant protocol stacks, that are, however, not all fully specified in this part of ISO/IEC 14496.

Figure 5 — The User Plane in an ISO/IEC 14496 terminal

9 Remote interactive scenario

9.1 Additional requirements

9.1.1 Introduction

This subclause only focuses on the Remote Interactive scenario, and defines the common features that each DMIF Instance for such a scenario should implement. These common features take care of the requirement of facilitating the possible evolution to Heterogeneous Networks and of allowing service providers to log resources per session for usage accounting, in order to facilitate the implementation of appropriate billing policies. Valid DMIF Instances may however ignore these requirements.

9.1.2 Billing policies

In order to facilitate the implementation of appropriate billing policies, DMIF allows to group in Network Sessions the resources consumed for the delivery of a certain service. This way network service providers may log and then charge the resources consumed in the course of a Network Session as a whole, instead of just summing the cost for each individual resource. Network Sessions may include multiple Service Sessions, as far as they relate to the same DMIF peers. These Service Sessions may relate to different application instances on one or both sides. This case is depicted in Figure 6. Valid DMIF Instances may however prefer to maintain a one-to-one relation between the services as requested by the DMIF User and the Network Sessions. In this case if extra capacity is available on one connection belonging to one Service Session it cannot be used by the other Service Session and as a result the end user will pay more.

Billing policies are out of the scope of this specification.

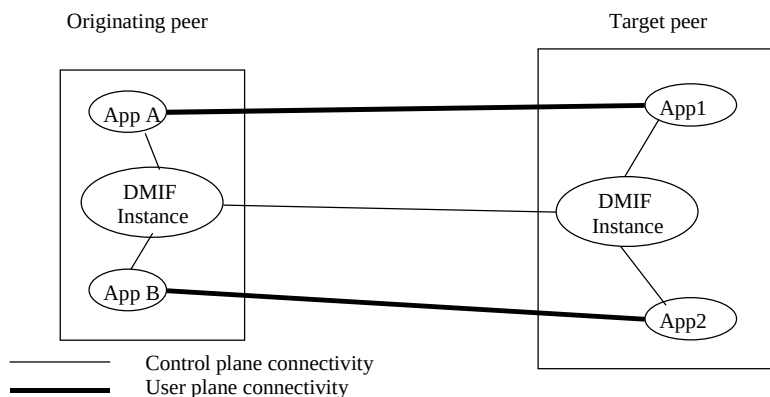


Figure 6 — A Network Session shared by multiple Service Sessions

9.1.3 Possible evolution to Heterogeneous Networks

This specification only targets homogeneous networks; however, possible evolution to Heterogeneous Networks has been considered. In order to facilitate this evolution, parts of a solution framework that was developed in ISO/IEC 13818-6 (Digital Storage Media Command & Control, User to Network – DSMCC-UN) have been adopted. In particular, the DMIF Default Signalling Protocol defined in this specification makes use of a generic descriptor architecture derived from ISO/IEC 13818-6 for describing and tagging the network resources being used.

9.2 DMIF within the context of the ISO/OSI layers

Figure 7 positions the DMIF-Application Interface in the DMIF architecture shown in Figure 3 and its networked component shown in Figure 4, and highlights the role of Delivery Layer with respect to the ISO/OSI layers: a DMIF Instance performs Session Layer functions, and the DMIF-Application Interface corresponds to a Session Service Access Point. The DMIF control primitives at the DMIF-Application Interface capture parameters that are maintained for the duration of the contract. Any other parameter that varies during the life of the contract is carried on the User Plane and is out of the scope of this specification. In the case of ISO/IEC 14496 refer to ISO/IEC 14496-1. One such example of the first kind is the QoS of the media stream that is fixed during the contract and therefore carried in the DMIF control primitives. In the case of ISO/IEC 14496 an example of the second kind is the SL packet header length which could vary during the life of the media stream and is therefore carried on the User Plane as specified in ISO/IEC 14496-1. Also while the DMIF control primitives are specified both at the receive and transmit ends, the User Plane is only specified at the receive end.

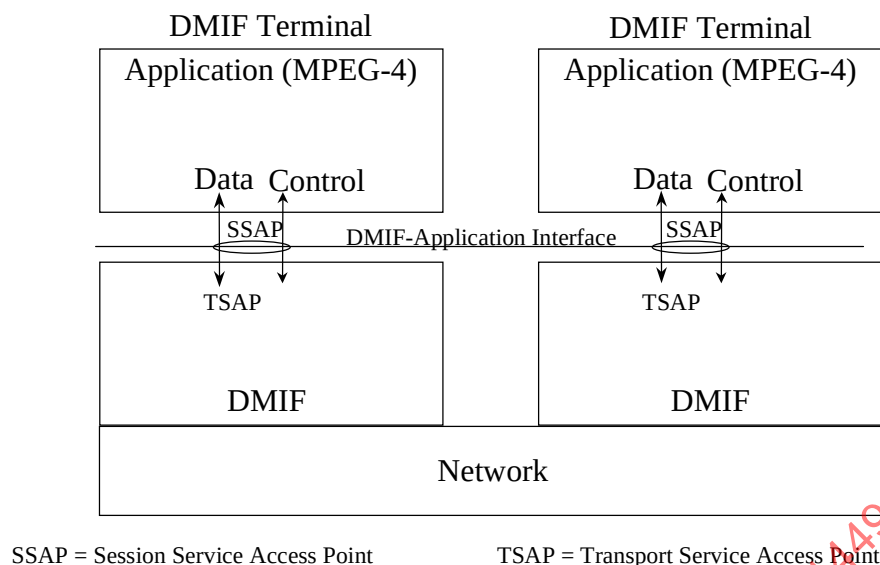


Figure 7 — Position of the DMIF-Application Interface in the DMIF architecture

10 The DMIF-Application Interface

10.1 The DMIF-Application Interface key concepts

The DMIF-Application Interface (DAI) is a semantic API that allows the development of applications transparent to the supported delivery technologies. By using the DAI, an application could seamlessly access content from broadcast networks, from local storage devices and from remote end-systems.

The DAI semantics is normative.

DAI does not impose any programming language, nor syntax (e.g., the exact format for specifying a particular parameter — within the bounds of its semantic definition — or the definition of reserved values), except for the URL defined in Annex C. Moreover the DAI provides only the minimal semantics for defining the behaviour of DMIF.

A real implementation of the DAI needs more than what is specified here (e.g., methods to initialise, reset, reconfigure, destroy, query the status, register services ...). Most of these aspects, as well as the detailed syntax, deeply depend on the language binding and on the implementation (part of an operating system, or a separate library, or a set of separate libraries, or a combination of the above, or ...; and for the syntax: synchronous or asynchronous implementation, callback functions or events or a polling technique, ...). Moreover such details have no impact on the DMIF model, and are therefore out of the scope of DMIF.

It is worth mentioning that this interface is available to any application, not just ISO/IEC 14496-1 (MPEG-4 System) applications, i.e., the DMIF-Application Interface is designed for generic use.

The DMIF-Application Interface defines the functions offered by the Delivery layer. The entity that uses this interface is named the DMIF User.

Through the DMIF-Application Interface DMIF Users are hidden from the delivery technology details (for both Data and Control Planes), and just manipulate Service Sessions and channels.

The DMIF-Application Interface is comprised of the following classes of primitives:

- Service primitives, which deal with the Control Plane, and allow the management of Service Sessions (attach and detach);
- Channel primitives, which deal with the Control Plane, and allow the management of channels (add and delete);
- Data primitives, which deal with the User Plane, and serve the purpose of transferring data through channels.

Note that since only the minimum DAI semantics is specified, additional primitives and parameters may be required in actual implementations, e.g., to initialize the DMIF Instance, to register callbacks, to resolve service names, to apply sophisticated statistical multiplexing at the FlexMux Layer on the sender side (including the usage of unequal protection schemes), etc.

10.2 Common semantic elements

10.2.1 QoS Descriptor

Two QoS Descriptors are defined in this part of ISO/IEC 14496: one is used at the DMIF-Application Interface, another one is used at the DMIF-Network Interface. Both are intended to specify the “transport” Quality of Service for an individual Elementary Stream, however only the traffic parameters (e.g., bitrate) have been specified, not the performance parameters (e.g., delay, loss probability, ...).

The exact syntax of the QoS_Descriptor used at the DAI is not specified in this part of ISO/IEC 14496, only the semantic is given.

The exact syntax of the QoS_Descriptor used at the DNI and in the messages through the network is instead fully specified in subclause QoS Descriptors.

DMIF may aggregate multiple Elementary Streams and carry them into a single TransMux, and compute the aggregate “transport” QoS. DMIF will then further map the “transport” QoS requirements for a particular TransMux into specific “network” QoS.

The QoS_Descriptor shall be able to carry a number of QoS metrics. The set of metrics currently defined and their semantic is summarized in Table 2.

Table 2 — ISO/IEC 14496-6 defined QoS metrics

QoS Metric	Semantic Description
PRIORITY	Priority for the stream
MAX_AU_SIZE	Maximum size of an AU
AVG_BITRATE	Average bit rate
MAX_BITRATE	Maximum bit rate

10.3 DMIF-Application Interface primitives

This subclause lists the primitives specified for the DMIF-Application Interface, and describes the parameters used. The only parameter with fixed syntax is the URL defined in Annex C. The C++ like formalism used to describe the primitives only aims at capturing their semantic meaning. IN and OUT keywords allow to clearly distinguish the parameters provided to the other side of the interface from those returned from it; by no means they are meant to force a synchronous implementation. The loop() construct allows to concisely represent an array of elements.

The primitives are:

- DA_ServiceAttach (IN: URL, uuDataInBuffer, uuDataInLen; OUT: response, serviceSessionId, uuDataOutBuffer, uuDataOutLen)
- DA_ServiceAttachCallback (IN: serviceSessionId, serviceName, uuDataInBuffer, uuDataInLen; OUT: response, uuDataOutBuffer, uuDataOutLen)
- DA_ServiceDetach (IN: serviceSessionId, reason; OUT: response)
- DA_ServiceDetachCallback (IN: serviceSessionId, reason; OUT: response)
- DA_ChannelAdd (IN: serviceSessionId, loop(qosDescriptor, direction, uuDataInBuffer, uuDataInLen); OUT: loop(response, channelHandle, uuDataOutBuffer, uuDataOutLen))
- DA_ChannelAddCallback (IN: serviceSessionId, loop(channelHandle, qosDescriptor, direction, uuDataInBuffer, uuDataInLen); OUT: loop(response, uuDataOutBuffer, uuDataOutLen))
- DA_ChannelDelete (IN: loop(channelHandle, reason); OUT: loop(response))
- DA_ChannelDeleteCallback (IN: loop(channelHandle, reason); OUT: loop(response))
- DA_UserCommand (IN: loop(channelHandle, uuDataInBuffer, uuDataInLen))
- DA_UserCommandCallback (IN: loop(channelHandle, uuDataInBuffer, uuDataInLen))
- DA_Data (IN: channelHandle, streamDataBuffer, streamDataLen)
- DA_DataCallback (IN: channelHandle, streamDataBuffer, streamDataLen, errorFlag)

The parameters have the following semantics:

channelHandle: is a local identifier that uniquely identifies a channel in the application space, no matter how many services the application attaches to, or how many DMIF Instances it is using. In this interface specification the channelHandle parameter is set by the DMIF Instance, however it would be also acceptable if it were set by the DMIF User. The algorithm used by the DMIF Instance (User) to set the channelHandle is a matter that does not affect this interface.

direction: Indicates the direction of the channel, either UPSTREAM i.e., from the receiver to the sender or DOWNSTREAM i.e., from the sender to the receiver.

URL: is a string that identifies the location the service as well as the service name. The local DMIF peer strips the protocol portion (<scheme>) of the URL in order to identify and activate the appropriate DMIF Instance; the remaining part of the URL represents the service location. URLs shall be absolute. Refer to Annex C for more information on the usage of URLs in DMIF, including the list of allowed URL schemes.

errorFlag: is a flag that indicates whether an error has been detected (but not corrected) on the streamDataBuffer.

qosDescriptor: is a parameter set by the DMIF User containing the complete description of the Quality of Service requested for that particular stream. Its semantic definition is given in subclause 10.2.1.

serviceName: at the target DMIF peer it identifies the actual service.

serviceSessionId: is a local identifier that uniquely identifies a Service Session in the application space, no matter how many services the application attaches to, or how many DMIF Instances it is using. In this interface specification the serviceSessionId parameter is set by the DMIF Instance, however it would be also acceptable if it were set by the DMIF User. The algorithm used by the DMIF Instance (User) to set the serviceSessionId is a matter that does not affect this interface.

streamDataBuffer: is the actual Data Unit generated by the DMIF User.

streamDataLen: is the length of the streamDataBuffer field.

uuDataInBuffer: is an opaque structure providing upper layer information; it is transparently transported through DMIF from the local peer to the remote peer.

uuDataInLen: is the length of the uuDataInBuffer field.

uuDataOutBuffer: is an opaque structure providing upper layer information; it is transparently transported through DMIF from the remote Peer to the local Peer.

uuDataOutLen: is the length of the uuDataOutBuffer field.

response: a code identifying the response.

reason: a code identifying the reason.

10.4 DMIF-Application Interface semantics

10.4.1 DA_ServiceAttach ()

DA_ServiceAttach (IN: URL, uuDataInBuffer, uuDataInLen; OUT: response, serviceSessionId, uuDataOutBuffer, uuDataOutLen)

This primitive is issued by a DMIF User to request the initialization of a Service Session: the service is unambiguously identified by its **URL** which conveys information for identifying both the delivery technology being used (i.e. protocol), the address of the target DMIF peer (this may have different meanings in the different scenarios) and the name of the service inside the domain managed by the target DMIF peer -which is then referred to as **serviceName**.

The DMIF User might provide additional information such as client credentials in **uuDataIn**: this additional information is opaque to the Delivery layer and is only consumed by the target DMIF User (which is locally emulated in Broadcast and Local Storage scenarios).

The target DMIF User might in turn provide additional information in **uuDataOut**: this additional information is opaque to the Delivery layer and is only consumed by the local DMIF User. In an ISO/IEC 14496 application the **uuDataOut** shall return a single Object Descriptor or Initial Object Descriptor if required by the context of this call.

In case of a positive **response**, the **serviceSessionId** parameter contains the Service Session identifier that the DMIF User should refer to in subsequent interaction through the DAI regarding this Service Session.

10.4.2 DA_ServiceAttachCallback ()

DA_ServiceAttachCallback (IN: serviceSessionId, serviceName, uuDataInBuffer, uuDataInLen; OUT: response, uuDataOutBuffer, uuDataOutLen)

This primitive is issued by the target DMIF Instance to the appropriate target DMIF User as identified through the **serviceName** field. The steps involved in identifying the appropriate DMIF User and delivering the primitives to it are out of the scope of this specification.

The target DMIF Instance also provides the **serviceSessionId** parameter, that contains the Service Session identifier that the target DMIF User should refer to in subsequent interaction through the DAI regarding this Service Session.

The target DMIF User (the Application Executive running the service) might also receive additional information (e.g.; client credentials) through the **uuDataIn** field.

The target DMIF User might in turn provide additional information in **uuDataOut**: this additional information is opaque to the Delivery layer and is only consumed by the local DMIF User. In an ISO/IEC 14496 application the **uuDataOut** shall return a single Object Descriptor or Initial Object Descriptor if required by the context of this call.

In case of a negative **response**, the **serviceSessionId** becomes invalid at the target DMIF Instance.

Real implementations of this primitive are likely to support an additional parameter identifying the calling peer.

10.4.3 DA_ServiceDetach ()

DA_ServiceDetach (IN: serviceSessionId, reason; OUT: response)

This primitive is issued by a DMIF User to request the termination of the service identified by **serviceSessionId**; a **reason** should be specified. The DMIF Instance returns a **response**.

10.4.4 DA_ServiceDetachCallback ()

DA_ServiceDetachCallback (IN: serviceSessionId, reason; OUT: response)

This primitive is issued by the target DMIF Instance to inform the target DMIF User that the service identified by **serviceSessionId** has been terminated due to the reason **reason**. The target DMIF User returns a **response**.

10.4.5 DA_ChannelAdd ()

DA_ChannelAdd (IN: serviceSessionId, loop(qosDescriptor, direction, uuDataInBuffer, uuDataInLen); OUT: loop(response, channelHandle, uuDataOutBuffer, uuDataOutLen))

This primitive is issued by a DMIF User to request the addition of one or more end-to-end channels in the context of a particular Service Session identified by **serviceSessionId**.

Each channel is requested by providing an (optional) **qosDescriptor** and a **direction**

The local DMIF User might provide additional information for each requested channel in **uuDataIn**. This additional information is opaque to the Delivery layer and is only consumed by the target DMIF User. In the case of an ISO/IEC 14496 application the **uuDataIn** shall always be present.

For each requested channel, in case of a positive **response**, the **channelHandle** parameter contains the channel identifier that the DMIF User should refer to in subsequent interaction through the DAI involving this channel.

10.4.6 DA_ChannelAddCallback ()

DA_ChannelAddCallback (IN: serviceSessionId, loop(channelHandle, qosDescriptor, direction, uuDataInBuffer, uuDataInLen); OUT: loop(response, uuDataOutBuffer, uuDataOutLen))

This primitive is issued by the target DMIF Instance to the appropriate target DMIF User as identified through the **serviceSessionId** field, to inform the target DMIF User that the addition of channels is requested.

For each requested channel, the target DMIF Instance provides the **direction** to the target DMIF User. It also provides the **channelHandle** parameter, that contains the channel identifier that the target DMIF User should refer to in subsequent interaction through the DAI involving this Channel.

The target DMIF User (the Application Executive running the service) might also receive additional information through the **uuDataIn** field. In the case of an ISO/IEC 14496-1 based application the **uuDataIn** shall always be present and may contain the Elementary Stream Identifier -ES_ID-.

For each requested channel, the target DMIF User returns a **response**. In case of a negative **response**, the **channelHandle** becomes invalid at the target DMIF Instance.

10.4.7 DA_ChannelDelete ()

DA_ChannelDelete (IN: loop(channelHandle, reason); OUT: loop(response))

This primitive is issued by a DMIF User to delete one or more channels as identified by **channelHandle**; a **reason** should be specified. The channels need not be all part of a single Service Session. The DMIF Instance returns a **response**.

10.4.8 DA_ChannelDeleteCallback ()

DA_ChannelDeleteCallback (IN: loop(channelHandle, reason); OUT: loop(response))

This primitive is issued by the target DMIF Instance to inform the target DMIF User that the channels identified by **channelHandle** have been closed due to the reason **reason**. The target DMIF User returns a **response**.

10.4.9 DA_UserCommand ()

DA_UserCommand (IN: loop(channelHandle, uuDataInBuffer, uuDataInLen))

This primitive is issued by a DMIF User to send **uuData** that refers to channels identified by **channelHandle**. This primitive is intended to support the delivery of control information in the upstream direction.

10.4.10 DA_UserCommandCallback ()

DA_UserCommandCallback (IN: loop(channelHandle, uuDataInBuffer, uuDataInLen))

This primitive is issued by the target DMIF Instance to inform the target DMIF User that there is **uuData** relative to channels identified by **channelHandle**.

10.4.11 DA_Data ()

DA_Data (IN: channelHandle, streamDataBuffer, streamDataLen)

This primitive is issued by a DMIF User to send **streamData** in the channel identified by **channelHandle**.

10.4.12 DA_DataCallback ()

DA_DataCallback (IN: channelHandle, streamDataBuffer, streamDataLen, errorFlag)

This primitive is issued by the DMIF Instance to the appropriate DMIF User (identified through the **channelHandle**) and provides the **streamData** along with an **errorFlag** for that channel.

11 The DMIF-Network Interface

11.1 The DMIF-Network Interface key concepts

The DMIF-Network Interface (DNI) is a semantic API for the Control Plane. It abstracts the signalling between DMIF peers irrespectively of the supported delivery technologies. The parameters conveyed through the DNI are then mapped onto network dependent native signalling when possible. The parameters which cannot be mapped to native signalling are carried opaque to the native signalling in which case the necessary syntax is defined in this specification. The mappings to native signalling standards are described in clause 12.

