INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

T.125

(02/98)

SERIES T: TERMINALS FOR TELEMATIC SERVICES

Multipoint communication service protocol specification

ITU-T Recommendation T.125

(Previously CCITT Recommendation)

ITU-T T-SERIES RECOMMENDATIONS

TERMINALS FOR TELEMATIC SERVICES

For further details, please refer to ITU-T List of Recommendations.

ITU-T RECOMMENDATION T.125

MULTIPOINT COMMUNICATION SERVICE PROTOCOL SPECIFICATION

Summary

This Recommendation defines a protocol operating through the hierarchy of a multipoint communication domain. It specifies the format of protocol messages and procedures governing their exchange over a set of transport connections. The purpose of the protocol is to implement the Multipoint Communication Service defined by Recommendation T.122.

Annex A defines a protocol allowing the use of multicast network services in a T.120 conference. It does this by performing data routing as part of T.125. Once it assumes responsibility for routing, it will use multicast whenever possible. This Annex handles sub-optimal networks by testing multicast pathways before depending on them. In the event that multicast does not work at some point within the conference, unicast is used in its place. This Annex is independent of the networks on which it is used. All issues that are specific to a given network are handled by the protocol stacks that lie beneath this protocol.

The following changes have been incorporated into this version of the Recommendation:

– the MAP protocol has been incorporated as Annex A;

– MCS Protocol version 3 is specified;

– *Extended-Parameter* MCSPDUs have been added;

– *Capabilities-Notification* MCSPDUs have been added;

– fields were added to the Send-Data and Uniform-Send-Data MCSPDUs to support changes to Recommendation T.122;

– minor changes have been made to the document format.

Source

ITU-T Recommendation T.125 was revised by ITU-T Study Group 16 (1997-2000) and was approved under the WTSC Resolution No. 1 procedure on the 6th of February 1998.

FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, the ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

Recommandation T.125

MULTIPOINT COMMUNICATION SERVICE PROTOCOL SPECIFICATION

(revised in 1998)

1 Scope

This Recommendation specifies:

a) procedures for a single protocol for the transfer of data and control information from one MCS provider to a peer MCS provider;

b) the structure and encoding of the MCS protocol data units used for the transfer of data and control information.

The procedures are defined in terms of:

a) the interactions between peer MCS providers through the exchange of MCS protocol data units;

b) the interactions between an MCS provider and MCS users through the exchange of MCS primitives;

c) the interactions between an MCS provider and a transport service provider through the exchange of transport service primitives.

These procedures are applicable to instances of multipeer communication among systems that support MCS and wish to interconnect in an open systems environment.

2 References

The following Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision: all users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

– ITU-T Recommendation T.122 (1998), *Multipoint Communication Service – Service definition.*

– ITU-T Recommendation T.123 (1996), *Network specific data protocol stacks for multimedia conferencing.*

– ITU-T Recommendation X.200 (1994) | ISO/IEC 7498-1:1994, *Information technology – Open Systems Interconnection – Basic reference model: The basic model.*

– ITU-T Recommendation X.214 (1995) | ISO/IEC 8072:1996, *Information technology – Open Systems Interconnection – Transport service definition.*

– ITU-T Recommendation X.680 (1997) | ISO/IEC 8824-1:1998, *Information technology – Abstract Syntax Notation One (ASN.1) – Specification of basic notation.*

– ITU-T Recommendation X.690 (1997) | ISO/IEC 8825-1:1998, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).*

– ITU-T Recommendation X.691 (1997) | ISO/IEC 8825-2:1998, *Information technology – ASN.1 encoding rules – Specification of Packed Encoding Rules (PER).*

3 Definitions

NOTE – These definitions make use of the abbreviations defined in clause 4.

This Recommendation is based on the concepts developed in Recommendation X.200 and makes use of the following terms defined therein:

a) flow control;

b) reassembling;

c) recombining;

d) segmenting;

e) sequencing;

f) splitting;

g) transfer syntax;

h) transport connection;

i) transport connection endpoint identifier;

j) transport service;

k) transport service access point;

l) transport service access point address;

m) transport service data unit.

This Recommendation is also based on the concepts developed in Recommendation T.122 and makes use of the following terms defined therein:

a) control MCSAP;

b) MCS attachment;

c) MCS channel;

d) MCS connection;

e) MCS domain;

f) MCS domain selector;

g) MCS private channel;

h) MCS private channel manager;

i) MCS provider;

j) MCS service access point;

k) MCS user;

l) MCS user id;

m) top MCS provider.

For the purposes of this Recommendation, the following definitions apply.

3.1 MCS service data unit: An amount of MCS user data whose identity is preserved during the transfer from transmitter to receivers. Specifically, the content of one MCS-SEND-DATA request or one MCS-UNIFORM-SEND-DATA request.

3.2 MCS interface data unit: The unit of information transferred across an MCSAP between an MCS user and an MCS provider in a single interaction. Each MCS interface data unit contains interface control information and may also contain all or part of an MCS service data unit.

3.3 MCS protocol data unit: A unit of information exchanged in the MCS protocol, consisting of control information transferred between MCS providers to coordinate their joint operation and possibly data transferred on behalf of MCS users to whom they are providing service.

3.4 MCS data transfer priority: One of four levels: top, high, medium, low. The value is communicated unchanged from transmitter to receivers. Depending on an MCS domain parameter for the number of distinct data transfer priorities implemented, two or more lowest priorities may receive the same quality of service.

3.5 valid MCSPDU: An MCSPDU whose structure and encoding complies with this Recommendation.

3.6 invalid MCSPDU: An MCSPDU that is not a valid MCSPDU.

3.7 protocol error: Use of an MCSPDU in a manner inconsistent with the procedures of this Recommendation.

3.8 connect MCSPDU: Any one of **Connect-Initial, Connect-Response, Connect-Additional, Connect-Result**.

3.9 extended parameters MCSPDU: Either **Extended-Parameter-Propose** or **Extended-Parameter-Accept**.

3.10 domain MCSPDU: Any MCSPDU that is not a connect MCSPDU or an extended parameter MCSPDU.

3.11 data MCSPDU: Any one of **SendDataRequest, SendDataIndication, UniformSendDataRequest, UniformSendDataIndication**.

3.12 control MCSPDU: Any domain MCSPDU that is not a data MCSPDU.

3.13 initial TC: The first transport connection of an MCS connection, used to exchange control MCSPDUs and data MCSPDUs of top priority.

3.14 additional TC: A subsequent transport connection belonging to an MCS connection, used to exchange data MCSPDUs of lesser priority.

3.15 subtree of an MCS provider: In the context of an MCS domain, this consists of the MCS provider itself and its MCS attachments plus all MCS providers hierarchically subordinate to it and their MCS attachments.

3.16 height of an MCS provider: In the context of an MCS domain, this is the maximum height of all hierarchically subordinate MCS providers. An MCS provider without subordinates has height zero.

3.17 version 3 or V3 summit: The MCS providers in a Domain Hierarchy which use MCS protocol version 3 or greater and are either:

- a Top Provider; or
- have an upward connection path to a version 3 Top Provider that flows only through other version 3 providers.

3.18 leaf node: A node with one upward connection and no downward connections.

3.19 intermediate node: A node with one upward connection and at least one downward connection.

3.20 superior node: Any node with an upward connection is connected to its superior node.

3.21 intervening node: With respect to a specific node, any node in the path from the specific node to the summit provider is an intervening node.

3.22 capability: A high level behavior of the system, most likely requiring the cooperation between two or more nodes.

3.23 capability ID: An identification assigned to a capability. If standardized, this ID will be documented in T.125.

3.24 participation indicator: A characteristic of a capability which indicates the level of participation expected of V3 nodes in a V3 summit in order for the capability to be exercised. The value of a participation indicator is either partial (two or more nodes must participate) or global (all nodes must participate).

3.25 partial capability: A capability with a participation indicator of "partial".

3.26 global capability: A capability with a participation indicator of "global".

3.27 summit capabilities list: A list of capabilities available to the V3 summit.

3.28 V3 summit provider: Any top provider which is a V3 node.

4 Abbreviations

This Recommendation uses the following abbreviations.

MAP	Multicast Adaptation Protocol as described in Annex A
MAPSAP	MAP Service Access Point
MC	MAP Connection
MCS	Multipoint Communication Service
MCSAP	MCS Service Access Point
MCSPDU	MCS Protocol Data Unit
PDU	Protocol Data Unit
TC	Transport Connection
TS	Transport Service
TSAP	Transport Service Access Point
TSDU	Transport Service Data Unit
V2	Version 2
V3	Version 3

5 Overview of the MCS protocol

5.1 Model of the MCS layer

An MCS provider communicates with MCS users through an MCSAP by means of the MCS primitives defined in Recommendation T.122. These primitives can be the cause or result of MCSPDU exchanges between peer MCS providers using an MCS connection, or they can be the

cause or result of actions taken within a single MCS provider. MCSPDU exchanges occur between MCS providers that host the same MCS domain.

An MCS provider can have multiple peers, each reached directly by a different MCS connection or indirectly through a peer MCS provider. An MCS connection is composed of either one MAP connection or one or more transport connections. MAP based connections will operate as described in Annex A. For transport based connections, the number of connections will depend on the number of data transfer priorities implemented in an MCS domain. Protocol exchanges are effected using the services of the transport layer through a pair of TSAPs.

This model of the MCS layer is illustrated in Figure 5-1.

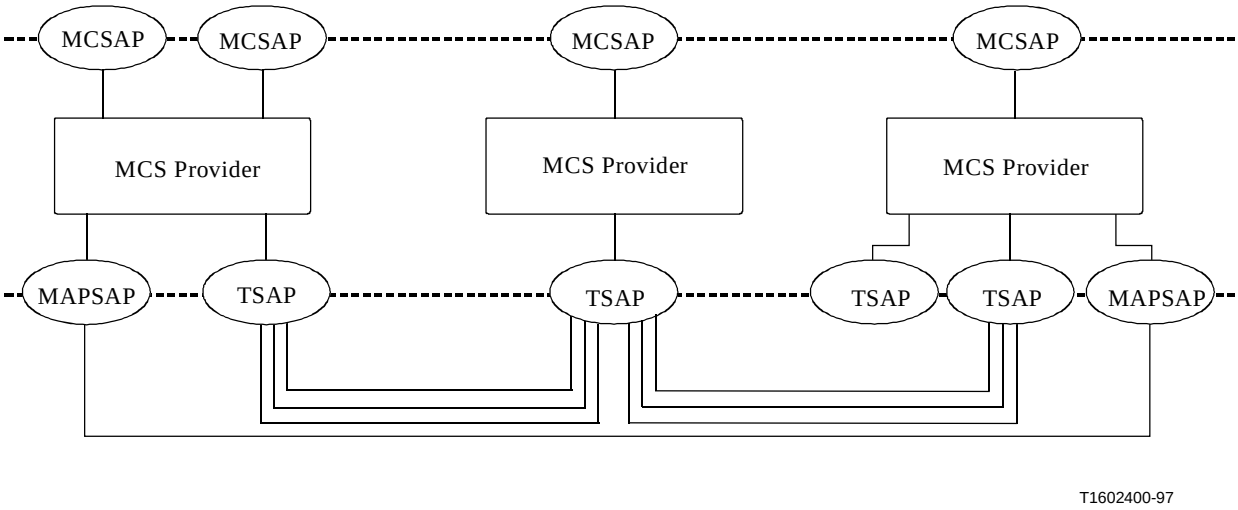

Figure 5-1/T.125 – Model of the MCS layer

5.2 Services provided by the MCS layer

The MCS protocol supports the services defined in Recommendation T.122. Information is transferred to and from an MCS user using the MCS primitives listed in Table 5-1.

5.2.1 MAP layer services

When using MAP connections, MCS will function as described in Annex A.

5.3 Services assumed from the transport layer

For transport connections, the MCS protocol assumes the use of a subset of the connection-oriented transport service defined in Recommendation X.214. Information is transferred to and from a TS provider by the primitives listed in Table 5-2.

5.4 Functions of the MCS layer

Table 5-1 identifies the functional units of MCS and the MCSPDUs associated with each MCS primitive. MCSPDUs are defined in clause 7. The relationship between primitives and MCSPDUs can be as simple as cause and effect, in either direction. For example, MCS-ATTACH-USER request generates **AttachUserRequest**, and **AttachUserConfirm** generates MCS-ATTACH-USER confirm. Other cases may be more complicated. The completion of MCS-CONNECT-PROVIDER, for example, requires the exchange of additional MCSPDUs as side effects of the four-phase primitive.

And any one of five MCSPDUs can cause an MCS-DETACH-USER indication, any of four an MCS-CHANNEL-EXPEL indication.

5.4.1 Domain management

The MCS layer maintains the integrity of the MCS connections comprising an MCS domain. An MCS connection is oriented, with one end hierarchically superior to the other. There is a single MCS provider at the top of each domain.

Establishing an MCS connection merges two domains into one. The MCS layer ensures that one top provider remains. It resolves any conflicts of unique identity or exclusive ownership that may arise.

Disconnecting an MCS connection splits a domain into two portions. The portion containing the top provider survives. The bottom portion eradicates itself.

The MCS layer uniquely identifies users attached to a domain. Users may become aware of each other through their interactions via MCS primitives. The MCS layer notifies all users of a domain when one of them detaches. The MCS layer recovers any resources of a detached user.

Table 5-1/T.125 – MCS primitives

Functional unit	Primitives	Associated MCSPDUs
Domain Management	MCS-CONNECT-PROVIDER request MCS-CONNECT-PROVIDER indication MCS-CONNECT-PROVIDER response MCS-CONNECT-PROVIDER confirm (side effects)	**Connect-Initial** **Connect-Initial** **Connect-Response** **Connect-Response** **Extended-Parameters-Propose** **Extended-Parameters-Accept** **Connect-Additional** **Connect-Result** **PlumbDomainIndication** **ErectDomainRequest** **MergeChannelsRequest** **MergeChannelsConfirm** **PurgeChannelsIndication** **MergeTokensRequest** **MergeTokensConfirm** **PurgeTokensIndication**
	MCS-DISCONNECT-PROVIDER request MCS-DISCONNECT-PROVIDER indication	**DisconnectProviderUltimatum** **DisconnectProviderUltimatum** **RejectMCSPDUUltimatum**
	MCS-ATTACH-USER request MCS-ATTACH-USER confirm	**AttachUserRequest** **AttachUserConfirm**
	MCS-DETACH-USER request MCS-DETACH-USER indication	**DetachUserRequest** **DetachUserIndication** **MergeChannelsConfirm** **PurgeChannelsIndication** **MergeTokensConfirm** **PurgeTokensIndication**

Table 5-1/T.125 – MCS primitives (*concluded*)

Functional unit	Primitives	Associated MCSPDUs
Channel Management	MCS-CHANNEL-JOIN request MCS-CHANNEL-JOIN confirm	**ChannelJoinRequest** **ChannelJoinConfirm**
	MCS-CHANNEL-LEAVE request MCS-CHANNEL-LEAVE indication	**ChannelLeaveRequest** **MergeChannelsConfirm** **PurgeChannelsIndication**
	MCS-CHANNEL-CONVENE request MCS-CHANNEL-CONVENE confirm	**ChannelConveneRequest** **ChannelConveneConfirm**
	MCS-CHANNEL-DISBAND request MCS-CHANNEL-DISBAND indication	**ChannelDisbandRequest** **MergeChannelsConfirm** **PurgeChannelsIndication**
	MCS-CHANNEL-ADMIT request MCS-CHANNEL-ADMIT indication	**ChannelAdmitRequest** **ChannelAdmitIndication**
	MCS-CHANNEL-EXPEL request MCS-CHANNEL-EXPEL indication	**ChannelExpelRequest** **ChannelExpelIndication** **ChannelDisbandIndication** **MergeChannelsConfirm** **PurgeChannelsIndication**
Data Transfer	MCS-SEND-DATA request MCS-SEND-DATA indication	**SendDataRequest** **SendDataIndication**
	MCS-UNIFORM-SEND-DATA request MCS-UNIFORM-SEND-DATA indication	**UniformSendDataRequest** **UniformSendDataIndication**
Token Management	MCS-TOKEN-GRAB request MCS-TOKEN-GRAB confirm	**TokenGrabRequest** **TokenGrabConfirm**
	MCS-TOKEN-INHIBIT request MCS-TOKEN-INHIBIT confirm	**TokenInhibitRequest** **TokenInhibitConfirm**
	MCS-TOKEN-GIVE request MCS-TOKEN-GIVE indication MCS-TOKEN-GIVE response MCS-TOKEN-GIVE confirm	**TokenGiveRequest** **TokenGiveIndication** **TokenGiveResponse** **TokenGiveConfirm**
	MCS-TOKEN-PLEASE request MCS-TOKEN-PLEASE indication	**TokenPleaseRequest** **TokenPleaseIndication**
	MCS-TOKEN-RELEASE request MCS-TOKEN-RELEASE confirm	**TokenReleaseRequest** **TokenReleaseConfirm**
	MCS-TOKEN-TEST request MCS-TOKEN-TEST confirm	**TokenTestRequest** **TokenTestConfirm**
Capabilities Notification		**CapabilitiesNotificationRequest** **CapabilitiesNotificationIndication**

Table 5-2/T.125 – Transport service primitives

Primitives	Use	Parameters	Use
T-CONNECT request T-CONNECT indication	X X	Called address Calling address Expedited data option Quality of service TS user-data	X X – X –
T-CONNECT response T-CONNECT confirm	X X	Responding address Expedited data option Quality of service TS user-data	– – X –
T-DATA request T-DATA indication	X X	TS user-data	X
T-EXPEDITED-DATA request T-EXPEDITED-DATA indication	– –	TS user-data	–
T-DISCONNECT request	X	TS user-data	–
T-DISCONNECT indication	X	Reason TS user-data	– –

X The MCS protocol assumes that this feature is always available.

– The MCS protocol does not use this feature.

5.4.2 Channel management

The MCS layer records which parts of an MCS domain contain one or more users joined to a given channel, so that it can optimize the transfer of data to destinations that wish to receive it.

The MCS layer treats user ids as single-member channels, which only the designated user is allowed to join. On request, it can create private channels to which only admitted users are allowed access or assign public channels to which no other users are currently joined.

5.4.3 Data transfer

The MCS layer maintains a sequenced flow of data to the users who have joined a channel. A channel becomes, in effect, a multicast distribution list, with a range somewhere between zero destinations and a complete broadcast.

By default, the MCS layer routes data to each receiver over the shortest path of MCS connections. Optionally, it routes specified MCS service data units through the top MCS provider, thereby guaranteeing their uniform receipt at all receivers, which may include the transmitter too.

The MCS layer recognizes one or more priorities of data transfer and extends them preferential processing. For fully-reliable data, the MCS layer allows MCS service data units of unlimited size through segmentation. However, for unreliable data, the size of MCS service data units may not exceed the maximum MCSPDU size minus PDU header overhead as determined by the domain parameters.

The MCS layer regulates the global flow of data within a domain. The inability of a receiver to accept data at the rate it is offered eventually creates back-pressure that causes transmitters to be blocked. A user may be detached involuntarily if it fails to maintain a minimum receiving rate.

The MCS layer provides both fully-reliable (i.e. guaranteed delivery) and unreliable (i.e. non-guaranteed delivery) data transfer. For fully-reliable data transfer, the MCS layer guarantees error-free receipt of transmitted data, as long as the source and destination users remain attached and

the destination user remains joined to the channel. For unreliable data transfers, MCS provides ordered receipt of transmitted data, although gaps in the data stream may occur. Also, for both fully-reliable and unreliable data transfers, higher priority data takes precedence, and a surfeit of it may indefinitely delay the delivery of lower priority data.

5.4.4 Token management

The MCS layer implements token operations at the top MCS provider, thereby ensuring consistency and exclusion.

5.4.5 Capabilities notification

The MCS layer provides a mechanism for the exchange of information regarding capabilities. Via this mechanism, nodes are able to determine the state of the domain with respect to a certain capability. This allows a node to make the decision whether to exercise that capability.

5.5 Hierarchical processing

Hierarchical processing in an MCS domain is illustrated in Figure 5-2.

The nodes in the figure represent MCS providers, and the labelled arrows represent MCSPDUs. This example focuses on a period of time after the domain has been established through connections between MCS providers and the use of data transfer is beginning to expand. At step 1 provider D requests on behalf of a user to join a channel over which data will be distributed, and at step 2 the request is confirmed as successful. At step 3 a user attached to provider A sends data, and the corresponding **SendDataRequest** begins to flow upward. Assuming that only attachments at providers A, C, and D are joined to the channel over which the data is being sent, the request MCSPDU is reflected downward at steps 6 and 7 as **SendDataIndication**. Provider E, aware that no other subordinate needs to receive the data, simply forwards **SendDataRequest** upward at step 4. Provider F forwards **SendDataRequest** upward at step 5 but also reflects it downward at step 6, knowing that provider C has expressed interest in the channel.

MCS providers are not especially concerned with their height in the hierarchy, except for their role in maintaining an overall limit on the height of the domain and in the broad sense that they are either the top provider or they are not. The top MCS provider has no upward connection. All others have exactly one.

An MCS provider records information about channels and tokens used in its subtree of an MCS domain.

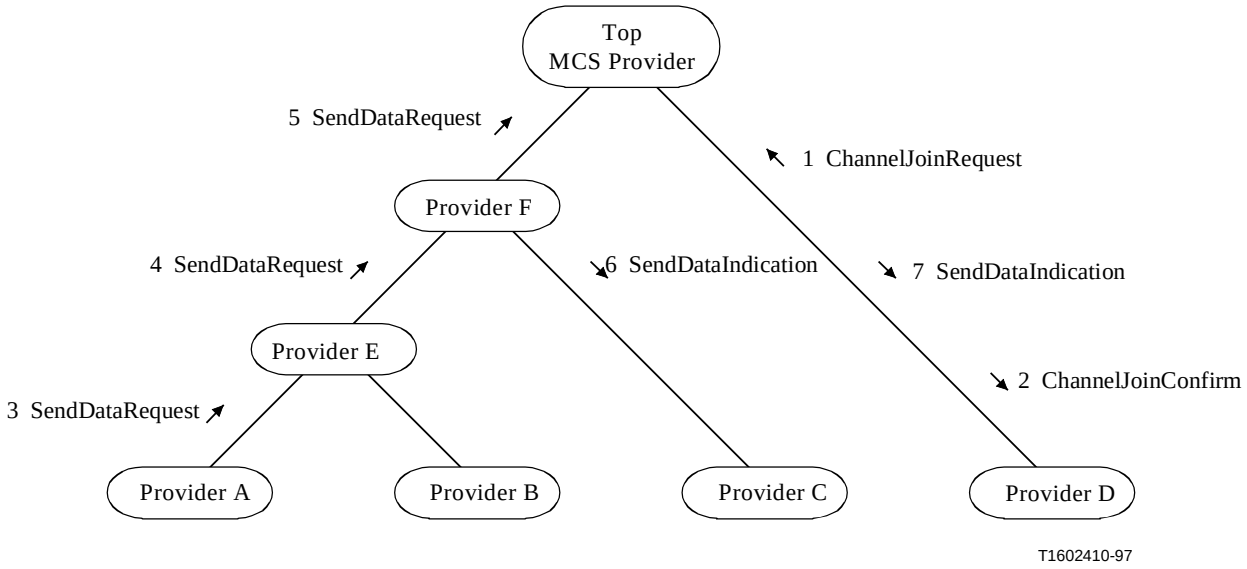

Figure 5-2/T.125 – Hierarchical processing in an MCS domain

An MCS provider records channels that are joined by users within the subtree and, for each such channel, where the joins originate, that is, from which attachments and from which downward MCS connections. It records user ids that are assigned in the subtree and where they originate. It records private channels that have either a manager or an admitted user in the subtree, and it records the associated user ids.

An MCS provider records tokens that are grabbed or inhibited by users within the subtree, and it records the associated user ids.

An MCS provider inspects requests arising from its subtree to verify that the initiating user id is legitimately assigned to the originating attachment or downward MCS connection. This creates rings of protection around the top MCS provider and limits how much disruption a malicious participant can cause in an otherwise cooperative domain.

In general terms, to which there are several exceptions, the operation of the MCS layer can be described as follows:

a) An MCS primitive request invoked at an MCS attachment generates an MCSPDU at the corresponding MCS provider and dispatches it upward towards the top MCS provider. There, where full information about the MCS domain is held, the MCSPDU is acted on.

b) A confirm MCSPDU may be generated at the top MCS provider to return results to the requesting attachment. MCS providers that pass it along update their records according to the impact of the operation on their subtree. A confirm is routed to the initiating user id by consulting local records at each successive downward hop.

c) An indication MCSPDU may be generated instead to inform other attachments about the action taken. Indications may be replicated and sent downward on several connections that lead to affected users. MCS providers may also update their records with the impact of an operation as part of processing an indication.

The preceding description is an overview intended to create a conceptual framework. Full details are specified in later clauses concerning the information recorded in an MCS provider and how specific MCSPDUs are processed.

Among exceptions are the following: Some requests, notably **ChannelJoinRequest** and **ChannelLeaveRequest**, may stop rising at a level short of the top provider, and some indications, notably **SendDataIndication**, may be generated at a level below the top provider. One MCSPDU, **TokenGiveResponse**, belongs to the response category. And some MCSPDUs may be generated by an MCS provider as a continuation of processing unlike MCSPDUs, such as **ChannelLeaveRequest** following **DetachUserIndication**.

5.6 Domain parameters

The MCS providers hosting a single MCS domain allocate resources and execute procedures according to the following parameters. The values of these parameters are identical throughout a domain with the exception of Protocol Version.

a) Maximum number of MCS channels that may be in use simultaneously. This includes channels that are joined by any user, user ids that have been assigned, and private channels that have been created.

b) Maximum number of user ids that may be assigned simultaneously. This is a sublimit within the constraint of the preceding parameter.

c) Maximum number of token ids that may be grabbed or inhibited simultaneously.

d) Number of data transfer priorities implemented. This equals the number of TCs in an MCS connection. An MCS user may still send and receive data with priorities outside the limit. However, such priorities may be treated the same as the lowest priority that is implemented.

e) Enforced throughput. Although global flow control limits data transfer within a domain to the rate of the slowest receiver, receivers must not be allowed to run arbitrarily slowly. Otherwise, one party in a conference may obstruct all others. This parameter instructs MCS providers to enforce a minimum receiving rate at each MCS attachment and over each downward MCS connection. Violators run the risk of being involuntarily detached or disconnected, respectively.

f) Maximum height. This constrains the height of all MCS providers, in particular the top provider.

g) Maximum size of domain MCSPDUs. Global flow control is based on buffering domain MCSPDUs within an MCS provider (but not connect MCSPDUs). For simplicity, fixed-size buffers are assumed. An MCS provider shall not generate larger MCSPDUs. This constrains the amount of information that can be packed into a single control MCSPDU and suggests where unlimited user data should be segmented in data MCSPDUs.

h) Protocol version. This takes one of three values defining different encodings for domain MCSPDUs. In version three of the MCS Protocol this value is on a per connection basis. This allows version two MCS providers to exist downstream in a version three enabled conference.

NOTE – A given instance of an MCS provider may operate with local resource constraints that are also parameterized. These may include the amount of memory available for buffering MCSPDUs awaiting transport, the maximum number of MCS attachments, and the maximum number of MCS connections to other providers. Such parameters are local matters and are not communicated across an MCS domain.

5.7 Extended parameters

The MCS providers who are members of the V3 summit of a single MCS domain will allocate resources and execute procedures according to the following parameters.

a) Unreliable data is supported. If unreliable data is supported in the V3 summit, this Boolean value will be True. The value of this parameter is identical throughout the V3 summit of a domain.

b) Domain Reference Identifier. This is not a negotiated value, but will be generated by an MCS provider to uniquely identify a specific domain. Although the same value may be used to identify different domains for different providers, it must uniquely represent one specific domain to the provider that generated the value.

6 Use of the transport service

6.1 Model of the transport service

This description paraphrases relevant parts of Recommendation X.214 assuming that no use is made of expedited data.

The transport service offers these features to a TS user:

a) Means to establish a TC with another TS user for the purpose of exchanging TSDUs. More than one TC may exist between the same pair of TS users.

b) Associated with each TC at its time of establishment, the opportunity to request, negotiate, and have agreed by the TS provider a certain Quality of Service as specified by parameters representing characteristics such as throughput, transit delay, residual error rate, and priority.

c) Means of transferring TSDUs on a TC. The transfer of TSDUs, which consist of an integral number of octets, is transparent, in that the boundaries of TSDUs and the contents of TSDUs are preserved unchanged by the TS provider.

d) Means by which a receiving TS user may control the rate at which the sending TS user may send data.

e) The unconditional and therefore possibly destructive release of a TC.