The DMIF-Network Interface comprises the following classes of primitives:

- Session primitives, which allow the management of sessions (setup and release);
- Service primitives, which allow the management of services (attach and detach);
- Transmux primitives, which allow the management of a transmux (setup, release and config);
- Channel primitives, which allow the management of channels (add and delete).

11.2 Common syntax elements

11.2.1 DMIF Descriptors

Table 3 describes the generic format of a DMIF Descriptor.

Table 3 — General format of the DMIF Descriptor

Syntax	Num. Of Bytes
dmifDescriptor { commonDescriptorHeader() descriptorDataFields() }	

commonDescriptorHeader: Table 4 defines the format of the commonDescriptorHeader.

descriptorDataFields: Provides the actual descriptor data.

Table 4 — DMIF commonDescriptorHeader

Syntax	Num. of Bytes
commonDescriptorHeader() { DmifDescriptorType 2 DmifDescriptorLen 2 }	

dmifDescriptorType: Defines the specific descriptor being carried. Table 5 defines the dmifDescriptorTypes defined in this specification.

dmifDescriptorLen: Defines the total length in bytes of the descriptorDataFields() that follow.

Table 5 — DMIF Descriptor Types

dmifDescriptorType	Value	Description
Reserved	0x0000	ISO/IEC 14496-6 reserved.
UuDataDescriptor	0x0001	Contains uuData as conveyed at the DAI
BypassFlexMuxDescriptor	0x0002	Indicates no flex-multiplexing is used
MPEG4SystemsFlexMuxDescriptor	0x0003	Supports MPEG-4 FlexMux as defined in ISO/IEC 14496-1.
MPEG4SystemsMuxCodeDescriptor	0x0004	Supports MPEG-4 FlexMux MuxCodeTables as defined in ISO/IEC 14496-1.
Reserved	0x0005-0xffff	ISO/IEC 14496-6 reserved.

11.2.1.1 uuDataDescriptor

This descriptor shown in Table 6 is used to carry uuData as exposed at the DAI. uuData is an opaque structure providing upper layer information; it is transparently transported through DMIF from Peer to Peer.

Table 6 — uuDataDescriptor

Syntax	Num. Of Bytes
UuDataBuffer	DmifDescriptorLen

uuDataBuffer: Carries the uuDataBuffer as exposed at the DAI.

11.2.1.2 BypassFlexMuxDescriptor

This descriptor shown in Table 7 is used to indicate that no additional multiplex tool is used on top of the particular TransMux Channel it is being associated.

Table 7 — BypassFlexMuxDescriptor

Syntax	Num. Of Bytes

This descriptor is empty. The dmifDescriptorLen is zero.

11.2.1.3 MPEG4SystemsFlexMuxDescriptor

This descriptor shown in Table 8 is used to identify a FlexMux Channel Number as defined in ISO/IEC 14496-1.

Table 8 — MPEG4SystemsFlexMuxDescriptor

Syntax	Num. Of Bytes
FlexMuxChannelNumber	1

FlexMuxChannelNumber: Provides the number of the channel in the MPEG-4 FlexMux.

11.2.1.4 MPEG4SystemsMuxCodeDescriptor

This descriptor shown in Table 9 is used to carry the FlexMux MuxCodeTables as defined in ISO/IEC 14496-1.

Table 9 — MPEG4SystemsMuxCodeDescriptor

Syntax	Num. Of Bytes
MuxCodeTable	DmifDescriptorLen

MuxCodeTable: Carries the FlexMux MuxCodeTables, whose syntax and semantics is defined in MPEG-4 Systems ISO/IEC 14496-1.

11.2.2 DMIF to DMIF data

Messages which contain information to be passed between DMIF peers but that need not be inspected by the Network use the `ddData` structure to convey such information. This structure contains a count and a list of DMIF descriptors which are defined in subclause 11.2.1. Table 10 defines the format of the `ddData()` structure.

Table 10 — `ddData()` format

Syntax	Num. of Bytes
<pre>ddData(){ dmifDescriptorCount for(i=0;i<dmifDescriptorCount;i++) { dmifDescriptor() } }</pre>	2

dmifDescriptorCount: Indicates the total number of `dmifDescriptor()` structures which are included in the list.

dmifDescriptor(): Refer to subclause 11.2.1 for DMIF descriptor definitions.

11.2.3 Resource Descriptors

A Network Session consists of a relationship between two DMIF peers. Messages which are used to request resources contain a `Resources()` data structure. This structure contains a count and a list of resource descriptors.

Table 11 defines the format of the `Resources()` structure.

Table 11 — `Resources()` format

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

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 and values of a resource which is being requested or which has been assigned to a session.

Table 12 describes the general format of a DMIF Resource Descriptor.

Table 12 — General format of the Resource Descriptor

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

The **commonDescriptorHeader** is normative and shall be included with every resource descriptor definition. Table 13 defines the format of the commonDescriptorHeader.

Table 13 — Format of the commonDescriptorHeader

Syntax	Num. of Bytes
commonDescriptorHeader() { resourceDescriptorType resourceLength resourceDataFieldCount if (resourceDescriptorType == 0xffff) { typeOwnerId typeOwnerValue } }	 2 2 2 3 3

The **resourceDescriptorType** field defines the specific resource being requested. The resourceDescriptorTypes, and their corresponding descriptors, are specified in ISO/IEC 13818-6, subclause 4.7.5. Annex E provides an extract from that subclause, where the resourceDescriptorTypes useful to this part of ISO/IEC 14496 are defined. The resourceDescriptorTypes used in DMIF are by no means limited to the ones identified in Annex E.

The **resourceLength** field defines the total length of the resourceDescriptorDataField section, which follows the commonDescriptorHeader. The value of the resourceLength 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, see Annex E.

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 that is outside of the scope of DSM-CC.

The **typeOwnerId** field is the first three bytes of an IEEE Organization Unique Identifier (OUI) as specified in the IEEE-802.1990.

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

Annex E provides extracts from ISO/IEC 13818-6 that describe the resourceDescriptorDataField and the resourceDescriptors that are useful to this part of ISO/IEC 14496.

11.2.4 QoS Descriptors

Two QoS Descriptors are defined in this part of ISO/IEC 14496: one is used at the DMIF-Application Interface, another one is used at the DMIF-Network Interface. Both are intended to specify the “transport” Quality of Service for an individual Elementary Stream, however only the traffic parameters (e.g., bitrate) have been specified, not the performance parameters (e.g., delay, loss probability, ...).

The exact syntax of the QoS_Descriptor used at the DAI is not specified in this part of ISO/IEC 14496, only the semantic is given (in subclause 10.2.1).

The exact syntax of the QoS_Descriptor used at the DNI and in the messages through the network is instead fully specified in this subclause.

DMIF may aggregate multiple Elementary Streams and carry them into a single TransMux, and compute the aggregate “transport” QoS. DMIF will then further map the “transport” QoS requirements for a particular TransMux into specific “network” QoS.

The QoS_Descriptor shall be able to carry a number of QoS metrics. The set of metrics currently defined and their semantic is summarized in Table 14.

Table 14 — ISO/IEC 14496-6 defined QoS metrics

QoS Metric	Semantic Description
PRIORITY	Priority for the stream
MAX_AU_SIZE	Maximum size of an AU
AVG_BITRATE	Average bit rate
MAX_BITRATE	Maximum bit rate

Table 15 describes the generic format of the QoS Descriptor.

Table 15 — QoSDescriptor

Syntax	Num. Of Bytes
streamPriority	1
QoS_QualifierCount	1
for(i=0; i<QoS_QualifierCount; i++) {	
QoS_QualifierTag	1
QoS_QualifierDataLength	1
QoS_QualifierData	QoS_QualifierDataLength
}	

streamPriority: Provides the priority of the channel. Lower values mean lower priority. Only the 5 least significant bits are used (values range from 0 to 31). The 3 most significant bits are reserved.

QoS_QualifierCount: Provides the number of QoS metrics specified in the descriptor.

QoS_QualifierTag: Identifies the metric type.

QoS_QualifierDataLength: Is the length of the following metric value.

QoS_QualifierData: Provides the metric value.

The **QoS_QualifierTags** are shown in Table 16.

Table 16 — ISO/IEC 14496-6 defined QoS_QualifierTags

QoS_QualifierTag	Value	Semantic Description
Reserved	0x00	ISO/IEC 14496-6 Reserved
PRIORITY	0x01	Priority for the stream
Reserved	0x02-0x40	ISO/IEC 14496-6 Reserved
MAX_AU_SIZE	0x41	Maximum size of an AU
AVG_BITRATE	0x42	Average bit rate
MAX_BITRATE	0x43	Maximum bit rate
Reserved	0x44-0x7f	ISO/IEC 14496-6 Reserved
User defined	0x80-0xff	User Private

Each QoS_QualifierTag is assigned the type and unit to be used in the QoS_QualifierData as specified in Table 17.

Table 17 — metric data syntax

QoS_QualifierTag	Type	Unit
PRIORITY	Byte	Higher values mean higher priority
MAX_AU_SIZE	Long	Bytes
AVG_BITRATE	Long	Bits/Second
MAX_BITRATE	Long	Bits/Second

11.2.5 Direction parameter

Table 18 — Direction codes

Value	Description
0x00	ISO/IEC 14496-6 Reserved
0x01	Downstream (Sender to Receiver)
0x02	Upstream (Receiver to Sender)
0x03-0xff	ISO/IEC 14496-6 Reserved

11.2.6 Reason parameter

Table 19 — Reason codes

Value	Description
0x0000	Reason Normal
0x0001	Reason Error
0x0002-0x0fff	ISO/IEC 14496-6 Reserved
0x1000-0xffff	User Private

11.2.7 Response parameter

Table 20 — Response codes

Value	Description
0x0000	Response OK
0x0001	Response Error
0x0002-0x0fff	ISO/IEC 14496-6 Reserved
0x1000-0xffff	User Private

11.3 DMIF-Network Interface primitives

This subclause lists the primitives specified for the DMIF-Network Interface and describes the parameters used. The C++ like formalism used to describe the primitives only aims at capturing their semantic meaning. IN and OUT keywords allow to clearly distinguish the parameters provided to the other side of the interface from those returned from it; by no means they are meant to force a synchronous implementation. The loop() construct allows to concisely represent an array of elements.

The optional [Callback] construct indicates that there are always pairs of primitives (with and without the suffix Callback) with identical parameters. A primitive without the 'Callback' is issued by the DMIF peer initiating the primitive, as a consequence the corresponding primitive with the 'Callback' is issued at the target DMIF peer (see Walkthrough in Annex B).

The primitives are:

- DN_SessionSetup[Callback] (IN: networkSessionId, calledAddress, callingAddress, compatibilityDescriptorIn; OUT: response, compatibilityDescriptorOut)
- DN_SessionRelease[Callback] (IN: networkSessionId, reason; OUT: response)
- DN_ServiceAttach[Callback] (IN: networkSessionId, serviceId, serviceName, ddDataIn(); OUT: response, ddDataOut())
- DN_ServiceDetach[Callback] (IN: networkSessionId, serviceId, reason; OUT: response)

- DN_TransMuxSetup[Callback] (IN: networkSessionId, loop(TAT, qosDescriptor; resources()); OUT: loop(response, resources()))
- DN_TransMuxRelease[Callback] (IN: networkSessionId, loop(TAT); OUT: loop(response))
- DN_ChannelAdd[Callback] (IN: networkSessionId, serviceId, loop(CAT, qosDescriptor, direction, ddDataIn()); OUT: loop(response, TAT, ddDataOut()))
- DN_ChannelAdded[Callback] (IN: networkSessionId, serviceId, loop(CAT, qosDescriptor, direction, TAT, ddDataIn()); OUT: loop(response, ddDataOut()))
- DN_ChannelDelete[Callback] (IN: networkSessionId, loop(CAT, reason); OUT: loop(response))
- DN_TransMuxConfig[Callback] (IN: networkSessionId, loop(TAT, ddDataIn()); OUT: loop(response))
- DN_UserCommand[Callback] (IN: networkSessionId, loop(CAT, ddDataIn()))

The parameters have the following semantics:

calledAddress: the network address of the Target DMIF peer. The Originating DMIF peer, issuing DN_SessionSetup, strips this addressing portion of a URL (see DAI definitions) to identify the location of the Target DMIF where a requested service resides. Depending on the form of the URL the originating DMIF peer may have to invoke directory service to obtain the calledAddress.

callingAddress: the network address of the Originating DMIF peer, i.e. the peer issuing DN_SessionSetup.

compatibilityDescriptor: in the case when native signalling does not provide capability exchange, this parameter instead (opaque to native signalling) provides a descriptor to enable capability exchange between DMIF peers as syntactically defined in subclause DMIF Default signalling with Compatibility Descriptors.

CAT: (Channel Association Tag) is an Association Tag which uniquely identifies a channel end-to-end within a Network Session (identified by networkSessionId) and is not changed during its lifetime within this Network Session. It is generated locally at a DMIF peer prior to issuing a DN_DownstreamChannelAdd() or DN_UpstreamChannelAdd().

ddDataIn(): is an opaque structure providing Delivery layer information; it is transparently transported through native signalling from the local Peer to the remote Peer. The generic syntax of such data is provided in subclause 11.2.2.

ddDataOut(): is an opaque structure providing Delivery layer information; it is transparently transported through native signalling from the remote Peer to the local Peer. The generic syntax of such data is provided in subclause 11.2.2.

direction: see DAI definitions in subclause DMIF-Application Interface primitives. See also subclause Direction parameter.

networkSessionId: uniquely identifies a Network Session end-to-end. It is generated locally at the originating DMIF peer prior to issuing DN_SessionSetup.

qosDescriptor: see DAI definitions in subclause DMIF-Application Interface primitives. See also subclause QoS Descriptors.

reason: see DAI definitions in subclause DMIF-Application Interface primitives. See also subclause Reason parameter.

resources(): is a structure containing a count and a list of DSM-CC resource descriptors which are defined in subclause 4.7 of ISO/IEC 13818-6. See also subclause Resource Descriptors.

response: see DAI definitions in subclause DMIF-Application Interface primitives. See also subclause Response parameter.

serviceld: uniquely identifies a service end-to-end within a Network Session (identified by `networkSessionId`). It is generated locally at a DMIF peer prior to issuing `DN_ServiceAttach`. It corresponds to a `serviceName` which is opaque to DMIF.

serviceName: see DAI definitions in subclause DMIF-Application Interface primitives.

TAT: (Transmux Channel Association Tag) is an Association Tag which uniquely identifies a Transmux Channel end-to-end within a Network Session (identified by `networkSessionId`) and is not changed during its lifetime within this Network Session. It is generated locally at a DMIF peer prior to issuing a `DN_TransMuxSetup`.

11.4 DMIF-Network Interface semantics

11.4.1 DN_SessionSetup ()

`DN_SessionSetup` (IN: `networkSessionId`, `calledAddress`, `callingAddress`, `compatibilityDescriptorIn`; OUT: `response`, `compatibilityDescriptorOut`)

`DN_SessionSetupCallback` (IN: `networkSessionId`, `calledAddress`, `callingAddress`, `compatibilityDescriptorIn`; OUT: `response`, `compatibilityDescriptorOut`)

`DN_SessionSetup()` is issued by the Originating DMIF to establish a Network Session with the Target DMIF. This is the very first action performed in establishing a relation between two peers.

The **calledAddress** is extracted by the **URL** provided by the DMIF User in the `DA_ServiceAttach()`. The **callingAddress** is automatically computed by the Originating DMIF entity. The **networkSessionId** is a network wide unique identifier assigned by the Originating DMIF entity.

Upon receiving the `DN_SessionSetupCallback()` the Target DMIF peer will analyze the set of tools described in the **compatibilityDescriptorIn** and reply to the Originating DMIF peer with a response code and the indication of the matching set of tools in the **compatibilityDescriptorOut**.

11.4.2 DN_SessionRelease ()

`DN_SessionRelease` (IN: `networkSessionId`, `reason`; OUT: `response`)

`DN_SessionReleaseCallback` (IN: `networkSessionId`, `reason`; OUT: `response`)

`DN_SessionRelease()` is issued by a DMIF peer to close all relations with the other peer. In normal conditions, it is only invoked when all services related to the indicated **networkSessionId** have been already detached (see `DN_ServiceDetach[Callback]`).

Upon receiving the `DN_SessionReleaseCallback()` the Target DMIF peer will reply to the Originating DMIF peer with a response code.

After completion of this procedure the **networkSessionId** is invalid.

11.4.3 DN_ServiceAttach ()

`DN_ServiceAttach` (IN: `networkSessionId`, `serviceld`, `serviceName`, `ddDataIn()`; OUT: `response`, `ddDataOut()`)

DN_ServiceAttachCallback (IN: networkSessionId, serviceId, serviceName, ddDataIn(); OUT: response, ddDataOut())

DN_ServiceAttach() is issued by the Originating DMIF to establish a Service Session with the Target DMIF. This Service Session is established inside a previously established Network Session, identified by the **networkSessionId**.

The **serviceName** is extracted by the URL provided by the DMIF User in the DA_ServiceAttach(). The **ddDataIn()** contains the **uuData()** provided by the DMIF User in the DA_ServiceAttach(). The **serviceId** is a Network Session wide unique identifier assigned by the Originating DMIF entity.

Upon receiving the DN_ServiceAttachCallback() the Target DMIF peer will issue DA_ServiceAttachCallback().

Upon returning of the DA_ServiceAttachCallback(), the Target DMIF peer will reply to the Originating DMIF peer with a response code. The **ddDataOut()** contains the **uuData()** provided by the DMIF User in return to the DA_ServiceAttachCallback().

11.4.4 DN_ServiceDetach ()

DN_ServiceDetach (IN: networkSessionId, serviceId, reason; OUT: response)

DN_ServiceDetachCallback (IN: networkSessionId, serviceId, reason; OUT: response)

DN_ServiceDetach() is issued by the Originating DMIF to detach a Service Session previously established with the Target DMIF. This Service Session is identified by the **serviceId** inside the Network Session identified by the **networkSessionId**.

Upon receiving the DN_ServiceDetachCallback() the Target DMIF peer will issue DA_ServiceDetachCallback().

Upon returning of the DA_ServiceDetachCallback(), the Target DMIF peer will reply to the Originating DMIF peer with a response code.

After completion of this procedure the **serviceId** is invalid.

11.4.5 DN_TransMuxSetup ()

DN_TransMuxSetup (IN: networkSessionId, loop(TAT, direction, qosDescriptor; resources()); OUT: loop(response, resources()))

DN_TransMuxSetupCallback (IN: networkSessionId, loop(TAT, direction, qosDescriptor; resources()); OUT: loop(response, resources()))

DN_TransMuxSetup() is issued by the Originating DMIF to establish one or more Transmux Channels inside a Network Session previously established with the Target DMIF. This Network Session is identified by the **networkSessionId**.

The **TAT** is a Network Session wide unique identifier assigned by the Originating DMIF entity. The **direction** determines the direction of the Transmux Channel. The **qosDescriptor** is set based on the information contained in the **qosDescriptors** passed in the DA_ChannelAdd() and related to the Elementary Streams being carried in the Transmux Channel. The **resources()** parameter contains the description of the network resources to be reserved for the Transmux Channel.

Upon receiving the DN_TransMuxSetupCallback() the Target DMIF peer will possibly complete and update the **resources()** parameter and reply to the Originating DMIF peer with a response code.

11.4.6 DN_TransMuxRelease ()

DN_TransMuxRelease (IN: networkSessionId, loop(TAT); OUT: loop(response))

DN_TransMuxReleaseCallback (IN: networkSessionId, loop(TAT); OUT: loop(response))

DN_TransMuxRelease() is issued by a DMIF peer to close all logical channels making use of the one or more indicated Transmux Channels. In normal conditions, it is only invoked when all logical channels related to the indicated **TATs** inside the Network Session identified by the **networkSessionId** have been already detached (see DN_ChannelDelete[Callback]).

Upon receiving the DN_TransMuxReleaseCallback() the Target DMIF peer will reply to the Originating DMIF peer with a response code.

After completion of this procedure the **TATs** is are invalid.