The operation of a TC is modelled in an abstract way by a pair of queues linking two TSAPs. There is one queue for each direction of information flow. Each TC is modelled by a separate pair of queues.

The queue model is used to express the flow control feature. A queue has a limited capacity, but this capacity is not necessarily either fixed or determinable. Connect, TSDU, and disconnect objects are entered and removed from a queue as the result of interactions at the two TSAPs. The ability of a TS user to add objects to a queue is determined by the behavior of the TS user removing objects from that queue and the state of the queue. The only objects that can be placed in a queue by the TS provider are disconnect objects. Objects are added to a queue subject to control by the TS provider. Objects are normally removed from the queue subject to control by the receiving TS user. Objects are normally removed in the same order that they were added. The only exception to normal removal is that an object may be deleted by the TS provider if, and only if, the following one is a disconnect object.

A TC endpoint identification mechanism must be provided locally if the TS user and the TS provider need to distinguish between several TCs at a TSAP. All primitives must then make use of this identification mechanism to identify the TC to which they apply. This implicit identification is not

shown as a parameter of the TS primitives and must not be confused with the address parameters of T-CONNECT.

6.2 Use of multiple connections

An MCS connection consists of one or more TCs between the same pair of MCS providers. The first TC established is called the initial TC; those established subsequently are called additional TCs. All the TCs that belong to one MCS connection are established by the same MCS provider, in reaction to an MCS-CONNECT-PROVIDER request. This request contains address parameters that are the calling and called TSAP addresses. These are used unmodified in the T-CONNECT requests that result.

The number of TCs per MCS connection is uniform throughout an MCS domain. This domain parameter equals the number of data transfer priority levels implemented. Separate TCs are required because each is a vehicle for flow control. Blockages in lower priority data should not result in back pressure against higher priority data. To be fully implemented, lower and higher priority data must be carried on different TCs.

The quality of service requested for a TC may vary depending on the data priority for which it is established. These quality of service targets need not be uniform across an MCS domain.

Aspects of quality of service that are of interest include maximum or average throughput and transit delay. High priority data may favour low transit delay for real-time response but may not require high throughput. Low priority data, on the other hand, may favour high throughput for bulk transfers but may not require low transit delay.

TC priority is another aspect of quality of service, although it does not align precisely with the concept of MCS data transfer priority. It specifies the relative order in which TCs are to have their quality of service degraded, if necessary. High TC priority may be requested along with other characteristics, like low transit delay, to ensure that MCS priority data receives the preferential treatment it deserves.

Connect MCSPDUs occur only as the first TSDU carried in either direction of a TC. **Connect-Initial** and **Connect-Response** traverse the initial TC of an MCS connection. **Connect-Additional** and **Connect-Result** traverse additional TCs, if any.

A calling MCS provider, issuing T-CONNECT requests, controls through its own actions which TCs are part of the same MCS connection and what data transfer priorities they represent.

A called MCS provider, receiving T-CONNECT indications, must in general accept a TC and read its first TSDU before learning its significance. **Connect-Initial** identifies an incoming TC as the beginning of a new MCS connection. **Connect-Additional** identifies an incoming TC as part of an MCS connection in progress.

Connect-Additional contains a value assigned by the called MCS provider and conveyed to the calling MCS provider in a **Connect-Response** over the initial TC that designates the additional TC as belonging to the same MCS connection. **Connect-Additional** also states explicitly the data priority that the TC represents.

Connect MCSPDUs are exchanged immediately following TC establishment. Once an MCS connection has been confirmed, it becomes part of a hierarchical MCS domain. Thereafter the MCS connection conveys domain MCSPDUs.

With the exception of data MCSPDUs, domain MCSPDUs traverse the initial TC of an MCS connection. Data MCSPDUs traverse the TC that corresponds to their data priority. If the priority specified is beyond the number implemented in an MCS domain, a data MCSPDU traverses the TC of the lowest priority that is implemented.

If only one priority is implemented in an MCS domain, its MCS connections each consist of a single TC, no use is made of **Connect-Additional** or **Connect-Result**, and all MCSPDUs travel in sequence between providers.

6.3 Transport connection release

A TS provider adds reliability to end-to-end connections by executing enough protocol to compensate for any weakness in underlying network services. An MCS provider does not duplicate this functionality. It does not attempt further automatic recovery in the event of a transport failure.

Unrecoverable errors are announced through a T-DISCONNECT indication. If any of the TCs belonging to an MCS connection is disconnected, the others are immediately disconnected too. Unless this was requested by the user, an MCS-DISCONNECT-PROVIDER indication is generated, and the reason is given as provider-initiated.

An MCS-DISCONNECT-PROVIDER request, on the other hand, should appear as an indication at the other side with the reason given as user-requested. Despite assurances in X.214, the simplest class of transport protocol does not allow passing user data in T-DISCONNECT. Hence, the disconnect reason code is transferred in an explicit MCSPDU. This MCSPDU compels an MCS provider, upon receipt, to disconnect the MCS connection that conveyed it.

7 Structure of Version 2 MCSPDUs

The structure of version 2 MCSPDUs is specified using the notation ASN.1 of Recommendation X.208. The use and significance of these MCSPDUs is further described in clauses 9 and 10.

```
MCS-PROTOCOL DEFINITIONS ::=

BEGIN

-- Part 1: Fundamental MCS types

ChannelId ::= INTEGER (0..65535)                -- range is 16 bits

StaticChannelId ::= ChannelId (1..1000)         -- those known permanently

DynamicChannelId ::= ChannelId (1001..65535)    -- those created and deleted

UserId ::= DynamicChannelId                     -- created by Attach-User
                                                -- deleted by Detach-User

PrivateChannelId ::= DynamicChannelId           -- created by Channel-Convene
                                                -- deleted by Channel-Disband

AssignedChannelId ::= DynamicChannelId          -- created by Channel-Join zero
                                                -- deleted by last Channel-Leave

TokenId ::= INTEGER (1..65535)                  -- all are known permanently

TokenStatus ::= ENUMERATED
{
        notInUse                (0),
        selfGrabbed             (1),
        otherGrabbed            (2),
        selfInhibited           (3),
        otherInhibited          (4),
        selfRecipient           (5),
```

```
        selfGiving                  (6),
        otherGiving                 (7)
}

DataPriority ::= ENUMERATED
{
        top                         (0),
        high                        (1),
        medium                      (2),
        low                         (3)
}

Segmentation ::= BIT STRING
{
        begin                       (0),
        end                         (1)
} (SIZE (2))

DomainParameters ::= SEQUENCE
{
        maxChannelIds               INTEGER (0..MAX),
                                    -- a limit on channel ids in use,
                                    -- static + user id + private + assigned
        maxUserIds                  INTEGER (0..MAX),
                                    -- a sublimit on user id channels alone
        maxTokenIds                 INTEGER (0..MAX),
                                    -- a limit on token ids in use
                                    -- grabbed + inhibited + giving + ungivable + given
        numPriorities               INTEGER (0..MAX),
                                    -- the number of TCs in an MCS connection
        minThroughput               INTEGER (0..MAX),
                                    -- the enforced number of octets per second
        maxHeight                   INTEGER  (0..MAX),
                                    -- a limit on the height of a provider
        maxMCSPDUsize               INTEGER  (0..MAX),
                                    -- an octet limit on domain MCSPDUs
        protocolVersion             INTEGER (0..MAX)
}

-- Part 2: Connect provider

Connect-Initial ::= [APPLICATION 101] IMPLICIT SEQUENCE
{
        callingDomainSelector       OCTET STRING,
        calledDomainSelector        OCTET STRING,
        upwardFlag                  BOOLEAN,
                                    -- TRUE if called provider is higher
        targetParameters            DomainParameters,
        minimumParameters           DomainParameters,
        maximumParameters           DomainParameters,
        userData                    OCTET STRING
}

Connect-Response ::= [APPLICATION 102] IMPLICIT SEQUENCE
{
        result                      Result,
        calledConnectId             INTEGER (0..MAX),
                                    -- assigned by the called provider
                                    -- to identify additional TCs of
                                    -- the same MCS connection
        domainParameters            DomainParameters,
```

```
        userData                    OCTET STRING
}

Connect-Additional  ::=  [APPLICATION 103] IMPLICIT SEQUENCE
{
        calledConnectId             INTEGER (0..MAX),
        dataPriority                DataPriority
}

Connect-Result  ::=  [APPLICATION 104] IMPLICIT SEQUENCE
{
        result                      Result
}

-- Part 3: Merge domain

PlumbDomainIndication  ::=  [APPLICATION 0] IMPLICIT SEQUENCE
{
        heightLimit                 INTEGER (0..MAX)
                                    -- a restriction on the MCSPDU receiver
}

ErectDomainRequest  ::=  [APPLICATION 1] IMPLICIT SEQUENCE
{
        subHeight                   INTEGER (0..MAX),
                                    -- height in domain of the MCSPDU transmitter
        subInterval                 INTEGER (0..MAX)
                                    -- its throughput enforcement interval in milliseconds
}

ChannelAttributes  ::=  CHOICE
{
        static                      [0] IMPLICIT SEQUENCE
        {
                channelId                   StaticChannelId
                                            -- joined is implicitly TRUE
        },

        userId                      [1] IMPLICIT SEQUENCE
        {
                joined                      BOOLEAN,
                                            -- TRUE if user is joined to its user id
                userId                      UserId
        },

        private                     [2] IMPLICIT SEQUENCE
        {
                joined                      BOOLEAN,
                                            -- TRUE if channel id is joined below
                channelId                   PrivateChannelId,
                manager                     UserId,
                admitted                    SET OF UserId
                                            -- may span multiple MergeChannelsRequest
        },

        assigned                    [3] IMPLICIT SEQUENCE
        {
                channelId                   AssignedChannelId
                                            -- joined is implicitly TRUE
        }
}
```

```
MergeChannelsRequest ::= [APPLICATION 2] IMPLICIT SEQUENCE
{
        mergeChannels                   SET OF ChannelAttributes,
        purgeChannelIds                 SET OF ChannelId
}

MergeChannelsConfirm ::= [APPLICATION 3] IMPLICIT SEQUENCE
{
        mergeChannels                   SET OF ChannelAttributes,
        purgeChannelIds                 SET OF ChannelId
}

PurgeChannelsIndication ::= [APPLICATION 4] IMPLICIT SEQUENCE
{
        detachUserIds                   SET OF UserId,
                                        -- purge user id channels
        purgeChannelIds                 SET OF ChannelId
                                        -- purge other channels
}

TokenAttributes ::= CHOICE
{
        grabbed                         [0] IMPLICIT SEQUENCE
        {
                tokenId                         TokenId,
                grabber                         UserId
        },
        inhibited                       [1] IMPLICIT SEQUENCE
        {
                tokenId                         TokenId,
                inhibitors                      SET OF UserId
                                                -- may span multiple MergeTokensRequest
        },

        giving                          [2] IMPLICIT SEQUENCE
        {
                tokenId                         TokenId,
                grabber                         UserId,
                recipient                       UserId
        },

        ungivable                       [3] IMPLICIT SEQUENCE
        {
                tokenId                         TokenId,
                grabber                         UserId
                                                -- recipient has since detached
        },

        given                           [4] IMPLICIT SEQUENCE
        {
                tokenId                         TokenId,
                recipient                       UserId
                                                -- grabber released or detached
        }
}

MergeTokensRequest ::= [APPLICATION 5] IMPLICIT SEQUENCE
{
        mergeTokens                     SET OF TokenAttributes,
        purgeTokenIds                   SET OF TokenId
```

```
}

MergeTokensConfirm ::= [APPLICATION 6] IMPLICIT SEQUENCE
{
        mergeTokens                     SET OF TokenAttributes,
        purgeTokenIds                   SET OF TokenId
}

PurgeTokensIndication ::= [APPLICATION 7] IMPLICIT SEQUENCE
{
        purgeTokenIds                   SET OF TokenId
}
-- Part 4: Disconnect provider

DisconnectProviderUltimatum ::= [APPLICATION 8] IMPLICIT SEQUENCE
{
        reason                          Reason
}

RejectMCSPDUUltimatum ::= [APPLICATION 9] IMPLICIT SEQUENCE
{
        diagnostic                      Diagnostic,
        initialOctets                   OCTET STRING
}

-- Part 5: Attach/Detach user

AttachUserRequest ::= [APPLICATION 10] IMPLICIT SEQUENCE
{
}

AttachUserConfirm ::= [APPLICATION 11] IMPLICIT SEQUENCE
{
        result                          Result,
        initiator                       UserId OPTIONAL
}

DetachUserRequest ::= [APPLICATION 12] IMPLICIT SEQUENCE
{
        reason                          Reason,
        userIds                         SET OF UserId
}

DetachUserIndication ::= [APPLICATION 13] IMPLICIT SEQUENCE
{
        reason                          Reason,
        userIds                         SET OF UserId
}

-- Part 6: Channel management

ChannelJoinRequest ::= [APPLICATION 14] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId
                                        -- may be zero
}
```

```
ChannelJoinConfirm ::= [APPLICATION 15] IMPLICIT SEQUENCE
{
        result                  Result,
        initiator               UserId,
        requested               ChannelId,
                                -- may be zero
        channelId               ChannelId OPTIONAL
}

ChannelLeaveRequest ::= [APPLICATION 16] IMPLICIT SEQUENCE
{
        channelIds              SET OF ChannelId
}

ChannelConveneRequest ::= [APPLICATION 17] IMPLICIT SEQUENCE
{
        initiator               UserId
}

ChannelConveneConfirm ::= [APPLICATION 18] IMPLICIT SEQUENCE
{
        result                  Result,
        initiator               UserId,
        channelId               PrivateChannelId OPTIONAL
}

ChannelDisbandRequest ::= [APPLICATION 19] IMPLICIT SEQUENCE
{
        initiator               UserId,
        channelId               PrivateChannelId
}

ChannelDisbandIndication ::= [APPLICATION 20] IMPLICIT SEQUENCE
{
        channelId               PrivateChannelId
}

ChannelAdmitRequest ::= [APPLICATION 21] IMPLICIT SEQUENCE
{
        initiator               UserId,
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}

ChannelAdmitIndication ::= [APPLICATION 22] IMPLICIT SEQUENCE
{
        initiator               UserId,
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}

ChannelExpelRequest ::= [APPLICATION 23] IMPLICIT SEQUENCE
{
        initiator               UserId,
        channelId               PrivateChannelId,
        userIds                 SET OF UserId
}
```

```
ChannelExpelIndication  ::=  [APPLICATION 24] IMPLICIT SEQUENCE
{
        channelId                       PrivateChannelId,
        userIds                         SET OF UserId
}

-- Part 7: Data transfer

SendDataRequest  ::=  [APPLICATION 25] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,
        dataPriority                    DataPriority,
        segmentation                Segmentation,
        userData                        OCTET STRING
}

SendDataIndication  ::=  [APPLICATION 26] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,
        dataPriority                    DataPriority,
        segmentation                Segmentation,
        userData                        OCTET STRING
}

UniformSendDataRequest  ::=  [APPLICATION 27] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,
        dataPriority                    DataPriority,
        segmentation                Segmentation,
        userData                        OCTET STRING
}

UniformSendDataIndication  ::=  [APPLICATION 28] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,
        dataPriority                    DataPriority,
        segmentation                Segmentation,
        userData                        OCTET STRING
}

-- Part 8: Token management

TokenGrabRequest  ::=  [APPLICATION 29] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId
}

TokenGrabConfirm  ::=  [APPLICATION 30] IMPLICIT SEQUENCE
{
        result  Result,
        initiator                       UserId,
        tokenId                         TokenId,
        tokenStatus                     TokenStatus
}
```

```
TokenInhibitRequest ::= [APPLICATION 31] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId
}

TokenInhibitConfirm ::= [APPLICATION 32] IMPLICIT SEQUENCE
{
        result                          Result,
        initiator                       UserId,
        tokenId                         TokenId,
        tokenStatus                     TokenStatus
}

TokenGiveRequest ::= [APPLICATION 33] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId,
        recipient                       UserId
}

TokenGiveIndication ::= [APPLICATION 34] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId,
        recipient                       UserId
}

TokenGiveResponse ::= [APPLICATION 35] IMPLICIT SEQUENCE
{
        result                          Result,
        recipient                       UserId,
        tokenId                         TokenId
}

TokenGiveConfirm ::= [APPLICATION 36] IMPLICIT SEQUENCE
{
        result                          Result,
        initiator                       UserId,
        tokenId                         TokenId,
        tokenStatus                     TokenStatus
}

TokenPleaseRequest ::= [APPLICATION 37] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId
}

TokenPleaseIndication ::= [APPLICATION 38] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId
}

TokenReleaseRequest ::= [APPLICATION 39] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId
}
```

```
TokenReleaseConfirm ::= [APPLICATION 40] IMPLICIT SEQUENCE
{
        result Result,
        initiator                       UserId,
        tokenId                         TokenId,
        tokenStatus                     TokenStatus
}

TokenTestRequest ::= [APPLICATION 41] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId
}

TokenTestConfirm ::= [APPLICATION 42] IMPLICIT SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId,
        tokenStatus                     TokenStatus
}

-- Part 9: Status codes

Reason ::= ENUMERATED       -- in DisconnectProviderUltimatum, DetachUserRequest, DetachUserIndication
{
        rn-domain-disconnected          (0),
        rn-provider-initiated           (1),
        rn-token-purged                 (2),
        rn-user-requested               (3),
        rn-channel-purged               (4)
}

Result ::= ENUMERATED       -- in Connect, response, confirm
{
        rt-successful                   (0),
        rt-domain-merging               (1),
        rt-domain-not-hierarchical      (2),
        rt-no-such-channel              (3),
        rt-no-such-domain               (4),
        rt-no-such-user                 (5),
        rt-not-admitted                 (6),
        rt-other-user-id                (7),
        rt-parameters-unacceptable      (8),
        rt-token-not-available          (9),
        rt-token-not-possessed          (10),
        rt-too-many-channels            (11),
        rt-too-many-tokens              (12),
        rt-too-many-users               (13),
        rt-unspecified-failure          (14),
        rt-user-rejected                (15)
}

Diagnostic ::= ENUMERATED -- in RejectMCSPDUUltimatum
{
        dc-inconsistent-merge           (0),
        dc-forbidden-PDU-downward       (1),
        dc-forbidden-PDU-upward         (2),
        dc-invalid-BER-encoding         (3),
        dc-invalid-PER-encoding         (4),
        dc-misrouted-user               (5),
        dc-unrequested-confirm          (6),
```

```
        dc-wrong-transport-priority     (7),
        dc-channel-id-conflict          (8),
        dc-token-id-conflict            (9),
        dc-not-user-id-channel          (10),
        dc-too-many-channels            (11),
        dc-too-many-tokens      (12),
        dc-too-many-users               (13)
}

-- Part 10: MCSPDU repertoire

ConnectMCSPDU ::= CHOICE
{
        connect-initial                 Connect-Initial,
        connect-response                Connect-Response,
        connect-additional              Connect-Additional,
        connect-result                  Connect-Result
}

DomainMCSPDU ::= CHOICE
{
        plumbDomainIndication           PlumbDomainIndication,
        erectDomainRequest              ErectDomainRequest,
        mergeChannelsRequest            MergeChannelsRequest,
        mergeChannelsConfirm            MergeChannelsConfirm,
        purgeChannelsIndication         PurgeChannelsIndication,
        mergeTokensRequest              MergeTokensRequest,
        mergeTokensConfirm              MergeTokensConfirm,
        purgeTokensIndication           PurgeTokensIndication,
        disconnectProviderUltimatum     DisconnectProviderUltimatum,
        rejectMCSPDUUltimatum           RejectMCSPDUUltimatum,
        attachUserRequest               AttachUserRequest,
        attachUserConfirm               AttachUserConfirm,
        detachUserRequest               DetachUserRequest,
        detachUserIndication            DetachUserIndication,
        channelJoinRequest              ChannelJoinRequest,
        channelJoinConfirm              ChannelJoinConfirm,
        channelLeaveRequest             ChannelLeaveRequest,
        channelConveneRequest           ChannelConveneRequest,
        channelConveneConfirm           ChannelConveneConfirm,
        channelDisbandRequest           ChannelDisbandRequest,
        channelDisbandIndication        ChannelDisbandIndication,
        channelAdmitRequest             ChannelAdmitRequest,
        channelAdmitIndication          ChannelAdmitIndication,
        channelExpelRequest             ChannelExpelRequest,
        channelExpelIndication          ChannelExpelIndication,
        sendDataRequest                 SendDataRequest,
        sendDataIndication              SendDataIndication,
        uniformSendDataRequest          UniformSendDataRequest,
        uniformSendDataIndication       UniformSendDataIndication,
        tokenGrabRequest                TokenGrabRequest,
        tokenGrabConfirm                TokenGrabConfirm,
        tokenInhibitRequest             TokenInhibitRequest,
        tokenInhibitConfirm             TokenInhibitConfirm,
        tokenGiveRequest                TokenGiveRequest,
        tokenGiveIndication             TokenGiveIndication,
        tokenGiveResponse               TokenGiveResponse,
        tokenGiveConfirm                TokenGiveConfirm,
        tokenPleaseRequest              TokenPleaseRequest,
        tokenPleaseIndication           TokenPleaseIndication,
        tokenReleaseRequest             TokenReleaseRequest,
```

```
        tokenReleaseConfirm         TokenReleaseConfirm,
        tokenTestRequest            TokenTestRequest,
        tokenTestConfirm            TokenTestConfirm
}

END
```