11.4.7 DN_ChannelAdd()

DN_ChannelAdd (IN: networkSessionId, servid, loop(CAT, direction, qosDescriptor, ddDataIn()); OUT: loop(response, TAT, ddDataOut()))

DN_ChannelAddCallback (IN: networkSessionId, servid, loop(CAT, direction, qosDescriptor, ddDataIn()); OUT: loop(response, TAT, ddDataOut()))

DN_ChannelAdd() is issued by the Originating DMIF to open one or more logical channels inside a Service Session. The Service Session is identified by the **servid** inside the Network Session identified by the **networkSessionId**.

For each logical channel to be established, a tuple of parameters is provided, some of which is derived from parameters passed in the DA_ChannelAdd() and related to the Elementary Stream being carried in the logical channel. The **CAT** is a Network Session wide unique identifier assigned by the Originating DMIF entity. The **direction** is set based on the related **direction** parameter passed in the DA_ChannelAdd(). The **qosDescriptor** contains the complete description of the Quality of Service requested for a particular channel and is set based on the information contained in the related **qosDescriptor** passed in the DA_ChannelAdd(). The **ddDataIn()** contains the related **uuData()** provided by the DMIF User in the DA_ChannelAdd().

Upon receiving the DN_ChannelAddCallback() the Target DMIF peer will issue DA_ChannelAddCallback().

Upon returning of the DA_ChannelAddCallback(), the Target DMIF peer will reply to the Originating DMIF peer: for each logical channel established (or not established), a tuple of parameters is provided, some of which is derived from parameters passed in the DA_ChannelAddCallback() and related to the Elementary Stream being carried in the logical channel. **ddDataOut()** contains the related **uuData()** provided by the DMIF User in return to the DA_ChannelAddCallback() and the DMIF descriptor containing the Flexmux information. The **TAT** contains the Association Tag of the Transmux Channel carrying the logical channel (see DN_TransMuxSetup[Callback]).

11.4.8 DN_ChannelAdded()

DN_ChannelAdded (IN: networkSessionId, servid, loop(CAT, direction, qosDescriptor, TAT, ddDataIn()); OUT: loop(response, ddDataOut()))

DN_ChannelAddedCallback (IN: networkSessionId, servid, loop(CAT, direction, qosDescriptor, TAT, ddDataIn()); OUT: loop(response, ddDataOut()))

DN_ChannelAdded() is issued by the Originating DMIF to notify the Target DMIF peer that one or more logical channels inside a Service Session were added. The Service Session is identified by the **servid** inside the Network Session identified by **networkSessionId**.

For each logical channel established, a tuple of parameters is provided, some of which is derived from parameters passed in the DA_ChannelAdd() and related to the Elementary Stream being carried in the logical channel. The **CAT** is a Network Session wide unique identifier assigned by the Originating DMIF entity. The **direction** is set based on the related **direction** parameter passed in the DA_ChannelAdd(). The **qosDescriptor** contains the complete description of the Quality of Service requested for a particular channel and is set based on the information contained in the related **qosDescriptor** passed in the DA_ChannelAdd(). The **ddDataIn()** contains the related **uuData()** provided by the DMIF User in the DA_ChannelAdd() and the DMIF descriptor containing the Flexmux information. The **TAT** contains the Association Tag of the Transmux Channel carrying the logical channel (see DN_TransMuxSetup[Callback]).

Upon receiving the DN_ChannelAdded() the Target DMIF peer will issue DA_ChannelAddCallback().

Upon returning of the DA_ChannelAddCallback(), the Target DMIF peer will reply to the Originating DMIF peer: for each logical channel established (or not established), a tuple of parameters is provided, some of which is derived from parameters passed in the DA_ChannelAddCallback() and related to the Elementary Stream being carried in the logical channel. **ddDataOut()** contains the related **uuData()** provided by the DMIF User in return to the DA_ChannelAddCallback().

11.4.9 DN_ChannelDelete ()

DN_ChannelDelete (IN: networkSessionId, loop(CAT, reason); OUT: loop(response))

DN_ChannelDeleteCallback (IN: networkSessionId, loop(CAT, reason); OUT: loop(response))

DN_ChannelDelete() is issued by the Originating DMIF to close one or more logical channels previously established inside a Network Session. The logical channels are identified by their **CAT** inside the Network Session identified by the **networkSessionId**.

Upon receiving the DN_ChannelDeleteCallback() the Target DMIF peer will issue DA_ChannelDeleteCallback().

Upon returning of the DA_ChannelDeleteCallback(), the Target DMIF peer will reply to the Originating DMIF peer with a response code.

NOTE Since DA_ChannelDelete() may refer to channels in different Network Sessions there may be several DN_ChannelDelete() primitives issued, one for each invoked Network Session.

11.4.10 DN_TransMuxConfig ()

DN_TransMuxConfig (IN: networkSessionId, loop(TAT, ddDataIn()); OUT: loop(response))

DN_TransMuxConfigCallback (IN: networkSessionId, loop(TAT, ddDataIn()); OUT: loop(response))

DN_TransMuxConfig() is issued by the Originating DMIF to reconfigure one or more Transmux Channels previously established inside a Network Session. The Transmux Channels are identified by their **TAT** inside the Network Session identified by the **networkSessionId**.

For each Transmux Channel a tuple of parameters is provided. The **TAT** contains the Association Tag of the Transmux Channel; **ddDataIn()** contains the DMIF descriptor containing the Flexmux information.

Upon receiving the DN_TransMuxConfigCallback() the Target DMIF peer will reply to the Originating DMIF peer with a response code.

11.4.11 DN_UserCommand ()

DN_UserCommand (IN: networkSessionId, loop(CAT, ddDataIn()))

DN_UserCommandCallback (IN: networkSessionId, loop(CAT, ddDataIn()))

DN_UserCommand() is issued by the Originating DMIF to pass user data to the corresponding peer that refers to specific channels. Each channel is identified by its CAT, that is scoped inside the Network Session identified by the **networkSessionId**.

Upon receiving the DN_UserCommandCallback() the Target DMIF peer will issue DA_UserCommandCallback().

NOTE the DA_UserCommand() user call returns immediately since there is no expected response.

12 Control Plane mappings

12.1 Default syntax

When DMIF operates on a network that cannot extend its signalling to map the DNI primitives, a DMIF signalling channel shall be established. DMIF signalling messages as defined in this specification will be exchanged over that channel.

The following DNI primitives are mapped into DMIF signalling messages:

- DN_SessionSetup ()
- DN_SessionRelease ()
- DN_ServiceAttach ()
- DN_ServiceDetach ()
- DN_TransMuxSetup ()
- DN_TransMuxRelease ()
- DN_ChannelAdd ()
- DN_ChannelAdded ()
- DN_ChannelDelete ()
- DN_TransMuxConfig ()
- DN_UserCommand ()

12.1.1 Syntax elements

12.1.1.1 General message format

All DMIF signalling messages have a common message format. Table 21 defines the DMIF Signalling Message format. This format is called the DMIFSignallingMessage().

Table 21 — General Format of DMIF Signalling Message

Syntax
<pre>DMIFSignallingMessage(){ DsmccMessageHeader() MessagePayload() Padding() }</pre>

The **dsmccMessageHeader** is defined in the subclause DSM-CC message header.

The **MessagePayload** is dependent on the particular message. Subclause 12.1.2 illustrates the syntax of each such message.

The **Padding** is a series of up to 3 bytes, encoded as 0s, that enforces 4 bytes alignment.

12.1.1.2 DSM-CC message header

All MPEG-4 DMIF signalling messages conform to the generic format of MPEG-2 DSM-CC messages and begin with the DSM-CC MessageHeader as defined in clause 2 of ISO/IEC 13818-6. This header contains information about the type of message being passed. Table 22 defines the format of a DSM-CC message header as used for MPEG-4 DMIF signalling messages.

Table 22 — 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
}	

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 23 defines the possible dsmccTypes.

Table 23 — MPEG-2 DSM-CC dsmccType values

DsmccType	Description
0x00	ISO/IEC 13818-6 Reserved.
0x01-0x05	ISO/IEC 13818-6 Defined.
0x06	Identifies the message as an ISO/IEC 14496-6 DMIF Signalling message.
0x07-0x7F	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 contained in the request-confirm command pair shall be identical. Its format follows the normative definition in ISO/IEC 13818-6.

The transactionId is constructed of a 2 bit transactionId originator indication and a 30 bit transaction number. Figure 8 describes the format of the transactionId field.

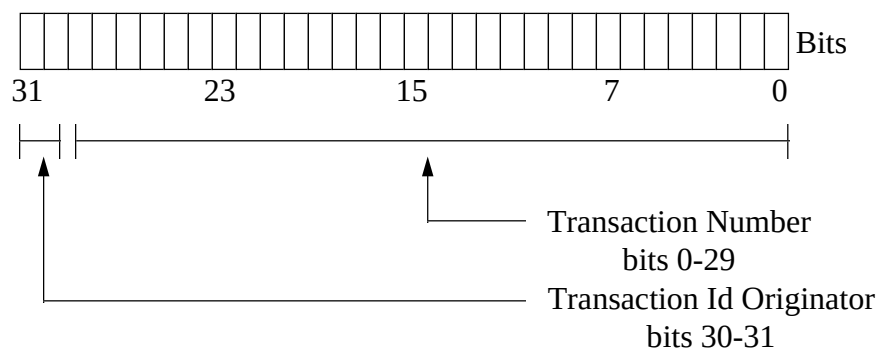


Figure 8 — Format of transactionId field

The coding of the transactionId originator indication is described in Table 24.

Table 24 — MPEG-2 DSM-CC transactionId originator

Originator	Description
0x00	TransactionId is assigned by the Session Originator.
0x01	TransactionId is assigned by the other Peer.
0x02	not used for ISO/IEC 14496-6 DMIF signalling messages.
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 shall be set to 0 for MPEG-4 DMIF signalling messages.

The **messageLength** field is used to indicate the total length in bytes of the message following this field. This length includes the padding required for 4 bytes alignment.

12.1.1.3 Message identifiers

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 subclause DSM-CC message header. Figure 9 defines the encoding of the messageId fields used in DMIF signalling messages. Bit 0 is the least significant bit and bit 15 is the most significant bit.

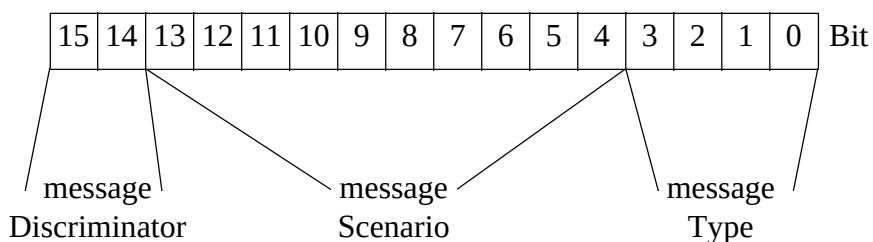


Figure 9 — Format of DMIF Signalling messageId

The **messageDiscriminator** field shall be set to 00. Other values are ISO/IEC 14496-6 Reserved.

The **messageScenario** field is used to indicate the message group to which the message belongs. Table 25 defines the possible values for the messageScenario field.

Table 25 — messageScenario field values

MessageScenario	Description
00 0000 0000	ISO/IEC 14496-6 Reserved.
00 0000 0001	SessionSetup
00 0000 0010	SessionRelease
00 0000 0011	ServiceAttach
00 0000 0100	ServiceDetach
00 0000 0101	TransmuxSetup
00 0000 0110	Transmux Release
00 0000 0111	ChannelAdd
00 0000 1000	ChannelAdded
00 0000 1001	ChannelDelete
00 0000 1010	TransmuxConfig
00 0000 1011	UserCommand
00 0000 1100 - 01 1111 1111	ISO/IEC 14496-6 Reserved.
10 0000 0000 - 11 1111 1111	User Defined Message Scenario

Most of the control messages in this part of ISO/IEC 14496 use a confirmation mechanism. When a DMIF peer issues a request message, the receiver of that message issues a definite response to that message. There are however some cases which do not use this mechanism.

The **messageType** field is used to indicate the directionality of the message. Table 26 defines the possible values for the messageType field.

Table 26 — messageType field values

MessageType	Description
0000	Request Message.
0001	Confirm Message. This indicates that the message is being sent in response to a Request message.
0010-1111	ISO/IEC 14496-6 Reserved.

Table 27 defines the messageId's which are used in the DMIF signalling messages.

Table 27 — DMIF signalling messages

Command	messageld
ISO/IEC 14496-6 reserved	0x0000 - 0x000f
DS_SessionSetupRequest	0x0010
DS_SessionSetupConfirm	0x0011
ISO/IEC 14496-6 reserved	0x0012 - 0x001f
DS_SessionReleaseRequest	0x0020
DS_SessionReleaseConfirm	0x0021
ISO/IEC 14496-6 reserved	0x0022 - 0x002f
DS_ServiceAttachRequest	0x0030
DS_ServiceAttachConfirm	0x0031
ISO/IEC 14496-6 reserved	0x0032 - 0x003f
DS_ServiceDetachRequest	0x0040
DS_ServiceDetachConfirm	0x0041
ISO/IEC 14496-6 reserved	0x0042 - 0x004f
DS_TransMuxSetupRequest	0x0050
DS_TransMuxSetupConfirm	0x0051
ISO/IEC 14496-6 reserved	0x0052 - 0x005f
DS_TransMuxReleaseRequest	0x0060
DS_TransMuxReleaseConfirm	0x0061
ISO/IEC 14496-6 reserved	0x0062 - 0x006f
DS_ChannelAddRequest	0x0070
DS_ChannelAddConfirm	0x0071
ISO/IEC 14496-6 reserved	0x0072 - 0x007f
DS_ChannelAddedRequest	0x0080
DS_ChannelAddedConfirm	0x0081
ISO/IEC 14496-6 reserved	0x0082 - 0x008f
DS_ChannelDeleteRequest	0x0090
DS_ChannelDeleteConfirm	0x0091
ISO/IEC 14496-6 reserved	0x0092 - 0x009f
DS_TransMuxConfigRequest	0x00a0
DS_TransMuxConfigConfirm	0x00a1
ISO/IEC 14496-6 reserved	0x00a2 - 0x00af
DS_UserCommand	0x00b0
ISO/IEC 14496-6 reserved	0x00b1
ISO/IEC 14496-6 reserved	0x00b2 - 0x00bf
ISO/IEC 14496-6 reserved	0x00c0 - 0x1fff
User defined messageld's.	0x2000 - 0x3fff
ISO/IEC 14496-6 reserved	0x4000 - 0xffff

12.1.1.4 Use of Resources() structure in DMIF signalling messages

A Network Session consists of a relationship between two DMIF peers. Messages which are used to request resources contain a Resources() data structure. The Resources() structure is defined in subclause 11.2.3.

12.1.1.5 Use of ddData() structure in DMIF signalling messages

Messages which contain information to be passed between DMIF peers but that need not be inspected by the Network use the ddData() structure to convey such information. In this part of ISO/IEC 14496 ddData() contains DMIF descriptors. The ddData() structure is defined in subclause 11.2.2.

12.1.1.6 The relation between DNI and DS messages

Each DNI primitive pair (i.e. primitives with and without callback suffix) has two associated messages. One which corresponds to the "IN:" parameters the and one which corresponds to the "OUT:" parameters. As follows:

DN_<xxx>[Callback](IN: <yyy>; OUT: <zzz>)

gives:

DS_<xxx>Request(){<yyy>}

and

DS_<xxx>Confirm(){<zzz>}

12.1.2 DNI mapping to DMIF signalling messages

12.1.2.1 DS_SessionSetupRequest

Table 28 — DS_SessionSetupRequest message

Syntax	Num. Of Bytes
DS_SessionSetupRequest(){ dsmccMessageHeader() networkSessionId compatibilityDescriptor() }	10

The format of the **compatibilityDescriptor()** is defined in subclause 13.1.

12.1.2.2 DS_SessionSetupConfirm

Table 29 — DS_SessionSetupConfirm message

Syntax	Num. of Bytes
DS_SessionSetupConfirm(){ dsmccMessageHeader() Response compatibilityDescriptor() }	2

The format of the **compatibilityDescriptor()** is defined in subclause 13.1.

12.1.2.3 DS_SessionReleaseRequest

Table 30 — DS_SessionReleaseRequest message

Syntax	Num. Of Bytes
DS_SessionReleaseRequest(){ dsmccMessageHeader() NetworkSessionId Reason }	 10 2

12.1.2.4 DS_SessionReleaseConfirm

Table 31 — DS_SessionReleaseConfirm message

Syntax	Num. of Bytes
DS_SessionReleaseConfirm(){ dsmccMessageHeader() response }	 2

12.1.2.5 DS_ServiceAttachRequest

Table 32 — DS_ServiceAttachRequest message

Syntax	Num. Of Bytes
DS_ServiceAttachRequest(){ dsmccMessageHeader() networkSessionId serviceId serviceNameLen serviceName ddData() }	 10 2 1 serviceNameLen

If uuData() is provided through DA_ServiceAttach(), **ddData()** shall contain a UuDataDescriptor containing the uuData().

12.1.2.6 DS_ServiceAttachConfirm

Table 33 — DS_ServiceAttachConfirm message

Syntax	Num. Of Bytes
DS_ServiceAttachConfirm(){ dsmccMessageHeader() response ddData() }	 2

If `uuData()` is provided through `DA_ServiceAttachCallback()`, **ddData()** shall contain a `UuDataDescriptor` containing the `uuData()`.

12.1.2.7 DS_ServiceDetachRequest

Table 34 — DS_ServiceDetachRequest message

Syntax	Num. Of Bytes
DS_ServiceDetachRequest(){ dsmccMessageHeader() networkSessionId serviceId reason }	 10 2 2

12.1.2.8 DS_ServiceDetachConfirm

Table 35 — DS_ServiceDetachConfirm message

Syntax	Num. Of Bytes
DS_ServiceDetachConfirm(){ dsmccMessageHeader() response }	 2

12.1.2.9 DS_TransMuxSetupRequest

Table 36 — DS_TransMuxSetupRequest message

Syntax	Num. Of Bytes
DS_TransMuxSetupRequest(){ dsmccMessageHeader() networkSessionId count loop(count) { TAT direction QoSDescriptor() resources() } }	 10 1 2 1

The format of the **QoSDescriptor()** is defined in subclause 11.2.3.

resources() contains DSM-CC resource descriptors.

12.1.2.10 DS_TransMuxSetupConfirm

Table 37 — DS_TransMuxSetupConfirm message

Syntax	Num. Of Bytes
DS_TransMuxSetupConfirm(){ dsmccMessageHeader() count loop(count) { response resources() } }	 1 2

resources() contains DSM-CC resource descriptors.

12.1.2.11 DS_TransMuxReleaseRequest

Table 38 — DS_TransMuxReleaseRequest message

Syntax	Num. Of Bytes
DS_TransMuxReleaseRequest(){ dsmccMessageHeader() networkSessionId count loop(count) { TAT } }	 10 1 2

12.1.2.12 DS_TransMuxReleaseConfirm

Table 39 — DS_TransMuxReleaseConfirm message

Syntax	Num. Of Bytes
DS_TransMuxReleaseConfirm(){ dsmccMessageHeader() count loop(count) { response } }	 1 2

12.1.2.13 DS_ChannelAddRequest

Table 40 — DS_ChannelAddRequest message

Syntax	Num. Of Bytes
DS_ChannelAddRequest(){	
dsmccMessageHeader()	
networkSessionId	10
servicId	2
count	1
loop(count) {	
CAT	2
direction	1
qosDescriptor()	
ddData()	
}	
}	

If uuData() is provided through DA_ChannelAdd(), **ddData()** shall contain a UuDataDescriptor containing the uuData().

12.1.2.14 DS_ChannelAddConfirm

Table 41 — DS_ChannelAddConfirm message

Syntax	Num. Of Bytes
DS_ChannelAddConfirm(){ dsmccMessageHeader() count loop(count) { response TAT ddData() } }	 1 2 2

ddData() shall contain a BypassFlexMuxDescriptor or an MPEG4SystemsFlexMuxDescriptor.

If `uuData()` is provided through `DA_ChannelAddCallback()`, **`ddData()`** shall also contain a `UuDataDescriptor` containing the `uuData()`.

12.1.2.15 DS_ChannelAddedRequest

Table 42 — DS_ChannelAddedRequest message

Syntax	Num. Of Bytes
DS_ChannelAddedRequest(){	
dsmccMessageHeader()	
NetworkSessionId	10
ServiceId	2
Count	1
loop(count) {	
CAT	2
direction	1
qosDescriptor()	
TAT	2
ddData()	
}	
}	

ddData() shall contain a BypassFlexMuxDescriptor or an MPEG4SystemsFlexMuxDescriptor.

If `uuData()` is provided through `DA_ChannelAdd()`, **`ddData()`** shall also contain a `UuDataDescriptor` containing the `uuData()`.

12.1.2.16 DS_ChannelAddedConfirm

Table 43 — DS_ChannelAddedConfirm message

Syntax	Num. Of Bytes
DS_ChannelAddedConfirm(){ dsmccMessageHeader() Count loop(count) { Response ddData() } }	 1 2

If `uuData()` is provided through `DA_ChannelAddCallback()`, **`ddData()`** shall also contain a `UuDataDescriptor` containing the `uuData()`.

12.1.2.17 DS_ChannelDeleteRequest

Table 44 — DS_ChannelDeleteRequest message

Syntax	Num. Of Bytes
DS_ChannelDeleteRequest(){ dsmccMessageHeader() networkSessionId count loop(count) { CAT reason } }	 10 1 2 2

12.1.2.18 DS_ChannelDeleteConfirm

Table 45 — DS_ChannelDeleteConfirm message

Syntax	Num. Of Bytes
DS_ChannelDeleteConfirm(){ dsmccMessageHeader() count loop(count) { response } }	 1 2

12.1.2.19 DS_TransMuxConfigRequest

Table 46 — DS_TransMuxConfigRequest message

Syntax	Num. Of Bytes
DS_TransMuxConfigRequest(){ dsmccMessageHeader() networkSessionId count loop(count) { TAT ddData() } }	 10 1 2

ddData() may contain an MPEG4SystemsMuxCodeDescriptor.

12.1.2.20 DS_TransMuxConfigConfirm

Table 47 — DS_TransMuxConfigConfirm message

Syntax	Num. Of Bytes
DS_TransMuxConfigConfirm(){ dsmccMessageHeader() count loop(count) { response } }	 1 2

12.1.2.21 DS_UserCommand

Table 48 — DS_UserCommand message

Syntax	Num. Of Bytes
DS_UserCommand(){ dsmccMessageHeader() networkSessionId count loop(count) { CAT ddData() } }	 10 1 2

If uuData() is provided through DA_UserCommand(), **ddData()** shall contain a UuDataDescriptor containing the uuData().

12.2 Syntax for IP networks with (or without) RSVP signalling, using TCP for DMIF signalling

When DMIF operates with IP networks with (or without) RSVP signalling, and uses TCP for DMIF signalling, it shall use a DMIF signalling channel and the Default Syntax for DMIF signalling messages as specified in subclause 12.1 for all DNI primitives except:

- DN_SessionSetup ()
- DN_SessionRelease ()
- DN_TransMuxSetup ()
- DN_TransMuxRelease ()

The mapping of the above primitives into DMIF signalling messages and socket actions is described in subclause DNI mapping to socket actions and RSVP signalling.

12.2.1 Approach overview

Whenever a new session is started, a DMIF signalling channel is established on a TCP socket. Whenever a new TransMux is requested, a new TCP or UDP socket is created.

12.2.2 DSM-CC Resource Descriptors used

The DSM-CC Resource Descriptor used is the IP resource descriptor that is defined in subclause 4.7.5.9 of ISO/IEC 13818-6, and is reported in Annex E for convenience.

12.2.3 Usage of networkSessionIds

A DMIF Signalling channel is established for each Network Session; there is a 1 to 1 relation between the socket carrying the DMIF signalling messages and the networkSessionId. Therefore the networkSessionId field contained in the DMIF signalling messages although present is not used.

12.2.4 Usage of transactionIds

The state machine for transactionIds (see Annex D) allows the repetition of a message up to a certain number of times when a certain timeout expires. Timeouts and number of retries are predefined locally.

Since in this case the DMIF signalling messages are carried on a reliable channel, the number of retries shall be set to 0.

12.2.5 DNI mapping to socket actions and RSVP signalling

12.2.5.1 DN_SessionSetup ()

The DN_SessionSetup () functionality is obtained through the setup of a DMIF Signalling channel, and the exchange of DS_SessionSetup messages. This procedure works as follows:

- 1) the target peer is supposed to listen on a well known TCP port number (DMIF_PORT)
- 2) the originating peer creates a TCP socket and connects to the target peer, using the DMIF_PORT port number and the TCP protocol.
- 3) the target peer accepts the connection: this connection carries the DMIF signalling channel.
- 4) the originating peer sends a DS_SessionSetupRequest message on the just established connection.
- 5) the target peer replies with a DS_SessionSetupConfirm message.

NOTE Steps 4 and 5 are only needed for the compatibility exchange.

12.2.5.2 DN_SessionRelease ()

The DN_SessionRelease () functionality is obtained through the deletion of the DMIF Signalling channel. Since the networkSessionId field in the DS messages is not used, the DS_SessionRelease messages are not transmitted. This procedure works as follows:

- 1) the originating peer closes the TCP socket corresponding to the DMIF signalling channel.
- 2) the target peer is notified that the peer has closed the socket corresponding to the DMIF signalling channel, and closes the socket.

NOTE 1 Since the networkSessionId parameter is not used, the DS_SessionRelease messages are omitted.

NOTE 2 The reason code is always set at the target peer to indicate "remote disconnection".

NOTE 3 The response code is always set at the originating peer to indicate "OK".

NOTE 4 In case the socket is automatically closed by the operating system or network, both sides behave as targets.

12.2.5.3 DN_TransMuxSetup ()

The DN_TransMuxSetup () functionality is obtained through the setup of a new socket, and the exchange of DS_TransMuxSetup messages. This procedure works as follows:

UDP sockets

- 1) the originating peer creates a UDP socket and binds it to some port number.
- 2) the originating peer creates an IP resource descriptor, setting the **sourceIpAddress**, **sourceIpPort** and **ipProtocol** fields.
- 3) the originating peer generates a Transmux Association Tag, and sends a DS_TransMuxSetupRequest message containing the above TAT and resource.
- 4) the originating peer associates the socket to the Transmux Association Tag carried in the DS_TransMuxSetup messages.
- 5) the target peer creates a UDP socket.
- 6) the target peer associates the socket to the Transmux Association Tag carried in the DS_TransMuxSetup messages.
- 7) the target peer updates the received IP resource descriptor, setting the **destinationIpAddress** and **destinationIpPort** fields.
- 8) the target peer sends a DS_TransMuxSetupConfirm message containing the above resource.
- 9) the originating peer receives the DS_TransMuxSetupConfirm message and retrieves the address and port number of the peer.
- 10) the originating peer generates a RSVP PATH with Sender_Tspec (containing the destination address and port, and the QoS requirements) and ADSpec.
- 11) the intermediate routers modify the ADSpec.
- 12) the target peer receives the RSVP message with Sender_Tspec and the updated ADSpec, and based on meeting the QoS designated for the socket generates a RESV message with the FlowSpec.
- 13) the originating peer receives the RESV message
- 14) the setup is complete

NOTE 1 Steps 10 to 13 are only executed if RSVP is used.

NOTE 2 RSVP may require a reduction of the MTU size which is presently not supported in this part of ISO/IEC 14496.

TCP sockets

- 1) the originating peer creates a TCP socket, binds it to some port number, and listens on that socket.
- 2) the originating peer creates an IP resource descriptor, setting the **sourceIpAddress**, **sourceIpPort** and **ipProtocol** fields.
- 3) the originating peer generates a Transmux Association Tag, and sends a DS_TransMuxSetupRequest message containing the above TAT and resource.
- 4) the originating peer provisionally associates the listening socket to the Transmux Association Tag carried in the DS_TransMuxSetup messages.
- 5) the target peer creates a TCP socket and connects to the target peer socket, as identified by the IP resource descriptor.

- 6) the originating peer accepts the connection, thus gets a new socket, and looks at the new socket's peer address and port; it associates the new socket to the TAT that was provisionally associated to the corresponding listening socket and closes the listening socket.
- 7) the target peer associates the socket to the Transmux Association Tag carried in the DS_TransMuxSetup messages.
- 8) the target peer updates the received IP resource descriptor, setting the **destinationIpAddress** and **destinationIpPort** fields.
- 9) the target peer sends a DS_TransMuxSetupConfirm message containing the above resource.
- 10) the originating peer receives the DS_TransMuxSetupConfirm message and retrieves the address and port number of the peer.
- 11) the originating peer compares the sockets' peer address and port with the peer address and port as conveyed in the fields of the IP resource descriptors and associates the socket to the Transmux Association Tag carried in the relevant DS_TransMuxSetup messages.
- 12) the setup is complete

NOTE The originating peer should be prepared to receive the DS_TransMuxSetupConfirm message and accept an incoming connection in any order, thus the comparison in step 11 shall support both sequences.

12.2.5.4 DN_TransMuxRelease ()

The DN_TransMuxRelease () functionality is obtained through the deletion of the socket corresponding to a particular TAT, and possibly the exchange of DS_TransMuxRelease messages (UDP case). Since the networkSessionId field in the DS messages is not used, the DS_TransMuxRelease messages are not transmitted in the TCP case. This procedure works as follows:

UDP sockets

- 1) the originating peer sends a DS_TransMuxReleaseRequest message containing the TAT associated to the socket being closed.
- 2) the target peer closes the UDP socket corresponding to the above TAT.
- 3) the target peer sends a DS_TransMuxReleaseConfirm message.
- 4) the originating peer closes the UDP socket corresponding to the above TAT.

TCP sockets

- 1) the originating peer closes the TCP socket corresponding to a particular TAT.
- 2) the target peer is notified that the peer has closed the socket corresponding to a particular TAT, and closes the socket.

NOTE 1 The reason code is always set at the target peer to indicate "remote disconnection".

NOTE 2 The response code is always set at the originating peer to indicate "OK".

NOTE 3 In case the socket is automatically closed by the operating system or network, both sides behave as targets.

12.3 Syntax for IP networks with (or without) RSVP Signalling, using UDP for DMIF signalling

When DMIF operates with IP networks with (or without) RSVP Signalling, and uses UDP for DMIF signalling, it shall use a DMIF signalling channel and the Default Syntax for DMIF signalling messages as specified in subclause 12.1 for all DNI primitives except:

- DN_SessionSetup ()
- DN_SessionRelease ()
- DN_TransMuxSetup ()
- DN_TransMuxRelease ()

The mapping of the above primitives into DMIF signalling messages and socket actions is described in subclause DNI mapping to socket actions and RSVP signalling.

12.3.1 Approach overview

Whenever a new session is started, a DMIF signalling channel is established on a UDP socket. Whenever a new TransMux is requested, a new TCP or UDP socket is created.

12.3.2 DSM-CC Resource Descriptors used

See subclause DSM-CC Resource Descriptors used.

12.3.3 Usage of networkSessionIds

A DMIF Signalling channel is established for each Network Session; there is a 1 to 1 relation between the full address of the socket carrying the DMIF signalling messages and the networkSessionId. Therefore the networkSessionId field contained in the DMIF signalling messages although present is not used.

12.3.4 Usage of transactionIds

The state machine for transactionIds (see Annex D) allows the repetition of a message up to a certain number of times when a certain timeout expires. Timeouts and number of retries are predefined locally.

12.3.5 DNI mapping to socket actions and RSVP signalling

12.3.5.1 DN_SessionSetup ()

The DN_SessionSetup () functionality is obtained through the setup of a DMIF Signalling channel, and the exchange of DS_SessionSetup messages. This procedure works as follows:

- 1) The target peer is supposed to have created a UDP socket on a well known UDP port number (DMIF_PORT).
- 2) The originating peer creates a UDP socket and sends a DS_SessionSetupRequest message on the just established connection.
- 3) The target peer replies with a DS_SessionSetupConfirm message.

NOTE 1 The target peer shares a single UDP socket among many Network Sessions.

NOTE 2 The originating peer always creates a new UDP socket, which shall not be shared with any other Network Session.

12.3.5.2 DN_SessionRelease ()

The DN_SessionRelease () functionality is obtained through the exchange of DS_SessionRelease messages and the deletion of the DMIF Signalling channel. This procedure works as follows:

- 1) The originating peer sends a DS_SessionReleaseRequest message on the just established connection.
- 2) The target peer replies with a DS_SessionReleaseConfirm message.
- 3) The originating peer closes the UDP socket corresponding to the DMIF signalling channel.

NOTE 1 The reason code is always set at the target peer to indicate "remote disconnection".

NOTE 2 The response code is always set at the originating peer to indicate "OK".

NOTE 3 The target peer does not close the UDP socket, since it is shared among multiple Network Sessions.

12.3.5.3 DN_TransMuxSetup ()

See subclause DN_TransMuxSetup ().

12.3.5.4 DN_TransMuxRelease ()

See subclause DN_TransMuxRelease ().

12.4 Syntax for ATM networks with Q.2931 signalling

When DMIF operates with ATM with Q.2931, it shall use the Default Syntax as specified in subclause 12.1 for all DNI primitives except:

- DN_SessionSetup ()
- DN_SessionRelease ()
- DN_TransMuxSetup ()
- DN_TransMuxRelease ()

The mapping of the above primitives into Q.2931 signalling messages is described in subclause DNI mapping to Q.2931 .

12.4.1 Approach overview

Whenever a new session is started, a DMIF signalling channel is established on an ATM connection. Whenever a new TransMux is requested, a new ATM connection is created.

12.4.2 DSM-CC Resource Descriptors used

Since the DN_TransMuxSetup () primitive is mapped into native Q.2931 signalling messages, no DSM-CC Resource Descriptors are used.

12.4.3 Usage of networkSessionIds

In order to be able to compress the networkSessionId + TAT field of 12 bytes to fit within an 8-byte B-HLI information field, the networkSessionId field carried in the B-HLI information field will contain the 5 least significant bytes of the 10-byte networkSessionId.

A DMIF Signalling channel is established for each Network Session; there is a 1 to 1 relation between the SVC carrying the DMIF signalling messages and the networkSessionId. Therefore the networkSessionId field contained in the DMIF signalling messages although present is not used.

12.4.4 Usage of transactionIds

The state machine for transactionIds (see Annex D) allows the repetition of a message up to a certain number of times when a certain timeout expires. Timeouts and number of retries are predefined locally.

12.4.5 DNI mapping to Q.2931 signalling messages

12.4.5.1 DN_SessionSetup ()

The DN_SessionSetup () functionality is obtained through the setup of a DMIF Signalling channel, and the exchange of DS_SessionSetup messages. This procedure works as follows:

- 1) The target peer listens on a BHLI for a bilaterally agreed DMIF_SelectorByte
- 2) the originating peer generates a 5 bytes NetworkSessionId and builds a Q.2931 SETUP message, with the following rules:

- Call Reference = X1
- Call Reference Flag = 0 (originator)
- Called Party Number = calledAddress
- Calling Party Number = callingAddress
- QoS parameter = 0 (forward and backward)
- ATM traffic descriptor - forward peak cell rate CLP=0 = 100 (suggested)
- ATM traffic descriptor - forward peak cell rate CLP=0+1 = 100 (suggested)
- ATM traffic descriptor - backward peak cell rate CLP=0 = 100 (suggested)
- ATM traffic descriptor - backward peak cell rate CLP=0+1 = 100 (suggested)
- AAL parameters - AAL type = AAL5
- AAL parameters – forward and backward maximum CPCS-SDU size = 1000 (suggested)
- AAL parameters - SSCS type = NULL
- Broadband bearer capability - bearer class = bcob-x (suggested)
- Broadband bearer capability - traffic type = no indication (suggested)
- Broadband bearer capability - timing requirements = no indication (suggested)
- Broadband bearer capability - susceptibility to clipping = not susceptible (suggested)
- Broadband bearer capability - bearer class = user plane connection configuration = point-to-point
- BHLI - High Layer Information Type = User specific
- BHLI - octets 1 = DMIF_SelectorByte
- BHLI - octets 2-6 = NetworkSessionId
- BHLI - octets 7-8 = 0x0000

3) the target peer receives the SETUP message with in particular the following fields:

- Call Reference = X2
- Call Reference Flag = 1 (target)
- Connection Identifier - VP = VPX2
- Connection Identifier - VC = VCX2
- ...
- BHLI - High Layer Information Type = User specific
- BHLI - octets 1 = DMIF_SelectorByte
- BHLI - octets 2-6 = NetworkSessionId
- BHLI - octets 7-8 = 0x0000

4) the target peer generates the CONNECT message with in particular the following fields:

- Call Reference = X2
- Call Reference Flag = 0 (originator)

5) the originating peer receives the CONNECT message with in particular the following fields:

- Call Reference = X1
- Call Reference Flag = 1 (target)
- Connection Identifier - VP = VPX1
- Connection Identifier - VC = VCX1

6) both peers associate the SVC just established to the 5-bytes networkSessionId carried in the Q.2931 SETUP message. All further message exchanges over this SVC shall be associated to this Network Session.

7) the originating peer sends a DS_SessionSetupRequest message on the just established connection.

8) the target peer replies with a DS_SessionSetupConfirm message.

NOTE 1 Steps 7 and 8 are only needed for the compatibility exchange.

NOTE 2 The CONNECT message may be preceeded by a CALL_PROCEEDING message. In this case, at the originating side, the CALL_PROCEEDING and not the CONNECT will carry the Connection Identifier.

12.4.5.2 DN_SessionRelease ()

The DN_SessionRelease () functionality is obtained through the deletion of the DMIF Signalling channel. Since the networkSessionId field in the DS messages is not used, the DS_SessionRelease messages are not transmitted. This procedure works as follows:

- 1) the originating peer generates a Q.2931 RELEASE message, with the following rules:
 - Call Reference = X1
 - Call Reference Flag = 0 (originator)
- 2) the target peer receives the RELEASE message with in particular the following fields:
 - Call Reference = X2
 - Call Reference Flag = 1 (target)
- 3) the target peer generates the RELEASE COMPLETE message with in particular the following fields:
 - Call Reference = X2
 - Call Reference Flag = 0 (originator)
- 4) the originating peer receives the RELEASE COMPLETE message with in particular the following fields:
 - Call Reference = X1
 - Call Reference Flag = 1 (target)

NOTE In case the SVC is automatically closed by the network, both sides behave as targets.

12.4.5.3 DN_TransMuxSetup ()

The DN_TransMuxSetup () functionality is obtained through the setup of a new ATM channel. This procedure works as follows:

- 1) the originating peer allocates a new Transmux Association Tag
- 2) the originating peer generates a Q.2931 SETUP message, with the following rules:
 - Call Reference = Y1
 - Call Reference Flag = 0 (originator)
 - Called Party Number = calledAddress
 - Calling Party Number = callingAddress
 - QoS parameter = 0
 - ATM traffic descriptor - forward peak cell rate CLP=0 = as appropriate
 - ATM traffic descriptor - forward peak cell rate CLP=0+1 = as appropriate
 - ATM traffic descriptor - backward peak cell rate CLP=0 = as appropriate
 - ATM traffic descriptor - backward peak cell rate CLP=0+1 = as appropriate
 - AAL parameters - AAL type = AAL5
 - AAL parameters - forward maximum CPCS-SDU size = as appropriate
 - AAL parameters - SSCS type = NULL
 - Broadband bearer capability - bearer class = bcob-x (suggested)
 - Broadband bearer capability - traffic type = as appropriate
 - Broadband bearer capability - timing requirements = as appropriate
 - Broadband bearer capability - susceptibility to clipping = as appropriate
 - Broadband bearer capability - bearer class = user plane connection configuration = point-to-point
 - BHLI - High Layer Information Type = User specific
 - BHLI - octets 1 = DMIF_SelectorByte
 - BHLI - octets 2-6 = NetworkSessionId
 - BHLI - octets 7-8 = Transmux Association Tag

3) the target peer receives the SETUP message with in particular the following fields:

- Call Reference = Y2
- Call Reference Flag = 1 (target)
- Connection Identifier - VP = VPY2
- Connection Identifier - VC = VCY2
- ...
- BHLI - High Layer Information Type = User specific
- BHLI - octets 1 = DMIF_SelectorByte
- BHLI - octets 2-6 = NetworkSessionId
- BHLI - octets 7-8 = Transmux Association Tag

4) the target peer generates the CONNECT message with in particular the following fields:

- Call Reference = Y2
- Call Reference Flag = 0 (originator)

5) the originating peer receives the CONNECT message with in particular the following fields:

- Call Reference = Y1
- Call Reference Flag = 1 (target)
- Connection Identifier - VP = VPY1
- Connection Identifier - VC = VCY1

6) the setup is complete: each side associates its own SVC and Call Reference to the same TAT

NOTE The CONNECT message may be preceeded by a CALL_PROCEEDING message. In this case, at the originating side, the CALL_PROCEEDING and not the CONNECT will carry the Connection Identifier.