8 Structure of Version 3 MCSPDUs

The structure of version 3 MCSPDUs is specified using the notation ASN.1 of Recommendation X.208. The use and significance of these MCSPDUs is further described in clauses 9 and 10.

```
MCS-PROTOCOL-3 DEFINITIONS AUTOMATIC TAGS::=

BEGIN

-- Part 1:  Fundamental MCS types

H221NonStandardIdentifier ::= OCTET STRING (SIZE (4..255))
                                                    -- First four octets shall be country
                                                    -- code and Manufacturer code, assigned
                                                    -- as specified in Annex A/H.221 for
                                                    -- NS-cap and NS-comm

Key ::= CHOICE                                      -- Identifier of a standard or non-standard object
{
        object                  OBJECT IDENTIFIER,
        h221NonStandard         H221NonStandardIdentifier
}

NonStandardParameter ::= SEQUENCE
{
        key                     Key,
        data                    OCTET STRING
}

ChannelId ::= INTEGER (0..65535)                    -- range is 16 bits

StaticChannelId ::= ChannelId (1..1000)             -- those known permanently

DynamicChannelId ::= ChannelId (1001..65535)        -- those created and deleted

UserId ::= DynamicChannelId                         -- created by Attach-User
                                                    -- deleted by Detach-User
PrivateChannelId ::= DynamicChannelId               -- created by Channel-Convene
                                                    -- deleted by Channel-Disband

AssignedChannelId ::= DynamicChannelId              -- created by Channel-Join zero
                                                    -- deleted by last Channel-Leave

TokenId ::= INTEGER (1..65535)                      -- all are known permanently

TokenStatus ::= CHOICE
{
        notInUse                NULL,
        selfGrabbed             NULL,
        otherGrabbed            NULL,
        selfInhibited           NULL,
        otherInhibited          NULL,
```

```
        selfRecipient                   NULL,
        selfGiving                      NULL,
        otherGiving                     NULL,
        ...
}

DataPriority ::= CHOICE
{
        top                             NULL,
        high                            NULL,
        medium                          NULL,
        low                             NULL,
        ...
}

Segmentation ::= BIT STRING
{
        begin                           (0),
        end                             (1)
} (SIZE (2))

-- Part 2:  Extended parameter

ExtendedParameters ::= SEQUENCE
{
        unreliableDataSupported         BOOLEAN,
        domainReferenceID               INTEGER (0 .. 65535),
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        …
}

ExtendedParameterPropose ::= SEQUENCE
{
        targetExtendedParameters            ExtendedParameters,
        minimumExtendedParameters           ExtendedParameters,
        maximumExtendedParameters           ExtendedParameters,
        nonStandard                         SEQUENCE OF NonStandardParameter OPTIONAL,
        …
}

ExtendedParameterAccept ::= SEQUENCE
{
        extendedParameters              ExtendedParameters,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        …
}

-- Part 3:  Merge domain

PlumbDomainIndication ::= SEQUENCE
{
        heightLimit                     INTEGER (0..MAX),
                                        -- a restriction on the MCSPDU receiver
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
```

```
ErectDomainRequest ::= SEQUENCE
{
        subHeight                       INTEGER (0..MAX),
                                        -- height in domain of the MCSPDU transmitter
        subInterval                     INTEGER (0..MAX),
                                        -- its throughput enforcement interval in milliseconds
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
ChannelAttributes ::= CHOICE
{
        static                          SEQUENCE
        {
                channelId                       StaticChannelId,
                                                -- joined is implicitly TRUE
                nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
        },

        userId                          SEQUENCE
        {
                joined                          BOOLEAN,
                                                -- TRUE if user is joined to its user id
                userId                          UserId,
                nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
        },

        private                         SEQUENCE
        {
                joined                          BOOLEAN,
                                                -- TRUE if channel id is joined below
                channelId                       PrivateChannelId,
                manager                         UserId,
                admitted                        SET OF UserId,
                                                -- may span multiple MergeChannelsRequest
                nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
        },

        assigned                        SEQUENCE
        {
                channelId                       AssignedChannelId,
                                                -- joined is implicitly TRUE
                nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
        },

        ...
}

MergeChannelsRequest ::= SEQUENCE
{
        mergeChannels                   SET OF ChannelAttributes,
        purgeChannelIds                 SET OF ChannelId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
```

```
MergeChannelsConfirm ::= SEQUENCE
{
        mergeChannels                   SET OF ChannelAttributes,
        purgeChannelIds                 SET OF ChannelId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

PurgeChannelsIndication ::= SEQUENCE
{
        detachChannelIds                SET OF ChannelId,
                                        -- purge user id channels
        purgeChannelIds                 SET OF ChannelId,
                                        -- purge other channels
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
TokenAttributes ::= CHOICE
{
        grabbed                 SEQUENCE
        {
                tokenId                 TokenId,
                grabber                 UserId,
                nonStandard             SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
        },

        inhibited               SEQUENCE
        {
                tokenId                 TokenId,
                inhibitors              SET OF UserId,
                                        -- may span multiple MergeTokensRequest
                nonStandard             SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
        },

        giving                  SEQUENCE
        {
                tokenId                 TokenId,
                grabber                 UserId,
                recipient               UserId,
                nonStandard             SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
        },

        ungivable               SEQUENCE
        {
                tokenId                 TokenId,
                grabber                 UserId,
                                        -- recipient has since detached
                nonStandard             SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
        },

        given                   SEQUENCE
        {
                tokenId                 TokenId,
                recipient               UserId,
                                        -- grabber released or detached
                nonStandard             SEQUENCE OF NonStandardParameter OPTIONAL,
                ...
```

```
        },

        ...
}

MergeTokensRequest ::= SEQUENCE
{
        mergeTokens                     SET OF TokenAttributes,
        purgeTokenIds                   SET OF TokenId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MergeTokensConfirm ::= SEQUENCE
{
        mergeTokens                     SET OF TokenAttributes,
        purgeTokenIds                   SET OF TokenId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

PurgeTokensIndication ::= SEQUENCE
{
        purgeTokenIds                   SET OF TokenId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

-- Part 4:  Disconnect provider

DisconnectProviderUltimatum ::= SEQUENCE
{
        reason                          Reason,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

RejectMCSPDUUltimatum ::= SEQUENCE
{
        diagnostic                      Diagnostic,
        initialOctets                   OCTET STRING,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

-- Part 5:  Attach/Detach user

AttachUserRequest ::= SEQUENCE
{
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

AttachUserConfirm ::= SEQUENCE
{
        result                          Result,
        initiator                       UserId OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
```

```
DetachUserRequest ::= SEQUENCE
{
        reason                          Reason,
        userIds                         SET OF UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

DetachUserIndication ::= SEQUENCE
{
        reason                          Reason,
        userIds                         SET OF UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

-- Part 6:  Channel management

ChannelJoinRequest ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,          -- may be zero
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelJoinConfirm ::= SEQUENCE
{
        result  Result,
        initiator                       UserId,
        requested                       ChannelId,          -- may be zero
        channelId                       ChannelId OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelLeaveRequest ::= SEQUENCE
{
        channelIds                      SET OF ChannelId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelConveneRequest ::= SEQUENCE
{
        initiator                       UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelConveneConfirm ::= SEQUENCE
{
        result                          Result,
        initiator                       UserId,
        channelId                       PrivateChannelId OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
```

```
ChannelDisbandRequest ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       PrivateChannelId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelDisbandIndication ::= SEQUENCE
{
        channelId                       PrivateChannelId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelAdmitRequest ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       PrivateChannelId,
        userIds                         SET OF UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelAdmitIndication ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       PrivateChannelId,
        userIds                         SET OF UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelExpelRequest ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       PrivateChannelId,
        userIds                         SET OF UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

ChannelExpelIndication ::= SEQUENCE
{
        channelId                       PrivateChannelId,
        userIds                         SET OF UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

-- Part 7:  Data transfer

SendDataRequest ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,
        reliability                     BOOLEAN,
        domainReferenceID               INTEGER (0 .. 65535)      OPTIONAL,
        dataPriority                    DataPriority,
        segmentation            Segmentation,
        userData                        OCTET STRING,
```

```
        totalDataSize                   INTEGER                 OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

SendDataIndication ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,
        reliability                     BOOLEAN,
        domainReferenceID               INTEGER (0 .. 65535)    OPTIONAL,
        dataPriority                    DataPriority,
        segmentation            Segmentation,
        userData                        OCTET STRING,
        totalDataSize                   INTEGER                 OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

UniformSendDataRequest ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,
        reliability                     BOOLEAN,
        domainReferenceID               INTEGER (0 .. 65535)    OPTIONAL,
        dataPriority                    DataPriority,
        segmentation            Segmentation,
        userData                        OCTET STRING,
        totalDataSize                   INTEGER                 OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

UniformSendDataIndication ::= SEQUENCE
{
        initiator                       UserId,
        channelId                       ChannelId,
        reliability                     BOOLEAN,
        domainReferenceID               INTEGER (0 .. 65535)    OPTIONAL,
        dataPriority                    DataPriority,
        segmentation            Segmentation,
        userData                        OCTET STRING,
        totalDataSize                   INTEGER                 OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

-- Part 8:  Token management

TokenGrabRequest ::= SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
```

```
TokenGrabConfirm ::= SEQUENCE
{
        result                          Result,
        initiator                       UserId,
        tokenId                         TokenId,
        tokenStatus                     TokenStatus,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenInhibitRequest ::= SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenInhibitConfirm ::= SEQUENCE
{
        result                          Result,
        initiator                       UserId,
        tokenId                         TokenId,
        tokenStatus                     TokenStatus,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenGiveRequest ::= SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId,
        recipient                       UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenGiveIndication ::= SEQUENCE
{
        initiator                       UserId,
        tokenId                         TokenId,
        recipient                       UserId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenGiveResponse ::= SEQUENCE
{
        result                          Result,
        recipient                       UserId,
        tokenId                         TokenId,
        nonStandard                     SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenGiveConfirm ::= SEQUENCE
{
        result                          Result,
        initiator                       UserId,
        tokenId                         TokenId,
        tokenStatus                     TokenStatus,
```

```
        nonStandard                 SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenPleaseRequest ::= SEQUENCE
{
        initiator                   UserId,
        tokenId                     TokenId,
        nonStandard                 SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenPleaseIndication ::= SEQUENCE
{
        initiator                   UserId,
        tokenId                     TokenId,
        nonStandard                 SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenReleaseRequest ::= SEQUENCE
{
        initiator                   UserId,
        tokenId                     TokenId,
        nonStandard                 SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenReleaseConfirm ::= SEQUENCE
{
        result                      Result,
        initiator                   UserId,
        tokenId                     TokenId,
        tokenStatus                 TokenStatus,
        nonStandard                 SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenTestRequest ::= SEQUENCE
{
        initiator                   UserId,
        tokenId                     TokenId,
        nonStandard                 SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TokenTestConfirm ::= SEQUENCE
{
        initiator                   UserId,
        tokenId                     TokenId,
        tokenStatus                 TokenStatus,
        nonStandard                 SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
-- Part 9:  Capabilities notification

CapabilityID ::= CHOICE
{
        standardID                  INTEGER (0 .. 65535),
        nonstandardID               Key
}
```

```
CapabilityClass ::= CHOICE
{
        null                            NULL,
        unsignedMin                     INTEGER (0 .. MAX),
        unsignedMax                     INTEGER (0 .. MAX)
}

ParticipationIndicator ::= CHOICE
{
        global                          NULL,
        partial                         INTEGER (1 .. 2)
}

RequestCapability ::= SEQUENCE
{
        capabilityID                    CapabilityID,
        capabilityClass                 CapabilityClass,
        participationIndicator          ParticipationIndicator,
        nonStandard                     SEQUENCE OF NonStandardParameter    OPTIONAL,
        …
}

SeqOfRequestCapabilities ::= SEQUENCE OF RequestCapability

IndicationCapability ::= SEQUENCE
{
        capabilityID                    CapabilityID,
        capabilityClass                 CapabilityClass,
        summitProviderSupported         BOOLEAN,
        intermediateNodeSupported       BOOLEAN,
        nonStandard                     SEQUENCE OF NonStandardParameter    OPTIONAL,
        …
}

SeqOfIndicationCapabilities ::= SEQUENCE OF IndicationCapability

CapabilitiesNotificationRequest ::= SEQUENCE
{
        v2NodePresent                   BOOLEAN,
        addList                         SeqOfRequestCapabilities            OPTIONAL,
        removeList                      SeqOfRequestCapabilities            OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter    OPTIONAL,
        …
}

CapabilitiesNotificationIndication ::= SEQUENCE
{
        v2NodePresent                   BOOLEAN,
        addList                         SeqOfIndicationCapabilities         OPTIONAL,
        removeList                      SeqOfIndicationCapabilities         OPTIONAL,
        nonStandard                     SEQUENCE OF NonStandardParameter    OPTIONAL,
        …
}
```

```
-- Part 10:  Status codes

Reason ::= CHOICE       -- in DisconnectProviderUltimatum, DetachUserRequest, DetachUserIndication
{
        rn-domain-disconnected          NULL,
        rn-provider-initiated           NULL,
        rn-token-purged                 NULL,
        rn-user-requested               NULL,
        rn-channel-purged               NULL,
        ...
}

Result ::= CHOICE       -- in Connect, response, confirm
{
        rt-successful                   NULL,
        rt-domain-merging               NULL,
        rt-domain-not-hierarchical      NULL,
        rt-no-such-channel              NULL,
        rt-no-such-domain               NULL,
        rt-no-such-user                 NULL,
        rt-not-admitted                 NULL,
        rt-other-user-id                NULL,
        rt-parameters-unacceptable      NULL,
        rt-token-not-available          NULL,
        rt-token-not-possessed          NULL,
        rt-too-many-channels            NULL,
        rt-too-many-tokens              NULL,
        rt-too-many-users               NULL,
        rt-unspecified-failure          NULL,
        rt-user-rejected                NULL,
        ...
}

Diagnostic ::= CHOICE   -- in RejectMCSPDUUltimatum
{
        dc-inconsistent-merge           NULL,
        dc-forbidden-PDU-downward       NULL,
        dc-forbidden-PDU-upward         NULL,
        dc-invalid-BER-encoding         NULL,
        dc-invalid-PER-encoding         NULL,
        dc-misrouted-user               NULL,
        dc-unrequested-confirm          NULL,
        dc-wrong-transport-priority     NULL,
        dc-channel-id-conflict          NULL,
        dc-token-id-conflict            NULL,
        dc-not-user-id-channel          NULL,
        dc-too-many-channels            NULL,
        dc-too-many-tokens      NULL,
        dc-too-many-users               NULL,
        ...
}

-- Part 11:  MCSPDU repertoire

NonStandardPDU ::= SEQUENCE
{
        data                            NonStandardParameter,
        ...
}
```

```
ExtendedParameterMCSPDU ::= CHOICE
{
        extendedParameterPropose        ExtendedParameterPropose,
        extendedParameterAcceptExtendedParameterAccept,
        nonStandard                     NonStandardPDU,
        …
}

DomainMCSPDU ::= CHOICE
{
        plumbDomainIndication           PlumbDomainIndication,
        erectDomainRequest              ErectDomainRequest,
        mergeChannelsRequest            MergeChannelsRequest,
        mergeChannelsConfirm            MergeChannelsConfirm,
        purgeChannelsIndication         PurgeChannelsIndication,
        mergeTokensRequest              MergeTokensRequest,
        mergeTokensConfirm              MergeTokensConfirm,
        purgeTokensIndication           PurgeTokensIndication,
        disconnectProviderUltimatum     DisconnectProviderUltimatum,
        rejectMCSPDUUltimatum           RejectMCSPDUUltimatum,
        attachUserRequest               AttachUserRequest,
        attachUserConfirm               AttachUserConfirm,
        detachUserRequest               DetachUserRequest,
        detachUserIndication            DetachUserIndication,
        channelJoinRequest              ChannelJoinRequest,
        channelJoinConfirm              ChannelJoinConfirm,
        channelLeaveRequest             ChannelLeaveRequest,
        channelConveneRequest           ChannelConveneRequest,
        channelConveneConfirm           ChannelConveneConfirm,
        channelDisbandRequest           ChannelDisbandRequest,
        channelDisbandIndication        ChannelDisbandIndication,
        channelAdmitRequest             ChannelAdmitRequest,
        channelAdmitIndication          ChannelAdmitIndication,
        channelExpelRequest             ChannelExpelRequest,
        channelExpelIndication          ChannelExpelIndication,
        sendDataRequest                 SendDataRequest,
        sendDataIndication              SendDataIndication,
        uniformSendDataRequest          UniformSendDataRequest,
        uniformSendDataIndication       UniformSendDataIndication,
        tokenGrabRequest                TokenGrabRequest,
        tokenGrabConfirm                TokenGrabConfirm,
        tokenInhibitRequest             TokenInhibitRequest,
        tokenInhibitConfirm             TokenInhibitConfirm,
        tokenGiveRequest                TokenGiveRequest,
        tokenGiveIndication             TokenGiveIndication,
        tokenGiveResponse               TokenGiveResponse,
        tokenGiveConfirm                TokenGiveConfirm,
        tokenPleaseRequest              TokenPleaseRequest,
        tokenPleaseIndication           TokenPleaseIndication,
        tokenReleaseRequest             TokenReleaseRequest,
        tokenReleaseConfirm             TokenReleaseConfirm,
        tokenTestRequest                TokenTestRequest,
        tokenTestConfirm                TokenTestConfirm,
        nonStandard                     NonStandardPDU,
        ...
}

END
```

9 Encoding of MCSPDUs

Each MCSPDU is transported as one TSDU across a TC belonging to an MCS connection. Connect MCSPDUs are unlimited in size. Domain MCSPDUs are limited in size by a parameter of the MCS domain.

A standard ASN.1 data value encoding is used to transfer MCSPDUs between peer MCS providers. Encoding rules are selected as part of protocol version negotiation. This negotiation involves the exchange of a **Connect-Initial** and a **Connect-Response** MCSPDU over the initial TC. Three versions of this protocol are defined:

– *Version 1* – All MCSPDUs are described in clause 7 and use the Basic Encoding Rules of Recommendation X.690.

– *Version 2* – All MCSPDUs are described in clause 7. This version uses Basic Encoding Rules for connect MCSPDUs and the Packed Encoding Rules of Recommendation X.691 for all subsequent domain MCSPDUs. Specifically, the ALIGNED variant of BASIC-PER shall be applied to the ASN.1 type **DomainMCSPDU**. The bit string produced shall be conveyed as an integral number of octets. The leading bit of this string shall coincide with the most significant bit of the first octet.

– *Version 3* – Connect MCSPDUs are described in clause 7 and use Basic Encoding Rules for backward compatibility with Version 2. Extended Parameters and Domain MCSPDUs are described in clause 8 and use Packed Encoding Rules as specified for Version 2, above. Version 3 enabled nodes must be able to translate version 3 protocol into version 2 protocol for transmission over a connection to a node which has specified version 2 protocol in the domain parameters. This translation is described in 13.11.

NOTE 1 – The Packed Encoding Rules yield more compact MCSPDU headers.

NOTE 2 – Both BER and PER are self-delimiting, in the sense that they contain enough information to locate the end of each encoded MCSPDU. It might be argued that this makes the use of TSDUs unnecessary and that this protocol could be implemented over non-standard transport services that convey octet streams without preserving TSDU boundaries. However, such an approach is more vulnerable to implementation errors. If the boundary between MCSPDUs were ever lost, recovery would be difficult.

10 Routing of MCSPDUs

10.1 Connect and extended parameters MCSPDUs

Figures 10-1 and 10-2 specify the exchange of connect and extended parameters MCSPDUs (the number and relative order of additional TCs may vary).

With the receipt of a **Connect-Response**, the calling MCS provider will determine if the new connection is between two members of the V3 summit by examining the negotiated version number. If so, it will initiate the exchange of the **Extended-Parameters** MCSPDUs. If either node is not a member of the V3 summit, the calling provider will not issue an **Extended-Parameters-Propose**, but will proceed with **Connect-Additional** MCSPDUs.

After receiving the **Connect-Response**, the calling MCS provider also learns the negotiated value for the number of data transfer priorities implemented in the domain. For illustration, connect MCSPDUs on additional TCs are shown obeying the strict sequence 2, 3, 4 at the called MCS provider. In reality, transport connections may not be established in the same order that they are requested. Indeed, a **Connect-Additional** may arrive in a different order than it was sent, causing a **Connect-Result** to be returned out of sequence too. Or the latter, even if sent in sequence, may still be reordered in transit. A calling MCS provider need not wait for all additional TCs to be established

before sending the first **Connect-Additional**. The called MCS provider need not wait for a full set of **Connect-Additional** MCSPDUs to arrive before returning the first **Connect-Result**. Receipt of a full set of successful results at the calling MCS provider, in whatever order, generates a successful MCS-CONNECT-PROVIDER confirm.

An unsuccessful **Connect-Response** or **Connect-Result** or a T-DISCONNECT indication at some intermediate point shall cause all TCs belonging to the MCS connection so far to be disconnected and shall generate an unsuccessful MCS-CONNECT-PROVIDER confirm.

An MCS-CONNECT-PROVIDER request specifies which of two MCS providers is higher than the other. This hierarchical relationship determines the subsequent routing of domain MCSPDUs, and the distinction between calling and called MCS provider is thereafter irrelevant. For example, a next step is for the MCS layer to merge the resources of two previously independent domains. **MergeChannelsRequest** and **MergeTokensRequest** are generated by the lower MCS provider and are transmitted across the new MCS connection to the higher MCS provider. The direction in which these MCSPDUs are sent could be either calling-to-called or called-to-calling, depending on how the upward flag was set.

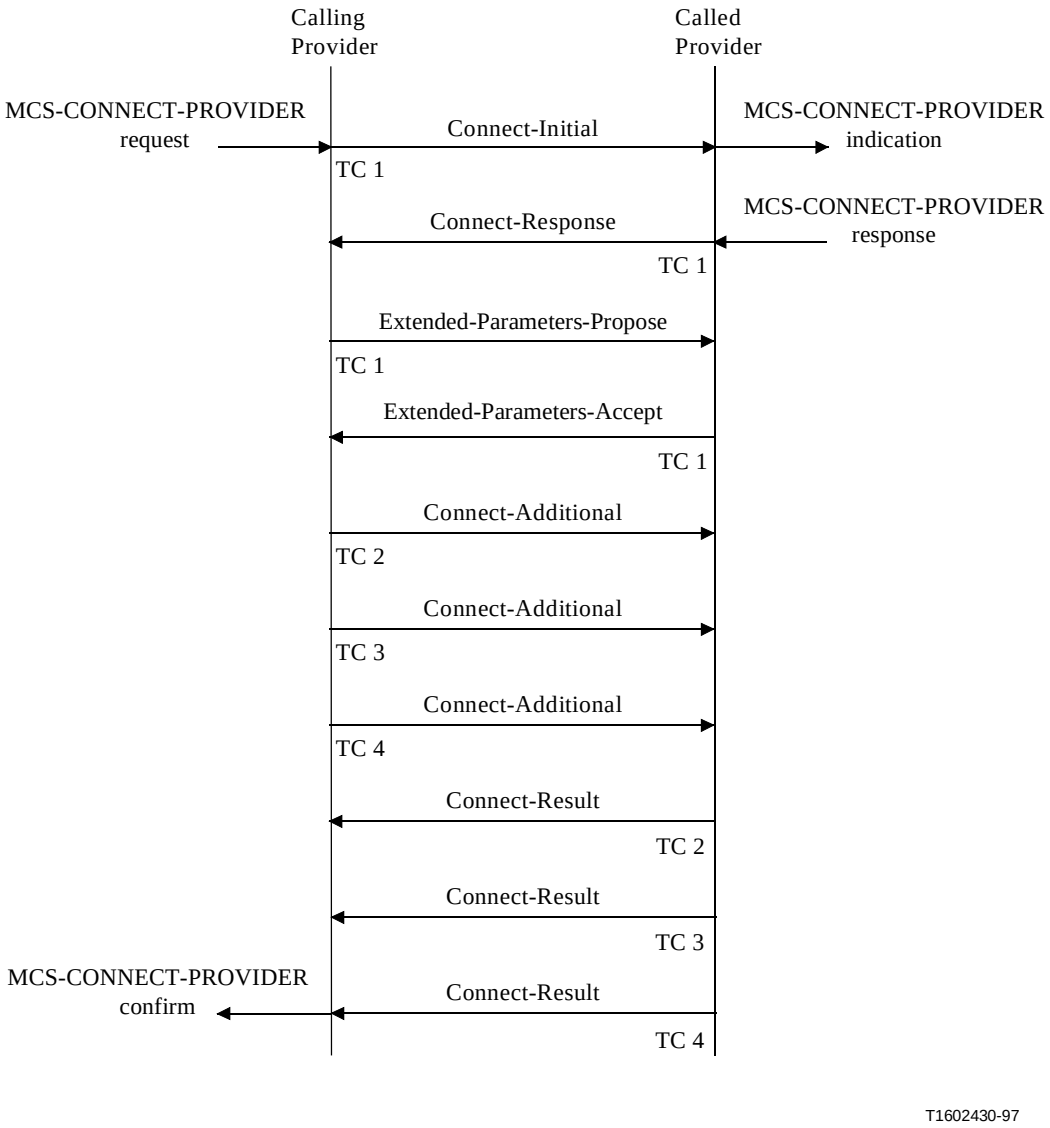

Figure 10-1/T.125 – Message flow of connect and extended parameters MCSPDUs between member nodes of the V3 Summit

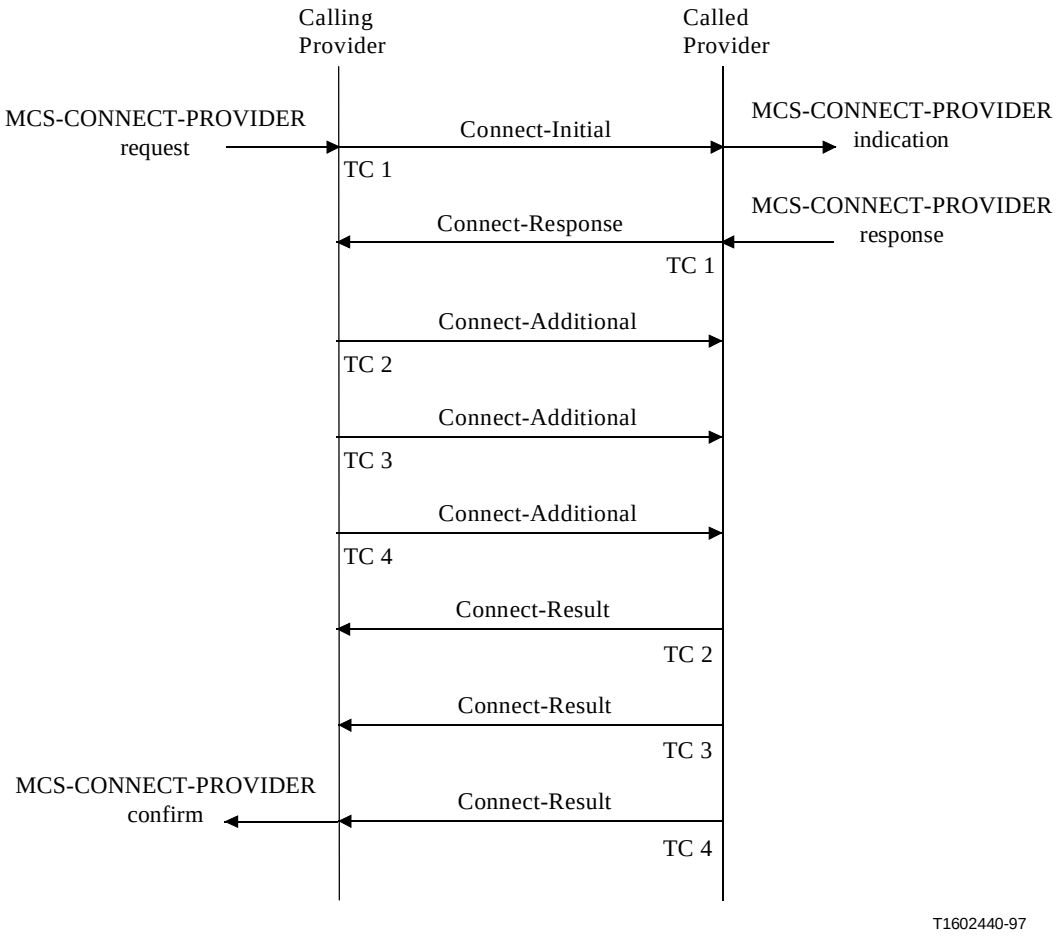

Figure 10-2/T.125 – Message flow of connect MCSPDUs between nodes that are not both members of the V3 Summit

10.2 Domain MCSPDUs

Table 10-1 specifies the routing of domain MCSPDUs.

If an MCS provider generates or forwards an MCSPDU of category *request*, it travels over the unique MCS connection upward. **ErectDomainRequest**, **ChannelJoinRequest**, **ChannelLeaveRequest**, and **CapabilitiesNotificationRequest** may be consumed at some intermediate MCS provider. Other requests rise to be acted on by the top MCS provider, unless the content of a request is determined to be invalid, in which case its MCSPDU may be ignored, without confirmation.

Table 10-1/T.125 – Routing of domain MCSPDUs

Category	MCSPDUs	TC	Direction
Request	**ErectDomainRequest** **MergeChannelsRequest** **MergeTokensRequest** **AttachUserRequest** **DetachUserRequest** **ChannelJoinRequest** **ChannelLeaveRequest** **ChannelConveneRequest** **ChannelDisbandRequest** **ChannelAdmitRequest** **ChannelExpelRequest** **TokenGrabRequest** **TokenInhibitRequest** **TokenGiveRequest** **TokenPleaseRequest** **TokenReleaseRequest** **TokenTestRequest** **CapabilitiesNotificationRequest**	I	Up
	SendDataRequest **UniformSendDataRequest**	A	
Indication	**PlumbDomainIndication** **PurgeChannelsIndication** **PurgeTokensIndication** **DetachUserIndication** **ChannelDisbandIndication** **ChannelAdmitIndication** **ChannelExpelIndication** **TokenGiveIndication** **TokenPleaseIndication** **CapabilitiesNotificationIndication**	I	Down
	SendDataIndication **UniformSendDataIndication**	A	

If an MCS provider generates or forwards an MCSPDU of category *indication*, copies of it, possibly amended in content, travel over zero or more MCS connections downward, according to the following rules:

a) **PlumbDomainIndication** is sent on all MCS connections downward. The height limit it contains is decremented by one. An MCS provider receiving this MCSPDU with a height limit of zero shall disconnect.

b) **PurgeChannelsIndication** is sent on all MCS connections downward. The set of user ids forwarded is unchanged, so that all detached users will be announced to those that remain. The set of other channel ids forwarded may be restricted to those in use in a subtree. At a former top provider that is still merging into an upper domain, both user ids and other channel ids are restricted to those whose acceptance into the upper domain has been confirmed.

c) **PurgeTokensIndication** is sent on all MCS connections downward. The set of token ids forwarded may be restricted to those in use in a subtree. At a former top provider that is still merging into an upper domain, token ids are restricted to those whose acceptance into the upper domain has been confirmed.

d) **DetachUserIndication** is sent on all MCS connections downward. The set of user ids forwarded is unchanged, so that all detached users will be announced to those that remain. At a former top provider that is still merging into an upper domain, user ids are restricted to those whose acceptance into the upper domain has been confirmed.

e) **ChannelDisbandIndication** is sent on all MCS connections downward that contain in their subtree the manager of the private channel or any admitted user.

f) **ChannelAdmitIndication** and **ChannelExpelIndication** are sent on all MCS connections downward that contain one or more of the affected users in their subtree. The set of user ids forwarded may be restricted to those residing in a subtree.

g) **TokenGiveIndication** is sent on a single MCS connection downward that contains the designated recipient in its subtree.

h) **TokenPleaseIndication** is sent on all MCS connections downward that contain in their subtree a user who has grabbed, inhibited, or is being given the token.

i) **CapabilitiesNotificationIndication** is sent on all MCS connections downward in the V3 summit. Each node receiving this MCSPDU alters its local copy of the V3 summit capabilities list based on the contents of the Add List and Remove List.

j) **SendDataIndication** and **UniformSendDataIndication** are sent on all MCS connections downward by which the specified channel is joined, except that when **SendDataIndication** is generated, it is not sent back on the connection by which **SendDataRequest** arrived.

Indications **PurgeChannelsIndication**, **PurgeTokensIndication**, **DetachUserIndication**, **ChannelAdmitIndication**, and **ChannelExpelIndication** need not be forwarded if the sets of ids they contain are empty.

If an MCS provider generates or forwards an MCSPDU of category *response*, it travels over the unique MCS connection upward. It rises to be acted on by the top MCS provider, unless its content is determined to be invalid.

If an MCS provider generates or forwards an MCSPDU of category *confirm*, it travels over a single MCS connection downward, according to the following rules.

a) **MergeChannelsConfirm** retraces, in the opposite direction, the path of the earliest **MergeChannelsRequest** that has not yet been answered by a confirm. This requires each MCS provider to maintain a first-in first-out queue of pending requests.

b) **MergeTokensConfirm** retraces, in the opposite direction, the path of the earliest **MergeTokensRequest** that has not yet been answered by a confirm. This requires each MCS provider to maintain a first-in first-out queue of pending requests.

c) **AttachUserConfirm** retraces, in the opposite direction, the path of an earlier **AttachUserRequest** that has not yet been answered by a confirm. It is not critical which **AttachUserRequest**, if more than one is pending, but to be fair each MCS provider should maintain a first-in first-out queue. Upon sending **AttachUserConfirm**, an MCS provider shall record to which subtree the user id it contains is thereby being assigned.

d) Other MCSPDUs of category confirm contain an initiator user id that was previously assigned through the action of **AttachUserConfirm** as just explained. These MCSPDUs are sent on the MCS connection downward that leads to the subtree where the user id was assigned. Continuing in this way, they eventually return to the provider that hosts the requesting MCS attachment.

Confirms are generated in the course of processing like requests. All but **ChannelJoinConfirm** are generated by the top MCS provider.

If an MCS provider generates an MCSPDU of category *ultimatum*, it travels over a single MCS connection, either upward or downward. **DisconnectProviderUltimatum** commands the receiving MCS provider to disconnect the MCS connection that conveys it. **RejectMCSPDUUltimatum** rejects an erroneous MCSPDU with a diagnostic code and invites the MCS provider that transmitted it to disconnect. Ultimatums are not forwarded.

11 Meaning of MCSPDUs

Tables 11-1 through 11-51 reiterate the contents of individual MCSPDUs as defined in clause 8.

11.1 Connect-Initial

Connect-Initial is generated by an MCS-CONNECT-PROVIDER request. It is sent as the first TSDU over the initial TC of a new MCS connection. At the receiver, it generates an MCS-CONNECT-PROVIDER indication.

Table 11-1/T.125 – Connect-Initial MCSPDU

Contents	Source	Sink
Calling Domain Selector	Request	Indication
Called Domain Selector	Request	Indication
Upward Flag	Request	Indication
Target Domain Parameters	Request	Indication
Minimum Domain Parameters	Request	Indication
Maximum Domain Parameters	Request	Indication
User Data	Request	Indication

Calling transport address and called transport address are additional parameters of the MCS-CONNECT-PROVIDER request and indication. They become parameters of T-CONNECT and are not passed explicitly in any MCSPDU. The same pair of transport addresses shall be used to request all TCs belonging to the same MCS connection.