12.4.5.4 DN_TransMuxRelease ()

The DN_TransMuxRelease () functionality is obtained through the deletion of the SVC corresponding to a particular TAT. Since the networkSessionId field in the DS messages is not used, the DS_TransMuxRelease messages are not transmitted. This procedure works as follows:

1) the originating peer generates a Q.2931 RELEASE message, with the following rules:

- Call Reference = Y1
- Call Reference Flag = 0 (originator)

2) the target peer receives the RELEASE message with in particular the following fields:

- Call Reference = Y2
- Call Reference Flag = 1 (target)

3) the target peer generates the RELEASE COMPLETE message with in particular the following fields:

- Call Reference = Y2
- Call Reference Flag = 0 (originator)

4) the originating peer receives the RELEASE COMPLETE message with in particular the following fields:

- Call Reference = Y1
- Call Reference Flag = 1 (target)

NOTE In case the SVC is automatically closed by the network, both sides behave as targets.

13 Terminal Capability Matching

Terminal Capability matching is required to check the ability of two peer end-systems to setup connections between them, and select the possible protocol stack supported. This stage is performed at session setup time. Whenever possible, DMIF makes use of the facilities provided by the network. When the existing network facilities cannot be used, DMIF defines its own mechanism (subclause 13.1).

13.1 DMIF Default signalling with Compatibility Descriptors

This stage is used when the native network signalling is not capable to accommodate the DMIF-Network Interface requirements. The originating DMIF compatibility descriptors are carried in the DMIF SessionSetUpRequest and a subset is returned from the target DMIF from which the originating DMIF can choose from. If no such subset exists then the originating terminal will release the session. The Compatibility Descriptor is defined in ISO/IEC 13818-6, clause 6.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14496-6:1999

Annex A (informative)

Overview of DAI and DNI parameters

This Annex provides the glue to link concepts and parameters as defined in subclause 10.3 for the DAI and in subclause 11.2.5 for the DNI.

A.1 Sessions and services

This clause correlates the following parameters:

URL: as defined in subclause 10.3.

serviceName: as defined in subclause 10.3.

serviceSessionId: as defined in subclause 10.3.

networkSessionId: as defined in subclause 11.2.5.

serviceld: as defined in subclause 11.2.5.

An application only identifies a particular service through its **URL**. The **serviceName** is then obtained from the **URL** by stripping out the addressing information. Each **URL** will have a 1 to 1 correspondance with a **serviceSessionId**. The service identified by the **URL** will be further referred to by its **serviceSessionId**.

Inside DMIF, the **serviceSessionId** is mapped 1 to 1 with a tuple <**networkSessionId**, **serviceld**>. This allows several **serviceSessionIds** to share a single **networkSessionId**. Depending on the network type, the **networkSessionId** can be implicit, in the sense that the DMIF Signalling channel established between two DMIF peers uniquely identifies a Network Session: for example a **networkSessionId** could be identified, on a homogeneous IP network, with the tuple <peer1IpAddress, peer1IpPort, peer2IpAddress, peer2IpPort, IpProtocol> identifying the socket providing the DMIF Signalling channel.

A.2 Channels

This clause correlates the following parameters:

channelHandle: as defined in subclause 10.3.

CAT: as defined in subclause 11.2.5.

TAT: as defined in subclause 11.2.5.

An ISO/IEC 14496-1 based application may identify a particular elementary stream through its **ES_ID**, scoped by the Service Session it belongs to. At the DAI the **ES_ID** parameter is made more generic to be ISO/IEC 14496-1 independent, and carried in **uuData**. Each <**serviceSessionId**, **ES_ID**> will have a 1 to 1 correspondance with a **channelHandle**. The stream identified by the <**serviceSessionId**, **ES_ID**> will be further referred to through its **channelHandle**.

Inside DMIF, the **channelHandle** is mapped 1 to 1 with the **CAT**. This allows to decouple a parameter (the **channelHandle**) that has to be unique at the local application / DMIF boundary, from another parameter (the **CAT**) that instead has to be unique in the scope of a Network Session. Moreover, the syntax of the **channelHandle** is a local matter, related to the DAI implementation, while the syntax of the **CAT** relates to the network, irrespectively of

the terminal type. This decoupling also allows to maintain uniformity at the DAI, regardless of the network, the file format or the broadcast/multicast technology being used.

Multiple streams can be multiplexed together on a single network connection (transmux). As a consequence, the concepts of **TAT** and *transmuxHandle* are defined in a similar manner. The latter however is not explicitly mentioned in this specification since it does not affect the DAI nor the network signalling (it is a purely internal entity).

CAT and **TAT** do not affect the DAI, but are useful concepts for the DMIF signalling messages. They convey a unique identifier (scoped by the Network Session) to identify the resources that are used by a particular stream or group of streams (transmux). Depending on the network type, the syntax of **CAT** and **TAT** may vary. **CAT** and **TAT** are useful concepts also for scenarios where no network exists, e.g., for local files. In this case, **CAT** and **TAT** may convey information which is appropriate for a particular file format, and which does not restrict necessarily to MPEG-4 only.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14496-6:1999

Annex B

(informative)

Information flows for DMIF

The following clauses provide information flows for Remote Interactive, Broadcast and Local Storage implementations. These flows show the ability of DAI to hide the delivery technologies. A DMIF terminal can simultaneously operate on Remote Interactive (clause B.1), Broadcast (clause B.2) and Local Storage delivery technologies (clause B.3) through a uniform procedure at the DAI.

The names for primitives and parameters in the following clauses are abbreviated to save space. It should be obvious to the reader how they relate to the primitives and parameters indicated in clause 10 and clause 11. In particular the suffix, Callback, has been dropped. The reader should understand that every DAI primitive initiated by DMIF results in a DAI callback. Also the information presented must be complemented with the specific mappings for each delivery technology.

The flows provided are not exhaustive, they represent frequent cases, and provide hints on how to deal with infrequent scenarios.

B.1 Information flows for Remote Interactive DMIF

The following figures represent information crossing the interfaces, for simplicity only the DNI primitives are shown. These need to be complemented with their mappings to supported network technologies. The central column in the figures thus represents the conceptual flows that occur between two DNI interfaces, without considering the details of how such primitives are mapped into signalling messages and reconstructed at the peer DNI.

The other columns in the figures represent the application, the Delivery layer, and the DAI.

B.1.1 Initiation of a service in a Remote Interactive DMIF

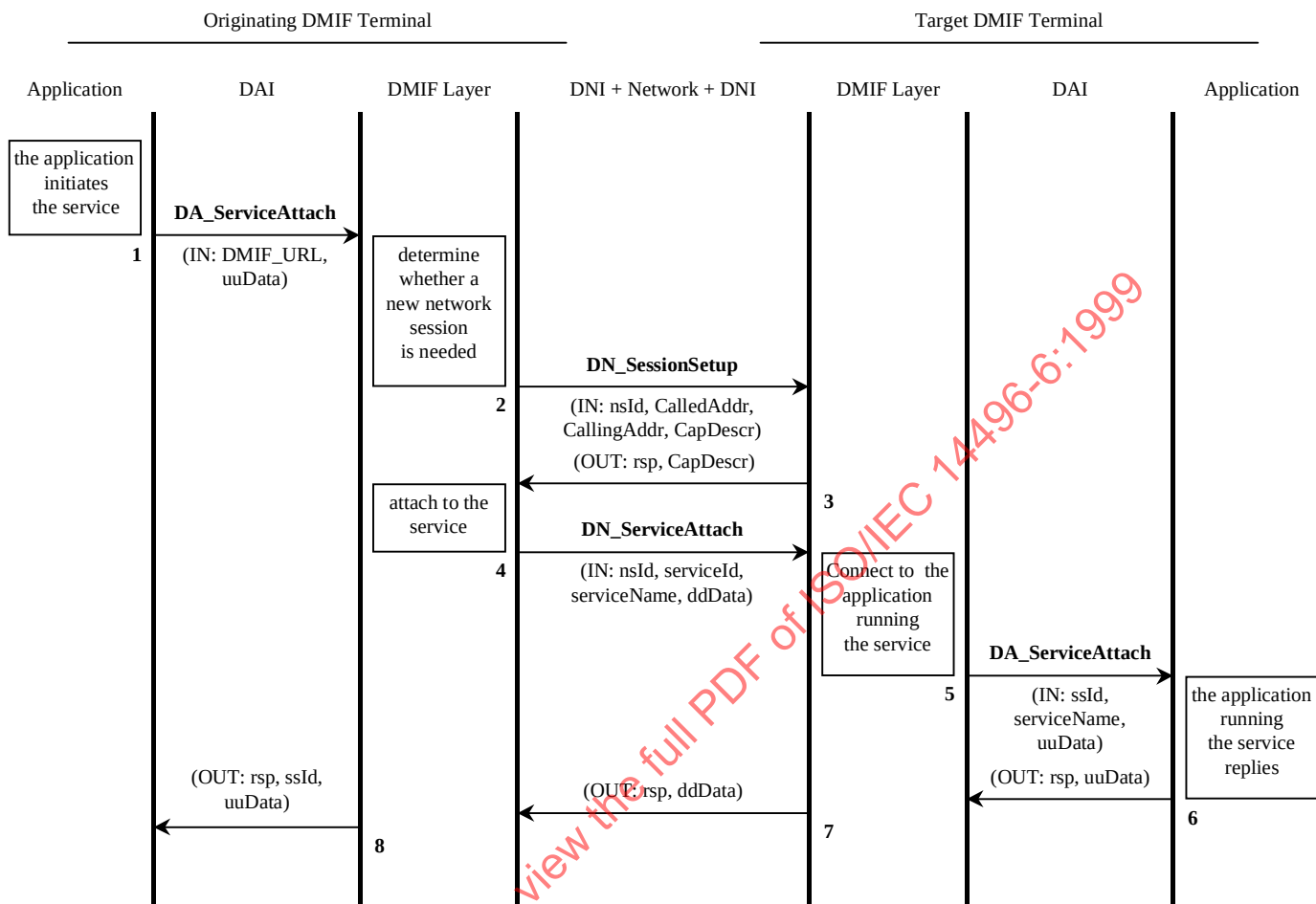


Figure B.1 — Initiation of a service in a Remote Interactive DMIF

Preconditions:

- The Application at the Originating DMIF has acquired a URL from a previous action (e.g., Web browsing, advertisement, specific contracts).
- The Service is registered at the Target DMIF Terminal by means that are outside this specification

Step 1

The Application at the Originating DMIF passes a `DA_ServiceAttach()` indicating the URL and additional `uuData` (e.g., for client credentials).

The URL is parsed and the appropriate DMIF Instance is invoked according to the protocol part of the URL.

Step 2

The Originating DMIF Instance strips the `<serviceName>` part of the URL for later use. It assigns a locally significant unique `serviceSessionId`; based on its own criteria, if there is already a Network Session active with the remote peer, the Originating DMIF Instance might decide to reuse that Network Session (and skip to step 4); otherwise it

shall create a new Network Session by passing through the DNI a DN_SessionSetup() primitive. In this case it assigns a network unique networkSessionId. Moreover Capability (and compatibility) Matching is performed (by means of a Compatibility Descriptor or by other means possibly supported by the Network).

Step 3

The Target DMIF Instance receives DN_SessionSetupCallback() and replies. Both peers now have knowledge of the networkSessionId. The compatibility is verified and the appropriate reply is returned identifying the common set of compatibility in the preferred order of choice.

Step 4

The Originating DMIF Instance assigns a serviceId which is unique in that particular Network Session, and passes the DN_ServiceAttach() through the DNI, indicating the networkSessionId and the serviceId along with the serviceName it has previously stripped from the URL and additional ddData that actually contains the uuData provided by the Originating Application.

Step 5

The Target DMIF Instance receives DN_ServiceAttachCallback() determines the Executive managing the services, assigns a locally significant unique serviceSessionId and then sends a DA_ServiceAttachCallback() to it, containing the serviceSessionId along with the serviceName and the uuData sent by the Originating Application. The mechanism used at the Target DMIF to identify the Application Executive running the service and to deliver the message to it is out of the scope of this specification.

The Target DMIF Instance maintains a table associating the locally meaningful <serviceSessionId> and the network wide meaningful tuple <networkSessionId, serviceId>

Step 6

The Target Application interprets the uuData and potentially performs tests on client credentials. Then it replies with uuData (which in the case of MPEG-4 contains the Initial OD) and a response code.

Step 7

The Target DMIF Instance replies to the DN_ServiceAttachCallback() through the DNI, providing a response code and ddData that contains the uuData provided by the Target Application.

Step 8

The Originating DMIF Instance uses the locally significant unique serviceSessionId and then replies to the DA_ServiceAttach() with the serviceSessionId, a response code and the uuData originally set by the Target Application.

The Originating DMIF Instance maintains a table associating the locally meaningful <serviceSessionId> and the network wide meaningful tuple <networkSessionId, serviceId>

B.1.2 Addition of Channels in a Remote Interactive DMIF

B.1.2.1 Addition of Channels by Target DMIF

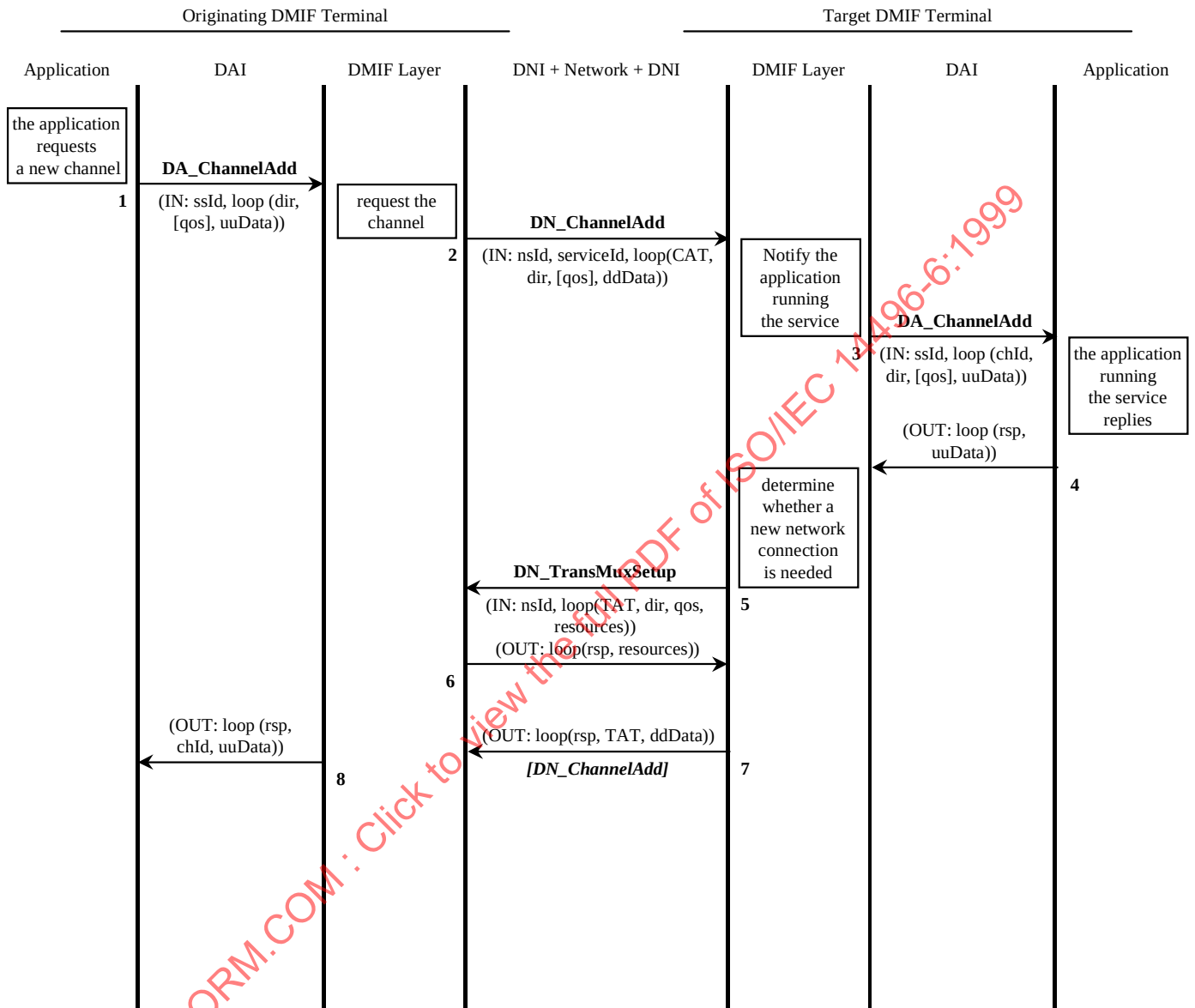


Figure B.2 — Addition of Channels in a Remote Interactive DMIF (by Target DMIF)

Preconditions:

- The service between the Originating and Target applications has been initiated successfully.
- The location of the source of the stream is available from previous interaction.

Step 1

The Application at the Originating DMIF passes a `DA_ChannelAdd()` indicating the channels it requires. The primitive contains the `serviceSessionId` for which the Channels are requested. Each channel is characterized by the `QoS` parameter (optional), the `direction` parameter and by `uuData`.

Step 2

The Originating DMIF Instance assigns a `channelAssociationTag` (CAT) for each requested channel. It then forwards the request to the peer by passing the `DN_ChannelAdd()` through the DNI, containing the network-wide unique tuple `<networkSessionId, serviceId>` corresponding to the `serviceSessionId`, and for each requested channel its CAT, `direction`, (optional) `qosDescriptor` and associated `ddData` (which conveys the original `uuData`).

Step 3

The Target DMIF Instance receives the `DN_ChannelAddCallback()` from the DNI; assigns a locally unique `channelHandle` for each requested channel and then issues a `DA_ChannelAddCallback()` to the Target Application, containing the locally unique `serviceSessionId` corresponding to the tuple `<networkSessionId, serviceId>`, and for each requested channel its locally unique `channelHandle`, the `QoS` parameter (optional), the `direction` parameter and associated `uuData` (which conveys the original `uuData`). At this point the Target DMIF Instance is able to associate the locally unique `channelHandle` to the end-to-end significant, `networkSession` unique CAT.

Step 4

The Target Application interprets the `uuData` to determine what stream is actually being requested (e.g., in the case of an ISO/IEC 14496-1 based application `uuData` may convey the `ES_ID`), and checks the availability of such stream. It then replies with a response code for each requested channel, along with (possibly) `uuData`.

Step 5

The Target DMIF Instance inspects the media stream `QoS` metric and values of its parameters, and decides whether existing network resources are sufficient for carrying the new Channel(s). If additional resources are needed, it starts network signalling to setup new connection(s) (TransMux Channel(s)) in the same Network Session, otherwise it skips to step 7. For each additional TransMux Channel needed it assigns an end-to-end significant `networkSession` unique `transmuxAssociationTag` (TAT); it passes a `DN_TransmuxSetup()` through the DNI, indicating the `networkSessionId`, and for each requested TransMux Channel indicates the `QoS`, the `direction` parameter, the TAT and the TransMux nested resource descriptors.

If required this step may be followed by `DN_TransmuxConfig()` in order for the Target DMIF to configure e.g., a FlexMux in MuxCode mode.

Step 6

The Originating DMIF Instance receives the `DN_TransmuxSetupCallback()` from the DNI, and is therefore able to associate a particular TAT to particular network resources and to a particular Network Session. It then replies through the DNI providing a response code. Note that the TAT is not necessarily bound to a specific `serviceId`, just to a specific `networkSessionId`. However it is reasonable that further restrictions be applied, such that TAT cannot be shared among different Service Sessions.

Step 7

The Target DMIF Instance replies to the original `DN_ChannelAddCallback()` providing for each channel TAT and further `ddData` that characterize it, along with a response code. In particular `ddData` would contain in this case further information on how a particular channel is flexmultiplexed in the TAT, that is in the case of MPEG-4 FlexMux, it provides the FlexMux Channel Number. At this point the Target DMIF Instance is able to associate the CAT to `networkSessionId`, `serviceId`, TAT and further flexmultiplexing configuration. `DdData` may also contain `uuData` returned by the Target Application.

Step 8

The Originating DMIF Instance receives the reply to the DN_ChannelAdd(), assigns a locally unique channelHandle for each requested channel, and replies to the original DA_ChannelAdd() by providing for each requested channel its channelHandle, uuData and a response code. At this point the Originating DMIF Instance is able to associate the locally unique channelHandle to the end-to-end significant, networkSession unique CAT and to associate the CAT to networkSessionId, servidId, TAT and further flexmultiplexing configuration.

Post channel establishment:

- The Receiving Application may request to start getting the data by issuing a DA_UserCommand() specifying the appropriate command in the uuData field.
- The Receiving DMIF Instance forwards the request to the peer by passing the DN_UserCommand() through the DNI, containing the networkSessionId, servidId and ddData that actually contains the uuData provided by the Receiving Application.
- The Sending DMIF Instance forwards the request to the Sending Application by issuing the DN_UserCommandCallback() specifying the appropriate command in the uuData field.
- The Sending Application inspects the uuData, determines the command, and in this case begins to send the data.
- The Elementary Streams are now received by the Receiving Application.

NOTE 1 DA_UserCommand() may be used for purposes other than stream control, and stream control may be performed by means other than DA_UserCommand()

NOTE 2 The Receiving Application is a euphemism for the decoder side and the Sender Application is euphemism for the source sending the encoded stream. Either an Originating or Target DMIF terminal may hold either or both receiving and sending applications.

B.1.2.2 Addition of Channels by Originating DMIF

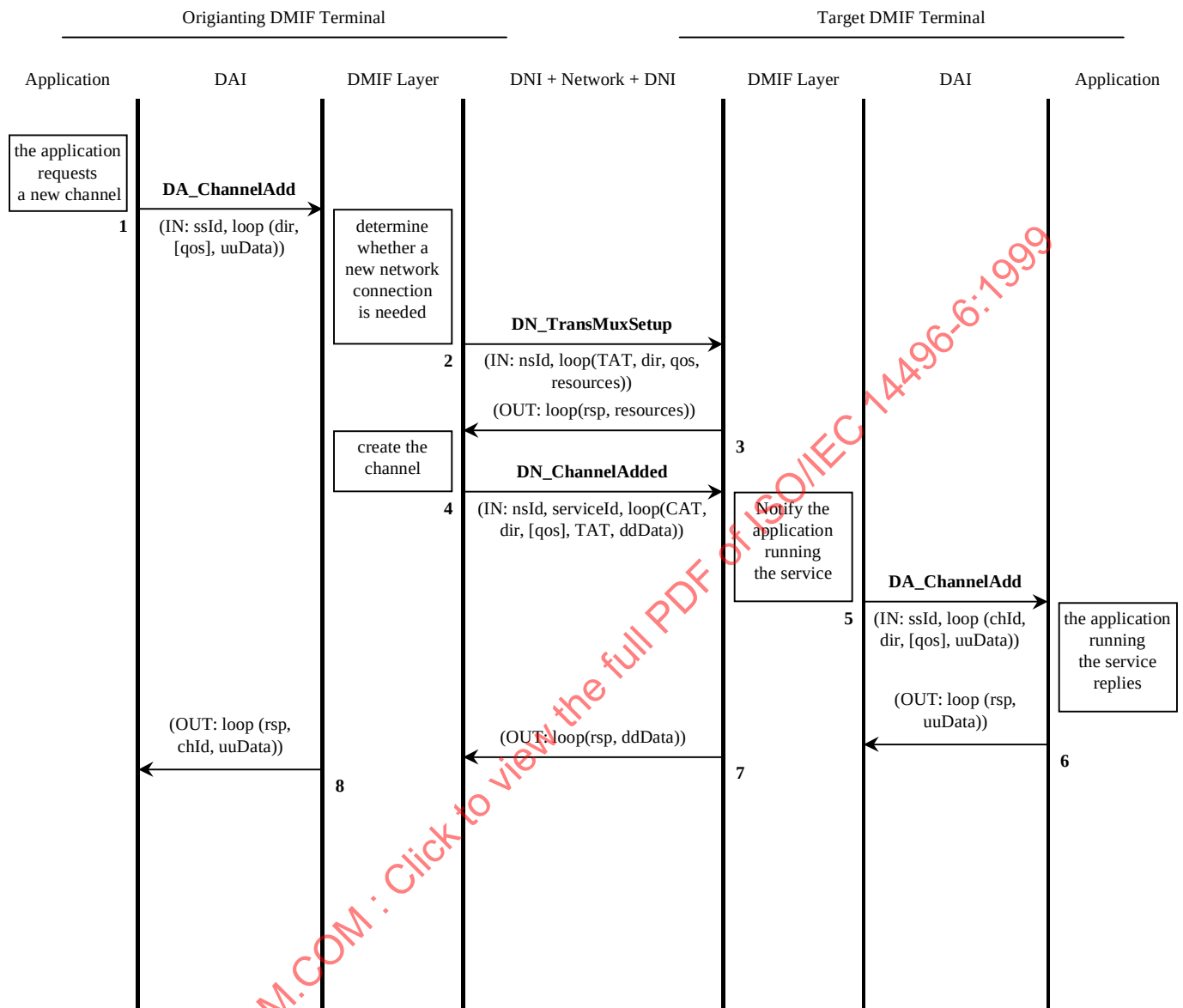


Figure B.3 — Addition of Channels in a Remote Interactive DMIF (by Originating DMIF)

Preconditions:

- The service between the Originating and Target applications has been initiated successfully.
- The location of the source of the stream is available from previous interaction.

Step 1

The Application at the Originating DMIF passes a `DA_ChannelAdd()` indicating the channels it requires. The primitive contains the `serviceSessionId` for which the Channels are requested. Each channel is characterized by the QoS parameter (optional), the direction parameter and by `uuData`.

Step 2

The Originating DMIF Instance inspects the media stream QoS metric and values of its parameters, and decides whether existing network resources are sufficient for carrying the new Channel(s). If additional resources are needed, it starts network signalling to setup new connection(s) (TransMux Channel(s)) in the same Network Session, otherwise it skips to step 4. For each additional TransMux Channel needed it assigns an end-to-end significant, networkSession unique transmuxAssociationTag (TAT); it passes a DN_TransmuxSetup() through the DNI, indicating the networkSessionId, and for each requested TransMux Channel indicates the QoS, the direction parameter, the TAT and the TransMux nested resource descriptors.

Step 3

The Target DMIF Instance receives the DN_TransmuxSetupCallback() from the DNI, and is therefore able to associate a particular TAT to particular network resources and to a particular Network Session. It then replies through the DNI providing a response code. Note that the TAT is not necessarily bound to a specific serviceId, just to a specific networkSessionId. However it is reasonable that further restrictions be applied, such that TAT cannot be shared among different Service Sessions.

If required this step may be followed by DN_TransmuxConfig() in order for the Target DMIF to configure e.g., a FlexMux in MuxCode mode.

Step 4

The Originating DMIF Instance assigns a channelAssociationTag (CAT) for each requested channel. It then forwards the request to the peer by passing the DN_ChannelAdded() through the DNI, containing the network-wide unique tuple <networkSessionId, serviceId> corresponding to the serviceSessionId, and for each requested channel its CAT, TAT, direction and associated ddData (which conveys the original uuData as well as further information on the nested TransMux stacks and how a particular channel is flexmultiplexed in the TAT, that is in the case of MPEG-4 FlexMux, the FlexMux Channel Number). At this point the Originating DMIF Instance is able to associate the CAT to networkSessionId, serviceId, TAT and further flexmultiplexing configuration.

Step 5

The Target DMIF Instance receives the DN_ChannelAddedCallback() from the DNI; assigns a locally unique channelHandle for each established channel and then issues a DA_ChannelAddCallback() to the Target Application, containing the locally unique serviceSessionId corresponding to the tuple <networkSessionId, serviceId>, and for each established channel its locally unique channelHandle, the QoS parameter (optional), the direction parameter and associated uuData (which conveys the original uuData). At this point the Target DMIF Instance is able to associate the locally unique channelHandle to the end-to-end significant, networkSession unique CAT, and to associate the CAT to networkSessionId, serviceId, TAT and further flexmultiplexing configuration.

Step 6

The Target Application interprets the uuData to determine what stream is actually being requested (e.g., in the case of an ISO/IEC 14496-1 based application uuData may convey the ES_ID), and checks the availability of such stream. It then replies with a response code for each requested channel, along with (possibly) uuData.

Step 7

The Target DMIF Instance replies to the original DN_ChannelAddedCallback() providing for each channel further ddData (ddData may contain uuData returned by the Target Application), along with a response code.

Step 8

The Originating DMIF Instance receives the reply to the DN_ChannelAdded(), assigns a locally unique channelHandle for each requested channel, and replies to the original DA_ChannelAdd() by providing for each requested channel its channelHandle, uuData and a response code. At this point the Originating DMIF Instance is able to associate the locally unique channelHandle to the end-to-end significant, networkSession unique CAT.

Post channel establishment:

- The Receiving Application may request to start getting the data by issuing a `DA_UserCommand()` specifying the appropriate command in the `uuData` field.
- The Receiving DMIF Instance forwards the request to the peer by passing the `DN_UserCommand()` through the DNI, containing the `networkSessionId`, `serviceId` and `ddData` that actually contains the `uuData` provided by the Receiving Application.
- The Sending DMIF Instance forwards the request to the Sending Application by issuing the `DN_UserCommandCallback()` specifying the appropriate command in the `uuData` field.
- The Sending Application inspects the `uuData`, determines the command, and in this case begins to send the data.
- The Elementary Streams are now received by the Receiving Application.

NOTE 1 `DA_UserCommand()` may be used for purposes other than stream control, and stream control may be performed by means other than `DA_UserCommand()`

NOTE 2 The Receiving Application is a euphemism for the decoder side and the Sender Application is euphemism for the source sending the encoded stream. Either an Originating or Target DMIF terminal may hold either or both receiving and sending applications.

B.1.3 Deletion of Channels in a Remote Interactive DMIF

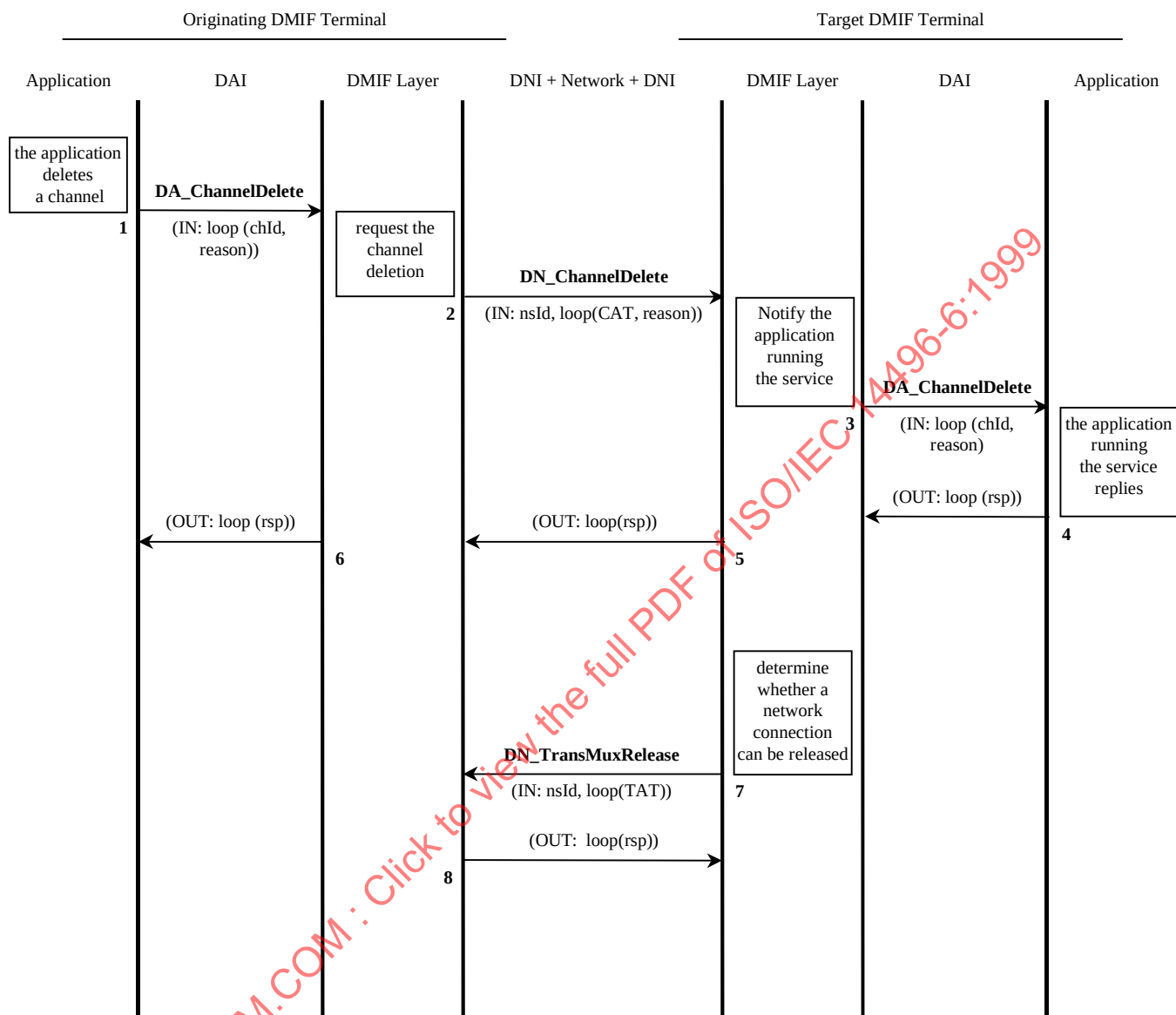


Figure B.4 — Deletion of Channels in a Remote Interactive DMIF (controlled by the Originating DMIF peer)

Preconditions:

- The service has been established between the Originating and Target applications.
- At least a channel has been successfully added.
- A stream being carried on a channel is no longer required.

Step 1

The Application at the Originating DMIF passes a `DA_ChannelDelete()` indicating the channels it wants to delete. The primitive contains the `channelHandle(s)` along with reason codes.

Step 2

The DMIF Instance stops the actual delivery of data on the indicated Channel(s).

The Originating DMIF Instance forwards the request to the peer by passing the DN_ChannelDelete() through the DNI, containing the network-wide unique networkSessionId, and for each requested channel its CAT and the reason code.

Step 3

The Target DMIF Instance receives the DN_ChannelDeleteCallback() from the DNI; and issues a DA_ChannelDeleteCallback() to the Target Application, containing for each requested channel its locally unique channelHandle and the reason code.

Step 4

The Target Application stops the actual delivery of data on the indicated Channel(s), and replies with response codes. At this point channelHandle(s) are invalidated at the Target Application.

Step 5

The Target DMIF Instance replies to the original DN_ChannelDeleteCallback() providing for each channel a response code. At this point channelHandle(s) and CAT(s) are invalidated at the Target DMIF Instance.

Step 6

The Originating DMIF Instance replies to the original DA_ChannelDelete() by providing for each channel a response code. At this point channelHandle(s) and CAT(s) are invalidated at the Originating DMIF Instance.

The Originating Application receives the reply and invalidates the channelHandle(s).

NOTE The Originating or the Target DMIF depending on which scenario was used to add channels can carry out the following steps 7 and 8.

Step 7

The Originating (Target) DMIF Instance determines that some network resource is not needed anymore, and might decide to release them. In that case it starts network signalling to release connection(s) (TransMux Channel(s)) in the same Network Session. It passes a DN_TransmuxRelease() through the DNI, indicating the networkSessionId and the TAT of the TransMux Channels to be released.

Step 8

The Target (Originating) DMIF Instance receives the DN_TransmuxReleaseCallback() from the DNI, and replies through the DNI providing a response code. TATs are invalidated at both sides.

B.1.4 Termination of a Service in a Remote Interactive DMIF

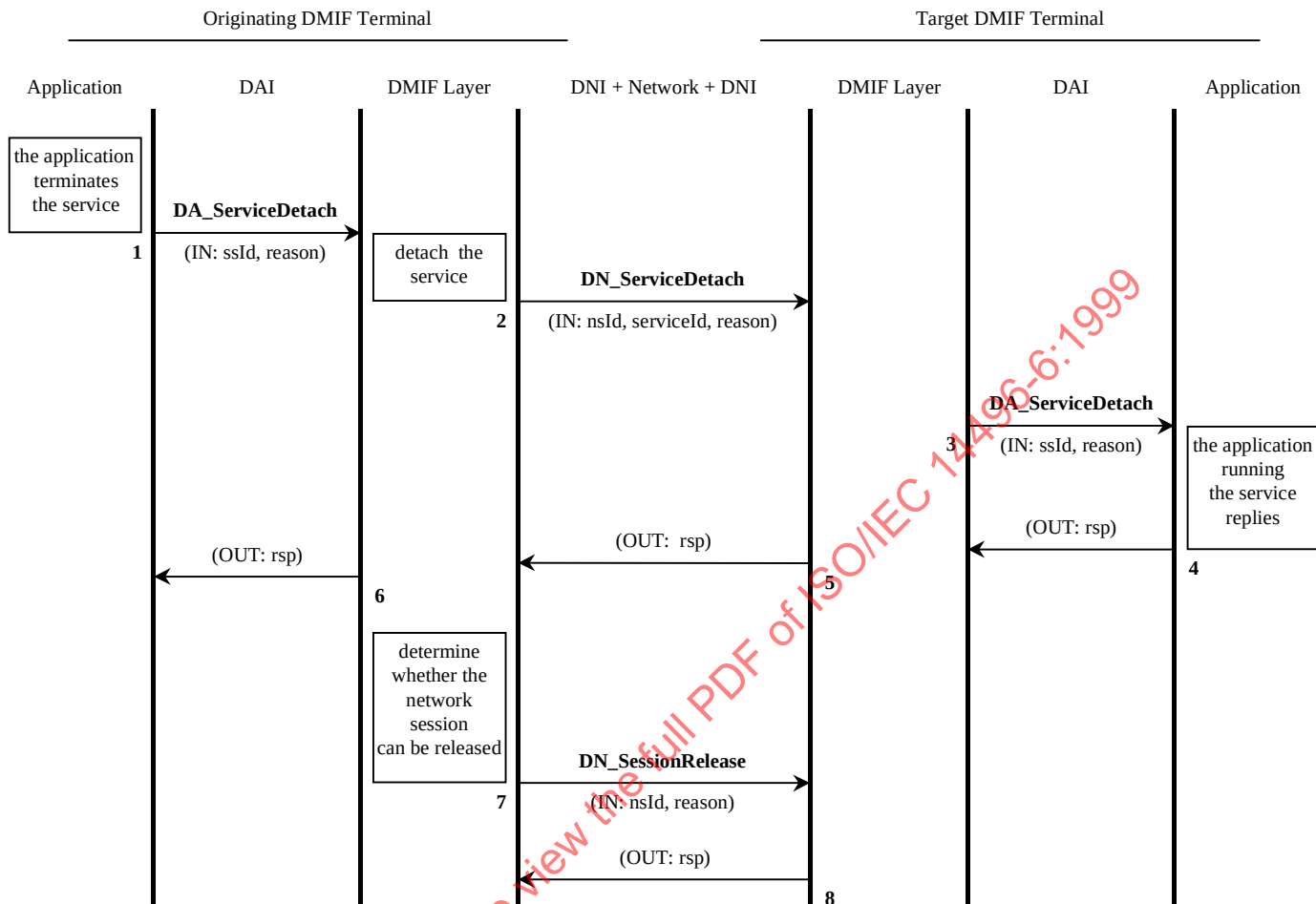


Figure B.5 — Termination of a service in a Remote Interactive DMIF

Preconditions:

- The service has been established between the Originating and Target applications.
- The Service Session is no longer required.