Transport quality of service is an additional parameter of the MCS-CONNECT-PROVIDER request but not of the indication. Quality of service can vary from one TC to another, and the quality available is only disclosed in the process of establishing a given TC. Since the number of additional

TCs needed is not known until MCS-CONNECT-PROVIDER has negotiated the domain parameter for number of data priorities implemented, this primitive cannot at the same time fully negotiate their transport quality of service. A called MCS provider shall therefore accept incoming TCs automatically at the offered quality of service, so long as this meets any minimum optionally specified by the calling MCS provider and indicated through each individual T-CONNECT.

The interpretation of domain selector values is a local matter for each MCS provider. These are octet strings that have the characteristics of an address. Acceptable values may be determined through the process of configuring an MCS provider. More than one value may select the same domain. An unspecified domain selector is an octet string of length zero. This may be resolved, through local convention, to some explicit value.

The upward flag specifies the direction of a new MCS connection: true if the called provider is to be considered higher than the calling provider and false otherwise. An MCS provider plays a role in the hierarchy of a domain based on the direction of MCS connections in which it participates. No provider shall allow two connections to higher providers. A provider without a connection to a higher provider shall act as top MCS provider.

The target domain parameters of **Connect-Initial** individually shall lie between the minimum and maximum values specified. An MCS provider shall revise the requested domain parameters to reflect limits of its implementation or to impose values already agreed among the existing members of a domain. It may increase the minimums and decrease the maximums. It shall change targets only to keep them within the interval. An MCS provider should be prepared to honour any response that falls within the range of values it proposes.

User data is an arbitrary octet string. It may have length zero.

An MCS provider automatically accepts each incoming TC to the limit of its capacity. User data in T-CONNECT is unused. The first TSDU received in data transfer, being either a **Connect-Initial** or a **Connect-Additional** MCSPDU, determines the nature of the TC. If the content is unacceptable, a called MCS provider may disconnect the TC immediately. The preferred reaction is to return a **Connect-Response** or **Connect-Result**, as the case may be, explaining why the MCS connection failed. The calling MCS provider shall then disconnect.

11.2 Connect-Response

Connect-Response is generated by an MCS-CONNECT-PROVIDER response. It is the first TSDU sent in reverse over the initial TC of a new MCS connection. It conveys the acceptance of an MCS connection to the calling MCS provider, which then proceeds to establish any additional TCs required.

Table 11-2/T.125 – Connect-Response MCSPDU

Contents	Source	Sink
Result	Response	Confirm
Domain Parameters	Response	Confirm
Called Connect Id	Called provider	Calling provider
User Data	Response	Confirm

If the result is successful, this MCSPDU fixes the domain parameters in effect. Among these is the number of MCS data transfer priorities implemented, equal to the number of TCs in an MCS

connection. If this exceeds one, additional TCs shall be created and bound to the MCS connection through the exchange of **Connect-Additional** and **Connect-Result** MCSPDUs.

The called connect id serves as the means for associating additional incoming TCs at the called MCS provider with this initial TC. Its value is chosen for this purpose alone. It shall uniquely identify one MCS connection in progress at the called provider. This id has no lasting significance following the completion of MCS-CONNECT-PROVIDER.

Most of the parameters of MCS-CONNECT-PROVIDER confirm are conveyed in **Connect-Response**. If the result is unsuccessful or no additional TCs are needed, confirm is generated immediately. Otherwise, it is deferred until the results are known for binding additional TCs to the MCS connection.

11.3 Connect-Additional

If the calling MCS provider determines that both providers are members of the V3 summit, **Connect-Additional** is generated following receipt of **Extended-Parameters-Accept**, otherwise it will be generated following receipt of **Connect-Response**. It is sent as the first TSDU over an additional TC of a new MCS connection.

Data priority takes on the values *high*, *medium*, and *low* in sequence, up to the number of additional TCs required.

Table 11-3/T.125 – Connect-Additional MCSPDU

Contents	Source	Sink
Called Connect Id	Calling provider	Called provider
Data Priority	Calling provider	Called provider

11.4 Connect-Result

Connect-Result□ is generated following receipt of **Connect-Additional**. It is the first TSDU sent in reverse over an additional TC of a new MCS connection.

Table 11-4/T.125 – Connect-Result MCSPDU

Contents	Source	Sink
Result	Called provider	Confirm

If any result is unsuccessful, MCS-CONNECT-PROVIDER confirm shall be generated immediately. All TCs associated with the MCS connection shall be disconnected and any MCSPDUs they convey shall be ignored.

Otherwise, successful results shall be awaited for each additional TC. These may return out-of-sequence. When all have been collected, a successful MCS-CONNECT-PROVIDER confirm shall be generated.

After MCS-CONNECT-PROVIDER confirm, domain MCSPDUs may flow across an MCS connection. Each MCS connection belongs to a single domain. In configurations where an MCS provider hosts more than one domain, the MCS connection that carries MCSPDUs determines which domain they apply to. The descriptions of domain MCSPDUs that occupy the remainder of this subclause are set in the context of a single domain.

11.5 Extended-Parameters-Propose

Following the receipt of **Connect-Response**, if the calling MCS provider determines that both providers are members of the V3 summit, **Extended-Parameters-Propose** is generated. It is sent over the initial TC of a new MCS connection.

Table 11-5/T.125 – Extended-Parameters-Propose MCSPDU

Contents	Source	Sink
Target Extended Parameters	Calling provider	Called provider
Minimum Extended Parameters	Calling provider	Called provider
Maximum Extended Parameters	Calling provider	Called provider

For extended parameters with a range of possible values, the target parameters of **Extended-Parameters-Propose** individually shall lie between the minimum and maximum values specified. An MCS provider shall revise the requested extended parameters to reflect limits of its implementation or to impose values already agreed among the existing members of a domain. It may increase the minimums and decrease the maximums. It shall change targets only to keep them within the interval. An MCS provider should be prepared to honour any accepted target that falls within the range of values it proposes. If the extended parameters are unacceptable, a called MCS provider will disconnect the TC immediately.

For the *Unreliable data is supported* value, if the calling provider wishes to allow the value to be negotiated, the Minimum value will be False, the Maximum value will be True and the target value will indicate the caller's preference. If support of unreliable data is nonnegotiable, Target, Minimum and Maximum will be set to the callers preference.

The value for *Domain Reference Identifier* will be identical for Target, Minimum and Maximum and will uniquely identify this domain to the calling provider. The called provider may use this value to refer to this domain to the calling provider.

11.6 Extended-Parameters-Accept

Extended-Parameters-Accept is generated following receipt of **Extended-Parameters-Propose**. It conveys the acceptance of the extended parameter values to the calling MCS provider, which then proceeds to establish any additional TCs required.

Table 11-6/T.125 – Extended-Parameters-Accept MCSPDU

Contents	Source	Sink
Extended Parameters	Called provider	Calling provider

The value for *Domain Reference Identifier* will be identical for Target, Minimum and Maximum and will uniquely identify this domain to the called provider. The calling provider may use this value to refer to this domain to the called provider.

11.7 PlumbDomainIndication

PlumbDomainIndication is generated following the successful completion of MCS-CONNECT-PROVIDER. It plumbs the hierarchy of MCS providers below a new MCS connection to ensure that

no cycle has been created. **PlumbDomainIndication** is also generated by the top MCS provider to enforce the maximum height of a domain.

Table 11-7/T.125 – PlumbDomainIndication MCSPDU

Contents	Source	Sink
Height Limit	Former top or top	Subordinates

PlumbDomainIndication is generated at the lower end of a new MCS connection, by the provider that has ceased to be top of a domain. Its content is initialized to the domain parameter for the maximum height of the domain. **PlumbDomainIndication** is transmitted over all MCS connections downward.

As needed, **PlumbDomainIndication** is generated in the same way at the top MCS provider.

Wherever **PlumbDomainIndication** is received, the height limit it contains is inspected. If greater than zero, the limit is decremented by one and **PlumbDomainIndication** is forwarded over all MCS connections downward. A value of zero, on the other hand, means that the receiver lies too far from the top provider. It shall react by disconnecting the MCS connection upward. This deletes an entire subtree and helps to repair the height of the domain.

In the presence of a cycle, the height limit of **PlumbDomainIndication** must decrease until it reaches zero. The provider that detects this will break the cycle, at the expense of deleting all providers in the cycle and their subordinates from the domain.

NOTE – An MCS provider, with purely local knowledge of MCS connections, cannot prevent the creation of cycles. It can ensure that there is at most one connection upward at all times, but it cannot ensure that the upward connection does not loop back to some provider below. When a cycle is created, the immediate cause is a faulty upward connection from the top MCS provider. Controller applications, which specify the MCS connections to be created, must strive to avoid such errors.

11.8 ErectDomainRequest

ErectDomainRequest is generated following the successful completion of MCS-CONNECT-PROVIDER. It communicates upward changes in the height of providers and their throughput enforcement intervals. **ErectDomainRequest** is generated by an MCS provider whenever its height or interval changes.

The height of an MCS provider may change when an MCS connection is added or dropped or when a subordinate provider reports a change through **ErectDomainRequest**. Its monitoring interval to enforce the minimum throughput specified as a domain parameter may change by adapting to the intervals reported by subordinates or for other reasons. If either value changes, an MCS provider shall transmit **ErectDomainRequest** to its immediate superior.

Table 11-8/T.125 – ErectDomainRequest MCSPDU

Contents	Source	Sink
Height in Domain	Subordinate	Higher provider
Throughput Enforcement Interval	Subordinate	Higher provider

11.9 MergeChannelsRequest

MergeChannelsRequest is generated following the successful completion of MCS-CONNECT-PROVIDER. It communicates upward the attributes of channels held by a former top provider so that they may be incorporated into the merged domain.

Table 11-9/T.125 – MergeChannelsRequest MCSPDU

Contents	Source	Sink
Merge Channels	Former top	Top provider
Purge Channel Ids	Intermediates	Top provider

MergeChannelsRequest may be filled with the attributes of multiple channels, up to the domain limit on MCSPDU size. As detailed in the ASN.1 definitions of clause 7, each of the four kinds of channel in use (static, user id, private, assigned) has its relevant set of attributes. These are held in the information base of the top MCS provider and are partially replicated into the subtrees where a channel is used. When two domains are merged, through the action of MCS-CONNECT-PROVIDER, channels that are in use in the lower domain must either be incorporated into the information base of the upper domain or be purged from the lower domain. This decision rests with the top provider of the merged domain.

Each channel shall be considered individually. If the domain limits on channels in use allow it and the channel id has not already been put to some conflicting use, the upper domain shall expand to include it. The use of a static channel id is never a conflict. The use of a private channel id in the lower domain is not a conflict if it is also used as a private channel id in the upper domain and has the same user id as manager. All other combinations of simultaneous use are disallowed, requiring the channel id to be purged from the lower domain.

If a private channel has a large set of admitted users, its attributes may not fit into a single **MergeChannelsRequest** and shall be sent upward in multiple MCSPDUs. However, the second and succeeding requests to merge the same private channel shall be delayed until a **MergeChannelsConfirm** has been received in reply to the first. Only then is it known if domain limits have allowed the channel to be put into use in the upper domain. If the first request fails, it shall not be repeated with a remaining subset of admitted users.

Each **MergeChannelsRequest** elicits a **MergeChannelsConfirm** reply from the top MCS provider, in the same sequence. A **MergeChannelsConfirm** contains nothing that explicitly identifies the preceding **MergeChannelsRequest**. Replies shall be routed solely by the order in which these MCSPDUs are received. MCS providers above the former top shall make a record of each unanswered **MergeChannelsRequest** and whence it arrived, so that the corresponding **MergeChannelsConfirm** may be returned via the same MCS connection.

Intermediate MCS providers shall validate the user ids proclaimed in private channel attributes, to ensure that they are legitimately assigned to the subtree where the **MergeChannelsRequest** originates. Invalid user ids shall be deleted. If the manager of a private channel is deleted, all of the channel attributes shall be deleted from the merge request and the channel id alone shall be included in the set to be purged. Except for this validation of user ids, intermediate MCS providers shall not modify the contents of a **MergeChannelsRequest**.

A former top provider shall await individual confirmation that all user ids and all token ids have been incorporated into a merged domain or purged before it begins to submit static, assigned, or private channel attributes for merger.

11.10 MergeChannelsConfirm

MergeChannelsConfirm replies to a preceding **MergeChannelsRequest**. It reflects the same set of channel ids and a subset of the attributes. Channel attributes not incorporated into the merged domain are reported as channel ids to be purged.

Accepted channel ids are reflected with the attributes that were entered into the information base of the top MCS provider. Intermediate providers shall update their information base to conform.

Table 11-10/T.125 – MergeChannelsConfirm MCSPDU

Contents	Source	Sink
Merge Channels	Top provider	Intermediates
Purge Channel Ids	Top provider	Former top

Channels to be purged from the lower domain are listed by id only. If the same channel ids are used in the upper domain, they are left undisturbed. Intermediate providers shall forward purged channel ids without acting on them.

MCS providers shall route **MergeChannelsConfirm** to the source of the antecedent **MergeChannelsRequest**, using the knowledge that there is a one-to-one reply. **MergeChannelsConfirm** returns to the former top provider that generated **MergeChannelsRequest**. There the merged channels may be ignored, as they have remained in the information base pending a reply. Purged channel ids shall be deleted as they are for **PurgeChannelsIndication**.

Intermediate MCS providers shall confirm that user ids proclaimed in private channel attributes are assigned to the subtree to which **MergeChannelsConfirm** is routed. If a private channel manager has been detached and reassigned elsewhere in the time since the antecedent **MergeChannelsRequest** was validated, an intermediate provider shall generate a **ChannelDisbandRequest** making the private channel a casualty of domain merger and shall move the channel id into the purged set. If any admitted users have been reassigned elsewhere, it shall exclude them from the channel.

11.11 PurgeChannelsIndication

PurgeChannelsIndication is generated at a former top provider following receipt of **MergeChannelsConfirm**. It is broadcast downward and purges the use of specified channel ids from subordinate providers.

Table 11-11/T.125 – PurgeChannelsIndication MCSPDU

Contents	Source	Sink
Detach User Ids	Former top	Subordinates
Purge channel Ids	Former top	Subordinates

Depending on the current use of a channel id, the effect of purging it is: MCS-DETACH-USER indication to all users if a user id channel; MCS-CHANNEL-LEAVE indication to the joined users if a static or assigned channel id; MCS-CHANNEL-DISBAND indication to the manager and MCS-CHANNEL-EXPEL indication to the admitted users if a private channel id.

A former top provider, knowing the use of all channels in its lower domain, can generate the proper indications from channel ids alone. Its subordinates, however, may have only partial knowledge. They must be told which channel ids represent detached users, for which an indication is always generated, and which represent other kinds of channels, for which an indication is generated only if the channel is in use at the subordinate provider. Therefore, **PurgeChannelsIndication** divides the channel ids to be purged into these two categories.

The purging of a user id through **PurgeChannelsIndication** shall have the same consequences as deletion through **DetachUserIndication**, except that, since the receiver of **PurgeChannelsIndication** is no longer top provider, it need not generate **ChannelDisbandIndication** or **TokenGiveConfirm** as a side effect.

NOTE – A provider may receive in **PurgeChannelsIndication** static and assigned channel ids to which it is not joined or private channel ids for which its attachments are neither managers nor admitted users. Records of use maintained in the information base allow a provider to suppress primitive indications for such channel ids.

11.12 MergeTokensRequest

MergeTokensRequest is generated following the successful completion of MCS-CONNECT-PROVIDER. It communicates upward the attributes of tokens held by a former top provider so that they may be incorporated into the merged domain.

Table 11-12/T.125 – MergeTokensRequest MCSPDU

Contents	Source	Sink
Merge Tokens	Former top	Top provider
Purge Token Ids	Intermediates	Top provider

MergeTokensRequest may be filled with the attributes of multiple tokens, up to the domain limit on MCSPDU size. As detailed in the ASN.1 definitions of clause 7, each state of a token in use (grabbed, inhibited, giving, ungiveable, given) has its relevant set of attributes. These are held in the information base of the top MCS provider and are partially replicated into the subtrees where a token is used. When two domains are merged, through the action of MCS-CONNECT-PROVIDER, tokens that are in use in the lower domain must either be incorporated into the information base of the upper domain or be purged from the lower domain. This decision rests with the top provider of the merged domain.

Each token shall be considered individually. If the domain limits on tokens in use allow it and the token id has not already been put to some conflicting use, the upper domain shall expand to include it. The inhibiting of a token id in the lower domain is not a conflict if it is also inhibited in the upper domain. All other combinations of simultaneous use are disallowed, requiring the token id to be purged from the lower domain.

If a token has a large set of inhibiting users, its attributes may not fit into a single **MergeTokensRequest** and shall be sent upward in multiple MCSPDUs. However, the second and succeeding requests for the same inhibited token shall be delayed until a **MergeTokensConfirm** has been received in reply to the first. Only then is it known if domain limits have allowed the token to be put into use in the upper domain. If the first request fails, it shall not be repeated with a remaining subset of inhibitors.

Each **MergeTokensRequest** elicits a **MergeTokensConfirm** reply from the top MCS provider, in the same sequence. A **MergeTokensConfirm** contains nothing that explicitly identifies the

preceding **MergeTokensRequest**. Replies shall be routed solely by the order in which these MCSPDUs are received. MCS providers above the former top shall make a record of each unanswered **MergeTokensRequest** and whence it arrived, so that the corresponding **MergeTokensConfirm** may be returned via the same MCS connection.

Intermediate MCS providers shall validate the user ids proclaimed in token attributes, to ensure that they are legitimately assigned to the subtree where the **MergeTokensRequest** originates. Invalid user ids shall be deleted. A token being given shall remain grabbed if either the grabber or the recipient, but not both, is deleted. If a token becomes released through this deletion of user ids, all of its attributes shall be deleted from the merge request and the token id alone shall be included in the set to be purged. An inhibited token shall remain inhibited in **MergeTokensRequest** even if all inhibitors are deleted, leaving an empty set in the attributes, because inhibitors may survive from other MCSPDUs. Except for this validation of user ids, intermediate MCS providers shall not modify the contents of an **MergeTokensRequest**.

A former top provider shall await individual confirmation that all user ids have been incorporated into a merged domain or purged before it begins to submit token attributes for merger.

11.13 MergeTokensConfirm

MergeTokensConfirm replies to a preceding **MergeTokensRequest**. It reflects the same set of token ids and a subset of the attributes. Token attributes not incorporated into the merged domain are reported as token ids to be purged.

Table 11-13/T.125 – MergeTokensConfirm MCSPDU

Contents	Source	Sink
Merge Tokens	Top provider	Intermediates
Purge Token Ids	Top provider	Former top

Accepted token ids are reflected with the attributes that have been entered into the information base at the top MCS provider. Intermediate providers shall update their information base to conform.

Tokens to be purged from the lower domain are listed by id only. If the same token ids are used in the upper domain, they are left undisturbed. Intermediate providers shall forward purged token ids without acting on them.

MCS providers shall route **MergeTokensConfirm** to the source of the antecedent **MergeTokensRequest**, using the knowledge that there is a one-to-one reply. **MergeTokensConfirm** returns to the former top provider that generated **MergeTokensRequest**. There the merged tokens may be ignored, as they have remained in the information base pending a reply. Purged token ids shall be deleted as they are for **PurgeTokensIndication**.

Intermediate MCS providers shall confirm that the user ids proclaimed in token attributes are assigned to the subtree to which **MergeTokensConfirm** is routed. If any user ids have been detached and reassigned elsewhere in the time since the antecedent **MergeTokensRequest** was validated, an intermediate provider shall generate for them a **DetachUserRequest** with reason code *channel purged*, making them casualties of domain merger. If a non-inhibited token id becomes released through this deletion of user ids, it shall be moved into the purged set.

11.14 PurgeTokensIndication

PurgeTokensIndication is generated at a former top provider following receipt of **MergeTokensConfirm**. It is broadcast downward and purges the use of specified token ids from subordinate providers.

Table 11-14/T.125 – PurgeTokensIndication MCSPDU

Contents	Source	Sink
Purge Token Ids	Former top	Subordinates

The effect of purging a token is severe: MCS-DETACH-USER indication to any user who has grabbed, inhibited, or is being given one of the token ids. A provider shall implement this by generating **DetachUserRequest** on behalf of the affected users with reason *token purged*.

NOTE – It is anticipated that Recommendation T.122 may be revised in the future to allow MCS-TOKEN-RELEASE indication in this situation. This would allow the affected user to remain attached even though its right to use the token is withdrawn.

11.15 DisconnectProviderUltimatum

DisconnectProviderUltimatum is generated by an MCS-DISCONNECT-PROVIDER request. In turn, it generates an MCS-DISCONNECT-PROVIDER indication at the other end of an MCS connection. **DisconnectProviderUltimatum** compels the receiver to disconnect the MCS connection that conveyed it.

DisconnectProviderUltimatum may also be generated by an MCS provider when it detects an error condition like the existence of a cycle in the domain hierarchy. In such cases, the reason is other than *user-requested*.

Table 11-15/T.125 – DisconnectProviderUltimatum MCSPDU

Contents	Source	Sink
Reason	Requesting provider	Indication

11.16 RejectMCSPDUUltimatum

RejectMCSPDUUltimatum is generated when an MCS provider receives an invalid MCSPDU or detects an MCS protocol error. It invites the peer provider at the other end of an MCS connection to disconnect, since recovery is uncertain from a situation that should not occur.

RejectMCSPDUUltimatum diagnoses the error and returns an initial portion of the offending TSDU, typically as many octets as will fit in the maximum size MCSPDU. The receiving provider has the option to disconnect or to persevere.

Table 11-16/T.125 – RejectMCSPDUUltimatum MCSPDU

Contents	Source	Sink
Diagnostic	Rejecting provider	Rejected provider
Initial Octets	Rejecting provider	Rejected provider

11.17 AttachUserRequest

AttachUserRequest is generated by an MCS-ATTACH-USER request. It rises to the top MCS provider, which returns an **AttachUserConfirm** reply. If the domain limit on number of user ids allows, a new user id is generated.

Table 11-17/T.125 – AttachUserRequest MCSPDU

Contents	Source	Sink
(None)	–	–

AttachUserRequest contains no information other than its MCSPDU type. The domain to which the user attaches is determined by the MCS connection conveying the MCSPDU. The only initial characteristic of the user id generated is its uniqueness.

An MCS provider shall make a record of each unanswered **AttachUserRequest** received and by which MCS connection it arrived, so that a replying **AttachUserConfirm** can be routed back to the same source. To distribute replies fairly, each provider should maintain a first-in, first-out queue for this purpose.

11.18 AttachUserConfirm

AttachUserConfirm is generated at the top MCS provider upon receipt of **AttachUserRequest**. Routed back to the requesting provider, it generates an MCS-ATTACH-USER confirm.

Table 11-18/T.125 – AttachUserConfirm MCSPDU

Contents	Source	Sink
Result	Top provider	Confirm
Initiator (optional)	Top provider	Confirm

AttachUserConfirm contains a user id if and only if the result is successful. Providers that receive a successful **AttachUserConfirm** shall enter the user id into their information base.

MCS providers shall route **AttachUserConfirm** to the source of an antecedent **AttachUserRequest**, using the knowledge that there is a one-to-one reply. A provider that transmits **AttachUserConfirm** shall note to which downward MCS connection the new user id is thereby assigned, so that it may validate the user id when it arises later in other requests.

11.19 DetachUserRequest

DetachUserRequest is generated by an MCS-DETACH-USER request. If valid, it rises to the top MCS provider, which deletes the user from its information base and broadcasts **DetachUserIndication** to advise other providers of the change.

Table 11-19/T.125 – DetachUserRequest MCSPDU

Contents	Source	Sink
Reason	Requesting provider	Top provider
User Ids	Requesting provider	Top provider

An MCS-DETACH-USER request generates a **DetachUserRequest** containing reason *user-requested* and a single user id.

DetachUserRequest shall also be generated by an MCS provider when a downward MCS connection is disconnected. At that point, all users in the affected subtree are lost and shall be reported as detached with reason *domain disconnected*. If user id assignments are pending, a later reply, either **MergeChannelsConfirm** or **AttachUserConfirm**, may then be left with no route back to its source in the disconnected subtree. A provider confronted with this shall also generate **DetachUserRequest** to delete the unassignable user ids.

Providers that receive **DetachUserRequest** shall validate the user ids it contains to ensure that they are legitimately assigned to the subtree of origin. Invalid user ids shall be deleted. If no user ids remain, a **DetachUserRequest** shall be ignored.

The user ids contained in **DetachUserRequest** shall not be deleted from the information base until a provider receives **DetachUserIndication**. This maintains consistency with the top MCS provider.

NOTE – If more than one data priority is implemented in an MCS domain, **DetachUserIndication** may arrive at a given provider before data sent earlier but at a low priority by the same user. This protocol does not prevent data being delivered to an attachment even after it was reported through **DetachUserIndication** that the sender had detached.

11.20 DetachUserIndication

DetachUserIndication is generated at the top MCS provider upon receipt of **DetachUserRequest**. It is broadcast downward to all other providers and generates MCS-DETACH-USER indications at all attachments.

At a surviving attachment, **DetachUserIndication** generates one MCS-DETACH-USER indication for each user id it contains. It does not matter whether the notified user was previously aware of the existence of a detached user.

Upon receipt of a **DetachUserIndication** containing its own user id, an MCS attachment ceases to exist. Any static or assigned channels that become unjoined as a result of a user detaching shall be left via **ChannelLeaveRequest**.

Table 11-20/T.125 – DetachUserIndication MCSPDU

Contents	Source	Sink
Reason	Top provider	Indication
User Ids	Top provider	Indication

Providers that receive **DetachUserIndication** shall delete the specified user ids from their information base, including as the manager or as admitted user of a private channel.

Private channels managed by a detached user shall be deleted if there are no other admitted users. If other users remain, deletion of the manager shall cause the top provider to multicast a

ChannelDisbandIndication towards them. If the set of users admitted to a private channel becomes empty and the manager does not reside in the subtree, the channel id shall be deleted from the information base. Otherwise, if a private channel becomes unjoined as a result of users detaching, a provider shall generate a corresponding **ChannelLeaveRequest**.

A provider broadcasting **DetachUserIndication** shall compute, for each affected private channel and for each destination subtree, whether the subtree afterwards contains any attachments admitted to the private channel. If none, a provider shall conclude that the corresponding subordinate provider is no longer joined to the private channel and shall update its information base to this effect immediately, without waiting for **ChannelLeaveRequest**.

Any tokens grabbed, being given to, or inhibited by a detached user shall have their state adjusted accordingly. The deletion of an intended token recipient shall cause the top provider to generate an unsuccessful **TokenGiveConfirm** towards the donor, unless it has released the token or itself detached.

11.21 ChannelJoinRequest

ChannelJoinRequest is generated by an MCS-CHANNEL-JOIN request. If valid, it rises until it reaches an MCS provider with enough information to generate a **ChannelJoinConfirm** reply. This may be the top MCS provider.

Table 11-21/T.125 – ChannelJoinRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting Provider	Higher provider
Channel Id	Request	Higher provider

The user id of the initiating MCS attachment is supplied by the MCS provider that receives the primitive request. Providers that receive **ChannelJoinRequest** subsequently shall validate the user id to ensure that it is legitimately assigned to the subtree of origin. If the user id is invalid, the MCSPDU shall be ignored.

NOTE – This allows for the possibility that **ChannelJoinRequest** may be racing upward against a purge of the initiating user id flowing down. A provider that receives **PurgeChannelsIndication** first might receive a **ChannelJoinRequest** soon thereafter that contains an invalid user id. This is a normal occurrence and is not cause for rejecting the MCSPDU.

ChannelJoinRequest may rise to an MCS provider that has the requested channel id in its information base. Any such provider, being consistent with the top MCS provider, will agree whether the request should succeed. If the request should fail, the provider shall generate an unsuccessful **ChannelJoinConfirm**. If it should succeed and the provider is already joined to the same channel, the provider shall generate a successful **ChannelJoinConfirm**. In these two cases, MCS-CHANNEL-JOIN completes without necessarily visiting the top MCS provider. Otherwise, if the request should succeed but the channel is not yet joined, a provider shall forward **ChannelJoinRequest** upward.

If **ChannelJoinRequest** rises to the top MCS provider, the channel id requested may be zero, which is in no information base because it is an invalid id. If the domain limit on the number of channels in use allows, a new assigned channel id shall be generated and returned in a successful **ChannelJoinConfirm**. If the channel id requested is in the static range and the domain limit on the number of channels in use allows, the channel id shall be entered into the information base and shall likewise be returned in a successful **ChannelJoinConfirm**.

Otherwise, the request will succeed only if the channel id is already in the information base of the top MCS provider. A user id channel can only be joined by the same user. A private channel id can be joined only by users previously admitted by its manager. An assigned channel id can be joined by any user.