Step 1

The Application at the Originating DMIF passes a `DA_ServiceDetach()` indicating the service it wants to terminate. The primitive contains the `serviceSessionId` along with a reason code.

Step 2

The Originating DMIF Instance passes a `DN_ServiceDetach()` through the DNI indicating the service it wants to terminate (which is now identified by the tuple `<networkSessionId, serviceId>` along with a reason code.

Step 3

The Target DMIF Instance receives the DN_ServiceDetachCallback() from the DNI; and passes a DA_ServiceDetachCallback() to the Target Application indicating the service that must be terminated (which is now identified by the locally meaningful serviceSessionId) along with a reason code.

Step 4

The Target Application stops the provision of the service, and frees all resources used for it. It then replies to the DA_ServiceDetachCallback() providing a response code. At this point the locally meaningful serviceSessionId is no more valid.

Step 5

The Target DMIF Instance replies to the DN_ServiceDetachCallback() through the DNI along with the response code. At this point the Network Session unique serviceId is no more valid.

Step 6

The Originating DMIF Instance replies to the DA_ServiceDetach() along with the response code. At this point the locally meaningful serviceSessionId is no more valid.

NOTE The Originating or the Target DMIF irrespective of which one initiated the service detach can carry out the following steps 7 and 8.

Step 7

The Originating (Target) DMIF Instance might recognize that the Network Session need not be active anymore (this was the last service present on that Network Session). It might therefore decide to release the Network Session, and passes a DN_SessionRelease() through the DNI.

Step 8

The Target (Originating) DMIF Instance receives a DN_SessionReleaseCallback() and replies to it through the DNI. At this point the network wide unique networkSessionId is no longer valid.

B.2 Information Flows for Broadcast DMIF

The following figures represent the semantic information crossing the borders of a DMIF implementation for a Broadcast scenario. No details are provided on the internal exchanges between modules in DMIF.

The first two columns in the figures represent the application and the DAI, with the same information and formalism as in clause B.1; the third column represents a generic DMIF implementation, including all the internal conceptual modules according to the DMIF communication architecture of Figure 3, that is Originating DMIF Instance, Target DMIF Instance and Target Application. The right side of the figures represents Broadcast Network and shows the data entering the DMIF implementation. Note that such data flows purely provide a hint of what could be exchanged, while the real flows could differ significantly: therefore, they are not tagged with any number, and their description is purely textual.

B.2.1 Initiation of a service in a Broadcast DMIF

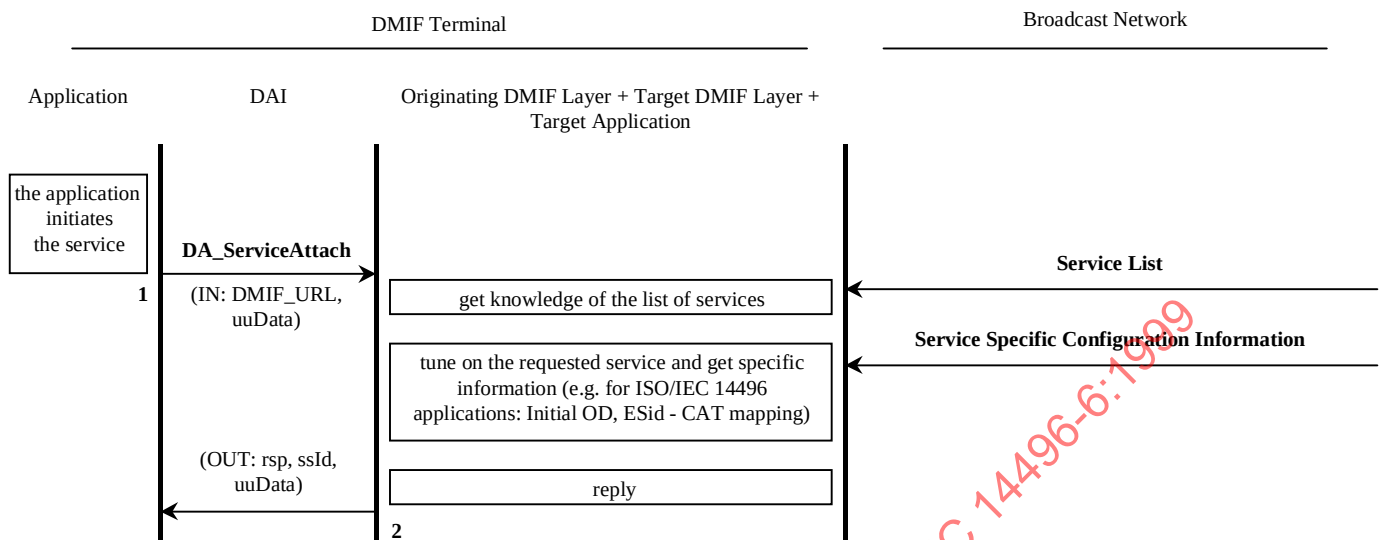


Figure B.6 — Initiation of a service in a Broadcast DMIF

Preconditions:

- The Application at the Originating DMIF has acquired an URL from a previous action (e.g.; Web browsing, advertisement, specific contracts).

Step 1

The Application at the Originating DMIF passes a `DA_ServiceAttach()` indicating the URL and additional `uuData` (e.g., for client credentials).

The URL is parsed and the appropriate DMIF Instance is invoked according to the protocol part of the URL.

The Originating DMIF Instance assigns a locally significant unique `serviceSessionId`. The Target Application module interprets the whole URL including the `serviceName`, and checks whether the requested service is available. It possibly gets additional service specific data from the broadcast network (e.g., in the case of an ISO/IEC 14496-1 based application, the Initial OD and possibly the table mapping the `ES_IDs` with the actual channels delivering them). Moreover, the Target Application module potentially interprets the `uuData` field for client verification.

Additional checks are performed through compatibility matching procedures: the mechanism adopted depends on the Broadcast Network.

Step 2

After the successful conclusion of the compatibility match and the correct tuning to the requested service, the Originating DMIF Instance replies to the `DA_ServiceAttach()` along with the locally significant `serviceSessionId` and possibly `uuData`. In the case of ISO/IEC 14496 applications the `uuData` would contain the `InitialOD`.

B.2.2 Addition of Channels in a Broadcast DMIF

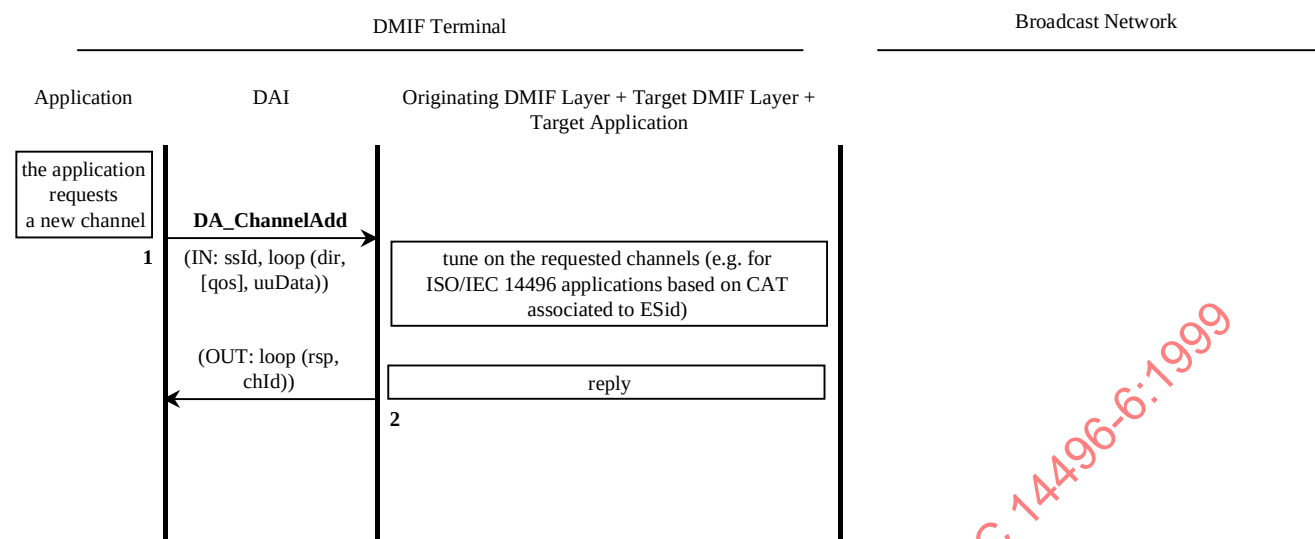


Figure B.7 — Addition of Channels in a Broadcast DMIF

Preconditions:

- The service between the Originating and Target applications has been initiated succesfully.
- The location of the source of the stream is available from previous interaction.

Step 1

The Application at the Originating DMIF passes a DA_ChannelAdd() indicating the channels it requires. The primitive contains the serviceSessionId for which the Channels are requested. Each channel is characterized by a QoS parameter, by a direction parameter (usually DOWNSTREAM; UPSTREAM for back-channels, that will be terminated at the Target Application module), and uuData. The Target Application module interprets the uuData to determine what stream is actually being requested (e.g., in the case of an ISO/IEC 14496-1 based application uuData may convey the ES_ID), and checks the availability of this stream.

Step 2

After the successful conclusion of the above check, the Originating DMIF Instance replies to the DA_ChannelAdd() along with the locally significant channelHandle(s).

Post channel establishment:

- The Originating Application may request to start getting the data by issuing a DA_UserCommand() specifying the appropriate command in the uuData field.
- The Target Application portion of the DMIF Instance inspects the uuData, determines the command, and in this case begins to send the data.
- The Elementary Streams are now received by the Originating Application.

NOTE DA_UserCommand() may be used for purposes other than stream control, and stream control may be performed by means other than DA_UserCommand()

B.2.3 Deletion of Channels in a Broadcast DMIF

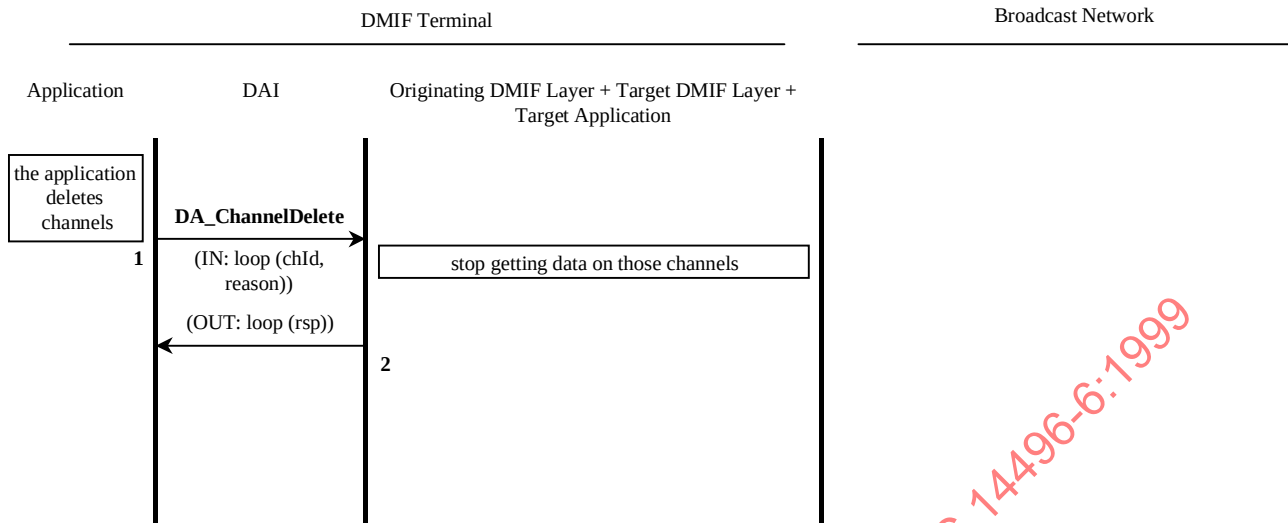


Figure B.8 — Deletion of Channels in a Broadcast DMIF

Preconditions:

- The service has been established between the Originating and Target applications.
- At least a channel has been successfully added.
- A stream being carried on a channel is no longer required.

Step 1

The Application at the Originating DMIF passes a `DA_ChannelDelete()` indicating the channels it wants to delete. The primitive contains the `channelHandle(s)` along with reason codes.

The DMIF Instance stops the actual delivery of data on the indicated Channel(s).

Step 2

The Originating DMIF Instance replies to the `DA_ChannelDelete()` along with response codes. At this point the `channelHandle(s)` are no longer valid.

B.2.4 Termination of a service in a Broadcast DMIF

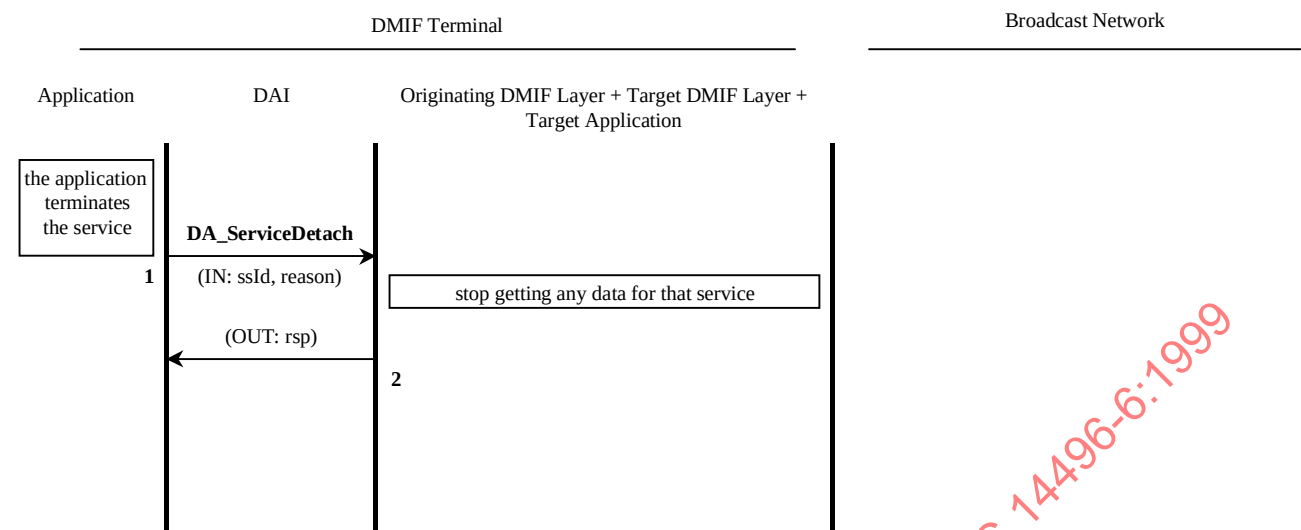


Figure B.9 — Termination of a service in a Broadcast DMIF

Preconditions:

- The service has been established between the Originating and Target applications.
- The Service Session is no longer required.

Step 1

The Application at the Originating DMIF passes a DA_ServiceDetach() indicating the service it wants to terminate. The primitive contains the serviceSessionId along with a reason code.

The DMIF Instance stops the provision of the service, and frees all resources used for that.

Step 2

The Originating DMIF Instance replies to the DA_ServiceDetach() along with a response code. At this point the serviceSessionId is no longer valid.

B.3 Information Flows for Local Storage DMIF

The following figures represent the semantic information crossing the borders of a DMIF implementation for a Local Storage scenario. No details are provided on the internal exchanges between modules in DMIF.

The first two columns in the figures represent the application and the DAI, with the same information and formalism as in clause B.1; the third column represents a generic DMIF implementation, including all the internal conceptual modules according to the DMIF communication architecture of Figure 3, that is Originating DMIF Instance, Target DMIF Instance and Target Application. The right side of the figures represents the File Systems and shows the basic commands emitted by the DMIF implementation and the data entering the DMIF implementation. Note that such commands and data flows purely provide a hint of what could be exchanged, while the real flows could differ significantly; therefore, they are not tagged with any number, and their description is purely textual.

B.3.1 Initiation of a service in a Local Storage DMIF

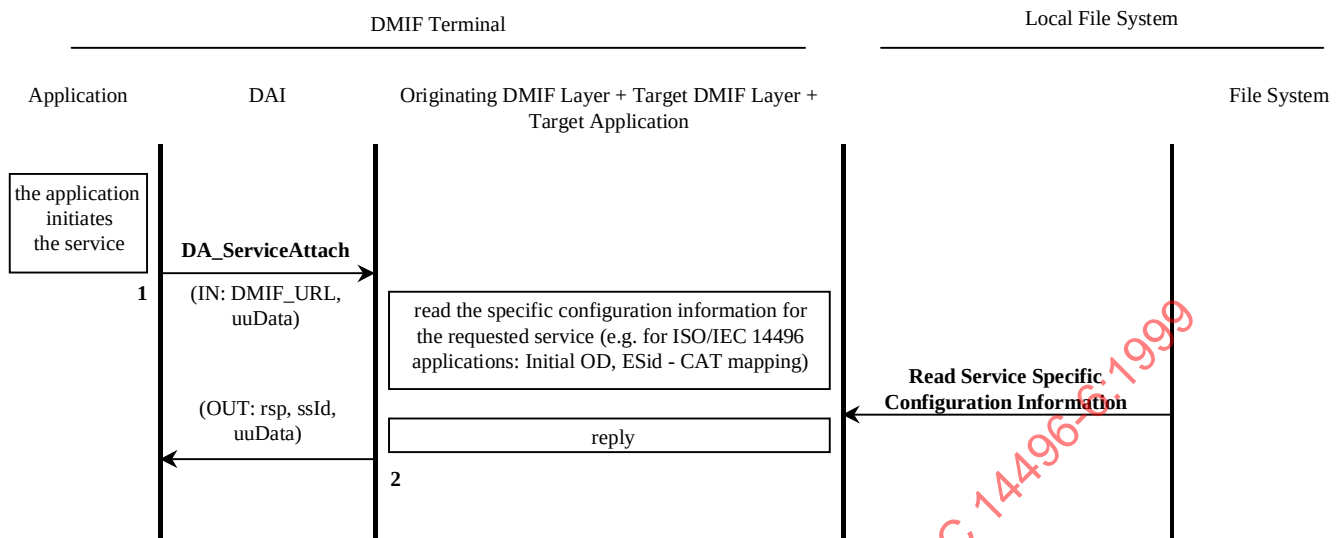


Figure B.10 — Initiation of a service in a Local Storage DMIF

Preconditions:

- The Application at the Originating DMIF has acquired an URL from a previous action (e.g., Web browsing, advertisement, specific contracts).

Step 1

The Application at the Originating DMIF passes a `DA_ServiceAttach()` indicating the URL and additional `uuData` (e.g., for client credentials).

The URL is parsed and the appropriate DMIF Instance is invoked according to the protocol part of the URL.

The Originating DMIF Instance assigns a locally significant unique `serviceSessionId`. The Target Application module interprets the whole URL including the `serviceName`, and checks whether the requested service is available. It possibly gets additional service specific data from the file system (e.g., in the case of ISO/IEC 14496-1 based applications, the Initial OD and possibly the table mapping the `ES_IDs` with the actual channels delivering them). Moreover, the Target Application module possibly interprets the `uuData` field.

Additional checks are performed through compatibility matching procedures: the mechanism adopted depends on the file system.

Step 2

After the successful conclusion of the compatibility match and the correct identification of the requested service, the Originating DMIF Instance replies to the `DA_ServiceAttach()` along with the locally significant `serviceSessionId` and possibly `uuData`. In the case of ISO/IEC 14496 applications the `uuData` would contain the `InitialOD`.