11.22 ChannelJoinConfirm

ChannelJoinConfirm is generated at a higher MCS provider upon receipt of **ChannelJoinRequest**. Routed back to the requesting provider, it generates an MCS-CHANNEL-JOIN confirm.

Table 11-22/T.125 – ChannelJoinConfirm MCSPDU

Contents	Source	Sink
Result	Higher provider	Confirm
Initiator	Higher provider	MCSPDU routing
Requested	Higher provider	Confirm
Channel Id (optional)	Higher provider	Confirm

ChannelJoinConfirm contains a joined channel id if and only if the result is successful.

The channel id requested is the same as in **ChannelJoinRequest**. This helps the initiating attachment relate MCS-CHANNEL-JOIN confirm to an antecedent request. Since **ChannelJoinRequest** need not rise to the top provider, confirms may occur out of order.

If the result is successful, **ChannelJoinConfirm** joins the receiving MCS provider to the specified channel. Thereafter, higher providers shall route to it any data that users send over the channel. A provider shall remain joined to a channel as long as any of its attachments or subordinate providers does. To leave the channel, a provider shall generate **ChannelLeaveRequest**.

Providers that receive a successful **ChannelJoinConfirm** shall enter the channel id into their information base. If not already there, the channel id shall be given type static or assigned, depending on its range.

ChannelJoinConfirm shall be forwarded in the direction of the initiating user id. If the user id is unreachable because an MCS connection no longer exists, the provider shall decide whether it has reason to remain joined to the channel. If not, it shall generate **ChannelLeaveRequest**.

11.23 ChannelLeaveRequest

ChannelLeaveRequest is generated by an MCS provider to remove itself from a set of channels. The motivation may be an MCS-CHANNEL-LEAVE request from the last attachment joined to a channel. **ChannelLeaveRequest** continues to rise if higher providers, as a consequence, also lose their reason for being joined.

Providers that receive **ChannelLeaveRequest** shall stop routing to the MCS connection that conveyed it any data that users send over the specified channels. When the last attachment or subordinate provider leaves a channel, an MCS provider shall generate a corresponding **ChannelLeaveRequest**.

Table 11-23/T.125 – ChannelLeaveRequest MCSPDU

Contents	Source	Sink
Channel Ids	Requesting provider	Higher provider

11.24 ChannelConveneRequest

ChannelConveneRequest is generated by an MCS-CHANNEL-CONVENE request. If valid, it rises to the top MCS provider, which returns a **ChannelConveneConfirm** reply. If the domain limit on number of channel ids allows, a new private channel id is generated.

ChannelConveneRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

The requester becomes manager of the private channel. Initially the channel is unjoined and its manager is the only admitted user.

Table 11-24/T.125 – ChannelConveneRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Higher provider

11.25 ChannelConveneConfirm

ChannelConveneConfirm is generated at the top MCS provider upon receipt of **ChannelConveneRequest**. Routed back to the requesting provider, it generates an MCS-CHANNEL-CONVENE confirm.

TABLE 11-25/T.125 – ChannelConveneConfirm MCSPDU

Contents	Source	Sink
Result	Top provider	Confirm
Initiator	Top provider	MCSPDU routing
Channel Id (optional)	Top provider	Confirm

ChannelConveneConfirm contains a private channel id if and only if the result is successful.

Providers that receive a successful **ChannelConveneConfirm** shall enter the channel id into their information base as a private channel with the initiating user id as its manager.

ChannelConveneConfirm shall be forwarded in the direction of the initiating user id. If the user id is unreachable because an MCS connection no longer exists, no special actions need be taken, as a **DetachUserIndication** must arrive later to report that the initiator has detached. Since the initiator is its manager, this will delete the channel id from the information base.

11.26 ChannelDisbandRequest

ChannelDisbandRequest is generated by an MCS-CHANNEL-DISBAND request. If valid, it rises to the top MCS provider, which deletes the private channel id and generates **ChannelDisbandIndication**.

Table 11-26/T.125 – ChannelDisbandRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Channel Id	Request	Top provider

ChannelDisbandRequest may also be generated by an MCS provider on its own initiative to disband a channel.

ChannelDisbandRequest contains the initiating user id, which shall be validated to ensure that it is legitimately assigned to the subtree of origin. If the initiator does not equal the manager of the private channel, as recorded in the information base, the MCSPDU shall be ignored.

11.27 ChannelDisbandIndication

ChannelDisbandIndication is generated at the top MCS provider upon receipt of **ChannelDisbandRequest**. It is multicast downward to providers that contain the manager or an admitted user in their subtree. It generates MCS-CHANNEL-EXPEL indications to admitted users with reason *channel disbanded*.

Table 11-27/T.125 – ChannelDisbandIndication MCSPDU

Contents	Source	Sink
Channel Id	Top provider	Indication

ChannelDisbandIndication shall also be generated by the top MCS provider when the manager of a private channel is detached.

Providers that receive **ChannelDisbandIndication** shall delete the channel from their information base.

11.28 ChannelAdmitRequest

ChannelAdmitRequest is generated by an MCS-CHANNEL-ADMIT request. If valid, it rises to the top MCS provider, which admits the specified users to the private channel and multicasts **ChannelAdmitIndication** to advise providers in whose subtree they reside.

ChannelAdmitRequest contains the initiating user id, which shall be validated as explained for **ChannelDisbandRequest**.

The other user ids of **ChannelAdmitRequest**, representing users to be admitted, shall be validated at the top MCS provider, which alone knows the entire user population. Those that are invalid shall be omitted from the resulting **ChannelAdmitIndication**.

Table 11-28/T.125 – ChannelAdmitRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Channel Id	Request	Top provider
User Ids	Request	Top provider

The user ids contained in **ChannelAdmitRequest** shall not be admitted to the private channel until a provider receives **ChannelAdmitIndication**. This maintains consistency with the top MCS provider.

11.29 ChannelAdmitIndication

ChannelAdmitIndication is generated at the top MCS provider upon receipt of **ChannelAdmitRequest**. It is multicast downward to providers that contain a newly-admitted user in their subtree. It generates MCS-CHANNEL-ADMIT indications at the affected attachments.

Providers that receive **ChannelAdmitIndication** shall ordinarily update the channel in their information base, admitting the specified users that reside in their subtree. However, if a provider is the former top of a lower domain that is being merged as a result of MCS-CONNECT-PROVIDER, it may refuse the admission by generating **DetachUserRequest** for the affected user ids with reason *channel purged*.

Table 11-29/T.125 – ChannelAdmitIndication MCSPDU

Contents	Source	Sink
Initiator	Top provider	Indication
Channel Id	Top provider	Indication
User Ids	Top provider	MCSPDU routing

11.30 ChannelExpelRequest

ChannelExpelRequest is generated by an MCS-CHANNEL-EXPEL request. If valid, it rises to the top MCS provider, which expels the specified users from the private channel and multicasts **ChannelExpelIndication** to advise providers in whose subtree they reside.

ChannelExpelRequest contains the initiating user id, which shall be validated as explained for **ChannelDisbandRequest**.

The other user ids of **ChannelExpelRequest**, representing users to be expelled, shall be validated at the top MCS provider, which alone knows the entire set of admitted users. Those that were not admitted shall be omitted from the resulting **ChannelExpelIndication**.

The user ids contained in **ChannelExpelRequest** shall not be expelled from the private channel until a provider receives **ChannelExpelIndication**. This maintains consistency with the top MCS provider.

Table 11-30/T.125 – ChannelExpelRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Channel Id	Request	Top provider
User Ids	Request	Top provider

11.31 ChannelExpelIndication

ChannelExpelIndication is generated at the top MCS provider upon receipt of **ChannelExpelRequest**. It is multicast downward to providers that contain an expelled user in their

subtree. It generates MCS-CHANNEL-EXPEL indications at the affected attachments with reason *user-requested*.

Table 11-31/T.125 – ChannelExpelIndication MCSPDU

Contents	Source	Sink
Channel Id	Top provider	Indication
User Ids	Top provider	MCSPDU routing

Providers that receive **ChannelExpelIndication** shall update the channel in their information base, deleting the specified users from the set of users admitted to the channel. If the set of users admitted to a private channel becomes empty and the manager does not reside in the subtree, the channel id shall be deleted from the information base. Otherwise, if the channel becomes unjoined as a result of expulsions, a provider shall generate a corresponding **ChannelLeaveRequest**.

A provider forwarding **ChannelExpelIndication** shall compute, for each destination subtree, whether it afterwards contains any attachments admitted to the private channel. If none, a provider shall conclude that the corresponding subordinate provider is no longer joined to the private channel and shall update its information base to this effect immediately, without waiting for **ChannelLeaveRequest**.

11.32 SendDataRequest

SendDataRequest is generated by an MCS-SEND-DATA request. If valid, it rises toward the top MCS provider. Along the way, providers may generate from it an **SendDataIndication** with identical contents and multicast this downward.

SendDataRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

If the channel id is listed in the information base of the receiving MCS provider as a private channel and the initiator of **SendDataRequest** is not an admitted user, the MCSPDU shall be ignored.

SendDataRequest contains the reliability level that was specified by the original MCS-SEND-DATA request. If the data is unreliable, the domain reference ID is also specified.

The initial or additional TC that conveys **SendDataRequest** shall match its data priority, taking into account the number of priorities implemented in the domain. MCSPDUs arriving over an MCS connection by the wrong TC shall be rejected.

Table 11-32/T.125 – SendDataRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Higher provider
Channel Id	Request	Higher provider
Reliability	Request	Higher provider
Domain Reference ID	Request	Higher provider
Data Priority	Request	Higher provider
Segmentation	Requesting provider	Higher provider
Total Data Size	Requesting provider	Higher provider
User Data	Request	Higher provider

The segmentation flags *begin* and *end* shall be set by a provider to show the relationship of user data in the **SendDataRequest** to the boundaries of an MCS service data unit. Providers have freedom to fragment and reassemble MCSPDUs that are part of the same MCS service data unit, so long as this does not disturb the integrity of user data. However, there should be little advantage in such manipulation, as the maximum size of an MCSPDU is constant throughout a domain.

For a given **SendDataRequest**, if the *begin* segmentation flag is true, and the *end* flag is false, the Total Data Size of the segmented user data will be provided. This field will only be present if both the source and sink nodes are members of the V3 Summit.

A provider shall generate from **SendDataRequest** an **SendDataIndication** with the same content and shall transmit it to all providers that are joined to the specified channel, excepting the subordinate provider that transmitted **SendDataRequest** upward. Unless the channel is listed in the provider's information base as a user id residing in its subtree, it shall also forward **SendDataRequest** upward.

11.33 SendDataIndication

SendDataIndication is generated at a higher MCS provider upon receipt of **SendDataRequest**. It is multicast downward and generates MCS-SEND-DATA indications at all attachments that are joined to the channel.

Table 11-33/T.125 – SendDataIndication MCSPDU

Contents	Source	Sink
Initiator	Higher provider	Indication
Channel Id	Higher provider	Indication
Reliability	Higher provider	Indication
Domain Reference ID	Higher provider	Indication
Data Priority	Higher provider	Indication
Segmentation	Higher provider	Indicating provider
Total Data Size	Higher provider	Indicating provider
User Data	Higher provider	Indication

The initial or additional TC that conveys **SendDataIndication** shall match its data priority, taking into account the number of priorities implemented in a domain. MCSPDUs arriving over an MCS connection by the wrong TC shall be rejected.

The segmentation flags *begin* and *end* permit user data to be reassembled into a complete MCS service data unit. These flags shall be interpreted in the context of **SendDataIndication** MCSPDUs arriving from the same user over the same channel and at the same priority. A stream of fragments to be reassembled may be interleaved with other MCSPDUs and data from other users over other channels at other priorities.

For a given **SendDataIndication**, if the *begin* segmentation flag is true, and the *end* flag is false, the Total Data Size of the segmented user data will be provided. This field will only be present if both the source and sink nodes are members of the V3 Summit.

It is a matter of local implementation how service data units are indicated to attached MCS users. One possibility is to deliver each MCSPDU as a separate interface data unit, with segmentation flags included. Alternative approaches that may seek to reassemble within the receiving provider should

make some provision for large service data units and should reflect to the user the relative order in which service data units begin to arrive.

Providers that receive **SendDataIndication** shall forward it to all subordinates that are joined to the channel.

11.34 UniformSendDataRequest

UniformSendDataRequest is generated by an MCS-UNIFORM-SEND-DATA request. If valid, it rises to the top MCS provider, which generates from it a **UniformSendDataIndication** with identical contents and multicasts this downward.

Table 11-34/T.125 – UniformSendDataRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Channel Id	Request	Top provider
Reliability	Request	Top provider
Domain Reference ID	Request	Top provider
Data Priority	Request	Top provider
Segmentation	Requesting provider	Top provider
Total Data Size	Requesting provider	Top provider
User Data	Request	Top provider

UniformSendDataRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

If the channel id is listed in the information base of the receiving MCS provider as a private channel and the initiator of **UniformSendDataRequest** is not an admitted user, the MCSPDU shall be ignored.

The initial or additional TC that conveys **UniformSendDataRequest** shall match its data priority, taking into account the number of priorities implemented in the domain. MCSPDUs arriving over an MCS connection by the wrong TC shall be rejected.

The segmentation flags *begin* and *end* shall be set by a provider to show the relationship of user data in the **UniformSendDataRequest** to the boundaries of an MCS service data unit. Providers have freedom to fragment and reassemble MCSPDUs that are part of the same MCS service data unit, so long as this does not disturb the integrity of user data. However, there should be little advantage in such manipulation, as the maximum size of an MCSPDU is constant throughout a domain.

For a given **UniformSendDataRequest**, if the *begin* segmentation flag is true, and the *end* flag is false, the Total Data Size of the segmented user data will be provided. This field will only be present if both the source and sink nodes are members of the V3 Summit.

The top MCS provider shall generate from **UniformSendDataRequest** a **UniformSendDataIndication** with the same content.

11.35 UniformSendDataIndication

UniformSendDataIndication is generated at the top MCS provider upon receipt of **UniformSendDataRequest**. It is multicast downward and generates MCS-UNIFORM-SEND-DATA indications at all attachments that are joined to the channel.

Table 11-35/T.125 – UniformSendDataIndication MCSPDU

Contents	Source	Sink
Initiator	Top provider	Indication
Channel Id	Top provider	Indication
Reliability	Top provider	Indication
Domain Reference ID	Top provider	Indication
Data Priority	Top provider	Indication
Segmentation	Top provider	Indicating provider
Total Data Size	Top provider	Indicating provider
User Data	Top provider	Indication

The initial or additional TC that conveys **UniformSendDataIndication** shall match its data priority, taking into account the number of priorities implemented in the domain. MCSPDUs arriving over an MCS connection by the wrong TC shall be rejected.

The segmentation flags *begin* and *end* permit user data to be reassembled into a complete MCS service data unit. These flags shall be interpreted in the context of **UniformSendDataIndication** MCSPDUs arriving from the same user over the same channel and at the same priority. A stream of fragments to be reassembled may be interleaved with other MCSPDUs and data from other users over other channels at other priorities.

For a given **UniformSendDataIndication**, if the *begin* segmentation flag is true, and the *end* flag is false, the Total Data Size of the segmented user data will be provided. This field will only be present if both the source and sink nodes are members of the V3 Summit.

It is a matter of local implementation how service data units are indicated to attached MCS users.

Providers that receive **UniformSendDataIndication** shall forward it to all subordinates that are joined to the channel.

11.36 TokenGrabRequest

TokenGrabRequest is generated by an MCS-TOKEN-GRAB request. If valid, it rises to the top MCS provider, which returns a **TokenGrabConfirm** reply.

Table 11-36/T.125 – TokenGrabRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Token Id	Request	Top provider

TokenGrabRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

If the token is free and the domain limit on the number of tokens in use allows, it shall become grabbed. If the token is inhibited by the requesting user only, it shall become grabbed. Otherwise, the state of the token shall not change.

11.37 TokenGrabConfirm

TokenGrabConfirm is generated at the top MCS provider upon receipt of **TokenGrabRequest**. Routed back to the requesting provider, it generates an MCS-TOKEN-GRAB confirm.

Table 11-37/T.125 – TokenGrabConfirm MCSPDU

Contents	Source	Sink
Result	Top provider	Confirm
Initiator	Top provider	MCSPDU routing
Token Id	Top provider	Confirm
Token Status	Top provider	Intermediate Providers

The result shall be *successful* if the token was previously free or if the token was converted from inhibited to grabbed by the same user. Other results are *too many tokens* and *token not available*. The latter applies to a token already grabbed by the requester; this may be discerned by examining the token status.

Providers that receive **TokenGrabConfirm** shall update the token state in their information base to agree with the status returned.

TokenGrabConfirm shall be forwarded in the direction of the initiating user id. If the user id is unreachable because an MCS connection no longer exists, no special actions need be taken, as a **DetachUserIndication** must arrive later to report that the initiator has detached. This will release its hold on the token id in the information base.

11.38 TokenInhibitRequest

TokenInhibitRequest is generated by an MCS-TOKEN-INHIBIT request. If valid, it rises to the top MCS provider, which returns a **TokenInhibitConfirm** reply.

Table 11-38/T.125 – TokenInhibitRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Token Id	Request	Top provider

TokenInhibitRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

If the token is free and the domain limit on the number of tokens in use allows, it shall become inhibited. If the token is grabbed by the requesting user, it shall become inhibited. If the token is

already inhibited, the requester shall be added to the set of inhibitors. Otherwise, the state of the token shall not change.

11.39 TokenInhibitConfirm

TokenInhibitConfirm is generated at the top MCS provider upon receipt of **TokenInhibitRequest**. Routed back to the requesting provider, it generates an MCS-TOKEN-INHIBIT confirm.

The result shall be *successful* if the token was previously free or inhibited or if the token was converted from grabbed to inhibited by the same user. Other results are *too many tokens* and *token not available*.

Table 11-39/T.125 – TokenInhibitConfirm MCSPDU

Contents	Source	Sink
Result	Top provider	Confirm
Initiator	Top provider	MCSPDU routing
Token Id	Top provider	Confirm
Token Status	Top provider	Intermediate Providers

Providers that receive **TokenInhibitConfirm** shall update the token state in their information base to agree with the status returned.

This MCSPDU is routed in the same way as **TokenGrabConfirm**.

11.40 TokenGiveRequest

TokenGiveRequest is generated by an MCS-TOKEN-GIVE request. If valid, it rises to the top MCS provider, which generates either **TokenGiveIndication** or an unsuccessful **TokenGiveConfirm**.

Table 11-40/T.125 – TokenGiveRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Token Id	Request	Top provider
Recipient	Request	Top provider

TokenGiveRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

If the token is grabbed by the requester and the intended recipient exists, **TokenGiveIndication** shall be transmitted toward the recipient. Otherwise, the request shall fail, the state of the token shall be unchanged, and **TokenGiveConfirm** shall be transmitted toward the requester with the result *token not possessed* or *no such user*.

11.41 TokenGiveIndication

TokenGiveIndication is generated at the top MCS provider upon receipt of **TokenGiveRequest**. Routed to the intended recipient, it generates an MCS-TOKEN-GIVE indication.

Providers that receive **TokenGiveIndication** shall ordinarily update the token id in their information base to the state of being given from initiator to recipient. However, if a provider is the former top of a lower domain that is being merged as a result of MCS-CONNECT-PROVIDER, it may refuse the offered token by generating **TokenGiveResponse** with reason *domain merging*.

Table 11-41/T.125 – TokenGiveIndication MCSPDU

Contents	Source	Sink
Initiator	Top provider	Indication
Token Id	Top provider	Indication
Recipient	Top provider	MCSPDU routing

TokenGiveIndication shall be forwarded in the direction of the recipient user id. If the user id is unreachable because an MCS connection no longer exists, no special actions need be taken, as a **DetachUserIndication** must arrive later to report that the recipient has detached. This will release its hold on the token id in the information base.

11.42 TokenGiveResponse

TokenGiveResponse is generated by an MCS-TOKEN-GIVE response. If valid, it rises to the top MCS provider, which generates **TokenGiveConfirm** to inform the token's donor of the outcome.

Table 11-42/T.125 – TokenGiveResponse MCSPDU

Contents	Source	Sink
Result	Response	Top provider
Recipient	Responding provider	Top provider
Token Id	Responding provider	Top provider

A *successful* result shall signify the recipient's acceptance of the offered token.

The user id of the responding MCS attachment is supplied by the MCS provider that receives the primitive response. Providers that receive **TokenGiveResponse** subsequently shall validate the user id to ensure that it is legitimately assigned to the subtree of origin. If the user id is invalid, the MCSPDU shall be ignored.

If the token id is not listed in the provider's information base as being given to the recipient, the MCSPDU shall be ignored. If the token id is still grabbed by the donor, its state shall be updated to grabbed by the recipient if the result is successful; otherwise it shall revert to grabbed by the donor or shall be deleted from the information base, depending on whether the donor resides in the subtree of the provider. If the token id has since been released by the donor and the result is not successful, the token shall be deleted from the provider's information base.

If the MCSPDU is not invalid and ignored, it shall be forwarded upward. The top MCS provider shall act on **TokenGiveResponse** as specified above. In addition, if the donor has not already released the token, the top provider shall generate **TokenGiveConfirm** containing the same result as **TokenGiveResponse**.

11.43 TokenGiveConfirm

TokenGiveConfirm is generated at the top MCS provider upon receipt of **TokenGiveResponse**. Routed back to the requesting provider, it generates a MCS-TOKEN-GIVE confirm.

Table 11-43/T.125 – TokenGiveConfirm MCSPDU

Contents	Source	Sink
Result	Top provider	Confirm
Initiator	Top provider	MCSPDU routing
Token Id	Top provider	Confirm
Token Status	Top provider	Intermediate Providers

TokenGiveConfirm shall also be generated by the top MCS provider upon receipt of **TokenGiveRequest** if a token cannot be offered to the intended recipient. This takes the place of generating **TokenGiveIndication**. **TokenGiveConfirm** shall also be generated with result *no such user* if the recipient is detached before **TokenGiveResponse** is received.

Providers that receive **TokenGiveConfirm** shall update the token state in their information base to agree with the status returned.

This MCSPDU is routed in the same way as **TokenGrabConfirm**.

11.44 TokenPleaseRequest

TokenPleaseRequest is generated by an MCS-TOKEN-PLEASE request. If valid, it rises to the top MCS provider, which multicasts **TokenPleaseIndication** to alert current users of the token.

TokenPleaseRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

Table 11-44/T.125 – TokenPleaseRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Token Id	Request	Top provider

11.45 TokenPleaseIndication

TokenPleaseIndication is generated at the top MCS provider upon receipt of **TokenPleaseRequest**. It is multicast downward and generates MCS-TOKEN-PLEASE indications.

Providers that receive **TokenPleaseIndication** shall forward it to all subordinates that contain in their subtree a user who has grabbed, inhibited, or is being given the specified token.

Table 11-45/T.125 – TokenPleaseIndication MCSPDU

Contents	Source	Sink
Initiator	Top provider	Indication
Token Id	Top provider	Indication

11.46 TokenReleaseRequest

TokenReleaseRequest is generated by an MCS-TOKEN-RELEASE request. If valid, it rises to the top MCS provider, which returns a **TokenReleaseConfirm** reply.

Table 11-46/T.125 – TokenReleaseRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Token Id	Request	Top provider

TokenReleaseRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

If the token is grabbed by the requester, it shall become free. If it is inhibited, the requester shall be removed from the set of inhibitors; if this set becomes empty, the token shall become free. If the token is in the process of being given away by the requester, it shall enter a distinct intermediate state of given to the intended recipient, pending receipt of **TokenGiveResponse**. Otherwise the state of the token shall not change.

11.47 TokenReleaseConfirm

TokenReleaseConfirm is generated at the top MCS provider upon receipt of **TokenReleaseRequest**. Routed back to the requesting provider, it generates an MCS-TOKEN-RELEASE confirm.

Table 11-47/T.125 – TokenReleaseConfirm MCSPDU

Contents	Source	Sink
Result	Top provider	Confirm
Initiator	Top provider	MCSPDU routing
Token Id	Top provider	Confirm
Token Status	Top provider	Intermediate Providers

The result shall be *successful* if the token was grabbed or inhibited by the requester or if the requester was in the process of giving it away. The other possible result is *token not possessed*.

Providers that receive **TokenReleaseConfirm** shall update the token state in their information base to agree with the status returned.

This MCSPDU is routed in the same way as **TokenGrabConfirm**.

11.48 TokenTestRequest

TokenTestRequest is generated by an MCS-TOKEN-TEST request. If valid, it rises to the top MCS provider, which returns a **TokenTestConfirm** reply.

TokenTestRequest contains the initiating user id, which shall be validated as explained for **ChannelJoinRequest**.

Table 11-48/T.125 – TokenTestRequest MCSPDU

Contents	Source	Sink
Initiator	Requesting provider	Top provider
Token Id	Request	Top provider

11.49 TokenTestConfirm

TokenTestConfirm is generated at the top MCS provider upon receipt of **TokenTestRequest**. Routed back to the requesting provider, it generates an MCS-TOKEN-TEST confirm.

Providers that receive **TokenTestConfirm** should find that the token state in their information base agrees with the status returned.

This MCSPDU is routed in the same way as **TokenGrabConfirm**.

Table 11-49/T.125 – TokenTestConfirm MCSPDU

Contents	Source	Sink
Initiator	Top provider	MCSPDU
Token Id	Top provider	Confirm
Token Status	Top provider	Confirm

11.50 CapabilitiesNotificationRequest

CapabilitiesNotificationRequest is an upward flowing PDU which is used to alert higher nodes in the hierarchy of desired changes to the summit capabilities list, or to alert the summit of a change in the status of V2 node connections. It is also sometimes sent by intermediate nodes which detect the loss of a downward connection. Any node which detects the establishment or loss of a connection with a V2 node might send this PDU.

Table 11-50/T.125 – CapabilitiesNotificationRequest MCSPDU

Contents	Source	Sink
V2 Node Present	Requesting provider	Higher provider
Add List	Requesting provider	Higher provider
Remove List	Requesting provider	Higher provider

The contents of the **CapabilitiesNotificationRequest** are as follows:

V2 Node Present: This is a flag which will be set if a V2 node connects to the V3 summit. At such time as the V2 node disconnects, this flag will be reset.

Add List: This is a list of request capabilities (described in 11.50.1) which a given node requests to be added to the summit capabilities list. This field is optional because not every **CapabilitiesNotificationRequest** generated will be for the purpose of adding capabilities to the list.

Remove List: This is a list of request capabilities which a given node requests to be removed from the summit capabilities list. This field is optional because not every **CapabilitiesNotificationRequest** generated will be for the purpose of removing capabilities from the list.

11.50.1 Request Capability

The following table summarizes the contents of the Request Capability. The AddList and RemoveList in a **CapabilitiesNotificationRequest** PDU is comprised of Request Capabilities.

Capability ID
Capability Class
Participation Indicator

Capability ID: This is an identification assigned to the capability. The ID may be standard (i.e. documented in T.125) or non-standard.

Capability Class: This indicates whether the capability has an associated value, and the processing associated with that value. All capabilities are of class MIN, MAX, or NULL.

Participation Indicator: This field indicates the degree of participation required of other V3 nodes in the summit. The possible values of this field are "partial" or "global".

11.51 CapabilitiesNotificationIndication

CapabilitiesNotificationIndication is a downward flowing PDU which is used to alert nodes to a change in the summit capabilities, or to a change in the status of V2 node connections. This PDU is sent by the summit provider or any intermediate node which has a need to inform a lower node of a change.

Table 11-51/T.125 – CapabilitiesNotificationIndication MCSPDU

Contents	Source	Sink
V2 Node Present	Higher provider	Subordinate
Add List	Higher provider	Subordinate
Remove List	Higher provider	Subordinate

The contents of the **CapabilitiesNotificationIndication** are as follows:

V2 Node Present: This is a flag which will be set if a V2 node is connected to the V3 summit. At such time as all V2 nodes disconnect, this flag will be reset.

Add List: This is a list of indication capabilities (described in 11.51.1) which a given node has decided need to be added to the summit capabilities list. This field is optional because not every **CapabilitiesNotificationIndication** generated will be for the purpose of adding capabilities to the list.

Remove List: This is a list of indication capabilities which a given node has decided need to be removed from the summit capabilities list. This field is optional because not every **CapabilitiesNotificationIndication** generated will be for the purpose of removing capabilities from the list.

11.51.1 Indication Capability

The following table summarizes the contents of the Indication Capability. The AddList and RemoveList in a **CapabilitiesNotificationIndication** PDU is comprised of Indication Capabilities.

Capability ID
Capability Class
Summit Provider Supported
Intermediate Node Supported

Capability ID: This is an identification assigned to the capability. The ID may be standard (i.e. documented in T.125) or non-standard.

Capability Class: This indicates whether the capability has an associated value, and the processing associated with that value. All capabilities are of class MIN, MAX, or NULL.

Summit Provider Supported: This is a flag which indicates whether the V3 summit provider supports this capability.

Intermediate Node Supported: This is a flag which indicates whether the intervening nodes on the path to the summit provider support this capability. If there are no intervening nodes, meaning a node has a direct connection to the summit provider, then this flag is set.

12 MCS provider information base

12.1 Hierarchical replication

Although an MCS provider may host multiple domains, it serves each one independently. It maintains logically separate information bases for each to record the state of channel and token resources in use. The description that follows is set in the context of a single domain.

The MCS resources that need to be managed in a domain are channel ids and token ids. User ids are a subset of channel ids. Domain parameters limit how many ids of each category can be in use simultaneously. This allows a provider to compute how much memory is needed for the information base in the worst case of a fully-utilized domain.

In the hierarchy of a domain, the ids in use at any given MCS provider stabilize to a subset of those in use at its immediate superior. Information about an id is recorded where it can be used to support MCS services involving that id. Recording information more widely would entail extra costs in MCSPDU traffic to keep the information up to date. Since the information recorded at a provider is consistent with that recorded at superior providers, within the limit of MCSPDU propagation delay, it may be said that the provider information base is partially replicated through the domain hierarchy.

Domain parameters become fixed and unchangeable with the establishment of the first MCS connection of a domain. A provider that lacks capacity for the maximum number of ids specified in each category may negotiate to join a domain on false pretense. It may speculate that at its low position in a hierarchy it will not be called upon to retain more than a fraction of the total information base. Such a provider may not support attachments and subordinates with the full range of MCS services they expect. Nonetheless, until its capacity is actually exceeded, such a provider may appear to be an equal member of the domain. This strategy may appeal to terminal nodes with limited aspirations.

Ids are placed into use first at the top MCS provider. They are placed into use at subordinate providers by a selective downward flow of MCSPDUs. Most are deleted from use in the same top-down manner. There are necessarily intervals during which a subordinate provider records as in use an id that its superiors do not, because the MCSPDU that deletes the id remains in transit. This is not, however, a situation that endures. Control MCSPDUs are received and processed in the order of their

transmission. The consequences of processing an MCSPDU, including its creation or deletion of channel and token ids, take effect before attention shifts to the next input event.

An exception to the preceding paragraph is the deletion of static and assigned channel ids. Although placed into use by a downward flow of **ChannelJoinConfirm**, these channel ids are deleted in the opposite order – from the bottom up. Specifically, they are deleted when an accumulation of MCS-CHANNEL-LEAVE requests from attachments and **ChannelLeaveRequest** MCSPDUs from subordinate providers combine to leave a channel unjoined. Such transitions motivate the transmission of **ChannelLeaveRequest** further upward. Thus, for these two cases, the channel ids recorded as in use are a strict subset of those in use at the superior provider. This is an accidental corollary of optimizations designed to speed channel management as a prelude to data transfer.

Channels ids are put into use by **MergeChannelsConfirm**, **AttachUserConfirm**, **ChannelJoinConfirm**, **ChannelConveneConfirm**, and **ChannelAdmitIndication**; they are deleted by **MergeChannelsConfirm**, **PurgeChannelsIndication**, **DetachUserIndication**, **ChannelLeaveRequest**, **ChannelDisbandIndication**, and **ChannelExpelIndication**. Token ids are put into use by **MergeTokensConfirm**, **TokenGrabConfirm**, **TokenInhibitConfirm**, and **TokenGiveIndication**; they are deleted by **MergeTokensConfirm**, **PurgeTokensIndication**, **TokenReleaseConfirm**, **TokenGiveResponse** and **TokenGiveConfirm**. When an id is put into use at a given provider, the MCSPDU that is the cause may be forwarded to zero, one, several, or all subordinate providers. The use of an id may grow or contract gradually as, for example, individual users are admitted to and expelled from a private channel. When an id is deleted from a given provider, the MCSPDU that effects this is forwarded to all subordinates who may still be recording the id as in use.

The use of an id is ultimately tied to actions on a channel or token by a user attached to the domain (although there may be some delay, as explained, in communicating changes through the transmission of MCSPDUs). The ids recorded stably as in use at a given MCS provider are those that are actively employed by some user in the subtree of the provider. It follows that they are a subset of those recorded stably at any superior provider.

Deleting a user id has the corollary effect of deleting channel ids and token ids of which it is the sole user in a subtree.

Criteria for considering channel ids and token ids to be in use are specified in the following subclauses.

12.2 Channel information

The four kinds of channel have corresponding criteria to determine whether a given attachment is considered to be using the channel id, hence whether it is to be represented in a provider's information base:

a) A static channel id (range 1..1000) is in use if the user has joined the channel with a successful MCS-CHANNEL-JOIN confirm and not left by MCS-CHANNEL-LEAVE request or indication.

b) A user id channel is in use if it was assigned to the user by a successful MCS-ATTACH-USER confirm and the user has not detached by MCS-DETACH-USER request or indication.

c) A private channel id is in use if the user has created the channel with a successful MCS-CHANNEL-CONVENE confirm or been admitted to it with MCS-CHANNEL-ADMIT indication and not expelled by MCS-CHANNEL-EXPEL indication and the channel has not been disbanded by MCS-CHANNEL-DISBAND request or indication.

d) An assigned channel id is in use if the user has joined the channel with a successful MCS-CHANNEL-JOIN confirm and not left by MCS-CHANNEL-LEAVE request or indication.

This information shall be recorded for a channel id in use:

a) The kind of channel it represents (static, user id, private, or assigned).

b) By which MCS attachments and which MCS connections to subordinate providers the channel is joined.

c) If a user id channel, the direction to it, that is, either the local MCS attachment to which the user id is assigned or the downward MCS connection to a subordinate provider in whose subtree the user resides.

d) If a private channel id, the user id of the manager who convened it (whether or not the manager itself is in the subtree of the provider) and the set of all user ids in the subtree of the provider who have been admitted to the channel.

The information recorded for channel ids is employed as explained in clause 10 to validate request MCSPDUs and to route indication and confirm MCSPDUs.

12.3 Token information

The state transitions of a token id are shown in Figure 12-1. The following abbreviations apply:

DUin – DetachUserIndication

TGcf – TokenGrabConfirm

TIcf – TokenInhibitConfirm

TRcf – TokenReleaseConfirm

TVin – TokenGiveIndication

TVrs – TokenGiveResponse

TVcf – TokenGiveConfirm

An individual token id can be *grabbed* by a single user or *inhibited* by one or more. The action of **TokenGiveIndication** converts the state to *giving* along the branch of a domain hierarchy leading from the top MCS provider toward the intended recipient. This state decays to *ungiveable* if the recipient detaches before its provider responds with **TokenGiveResponse**. It resolves to *given* if instead the donor releases the token explicitly or detaches. During the giving of a token, the branch of a domain hierarchy leading from the donor intersects the branch leading toward the recipient at least at the top MCS provider. The token state changes from *grabbed* to *giving*, and possibly thereafter to *ungiveable* or *given*, only along this intersection.

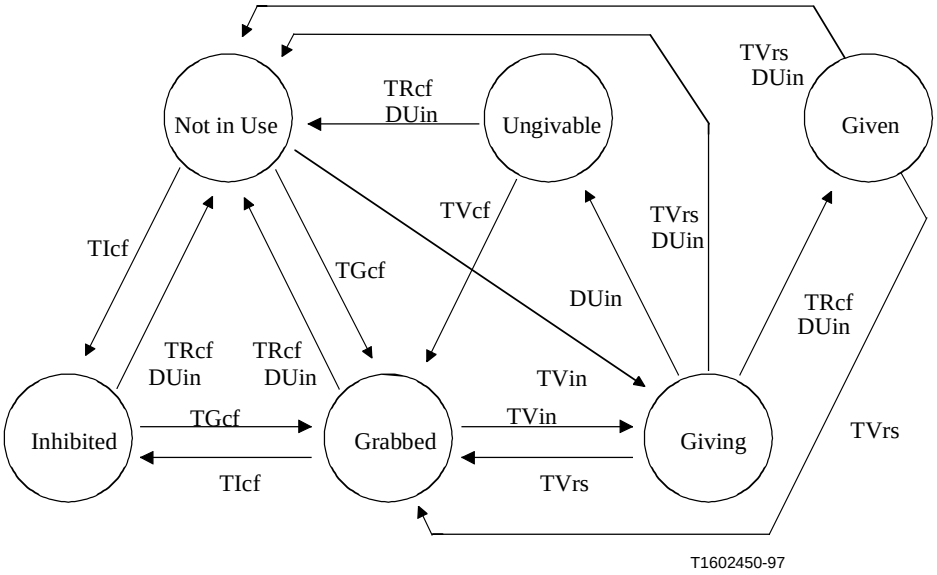

Figure 12-1/T.125 – State transitions of a token id

The user of a token stands in relationship to it as grabber, inhibitor, recipient, or both grabber and recipient (when giving a token to itself):

a) The user is a grabber if it has seized a token with a successful MCS-TOKEN-GRAB confirm and not released it by MCS-TOKEN-RELEASE request or successful MCS-TOKEN-GIVE confirm nor converted it with a successful MCS-TOKEN-INHIBIT confirm or if it has accepted an offered token with a successful MCS-TOKEN-GIVE response.

b) The user is an inhibitor if it has seized a token with a successful MCS-TOKEN-INHIBIT confirm and not released it with MCS-TOKEN-RELEASE request nor converted it with a successful MCS-TOKEN-GRAB confirm.

c) The user is a recipient if it has been offered a token by MCS-TOKEN-GIVE indication and not released it with an unsuccessful MCS-TOKEN-GIVE response.

This information shall be recorded for a token id in use:

a) The state of the token id at the MCS provider (not necessarily identical to that at the top provider).

b) If grabbed or ungiveable, the user id of the grabber in the subtree of the provider.

c) If giving, the user id of the grabber (whether or not the grabber is in the subtree of the provider).

d) If giving or given, the user id of the recipient in the subtree of the provider.

e) If inhibited, the set of all user ids in the subtree of the provider who have inhibited the token.

The information recorded for token ids in use is employed as explained in clause 10 to validate response MCSPDUs and to route indication MCSPDUs.

The state of a token id at a subordinate provider need not be identical to that at the top MCS provider. This is due to the fact that a token donor does not in general process **TokenGiveIndication** or **TokenGiveResponse** and that a recipient does not in general process a donor's **TokenReleaseConfirm**. Figure 12-2 shows states that may arise in a complex token interaction.

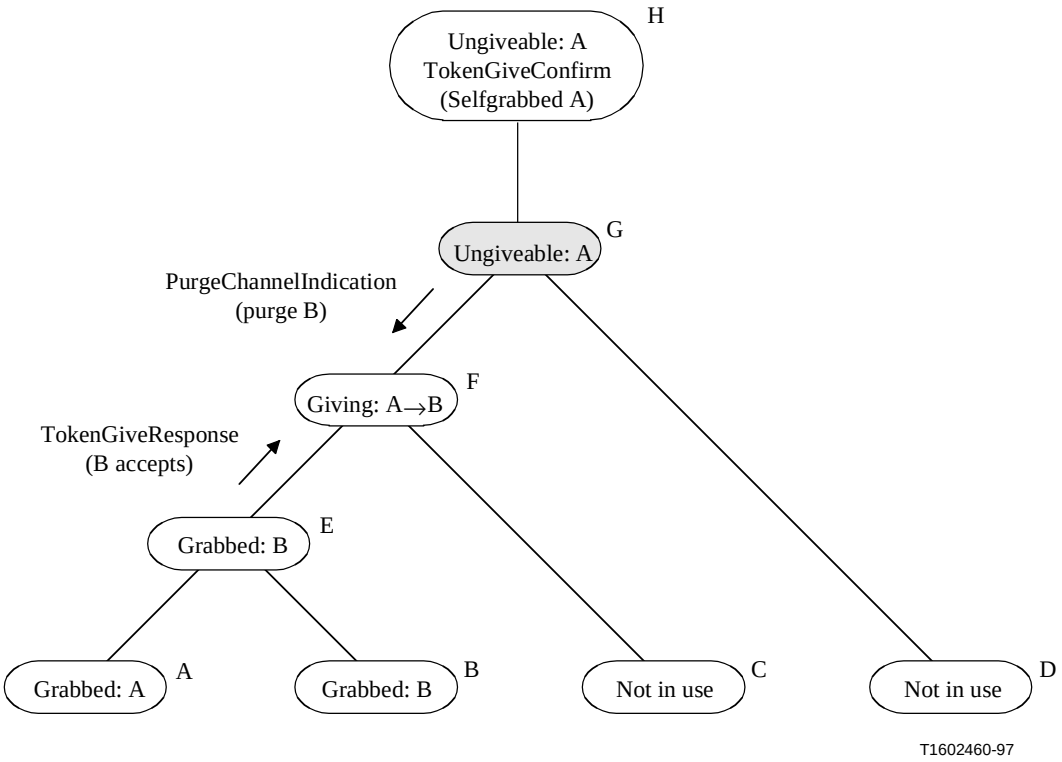

Figure 12-2/T.125 – States of a token that may arise in a complex interaction

The figure focuses on one token id in the information base of providers A through H. A plausible history is that the token was given by user A, attached to provider A, to user B, attached to provider B. Before user B could respond, however, provider G connected the domain to a new top provider H and began a merger. The new domain, unable to accept user B because of a conflict in channel ids, initiated its purge through **PurgeChannelsIndication**. This MCSPDU is shown having completed part of its journey through providers G and D. Provider G, as a result, has adjusted its token state from *giving* to *ungiveable* and merged the token, with this state, into the new domain. The new top provider H, presented with a token in such a state, has queued for transmission an unsuccessful **TokenGiveConfirm** to return the token to user A. At the time shown, the token has also finally been accepted by user B, and a **TokenGiveResponse** is making its way up the recipient's branch of the domain hierarchy, changing the token state at each provider from *giving* to *grabbed* by B. Two MCSPDUs now converge on provider F, and whichever arrives first determines its subsequent state there: either returned to *grabbed* by A or transiently *grabbed* by B and then *not in use* again after user B is detached. In either case, the state of the token will stabilize to *grabbed* by A when the new top provider transmits the **TokenGiveConfirm** it has pending.

13 Elements of procedure

13.1 MCSPDU sequencing

Control MCSPDUs remain in sequence between any pair of MCS providers because they travel over a single TC, the initial TC of an MCS connection. MCS providers shall process received MCSPDUs and shall transmit any resulting output MCSPDUs in the same order. This applies to MCSPDUs that are simply forwarded up or down the domain hierarchy as well as to MCSPDUs that are transformed, such as requests or responses into indications or confirms. The sequencing of MCSPDUs shall be maintained within an MCS provider if it is necessary to queue outputs for later transmission due to the back-pressure of flow control along a TC.

Data MCSPDUs of different priorities need not remain in sequence. On the contrary, the benefit of relative priorities is only achieved when higher priority data is advanced ahead of lower priority data. This means that it should be transmitted over separate TCs and queued separately within each MCS provider. If the number of data priorities implemented in a domain is less than the maximum, fewer TCs are available, but providers that elect to do so may still maintain separate queues internally. This realizes some but not all the benefits of relative priority.

An MCS provider shall maintain the sequence of data MCSPDUs transmitted at a given priority. This is a tighter restriction than that imposed by Recommendation T.122, which guarantees only the sequencing of service data units transmitted at a given priority over the same destination channel.

Control MCSPDUs and data MCSPDUs of top priority are transmitted over the same initial TC and shall receive equal attention by an MCS provider. Data of lower priorities may lag behind. Control indications advancing ahead may arrive before lower priority data that was actually transmitted first.

13.2 Input flow control

An MCS provider has goals that sometimes conflict: to keep data moving briskly through a domain despite transient blockages towards some receivers, to give transmitters fair access to the available bandwidth, and to prevent any party from lagging far behind peers who are receiving the same multicast data. MCS is a reliable service that preserves the integrity of user data. Since a provider has limited capacity to store MCSPDUs when they cannot be transmitted immediately, it must have a defense of sometimes refusing further inputs. While details of the interface to transport services are a local matter, the abstract effect must be that incoming TSDUs are held in TC pipelines, intact and in sequence, to be received at some future time when flow control is lifted. As TC pipelines fill, remote MCS providers may find that they are blocked by back-pressure from transmitting further MCSPDUs and may need to invoke a similar defense.

Flow control is not signalled explicitly in the MCS protocol. This is a function of lower layers that would be wasteful to duplicate. As a result, it is difficult to sense, through the medium of an intervening TC, whether a remote provider is resisting further inputs. To meet conflicting goals as well as possible, the policies described next may be recommended.

An MCS provider may grant each incoming TC a fixed quota of buffers that it may fill with MCSPDUs before back pressure is applied. Each buffer is processed as specified in this protocol, then assigned for output to zero or more outgoing TCs. Output may occur immediately or may be delayed because a transport pipeline is full. Until it is output to the last TC, a buffer is charged against the input quota of the TC by which it arrived. After it is output to all requisite TCs, it is recycled as an increment to that quota. When a quota is exhausted, further inputs over the corresponding TC are halted. Input quotas may be set taking into account whether a TC is part of an upward or downward MCS connection and what data priority it represents.

Buffering can mitigate a disparity of rates between transmitters and receivers. A quota on inputs can prevent any one TC from monopolizing resources. It can also bound how far out of step two receivers of the same multicast data can get. The recommended scheme is not clever enough, however, to anticipate all patterns of use and may sometimes slow the rate of data transfer through a domain when there are acceptable alternatives. The invention of better flow control policies (implemented locally and requiring no additional communication of MCSPDUs) may be a means of product differentiation.

13.3 Throughput enforcement

In contrast to input flow control, some support for throughput enforcement is explicit in the MCS protocol. Firstly, the rate that is enforced is a domain parameter negotiated through

MCS-CONNECT-PROVIDER. Secondly, the time interval over which throughput is monitored before taking adverse action is communicated by each MCS provider to its superior through **ErectDomainRequest**. To describe the throughput enforcement interval requires that providers share some common principles of behavior. Still, enforcement remains a heuristic technique with room for invention.

Enforcing a minimum input rate at each receiver is an option available to controller applications, which establish the MCS connections of a domain. This option is selected through the domain parameter for enforced throughput, which is stated in octets per second. While it seems intuitive that a party should not be allowed to run arbitrarily slowly and thereby obstruct data transfer among others, there is danger in seeking to enforce throughput too strictly.

Complex patterns of multiple transmitters are a problem. The kinds of back-pressure to which an enforcement policy reacts may not result from a single abnormally slow receiver. In the first place, MCSPDUs heading downward must compete for attention with MCSPDUs that rise from below but are reflected back, notably **SendDataIndication**. The downward flow experienced through a peer provider may therefore attain only a fraction of the nominal bandwidth and may vary dynamically with the number of other connections and attachments to the peer. Secondly, only top priority MCSPDUs can be expected to flow continuously over an MCS connection. Lesser priorities can properly be blocked by a peer provider for long periods of time due to the intensity of traffic received from other sources at higher priorities. Finally, there can be great variation in instantaneous throughput if it is measured by how long a blocked MCSPDU must wait before being accepted onto an MCS connection. Among other things, this can depend on how many and in what order MCSPDUs from other sources are queued inside the peer provider.

Nonetheless, throughput enforcement is a valuable option in practical situations where patterns of data transfer are known to be more uniform. It shall be interpreted as requiring a minimal output of MCSPDUs to each direct MCS attachment and to each MCS connection downward over a time interval to be specified by the enforcing MCS provider. Each MCSPDU output, both control and data, shall count towards throughput as though it were the maximum size allowed by domain parameters. The output of an MCSPDU to an MCS attachment shall mean delivery of the associated primitive indication or confirm. Output to a downward MCS connection shall mean the absence of back-pressure at the transport service interface and acceptance into the corresponding TC pipeline.

Output shall be monitored as long as one or more MCSPDUs are queued, regardless of data priority, towards a given attachment or downward connection. Whenever the queues empty, monitoring shall cease, with no enforcement action taken. At the same time, no credit for good behavior shall be recorded to offset future slowdowns. Monitoring shall resume as soon as back-pressure prevents any MCSPDU from being output and requires it to be queued instead. While one or more MCSPDUs remain queued, the number actually output shall be counted over a set interval of time.

A throughput enforcement interval shall be chosen by each MCS provider. The interval shall be long enough to allow at least one MCSPDU of maximum size to be output at the minimum throughput. An MCS provider shall advise its superior of the interval chosen and of any subsequent changes to it by transmitting **ErectDomainRequest** upward. A provider shall act against the offending MCS attachment or downward MCS connection at the end of any interval during which the monitored throughput fails to meet expectations. It shall detach the user or disconnect the connection.

Superior providers may set their throughput enforcement interval to be longer than that of any subordinate plus some margin of reaction time. The aim is to encourage enforcement action first at the lowest provider in a position to detect a problem. When a violator is removed at the end of some lower interval, superior providers should have enough time to sense the restoration of throughput to adequate levels. If they act too quickly, they may penalize a larger subtree than necessary and disrupt innocents in the domain.

The controller applications that set domain parameters should be conservative in their demands, expecting that throughput may occasionally dip during periods of intense application stimuli. If their concern is simply to defend against receivers that stop accepting anything at all, they may set the minimum throughput very low. Knowing the maximum MCSPDU size and enforced throughput rate, controllers can calculate the minimum interval that must elapse before any blockage is detected and overcome.

13.4 Domain configuration

Recommendation T.122 provides no mechanism for configuring the set of domains supported by an MCS provider. This must be considered a local matter whose standardization may be the subject of further study. This protocol assumes that an MCS provider will recognize some domain selectors as valid and others as invalid. It provides for domain selectors to be communicated as part of establishing an MCS connection.

An MCS provider participates implicitly in the negotiation of domain parameters. Whether the calling or the called side, it constrains the range of allowed parameter values according to limits for which it is configured. An MCS provider shall freeze the negotiability of domain parameters once any user has attached to a domain or once the first MCS connection has been established.

13.5 Domain merger

Domains are merged as a consequence of MCS-CONNECT-PROVIDER. If it is convenient to arrange that one domain or the other be empty at this point, there is little complication in a merger. In the most general case, however, provision must be made for updating the information base at the remaining top provider to contain the information base of the former top provider and for resolving any conflicts that become apparent. The details of this are explained in clause 10.

To aid understanding, an example of domain merger is illustrated in the sequence of Figures 13-1 through 13-4. Here provider E represents a former top that has joined a new domain by the highlighted MCS connection to an intermediate provider F. It is irrelevant whether provider E or provider F initiated the MCS connection.

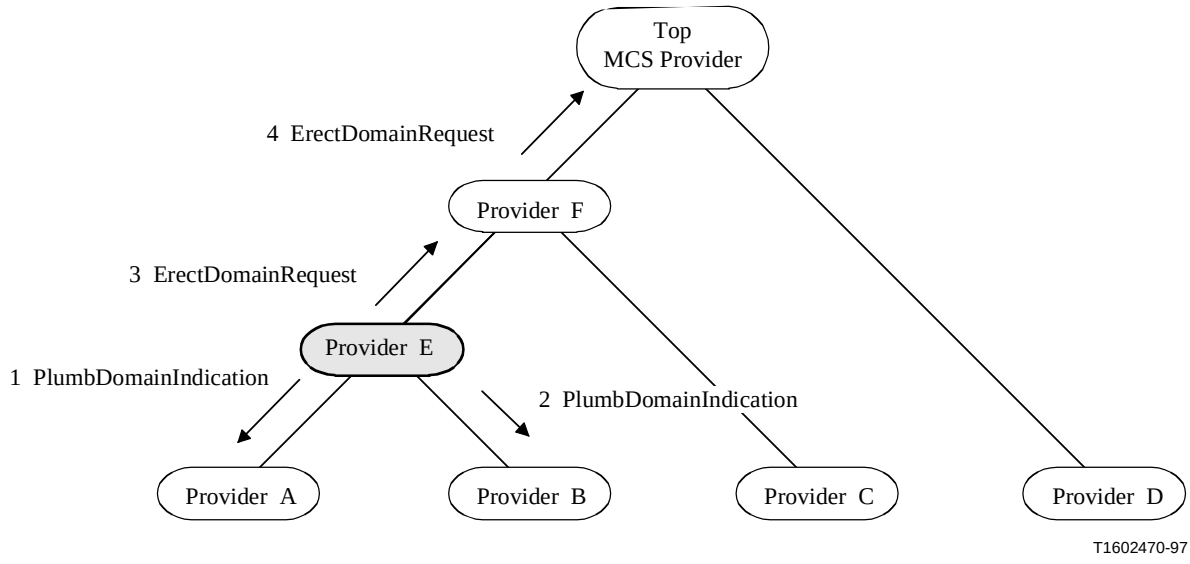

Figure 13-1/T.125 – Domain merger step one – Establish hierarchy

Provider E, finding itself at the lower end of a connection upward, assumes responsibility for executing the merger. Since it is unsafe to transact other activity while the information base is in flux, provider E stops accepting inputs from its subtree. Any MCSPDUs already in transit from providers A and B or new ones generated before the merger is completed will be safeguarded and processed later. The downward flow of MCSPDUs, however, is not impeded, whether generated by provider E or forwarded from further above.

Until the merger completes, the only confirm MCSPDUs provider E will receive are **MergeChannelsConfirm** and **MergeTokensConfirm**, since user requests are not allowed upward. These will confirm or purge channel and token ids already in use. Indication MCSPDUs **PurgeChannelsIndication**, **PurgeTokensIndication**, **DetachUserIndication**, and **ChannelDisbandIndication** of the upper domain may also arrive, deleting channel and token ids from the lower domain whose merger has been individually confirmed. Unconfirmed ids of the lower domain are protected against deletion, because as yet they mean something different from the same id of the upper domain. **PlumbDomainIndication** must be obeyed to enforce the domain height limit. Remaining indications need not be applied by provider E while merger is in progress. Data transfers, in particular, need not flow between the formerly separate domains until merger is completed; they cannot flow until at least the channels conveying them have been sorted out. Two indications that can place new ids into use may be inconvenient to apply. To keep the information base consistent, if provider E refuses **ChannelAdmitIndication** or **TokenGiveIndication**, it will react as specified in clause 10.

The first actions of provider E are to send **PlumbDomainIndication** downward, to ensure that the latest MCS connection has not created a cycle that would invalidate the principle that each domain hierarchy have exactly one top provider, and to send **ErectDomainRequest** upward to report its existing height and throughput enforcement interval. Provider F, advancing thereby from height 2 to 3, passes **ErectDomainRequest** along to the top provider, which then attains height 4.

In the second stage of domain merger, provider E sends upward as many instances of **MergeChannelsRequest** as it takes to contain the user ids in its information base. User ids that do not conflict with the upper domain are confirmed and the rest are purged in equally many instances of **MergeChannelsConfirm**. Provider E generates **PurgeChannelsIndication** from **MergeChannelsConfirm** to report any purges throughout its subtree. This stage ends when all user ids have either been explicitly confirmed or purged.