B.3.2 Addition of Channels in a Local Storage DMIF

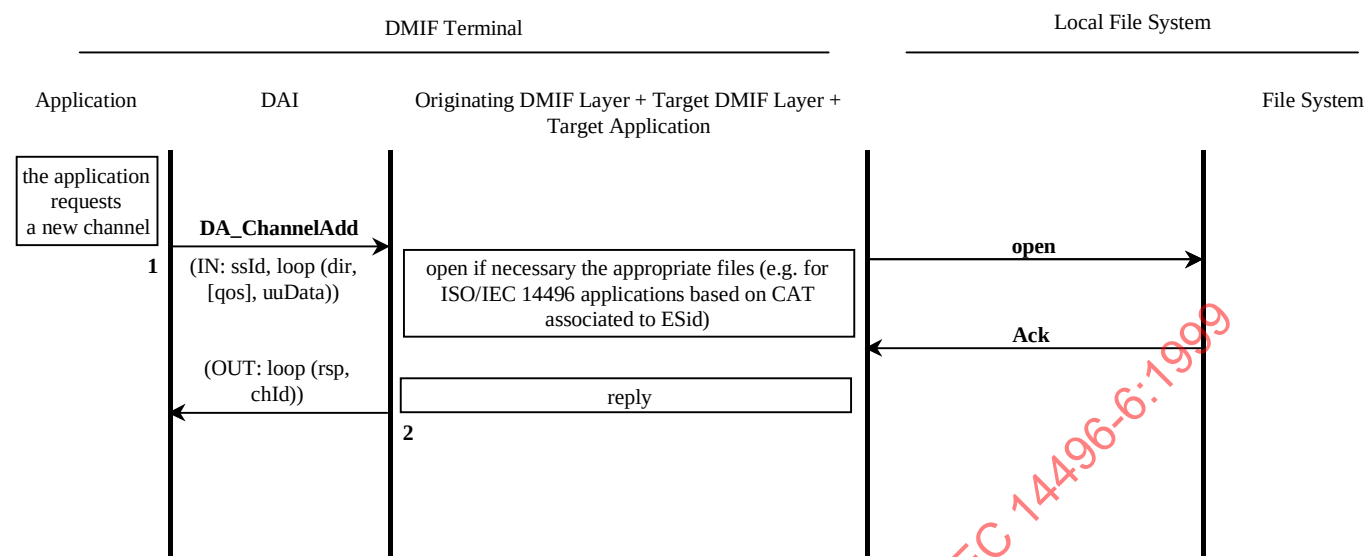


Figure B.11 — Addition of Channels in a Local Storage DMIF

Preconditions:

- The service between the Originating and Target applications has been initiated succesfully.
- The location of the source of the stream is available from previous interaction.

Step 1

The Application at the Originating DMIF passes a DA_ChannelAdd() indicating the channels it requires. The primitive contains the serviceSessionId for which the Channels are requested. Each channel is characterised by a QoS parameter, by a direction parameter (usually DOWNSTREAM; UPSTREAM for back-channels, that will be terminated at the Target Application module), and uuData. The Target Application module interprets the uuData to determine what stream is actually being requested (e.g., in the case of an ISO/IEC 14496-1 based application uuData may convey the ES_ID), and checks the availability of such stream.

Step 2

After the successful conclusion of the above check, the Originating DMIF Instance replies to the DA_ChannelAdd() along with the locally significant channelHandle(s).

Post channel establishment:

- The Originating Application may request to start getting the data by issuing a DA_UserCommand() specifying the appropriate command in the uuData field.
- The Target Application portion of the DMIF Instance inspects the uuData, determines the command, and in this case begins to send the data.
- The Elementary Streams are now received by the Originating Application.

NOTE DA_UserCommand() may be used for purposes other than stream control, and stream control may be performed by means other than DA_UserCommand()

B.3.3 Deletion of Channels in a Local Storage DMIF

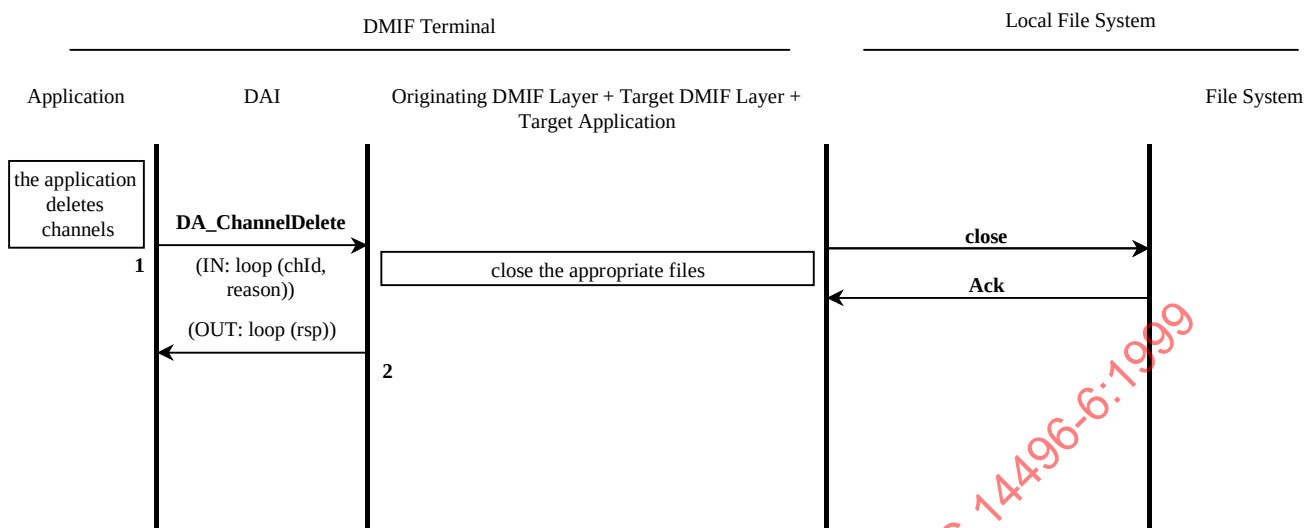


Figure B.12 — Deletion of Channels in a Local Storage DMIF

Preconditions:

- The service has been established between the Originating and Target applications.
- At least a channel has been successfully added.
- A stream being carried on a channel is no longer required.

Step 1

The Application at the Originating DMIF passes a `DA_ChannelDelete()` indicating the channels it wants to delete. The primitive contains the `channelHandle(s)` along with reason codes.

The DMIF Instance stops the actual delivery of data on the indicated Channel(s).

Step 2

The Originating DMIF Instance replies to the `DA_ChannelDelete()` along with response codes. At this point the `channelHandle(s)` are no more valid.

B.3.4 Termination of a service in a Local Storage DMIF

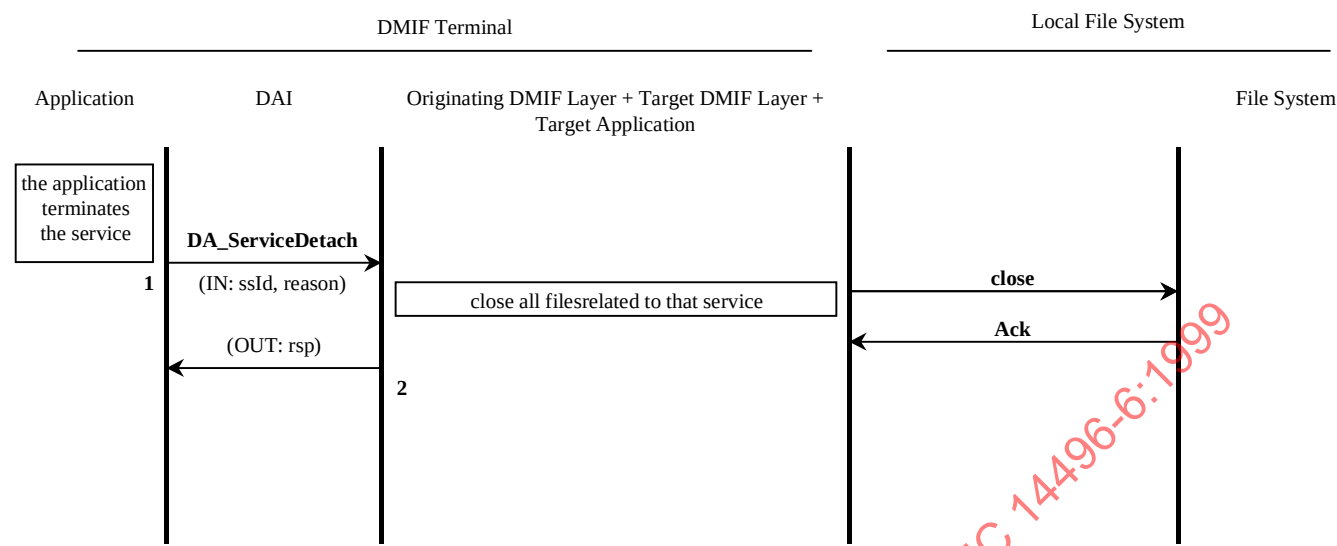


Figure B.13 — Termination of a service in a Local Storage DMIF

Preconditions:

- The service has been established between the Originating and Target applications.
- The Service Session is no longer required.

Step 1

The Application at the Originating DMIF passes a DA_ServiceDetach() indicating the service it wants to terminate. The primitive contains the serviceSessionId along with a reason code.

The DMIF Instance stops the provision of the service, and frees all resources used for that.

Step 2

The Originating DMIF Instance replies to the DA_ServiceDetach() along with a response code. At this point the serviceSessionId is no more valid.

Annex C (informative)

Use of URLs in DMIF

C.1 Introduction

This Annex provides a list of URL schemes that are allowed in DMIF (see clause C.3); moreover it provides a specification for those schemes that are new, i.e., not yet registered with the Internet Corporation for Assigned Names and Numbers (ICANN) (see clause C.4).

C.2 Generic concepts

A URL contains the name of the scheme being used (<scheme>) followed by a colon and then a string (the <scheme-specific-part>) whose interpretation depends on the scheme.

<scheme>:<scheme-specific-part>

In DMIF URLs always represent a service location, and can only be absolute.

The local DMIF parses the <scheme> part of the URL in order to identify and activate the appropriate DMIF Instance. The local DMIF Instance in turn parses a portion of the <scheme-specific-part> in order to establish a session with the designated remote (or emulated remote in the case of local storage and broadcast scenarios) DMIF; the remote (or emulated remote) DMIF parses a further portion of the <scheme-specific-part> to identify the remote application program; the remote (or emulated remote) application program parses the remaining portion of the <scheme-specific-part> to identify and activate the desired service.

C.3 URL schemes allowed in DMIF

The following are the URL schemes currently allowed in this part of ISO/IEC 14496:

- "dudp:"
- "dtcp:"
- "datm:"

C.4 New URL schemes

Warning:

The URL schemes here specified are new schemes, which are not yet officially registered at the time when this Annex is written. URLs for experimental schemes may be used by mutual agreement between parties. Scheme names starting with the characters "x-" are reserved for experimental purposes. In time the URL schemes here specified shall be registered with the Internet Corporation for Assigned Names and Numbers (ICANN) that maintains the registry of URL schemes. As long as the URL schemes here specified are not officially registered, characters "x-" have to be added in front of it.

C.4.1 URL scheme for DMIF signalling over UDP/IP

The URL for default DMIF signalling over UDP/IP follows the generic syntax for new URL schemes defined in RFC1738. Usually the URLs that involve the direct use of an IP-based protocol to a specified target on the Internet use a common Internet specific syntax:

```
//<user>:<password>@<host>:<port>/<url-path>
```

The URL for default DMIF signalling over UDP/IP consists of the following:

x-dudp://<user>:<password>@<target dmif>:<dmif port>/<service-entity-path>

The <host> is the target DMIF and hence for the "x-dudp:" scheme it will be named <target dmif>.

The <port> indicates the UDP/IP socket port number over which the DMIF signalling messages will be delivered and hence for the "x-dudp:" scheme it will be named <dmif port>.

The <url-path> is the path to access a dmif service entity and hence for the "x-dudp:" scheme it will be named <service-entity-path>.

C.4.2 URL scheme for DMIF signalling over TCP/IP

The URL for default DMIF signalling over TCP/IP follows the generic syntax for new URL schemes defined in RFC1738. Usually the URLs that involve the direct use of an IP-based protocol to a specified target on the Internet use a common Internet specific syntax:

```
//<user>:<password>@<host>:<port>/<url-path>
```

The URL for default DMIF signalling over TCP/IP consists of the following:

x-dtcp://<user>:<password>@<target dmif>:<dmif port>/<service-entity-path>

The <host> is the target DMIF and hence for the "x-dtcp:" scheme it will be named <target dmif>.

The <port> indicates the TCP/IP socket port number over which the DMIF signalling messages will be delivered and hence for the "x-dtcp:" scheme it will be named <dmif port>.

The <url-path> is the path to access a dmif service entity and hence for the "x-dtcp:" scheme it will be named <service-entity-path>.

C.4.3 URL scheme for DMIF signalling over networks using NSAP addresses format

The URL for default DMIF signalling over ATM follows the generic syntax for new URL schemes defined in RFC1738. Usually the URLs that involve the direct use of an IP-based protocol to a specified target on the Internet use a common Internet specific syntax:

```
//<user>:<password>@<host>:<port>/<url-path>
```

This scheme does not involve direct use of an IP-based protocol, thus the "://" is omitted. The other fields have however similar meaning, as explained below.

The URL for default DMIF signalling over ATM consists of the following:

x-datm:<user>:<password>@<target dmif>:<dmif selector>/<service-entity-path>

The <host> is the target DMIF and hence for the "x-datm:" scheme it will be named <target dmif>. The <target dmif> represents an E.164 number that will be used for the E.164 portion of the NSAP address of the calledAddress in the Q.2931 signalling messages.

The <port> indicates the Selector Byte that will be used for the Selector Byte portion of the NSAP address of the calledAddress in the Q.2931 signalling messages and hence for the "x-datm:" scheme it will be named <dmif selector>.

The <url-path> is the path to access a dmif service entity and hence for the "x-datm:" scheme it will be named <service-entity-path>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 14496-6:1999

Annex D (informative)

Protocol error recovery

D.1 Timeouts and retransmission

The DMIF Default Signalling Protocol includes facilities to detect the non-receipt of messages and message retransmission algorithms to provide error recovery from lost or corrupt messages. Detection of the non-receipt of a message is accomplished by the use of the `Tid.tMessage` timer. When a timeout occurs an event is generated of the form `[Tid.tMessage]`. At that time error recovery procedures will be performed.

Because the DMIF Default Signalling Protocol uses a basic request/reply transaction-based messaging scheme the transmit and receive windows for the protocol are at most 1 at any given time for each message. The Transaction ID used in each message allows multiple messages to be sent with simultaneous outstanding replies. The receipt of the associated Confirm indicates acknowledgment that a Request was received. Similarly the non-receipt of a Confirm message is indicated by the timeout of the `Tid.tMessage` timer for the associated Request and its subsequent retransmission. Therefore the sending entity of a Request message uses the `Tid.tMessage` timer as a retransmission timer whereas the sending entity of a Confirm message uses the `Tid.tMessage` timer as a holding timer to keep the message available in case the Request message is resent. Each time either of these conditions occurs the entity will resend the message and increment its `Tid.numRetrans` state variable. If `Tid.numRetrans` exceeds `Tid.retransBound` the entity will perform any session protocol actions and state transitions necessary to abort the current message sequence and move the transaction into the `TExpire` state.

The initial value of the `Tid.tMessage` timer as well as the algorithm chosen to compute its successive values are out of the scope of this part of ISO/IEC 14496; the same applies to the `Tid.retransBound` parameter.

D.2 Transaction state variables

Each transaction whose completion has not yet been verified has an instance of the Transaction State Machine alive to encode its status. The Transaction State Machine is realized by the following state variables.

D.2.1 Transaction identifier: `Tid`

The `transactionId` field value used in the `DSMCCMessageHeader()` to identify the transaction of which this message is a part.

D.2.2 Transaction state: `Tid.state`

The current state of the transaction with `transactionId` value of `Tid`.

D.2.3 Associated session: `Tid.Sid`

The value used to identify the `networkSession` in which the `transactionId` `Tid` pertains. This variable is used to correlate the transaction state variables with its associated `networkSession`.

D.2.4 Transaction message timeout: `Tid.tMessage`

The holding timer for the transaction message with `transactionId` value `Tid`. When initiating a Request message this timer is used to detect non-receipt of the corresponding Confirm message and initiates retransmission procedures. When initiating a Confirm message this timer serves as a holding timer used to keep the message available for possible retransmission if its associated Request message is retransmitted.

D.2.5 Expired transactionId holding timer: Tid.tHold

The timer set when the transaction state machine moves to the expired state. The Tid will be held in this state until Tid.tHold expires and transition is made to the TUndefined state and the corresponding Tid may be used again. This will prevent possible collisions of messages with the same transactionId in the network due to premature reuse of transactionIds.

The value assigned to Tid.tHold is not within the scope of this part of ISO/IEC 14496.

Note that although this functionality is represented as a timer in the Transaction Identifier State Machine, alternative methods can be used in implementations to achieve the same results without timers. For example, a cyclic allocation space sufficiently large to guarantee non-reuse within the required period of time is acceptable.

D.2.6 Transaction message: Tid.message

The message contents of the transaction with transactionId value of Tid. This is what will be retransmitted in the event that a transaction remains unacknowledged for an extended period of time.

D.2.7 Tid.retransBound

The number of retransmissions of an unacknowledged transaction before the protocol state machine is to give up and invalidate the transaction.

Tid.retransBound is assigned when Tid is allocated.

D.2.8 Tid.numRetrans

The current number of retransmissions that have been performed on this transaction.

D.3 Transaction Identifier State Machine

A transaction is a pair of messages which share the same transactionId value in their DSMCCMessageHeader(). The transaction pair consists of either a Request message followed by a Confirm message. The transactionId value is used to correlate the message pairs on the sending or receiving entities. To insure integrity of messages, it is important that transactionId value not be re-used by an entity until it can guarantee that all copies of a transaction's messages have been cleared from the session. In the Transaction Identifier State Machine this is accomplished by moving all transactionIds into the Texpired state for some period of time before they can be reused.

The following states are defined:

TUndefined: The Transaction ID has not been assigned.

TActiveRetrans: The Transaction ID has been assigned and the state is permitted to retransmit the message.

TActiveNoRetrans: The Transaction ID has been assigned and the state is not permitted to retransmit the message.

TExpire: The Transaction ID is retired.

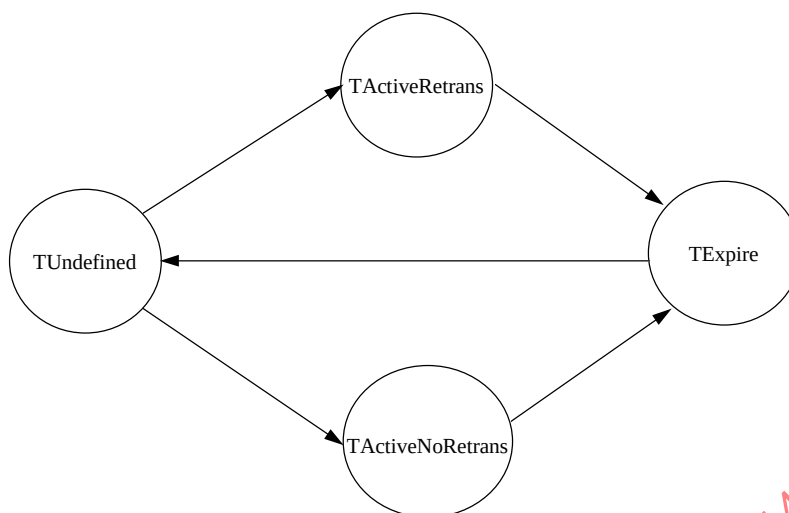


Figure D.1 — Transaction Identifier State Machine

D.4 Transaction Identifier State table

Description of the events:

[new-tid-retrans]	Sending a Request message with Tid.retransBound > 0
[new-tid-noretrans]	Sending a Request message with Tid.retransBound = 0
[free-tid]	Transaction ID no longer required
[Tid.tMessage]	Tid.tMessage timer has expired
[retrans-req]	An administrative request to retransmit
[Tid.tHold]	Tid.tHold timer has expired