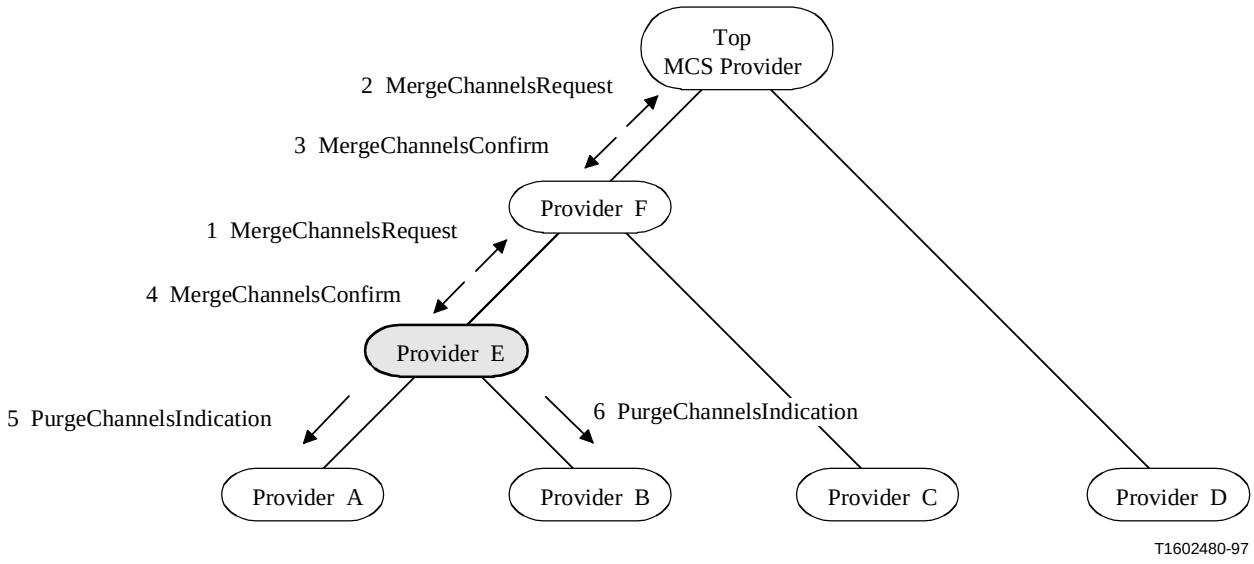

Figure 13-2/T.125 – Domain merger step two – Merge user id channels

Stage three resembles stage two, but concerns token ids rather than user id channels, using a parallel set of MCSPDUs. If user ids had not been merged first, portions of a later **MergeTokensRequest** could be refused as invalid and the affected token ids would be unnecessarily purged. In the case of an inhibited token, the entire set of inhibiting users may not fit into a single MCSPDU. Provider E waits for confirmation of the first subset it sends upward before sending the inhibited token again with the remaining user ids. This protects against the first set being refused due to too many token ids in use but the remainder later being accepted, which would corrupt the information base. This stage ends when all token ids have either been explicitly confirmed or purged.

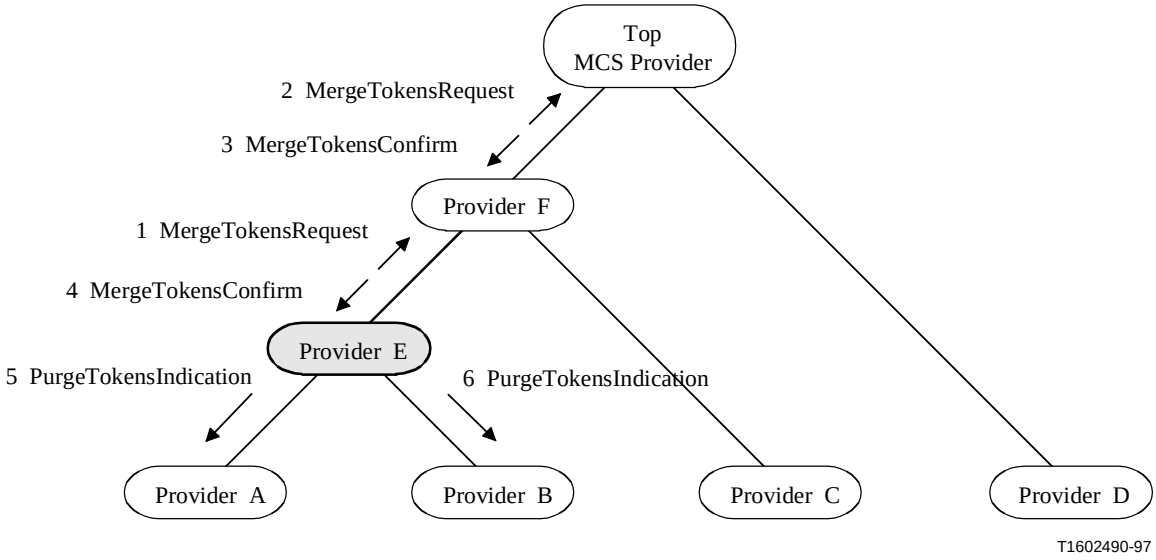

Figure 13-3/T.125 – Domain merger step three – Merge token ids

Stage four involves the same MCSPDUs as stage two, but they contain different channel ids. User id channels having been taken care of, what remains are static, private, and assigned channel ids. If user ids had not been merged first, portions of a later **MergeChannelsRequest** could be refused as invalid and the affected channel ids would be unnecessarily purged. In the case of a private channel, the entire set of users admitted by the channel manager may not fit into a single MCSPDU. Provider E waits for confirmation of the first subset it sends upward before sending the private channel again with the remaining user ids. This protects against the first set being refused due to too many channel ids in use but the remainder later being accepted, which would corrupt the information base. This stage ends and domain merger completes when all remaining channel ids have either been explicitly confirmed or purged.

NOTE – Merging token ids first makes their exclusive possession more effective in suppressing data flow conflicts. Otherwise, data may leak between domains, as channel ids are confirmed, before the conflict is revealed through a token purge.

13.6 Domain disconnection

When an upward MCS connection is disconnected, an MCS provider shall eradicate its subtree of the domain by detaching all its direct MCS attachments and disconnecting all its other MCS connections. The affected provider cannot in general establish a residual domain in its own subtree, because it has no record of request MCSPDUs that were sent upward for which it will never receive a matching confirm MCSPDU.

When a downward MCS connection is disconnected, an MCS provider shall generate **DetachUserRequest** MCSPDUs for all users residing in that portion of its subtree, giving as reason *domain disconnected*.

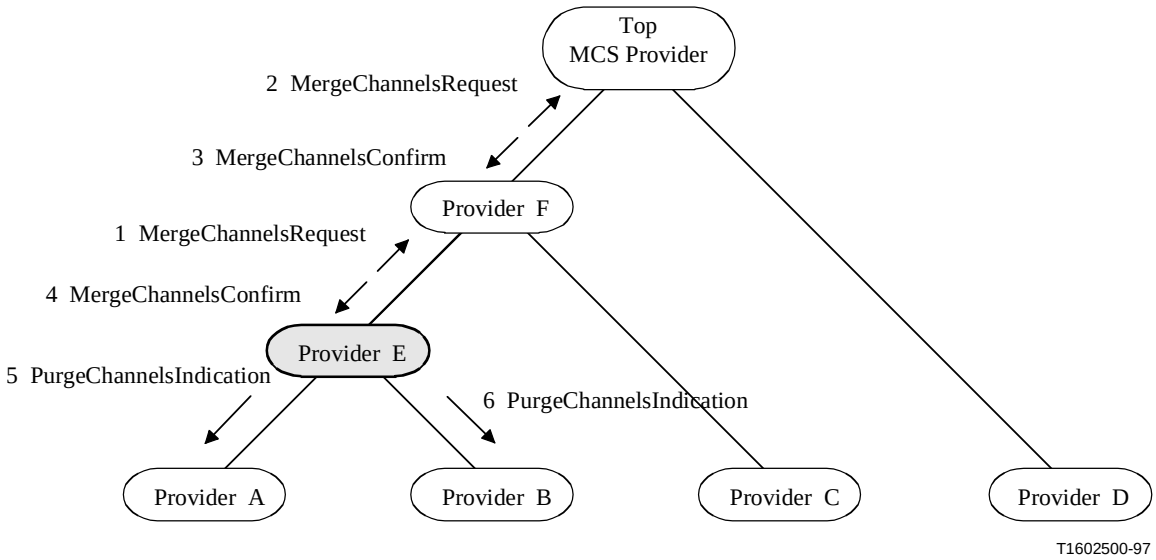

Figure 13-4/T.125 – Domain merger step four – Merge remaining channel ids

13.7 Channel id allocation

Channel ids in the range of 1001 and above are allocated dynamically at the top MCS provider during the processing of primitive requests MCS-ATTACH-USER, MCS-CHANNEL-JOIN zero, and MCS-CHANNEL-CONVENE. There is no requirement that the values allocated fit any particular pattern. Rather, it is desirable that the values be randomly dispersed over the allowed range. This makes it more likely that two domains that operate independently for a substantial period of time may later be merged without conflicts in their respective allocation of channel ids. It also defends against recycling ids too quickly within a single domain as they are released from one use and then reallocated for a different use. Applications using MCS should have time to adjust to the disappearance of a channel or user id before it returns in a new incarnation.

In situations where the seamless merging of active domains is required, conflicts may be avoided by selecting channel ids from a subrange unique to each provider. Subranges may be created by dividing the dynamically allocated channel ids 1001..65535 into several bands. Furthermore, within a subrange, ids may be allocated sequentially in one direction, from top or bottom. Providers that employ subrange allocation shall still obey all aspects of this protocol, including the procedure for domain merger. They shall thereby accommodate peer providers that may lack unique subranges.

NOTE – It is a matter of local implementation how a provider chooses a subrange from which to allocate. Pre-arranged conferences may be banded by a network-management system.

Token ids, unlike channel ids, are not allocated, and the dividing line of value 1001 has no significance for them. A free token id with a given value is statically available to be grabbed or inhibited at any time, subject only to a domain limit on the total number in use at once.

13.8 Token status

Token status is defined formally in clause 7. It is returned as a component of token confirm MCSPDUs and is used to update the record for a token id in the information base of subordinate

providers. Token status is not necessarily reported directly through a confirm primitive to the initiating user but may be reported indirectly through a result value.

When more that one token status value describes the relationship of a given user to a specified token id, the order of preference shall be as follows. *Self-recipient* shall be reported first, if it fits, to remind the user that they must respond to an MCS-TOKEN-GIVE indication. Next preferred is *self-giving*, to remind the user that they are engaged in an uncompleted operation, followed by *self-grabbed* or *self-inhibited*. Last in order are the remaining status values, reflecting the token's current state as a result of its sole use by other parties.

An MCS provider relies on the specified preference for token status values in order to update the token state correctly in its information base.

13.9 Capabilities Notification

Through extensibility mechanisms built into the V3 protocol, capabilities can be implemented in MCS by adding components to T.125 PDUs, or even by adding new PDUs, without requiring a new version of the protocol. A given node can act on PDU parameters it understands and simply relay unrecognized parameters intact.

In order for a given capability to be exercised, certain conditions may have to be met by the nodes which comprise the domain. These conditions, or rules, could include such things as a degree of participation by other nodes, or the presence or absence of Version 2 nodes. For example, a certain capability may require the participation of every node in the domain in order to be exercised. On the other hand, another capability may require only two or more nodes to participate in order to be useful. In other words, a set of rules is associated with each capability and a given node must make a decision whether to exercise this capability based on these rules and the state of the domain.

The **CapabilitiesNotificationRequest** and **CapabilitiesNotificationIndication** PDUs provide a means to facilitate capabilities notification in a V3 summit. The **CapabilitiesNotificationRequest** PDU carries capabilities which various nodes wish to be added to or removed from the summit capabilities list. In addition, this PDU is also used to notify other nodes of status changes in regard to V2 connections. The **CapabilitiesNotificationIndication** PDU is used to effect changes to the summit capabilities list and to inform of V2 connection status. For the remainder of this subclause, a **CapabilitiesNotificationRequest** PDU will be referred to as a "**Request**", and a **CapabilitiesNotificationIndication** PDU will be referred to as an "**Indication**".

13.9.1 Capability IDs

A capability is a high level behavior of the system, most likely requiring the cooperation between two or more nodes. As capabilities are developed, they are assigned a Capability ID. This ID could be standardized and documented in T.125. A Capability ID thus designates a set of PDU parameters (not necessarily in the same PDU) , possibly a set of values for those parameters, and the required processing of those PDUs by participating nodes.

13.9.2 Version 3 Summits

It cannot be expected that a domain hierarchy will consist solely of V3 nodes. Due to the stipulation of T.125 (Revised) that the protocol version be negotiated on a per connection basis, it is possible for a domain to consist of a mixture of V2 and V3 nodes. T.125 (Revised) also stipulates that a V3 capable node must source and sink only V2 PDUs if its upward connection negotiates to the V2 protocol with its superior node. These nodes are equivalent to V2 nodes, and are considered as such for the remainder of subclause 13.9.

A V3 summit is a group of contiguously connected V3 nodes situated at the top of the domain hierarchy. The top provider in the domain must be a V3 node in order for a V3 summit to exist. The

nodes which comprise the summit are able to exchange capabilities information, but are not capable of exchanging this information with any V2 nodes in the domain. This may or may not hinder the use of a given capability. Within the V3 summit, the top provider for the domain is referred to as the "V3 summit provider".

13.9.3 Rules of capability selection

There are several factors which influence whether a given capability is to be exercised in a V3 summit. These factors are predominantly a measure of participation in the capability by other nodes. For example, suppose a capability required all nodes in the domain to perform a given task at the same time. A PDU could be exchanged between all nodes which contained a reference time for when the given task was to be performed. In order for this to happen, all nodes in the domain would have to participate, by definition. This would not be possible if the domain contained either a V2 node, or a V3 node which did not have this capability implemented. On the other hand, some capabilities may not require domain-wide or summit-wide participation.

13.9.3.1 Presence of V2 nodes in the domain

Any V3 node in a summit will be aware if it has a downward connection to a V2 node. Via the proposed V3 protocol changes, it will be possible for the V3 node to notify all V3 nodes in the summit that a V2 node has a connection to the summit. Some capabilities may be useless in a domain where V2 nodes are present. Other capabilities may not be affected at all. By knowing whether a V2 node is connected to the summit, an individual node can make the decision whether to exercise a given capability.

13.9.3.2 V3 node participation

In order to be exercised, capabilities will require a certain amount of participation by nodes in the V3 summit. Each capability will be categorized as requiring either "global" or "partial" participation. Global participation means that every node in the summit must participate in order for the capability to be exercised. Partial participation requires the cooperation of two or more nodes. This participation requirement is known as a "participation indicator". Capabilities with a partial participation indicator are known as "partial capabilities". Capabilities with a global participation indicator are known as "global capabilities". Knowing the participation indicator for a given capability allows the summit provider to make a decision as to whether that capability should be made available to the summit.

13.9.3.3 Intervening V3 nodes

Because of ASN.1 extension marks, it is possible for a V3 node to receive a PDU with components the node does not recognize. The node will, however, recognize the PDU these components are associated with and relay the unrecognized components intact along with the PDU.

In addition to extending existing PDUs, the V3 protocol allows for the addition of new PDUs. The problem with this is that nodes which do not recognize new PDUs have no idea how to route them. In this situation, it would be advantageous for a node which sourced such a PDU to know whether the intervening nodes on the path to the V3 summit provider supported the PDU. If so, the sending node could assume that each node along the path would know how to deal with the PDU. The proposed changes to the V3 protocol contains a facility whereby any node, for each capability, is informed whether that capability is supported by the intervening nodes.

13.9.3.4 Capability class

Once it has been decided that a certain capability is to be made available to the V3 summit, there must be a facility to associate a value with that capability, if need be. This is done via the Capability

Class. Each capability would be classified as being in the class MIN, MAX, or NULL. For capabilities of class MIN, the V3 summit provider is responsible for determining the minimum value to be used for a capability by selecting the minimum value requested of every participating node. Likewise for capabilities of class MAX, the V3 summit provider is responsible for determining the maximum value to be used. A capability of class NULL has no value associated with it.

Once a capability is standardized, it is expected that the associated capability class is also standardized. Every time a given capability is requested from a lower node, the summit provider should see only a single capability class ever associated with that capability. If the summit provider ever receives a request for a capability with a capability class different than that of an original request, the summit provider should consider this an error condition and ignore the request.

13.9.4 Establishing the V3 summit provider

Every V3 summit is initially established by a single V3 node. The V3 node will be the "top provider" in a single node domain. This summit provider is aware only of its own capabilities.

Each of the summit provider's capabilities require either partial or global participation within this one-node summit. None of the partial capabilities are put in the summit capabilities list because they require, by definition, the participation of two or more nodes. The question is whether the capabilities requiring global participation are put in the list. It can be argued that "all nodes" in the summit are participating and, therefore, all the node's global capabilities should be in the summit list. If any of these global capabilities can be exercised, however, they obviously must be a capability which requires only a single node's participation. Such capabilities really are not a topic of this Recommendation. If any capability can be exercised by a single node, this protocol is irrelevant to that capability. It is for this reason that the summit capabilities list is considered to be empty when the summit consists of a single V3 node.

13.9.5 Expanding the V3 summit

Once the V3 summit provider is established, the summit expands as other V3 nodes connect to the summit provider. A new leaf node generates a **Request** to be sent to the summit provider. This PDU contains the capabilities, and aspects of those capabilities, which the leaf node wishes to be placed in the summit capabilities list.

From this PDU, the summit provider is able to do two things. First, it can formulate state information about the capabilities requested by the new connection. Second, based on summit-wide state information, it can make a sound judgement as to which capabilities should appear in the summit capabilities list. An **Indication** is then generated and transmitted downward to inform the leaf node of the contents of the summit list, as well as the status of V2 nodes connected to the summit.

In this same manner, the V3 summit can continue to grow. As new leaf nodes connect to intermediate nodes, a **Request** is generated and sent up. From the information contained in this PDU, the intermediate node will update its state information. Based on state information for its subtree and the current contents of the summit capabilities list, the intermediate node will take action, possible resulting in a **Request** being transmitted up, or **Indications** being transmitted downward. Eventually, a **Request** may rise to the summit provider. If the summit provider determines that the summit capabilities list needs to be modified, it will generate and transmit an **Indication** downward for the purpose of informing other nodes in the summit.

13.9.6 Processing PDU's at a leaf node

Once a V3 leaf node establishes a connection to a superior V3 node, it must inform the superior node of its supported capabilities. For each capability requested, the leaf node will specify the capability ID and class. If standardized, the ID specified is that which is assigned to the capability in T.125. If

the class belongs to the category MIN or MAX, the leaf node must also specify an associated value. If the class is NULL, no value is specified.

In addition, the leaf node must specify the participation indicator for each capability. For any partial capability, the leaf node specifies a value of 1, indicating to the superior node that only the new leaf node itself has requested the capability below this new connection.

These capabilities are packaged into the AddList component of a single **Request.** The V2NodePresent flag will be reset in the **Request**. Because this is a newly-joining leaf node, there are no V2 nodes connected to it. Also, the RemoveList component will be empty. Once packaged, this PDU is transmitted to the superior node.

This newly-connected leaf node has no knowledge of the contents of the summit capabilities list. The node can expect to be informed of the contents of this list via a single **Indication** from the superior node. The contents of the list will be contained in the AddList component of the **Indication**. The RemoveList will be empty. The V2NodePresent flag will inform the leaf node whether a V2 node is currently connected to any node in the summit. From this point forward, the leaf node will be informed of any modifications to the summit list via an **Indication** from the superior node. Should the status of the connections to V2 nodes change, the leaf node will be informed of this via an **Indication** also.

For each capability in the AddList component of an **Indication** received, the leaf node will be informed whether that capability is supported by the summit provider, and whether that capability is supported by each intervening node between the leaf node and the summit provider. With this information, the leaf node can decide whether to exercise a given capability.

As long as a node remains a leaf node, it will never again need to send a **Request**. At such time as another node (V2 or V3) connects upward to this node, it then becomes an intermediate node. The processing of PDUs at an intermediate node is described in 13.9.7.

13.9.7 Processing PDUs at an intermediate node

When an intermediate node receives a **Request** from a downward connection, it will first update the state information for the associated connection. The node will then address each capability in the AddList and RemoveList components, as well as the V2NodePresent flag. Although several capabilities requiring action may arrive in a single **Request**, they are processed independently. As a result, the intermediate node may have need to transmit several pieces of information to adjacent nodes. The node will combine these actions so that no more than a single PDU is transmitted to any connection as a result of processing the original **Request**.

13.9.7.1 Maintaining state information

Intermediate nodes maintain a set of state information for each downward connection. Upon receipt of a **Request** from a subnode, the intermediate node updates the state information associated with that connection. Included in this information are two lists. The first list consists of all partial capabilities requested by nodes below that connection. Each of these capabilities is tagged with a count of either 1 or 2. A value of 1 signifies that this capability has only been requested by a single node below the connection. A value of 2 signifies that two or more nodes have requested the capability.

The second list consists of a set of global capabilities. In contrast to partial capabilities, global capabilities will go in the list only if they have been requested by every node below the connection. For any capability in either of the two lists, the intermediate node will also track the class of the capability and any value associated with the class.

Finally, for each downward connection, a flag is maintained. This flag indicates whether a V2 node is connected to the summit below that connection.

The algorithm for updating state information is simple. If the intermediate node receives a request to add a capability which is not currently in either of the two lists, the capability is added to the appropriate list, depending on the participation indicator. If a request is received to remove a capability that is currently in one of the two lists, then the capability is removed from that list.

If the intermediate node receives a request to add a capability which is already in one of the two lists, the processing depends on the class and participation indicator of the capability. In the case of the MIN or MAX class, the class value in the state information is updated if different from that in the **Request**. In the case of a partial participation indicator, the associated value in the state information is updated if different.

Finally, if the intermediate node ever receives a **Request** in which the V2NodePresent flag is present, the value of that flag is compared with the flag in the state information. If different, the state information is updated.

13.9.7.2 Processing the CapabilitiesNotificationRequest PDU

An intermediate node may transmit a **Request** under the following conditions:

1) The node receives a **Request** from a downward connection.

2) The node detects the loss of a V3 connection.

3) The node detects a direct connection/disconnection with a V2 node.

The following subclauses describe how an intermediate node handles these conditions.

13.9.7.2.1 Global capabilities in the add list

When an intermediate node receives a **Request** asking to add a global capability, the node examines its local copy of the summit capabilities list to determine if the requested capability is already in the list. The next action performed by the node depends on whether the capability is in the list.

If the capability is not in the list, the intermediate node must determine if this was the first **Request** to come up from the connection. If so, this means the **Request** is from a newly-connected leaf node and no further action is required. Otherwise, it may be necessary to transmit a **Request** asking that the capability be added, depending on the state information of any downward connections. If the intermediate node is able to determine from its state information that the capability is now supported by every node in its subtree, then the **Request** is transmitted. This action could eventually result in the capability being added to the summit list.

If the requested capability is in the list, this implies that every node in the summit supports the capability. Further processing depends on the class of the capability. If the capability is of class MIN, the intermediate node must determine if the associated value is the least of the values requested by every node in this subtree for that capability. Likewise, if the capability is of class MAX, the intermediate node must determine if the associated value is the greatest of the values requested by the other nodes. Should either of these two conditions be met, the intermediate node will transmit a **Request** upward, asking to add the capability with its associated value. This will serve to inform the superior node that there is a lesser MIN or greater MAX value down in its subtree for this specific capability. If the capability is of class NULL, no **Request** will be required. Finally, the intermediate node will generate an **Indication** to transmit to the lower node, notifying it to add the capability (and its associated value, if there is one) to its local copy of the summit capabilities list.

When an intermediate node receives a **Request** from a newly-connected leaf node, the intermediate node may be required to take action due to global capabilities NOT requested by the leaf node. The

first request from a leaf node contains a list of all global capabilities that node supports. If the intermediate node has global capabilities in its local copy of the summit list that are not requested by the leaf node, then the leaf node does not support those capabilities. By definition, these capabilities now become unavailable to the summit and must be removed from the summit list. At this point, the intermediate node generates a **Request** and places this capability in the RemoveList. This PDU is then transmitted upward, informing the superior node that the capability is no longer supported in this subtree. If the intermediate node has any other downward connections, it will generate an **Indication** to be transmitted downward. This **Indication** will contain the capability in the RemoveList. Finally, the intermediate node will remove the capability from its own copy of the summit capabilities list.

13.9.7.2.2 Global capabilities in the remove list

If an intermediate node receives a **Request** asking that a global capability be removed, this implies that somewhere below the connection, a node has joined the summit that does not support the capability. This capability must now be made unavailable to the summit. The processing required here is straightforward. The node generates a **Request** asking that the capability be removed and transmits it upward. If other downward connections exist, the intermediate node generates an **Indication** asking that the capability be removed and transmits it to all downward connections (except the connection which originated the **Request**). The node then removes it from the local copy of the summit capabilities list.

13.9.7.2.3 Partial capabilities in the add list

If an intermediate node receives a **Request** asking that a partial capability be added, the node must determine whether this is the first request, the second request, or greater than the second request for the capability. This can be determined from the intermediate node's state information.

If this is the first request, the intermediate node will generate a **Request** asking that the capability be added. The value supplied with the partial participation indicator will be 1, signifying that only 1 request has been made for this capability in the subtree of the intermediate node. If it is the second request, the intermediate node will generate **Request** asking that this capability be added and a value of 2 will be supplied for the partial participation indicator. If the intermediate node determines that this capability has already been asked for at least twice, no **Request** is required.

Next, the intermediate node must examine its local copy of the summit list to see if the capability is in the list. If so, the node will generate an **Indication** with the capability in the AddList component and transmit it to the requesting connection.

The intermediate node must also take the class of the capability into account. This aspect of the processing is similar to that explained previously for global capabilities. If a capability comes in with the least MIN value in the subtree or the greatest MAX value in the subtree, then the superior node must be made aware of this via a **Request**.

13.9.7.2.4 Partial capabilities in the remove list

When an intermediate node receives a **Request** asking that a partial capability be removed, the node must determine whether the removal would affect the number of requests for the capability in the subtree. If so, the intermediate node must generate a **Request** for that capability, asking that the capability be added with a partial participation indicator of 1, or ask that the capability be removed completely, depending on the change in status.

If the intermediate node can determine the direct effect on the summit list, the node can adjust its own local copy of the list and generate **Indications** to notify the other downward connections. If the

node cannot determine the effect, then any adjustment will eventually come down from a higher node which can determine the effect.

13.9.7.2.5 Processing the V2 node present flag

In practice, although not required, an intermediate node will receive a **Request** which contains a V2NodePresent flag or a capabilities list (AddList and/or RemoveList), but usually not both.

For each downward connection, an intermediate node keeps a status of whether a V2 node is connected to the summit below the connection. The first time an intermediate node learns that a V2 node has connected from below, including directly to the node itself, it will generate a **Request** containing only the V2NodePresent flag, which will be set. This **Request** will then be sent upward. Also, an **Indication** with the same contents will be sent to all downward connections, except for the connection which may have provided the original **Request**. In this manner, every node in the summit will learn of the existence of the V2 node. Each node can take this into account when deciding whether to exercise a given capability.

By the same token, an intermediate node may receive a **Request** informing the node that all previous V2 connections have broken from the subtree. With each of these **Requests**, the intermediate node can determine whether any V2 nodes are now connected to its subtree. If others still exist, the intermediate node takes no action. If all V2 nodes disconnect, however, the intermediate node generates a **Request** with the V2NodePresent flag reset and sends it upward.

If all the V2 nodes connected to the summit disconnect, the summit provider will detect this by seeing that none of the downward connections contains a V2 node in its subtree, and that there are no V2 nodes connected directly to the summit provider. When this occurs, the summit provider generates an **Indication** with the V2NodePresent flag reset and transmits it throughout the summit.

13.9.7.2.6 Processing V3 node disconnections

When a given node in the summit detects that a downward connection has been broken, all state information for that connection is discarded. The state information for the remaining downward connections, if any, could affect the contents of the summit capabilities list.

If any global capabilities are supported by every node in the summit except for one or more nodes below the lost connection, those capabilities can now be added to the summit list. If the number of requests for a partial capability in the summit drops below two as a result of losing nodes below the lost connection, those capabilities must be removed from the summit. The process to adapt to the lost connection begins with the node which detects that the connection is broken. The node handles this by treating the situation as if a **Request** has been received asking that every capability in the state information for that connection be removed. The connection would then process this in the manner as described in 13.9.7.2.

13.9.7.3 Processing the CapabilitiesNotificationIndication PDU

At times, an intermediate node will receive an **Indication** from its upward connection. The indication will instruct the intermediate node to either add or remove capabilities from the summit list. These modifications will be made to the "local" summit list, and the indication will be forwarded to each downward connection. If this is the first **Indication** sent on a connection, the intermediate node will simply transmit the entire summit list via the AddList component of the **Indication**. If an intermediate node does not support a capability, the node must reset the flag which indicates intermediate node support.

13.9.7.4 Processing PDUs at the summit provider

The manner in which the summit provider processes PDUs is nearly identical to that of an intermediate node. Upon receipt of a **Request** from a downward connection, the summit provider will update the state information for the associated connection the same as an intermediate node, as described in 13.9.7.1. The summit provider deals with global capabilities and partial capabilities identically to intermediate nodes, with the obvious exception that no **Requests** will be sent to a superior node. With each request to add or remove a capability, the summit provider decides whether the summit list is to be modified. If deemed so, the summit provider will generate an **Indication** filled in with the appropriate capabilities to add or subtract and transmit this PDU down appropriate connections.

This protocol has been designed so that each node that determines that the summit capabilities list must change will initiate the change by notifying its downward connections, and by notifying its superior node. In this manner, the summit list will get update across the summit.

If a node does not have enough information to determine whether a change to the summit list can be made, it will send a **Request** up to the next higher node, who may in turn be able to decide. All decisions of this matter can ultimately be resolved by the summit provider.

13.10 Protocol version arbitration

This Recommendation defines three protocol versions (see clause 9 for details). However, Recommendation T.120 specifies that version 1 is obsolete. A node can refuse to support version 1 by requiring version 2 (or higher) during connection establishment (if a remote node *only* supports version 1, the connection can be rejected). For nodes that wish to support version 1 for backward compatibility, it is suggested that no mixing of other protocol versions in the same domain be allowed [as per the rules of Recommendation T.125 (1994)].

Recommendation T.120 will be (or has been) modified to allow for versions 2 or 3. The protocol version number parameter of the domain parameters shall be used to negotiate the capability to source and sink version 3 PDUs. A node which implements version 3 must also be able to source and sink version 2 PDUs.

The protocol version number parameter is negotiated independently for each connection (with one restriction, which is described below). Each version 3 node must maintain information regarding protocol versions selected by its upward and downward connections. Translation between versions must be provided for on any non-leaf node which implements version 3 of the protocol (this is discussed in detail in 13.11).

A version 3 capable node that has negotiated the use of version 2 on its upward connection will *only* negotiate the use of version 2 on its downward connections. This has the effect of concentrating the use of version 3 at the top of the domain hierarchy (this area of version 3 usage is referred to as the "V3 Summit").

A version 3 capable node that has negotiated the use of version 3 on its upward connection will attempt to negotiate the use of version 3 on its downward connections (although version 2 will be accepted when necessary).

13.11 Protocol version interoperability

Translation from version 2 to version 3 is accomplished by copying the parameters from a version 2 PDU to an equivalent version 3 PDU.

Translation from version 3 to version 2 is accomplished by copying the parameters from a version 3 PDU to an equivalent version 2 PDU. Version 3 parameters that do not correspond to valid version 2 parameters are discarded. Version 3 PDUs with no version 2 equivalent are also discarded.

When a node receives a version 3 PDU that it does not recognize, the PDU is discarded (this is not an error).

When a node receives a version 3 PDU that is *does* recognize, the PDU is processed normally. If the PDU contains any parameters that the receiver does not recognize, they are simply ignored. If the PDU must be forwarded to another node using protocol version 3, all parameters are forwarded intact (including those that were not locally recognized).

Considerations for using non-standard parameters in multi-protocol conferences:

NOTE 1 – Prior to utilizing non-standard parameters, the implementer should ensure version 2 and version 3 nodes interoperate successfully. No information comprising a version 3 PDU in the form of a non-standard parameter, or as a standard parameter beyond the extension marker, can be delivered to a version 2 node via a version 2 PDU. Non-standard parameters are therefore only suitable for use for extensions whose absence will not effect basic operation at either V2 or V3 nodes (fail safe operation). The precise procedures to be used by an MCU or other intervening nodes should be thought about in detail to ensure successful interoperability.

NOTE 2 – When using non-standard parameters, it must be kept in mind that such usage be an interim solution until equivalent functionality has been standardized.

NOTE 3 – This Recommendation describes a mechanism for information exchange between nodes for the purpose of ascertaining non-standard capabilities available to a version 3 summit. Scalability aspects should be considered, it being likely that in larger conferences the probability will be decreased that any non-standard capabilities will be found to be in common.

NOTE 4 – The processing cost of translating between V2 and V3 PDUs should be considered. Besides an MCU having to generate version 2 PDUs by dropping certain parameters from an equivalent version 3 PDU, as well as generating version 3 PDUs by generating parameters from an equivalent version 2 PDU, the cost of actually including non-standard parameters in PDUs is relatively high. The minimal encoding appears to be about 8 octets, excluding the size of any data passed in the parameter. This is equal to the MCS header of standard parameters in a data PDU and is an additional drain on bandwidth. Upon receipt of any non-standard parameter, a version 3 node must parse its key and then look up the manufacturer codes to determine its meaning, if known. This is a drain on processing power that should be minimized whenever possible. Procedures to be followed by nodes that do not understand non-standard parameters are specified in this subclause.

ANNEX A

Multicast adaptation protocol

A.1 Scope

This Annex defines a protocol that allows multicast services to be used in a T.120 conference. This protocol is independent of the type of network on which it is used. Issues that are specific to a given network, such as address format and multicast group address allocation, are outside the scope of this Annex. This protocol also allows for the transfer of unreliable data.

Figure A.1-1 illustrates how the Multicast Adaptation Protocol (MAP) fits into the T.120 model. MAP handles the translation of T.122 services to multicast services. Underneath MAP can be a variety of data delivery protocols, such as TCP, UDP, MTP/SO, and RMP (MTP/SO and RMP are example proposals for reliable multicast on IP networks).

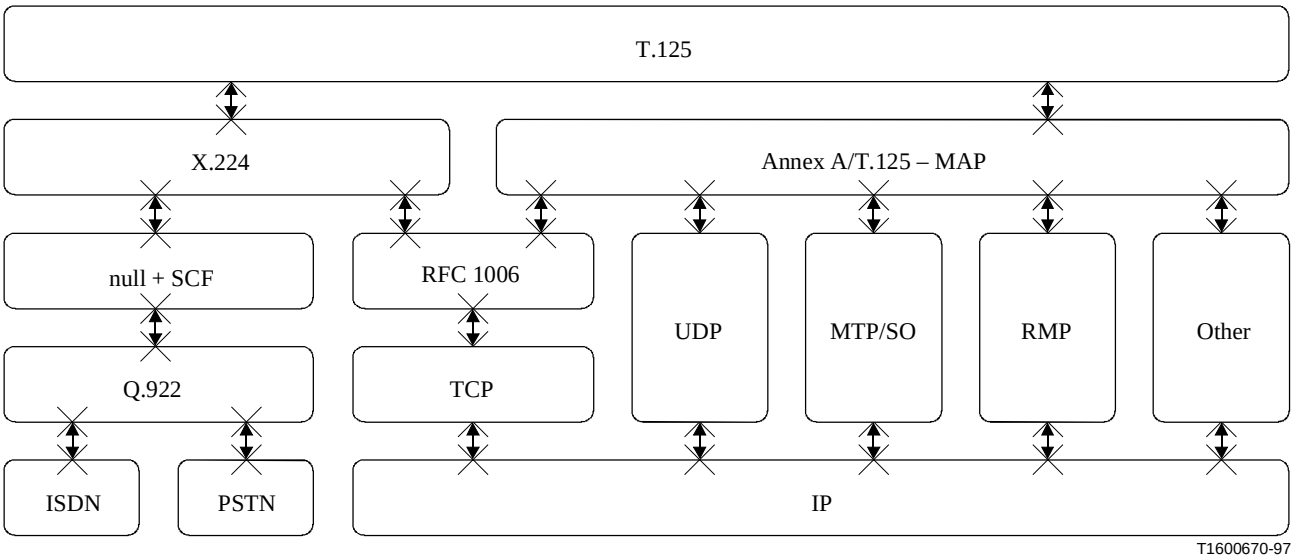

Figure A.1-1/T.125 – T.120 model showing relationship of MAP to other layers

A.2 Normative references

The following ITU Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Annex. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Annex are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The ITU-T Secretariat maintains a list of the currently valid ITU-T Recommendations.

– CCITT Recommendation T.35 (1991), *Procedure for the allocation of CCITT defined codes for non-standard facilities.*

– ITU-T Recommendation T.120 (1996), *Data protocols for multimedia conferencing.*

– ITU-T Recommendation T.122 (1998), *Multipoint Communication Service – Service definition.*

– ITU-T Recommendation T.123 (1996), *Network specific data protocol stacks for multimedia conferencing.*

– ITU-T Recommendation X.214 (1995) | ISO/IEC 8072:1996, *Information technology – Open Systems Interconnection – Transport service definition.*

– ITU-T Recommendation X.680 (1997) | ISO/IEC 8824-1:1998, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*

– ITU-T Recommendation X.691 (1997) | ISO/IEC 8825-2:1998, *Information technology – ASN.1 encoding rules – Specification of Packed Encoding Rules (PER).*

A.3 Definitions

This Annex defines the following terms.

A.3.1 data flow: This is a stream of data being transmitted from a sender to all interested receivers. Data flows are uniquely identified by the following five fields: sender ID; channel ID; reliability level; priority; and data type. Each data flow has a separate sequence counter, and may be routed differently from every other data flow.

A.3.2 data type: All MCS data PDUs are either non-uniform (source-ordered), uniform (globally-ordered); or proxy data (which is non-uniform data that is to be sent via multicast by the Multicast Group Provider on behalf of the originator).

A.3.3 MAP connection: This is an MCS connection between two nodes through MAP. One MAP connection is, in turn, made up of one or more transport connections. There is no need for adjacent nodes to have common multicast protocols in order to make a MAP connection.

A.3.4 MCS connection: This is a connection between two MCS providers that are adjacent in the domain hierarchy. This can be through MAP or directly through one of the protocol stacks defined in Recommendation T.123.

A.3.5 metachannel: This is an MCS channel at a given reliability level and in a given priority range. MAP assigns multicast groups to metachannels.

A.3.6 multicast: This refers to the ability to send a packet of data to more than one recipient with one operation.

A.3.7 multicast group: This is a group of receivers that have expressed interest in receiving certain data via multicast. The group is uniquely identified by a Multicast Group Address.

A.3.8 multicast group address: This is a network address that uniquely identifies a Multicast Group. Data sent to this address are delivered, via multicast, to all reachable nodes that have joined the Multicast Group.

A.3.9 multicast group provider: This is a node that either has no upward MAP connection, or has an upward MAP connection, but no common multicast protocols with the node above. The Multicast Group Provider is responsible for associating metachannels with multicast groups. Once this association is made, data transmitted to the metachannel can be delivered via multicast.

A.3.10 multicast island: This is a group of contiguous MAP connections within an MCS domain that have agreed to the use of a particular multicast transport protocol for a given metachannel.

A.3.11 multicast sender: This is the node in a multicast island that transmits a given data flow via multicast. For non-uniform data, this is the point-of-entry node. For uniform and proxy data, this is the Multicast Group Provider.

A.3.12 point-of-entry: This is the node in a multicast island at which a particular data flow enters the island.

A.3.13 reliability level: This refers to the level of guarantee that is provided for data transfer. MAP supports two reliability levels: fully reliable (guaranteed delivery); and unreliable (no guarantee).

A.3.14 sequence number: This is a rolling number that is associated with each data flow. This number is used for the re-assembly of unreliable data, and to facilitate seamless transitions between unicast and multicast data transfer.

A.3.15 transport connection: This is a unicast connection through one of the protocol stacks below MAP.

A.3.16 transport protocol: These are the protocols that are used by MAP to deliver data. There are four types of transport protocols: reliable unicast; reliable multicast; unreliable unicast; and unreliable multicast. Nodes in the domain arbitrate transport protocols before using them.

A.4 Abbreviations

This Annex uses the following abbreviations.

ATM	Asynchronous Transfer Mode
IP	Internet Protocol
ISDN	Integrated Services Digital Network
MAP	Multicast Adaptation Protocol
MAPPDU	Multicast Adaptation Protocol – Protocol Data Unit
MAPSAP	Multicast Adaptation Protocol – Service Access Point
MCS	Multipoint Communication Service
MTP/SO	Multicast Transport Protocol/Self-Organizing
PDU	Protocol Data Unit
PSTN	Public Switched Telephone Network
RMP	Reliable Multicast Protocol
SCF	Synchronization and Convergence Function
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

A.5 Overview

When using unicast-only transport protocols, T.125 creates point-to-point connections to "construct" a multipoint conference. Data flows across these connections, following the hierarchy of the domain, until it reaches all intended recipients. On networks that are capable of true multicast, this is extremely wasteful, since the same data is sometimes transmitted multiple times on the same physical device. This Annex describes a protocol which allows for the use of multicast data transfer in a T.120 conference. This protocol is referred to as the Multicast Adaptation Protocol (MAP).

MAP is completely independent of the underlying network. It is insulated from the network by a transport protocol that meets the service requirements defined in 5.2. Some of the examples in this Annex reference protocols that are implemented on top of the Internet Protocol (IP), but MAP can also be used on other types of multicast networks, such as ATM multicast or Frame Relay multicast.

To manage multicast, MAP needs to know about more than just unicast connections. MAP also needs to know about domains and channels. This information allows it to properly group unicast connections, and use multicast in the most efficient manner possible. Management of the domains and channels is still part of MCS, but this information is shared with MAP as needed.

At a high level, the problem of using multicast in T.120 can be broken into the following three parts:

1) *Allocation of multicast groups*: Before multicast can be used, someone must allocate a multicast group address that is not currently being used by anyone else on the reachable network. MAP determines which node allocates the address. How the allocation is done, however, is outside the scope of MAP (this is the focus of work in other standards groups).

2) *Distribution of multicast groups*: Once a multicast group address is allocated, it is necessary to distribute it to those nodes that need to use it. MAP handles the distribution of multicast group addresses.

3) *Management of multicast data transfer*: It is not enough to simply send the address of a multicast group and tell everyone to start using it. There are many reasons why multicast may not work uniformly throughout the domain. It is therefore necessary to "test" the underlying network to see who can receive multicast data (and from whom). MAP performs this test and uses multicast wherever it can. If a given node cannot receive data via multicast for some reason, then data is delivered to that node via unicast.

MAP also handles the dynamic selection of both unicast and multicast transport protocols. These protocols are arbitrated automatically between any two nodes that are communicating with MAP. Once these protocols are selected, MAP can facilitate the efficient delivery of data within the domain. MAP is a separate layer above these transport protocols.

A.5.1 The use of multicast

MAP supports the transmission of data at two different reliability levels, which directly affects the use of multicast. Different reliability levels will typically use different transport protocols, and will therefore usually use different multicast groups. MAP may also use different multicast groups for different priority ranges (to allow different priorities to be independently flow controlled). This Annex uses the term *metachannel* to indicate an MCS channel at a given reliability level and within a given priority range.

To enable the use of multicast, MAP assigns a multicast group to each metachannel in each multicast island (multicast islands are fully described in A.5.2). The only exception to this is User ID channels, which are not assigned multicast groups (there is only one potential receiver listening to a User ID channel, which violates the concept of multicast).

MAP can exercise a great deal of flexibility in how it performs group assignments. It could, for example, assign the same multicast group to every metachannel (to minimize the number of multicast groups). Or it could assign a separate multicast group to each metachannel (to reflect the fact that each metachannel can have a different set of interested receivers). These group assignments are controlled by a set of policies that can adapt over time, as network technology evolves.

When data is transmitted, MAP will use the channel, reliability level, and priority to select the appropriate multicast group. This allows the data to be delivered directly to the intended recipients, without propagating through the domain hierarchy (when multicast is working).

A.5.2 Multicast islands

Multicast cannot always be used to deliver data to every node in a domain. Some nodes may be connected to the domain via networks that are not capable of multicast (such as PSTN or ISDN). It is also possible for two groups of multicast capable nodes to be separated in the domain hierarchy by a multicast incapable connection. It is therefore necessary to recognize the existence of "multicast islands" within a domain.

A multicast island is a group of contiguous MAP connections within an MCS domain that have agreed to the use of a particular multicast transport protocol for a given metachannel.

Figure A.5-1 depicts a conference of 10 nodes. Each of the connections making up the conference is labeled with the transport protocol(s) that the adjacent nodes have agreed to use for data transfer. "PSTN" and "ISDN" refer to the protocol stacks defined in Recommendation T.123. "TCP | MTP/SO" indicates a MAP connection that is using TCP for unicast transport, and MTP/SO for multicast transport (for a given metachannel). Nodes 1, 2, 4, and 5 are in multicast island A,

while Nodes 3, 6, 7, 8, and 9 are in multicast island B. These islands are separate because the connection between Nodes 1 and 3 is PSTN (this connection cannot be used to convey multicast control information).

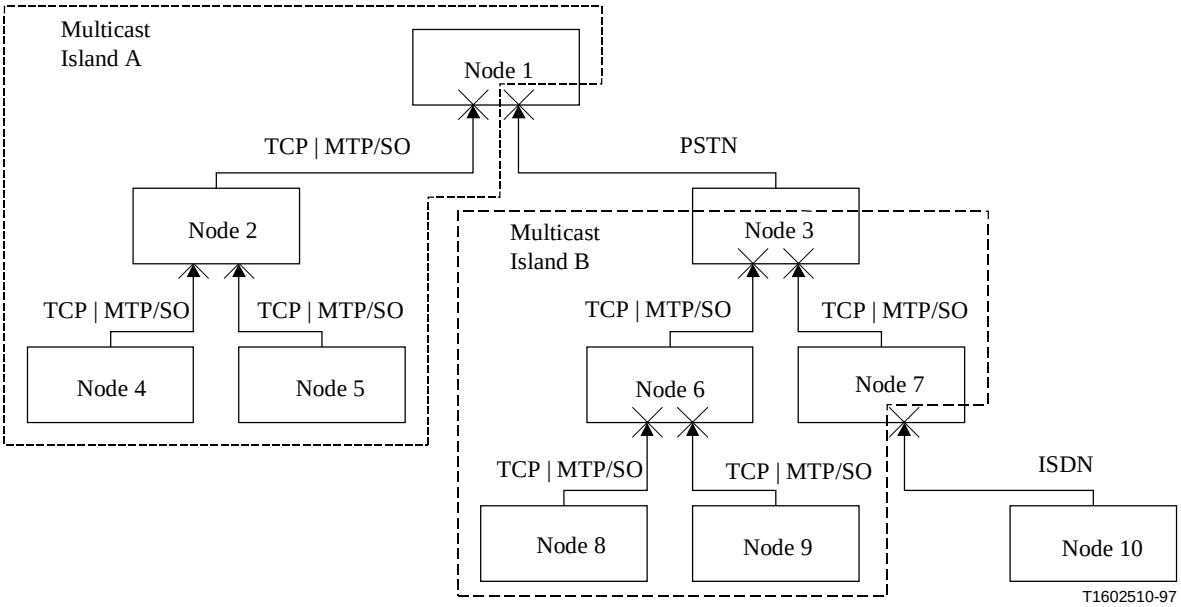

Figure A.5-1/T.125 – Multicast islands split by non-multicast connections

Multicast islands can also be split as a result of selecting different multicast transport protocols for a given metachannel. Figure A.5-2 depicts a conference with the same domain hierarchy as Figure A.5-1, but with different protocol selections. Nodes 1, 2, 3, 4, and 5 are in multicast island A, and have agreed to use RMP for multicast transport. Nodes 3, 6, 7, 8, 9, and 10 are in multicast island B, and have agreed to use MTP/SO for multicast transport. Note that Node 3 is in both multicast islands, acting as a bridge between them.

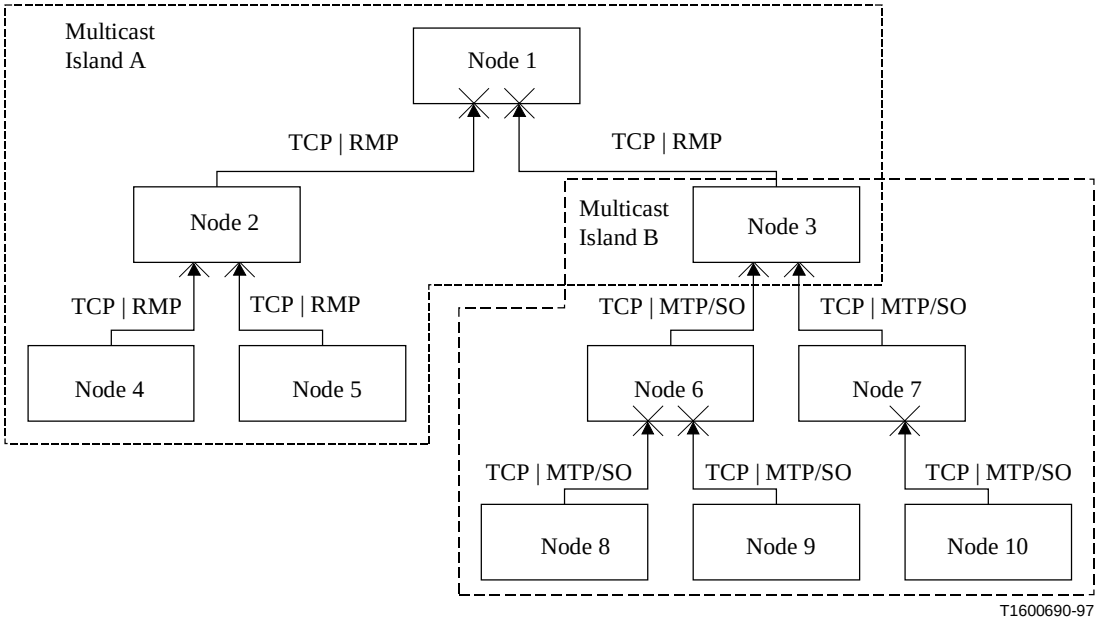

Figure A.5-2/T.125 – Multicast islands split by multicast protocol selection

Protocol selection is performed differently for unicast and multicast protocols. Unicast protocols are selected *per MCS connection*. The selected unicast protocols are used to transfer data between nodes that are adjacent in the MCS domain hierarchy. It is perfectly valid for different connections to use different unicast protocols. Multicast protocols are selected *per metachannel*. When mapping a metachannel to multicast, each MAP node will select the multicast protocols to be used by nodes beneath it. This selection process defines the boundaries of the multicast island *for that metachannel*. It *is* possible for multicast island boundaries to vary for different metachannels.

As a rule, each node should select the multicast protocols that will result in fewer, larger multicast islands. When possible, each node should select the same multicast protocol that was selected for it by the node above. In Figure A.5-2, Nodes 6 and 7 must have indicated that they do not support RMP, or else Node 3 would have selected it. This split in the multicast islands is the result of support for different multicast transport protocols.

This Annex refers to the unicast and multicast protocols collectively as *transport protocols*. Nodes that are adjacent in the MCS domain hierarchy will perform arbitration in order to agree on the set of transport protocols that can be used in the conference. During arbitration, the adjacent nodes *must* select a single reliable unicast protocol (and, optionally, a single unreliable unicast protocol). They will also agree on a set of zero or more multicast protocols that can be used at each reliability level. When mapping a metachannel to multicast, a node will only select multicast protocols that some downward nodes have agreed to support during arbitration.

A.5.3 Multicast group providers

Within each multicast island, there is one node who is responsible for the allocation of multicast group addresses. This node is referred to as the Multicast Group Provider.

The Multicast Group Provider is the node that either has no upward MAP connection, or has an upward MAP connection, but no common multicast protocols with the node above. The Top Provider for the domain may also be the Multicast Group Provider for a multicast island, but this is not necessary. In Figures A.5-1 and A.5-2, Node 1 is the Multicast Group Provider for multicast island A, while Node 3 is the Multicast Group Provider for multicast island B.

A.5.4 Data ordering

MCS allows for two types of data ordering: non-uniform (source ordered); and uniform (globally ordered).

Non-uniform data is normally sent to the multicast group at its initial "point-of-entry" to the multicast island. The point-of-entry is simply the node at which the data enters the multicast island. This is not the node at which the sending application is attached if that node is outside the multicast island. In Figure A.5-1, if Node 10 transmits data, Node 7 is the point-of-entry for multicast island B, while Node 1 is the point-of-entry for multicast island A.

Optionally, a point-of-entry node can choose to use the Multicast Group Provider as a multicast proxy. When using this option, non-uniform data is first sent upward via unicast, where it is then sent to the multicast group by the Multicast Group Provider. This is discussed more in A.8.4.

In order to guarantee global ordering, uniform data must be routed upward to the Top Provider via unicast. This data flow has an "upward" point-of-entry, but this is not an interesting concept, and will not be discussed further. The data are sent to the multicast group at the "downward" point-of-entry on its way back down (for uniform data, the downward point-of-entry will simply be referred to as the point-of-entry). In Figure A.5-1, if Node 10 transmits uniform data, it will first be unicast upward to the Top Provider. On the way back down, Node 1 is the point-of-entry for multicast island A, while Node 3 is the point-of-entry for multicast island B. For uniform data, the point-of-entry will *always* be the Multicast Group Provider for each multicast island.

A.5.5 Use of multicast and unreliable unicast

All control data is sent over the reliable unicast connections. **MAPData** is the only MAPPDU ever transmitted via multicast or unreliable unicast. Table A.5-1 summarizes how MAP uses each of the four transport protocol types.

Table A.5-1/T.125 – Use of transport protocol types

	Unicast	Multicast
Reliable	Any MAPPDU **MAPData** can contain any MCSPDU	Only **MAPData** **MAPData** can only contain one of the following MCSPDUs: **MCSSendDataRequest** **MCSSendDataIndication** **MCSUniformSendDataIndication**
Unreliable	Only **MAPData** **MAPData** can only contain one of the following MCSPDUs: **MCSSendDataRequest** **MCSSendDataIndication** **MCSUniformSendDataRequest** **MCSUniformSendDataIndication**	Only **MAPData** **MAPData** can only contain one of the following MCSPDUs: **MCSSendDataRequest** **MCSSendDataIndication** **MCSUniformSendDataIndication**

A.6 Use of MAP

This subclause describes how MCS manages connections and data transfer when using the Multicast Adaptation Protocol (MAP). It is not the intention of this subclause to exhaustively describe the interactions between MCS and MAP. Rather, this subclause describes how MCS behaves differently than when using a unicast-only protocol stack (such as those currently defined in Recommendation T.123).

To completely describe the differences in how MCS deals with MAP, it is useful to break T.125 into its four functional units: domain management; channel management; data transfer; and token management. Each of these units is described briefly below.

A.6.1 Domain management

When making a connection to a remote MCS Provider using a unicast-only protocol stack, MCS creates one Transport Connection (TC) for each data transfer priority (the number of priorities is arbitrated as part of the domain parameters). Control MCSPDUs are always sent on the TC with the highest priority. Data MCSPDUs can be sent on all TCs (which one depends on priority).

When making a connection to a remote MCS Provider using MAP, MCS creates only a single MAP Connection (MC), regardless of the number of data transfer priorities. Control MCSPDUs are sent on this MC. Data MCSPDUs are not actually sent on *any* MC, but are passed off to MAP as a whole (this is discussed in the subclause on Data Transfer).

The Connect MCSPDUs perform the same role of initializing a new MCS connection, and arbitrating the domain parameters. **Connect-Initial** is sent by the calling MCS as the first MCSPDU over a new

MC. **Connect-Response** is the first MCSPDU sent by the called MCS in reverse over a new MC. Since there is only one MC per MCS connection, there is no need for **Connect-Additional** and **Connect-Result**.

For MAP to properly perform its routing duties, MCS must share some information regarding all newly-created connections. This includes: what domain the connection is bound to; whether it is an upward or downward connection; and what MCS protocol version will be used on the connection. Furthermore, when MCS binds the first MC to a domain, it must share with MAP how many priorities and reliability levels are to be used in the domain.

All Domain MCSPDUs that deal with domain management are sent over an MC in exactly the same way they are sent over a TC when using a unicast protocol stack.

A.6.2 Channel management

All Domain MCSPDUs that deal with channel management are sent over an MC in exactly the same way they are sent over a TC when using a unicast protocol stack.

For MAP to be able to multicast data on behalf of MCS, it must know about channels (so that it knows where to send the data). MCS must share the following information with MAP:

1) For each active channel, MAP must know which domain the channel is associated with, and what type of channel it is.
2) For each active channel, MAP must know which MCs lead to joined users.
3) For each active channel, MAP must know if MCS needs to receive the data. MCS will need the data if it must redirect it to a locally-attached user application or a different protocol stack.

Static channels are considered active if there is at least one joined user application in the subtree of the local node. Dynamic channels are considered active as soon as they are allocated.

A.6.3 Data transfer

The most significant difference in using MAP lies in the handling of data transfer. When using a unicast-only protocol stack, MCS assumes all responsibility for redirecting data to TCs that lead to interested users. When using MAP, much of this responsibility is delegated to MAP.

MCS typically sends a given data MCSPDU to MAP only once, no matter how many nodes need to receive it. The only exception to this occurs if MCS has arbitrated a different protocol version for some of the MCs that are bound to the domain. MCS must send the data MCSPDU properly encoded for each active protocol version. MAP will see to it that everyone who needs the data will get it.

When MCS receives data from MAP, there is no need for MCS to redirect the data back to MAP (unless there are multiple protocol versions in use, requiring MCS to perform version translation).

When passing data to MAP, MCS uniquely identifies the target set of receivers by specifying a domain identifier, an MCS protocol version, and a data flow identifier. This is enough information since MAP already knows which MCs are bound to the domain, and which protocol version is in use on each. MAP can then use the data flow identifier to properly route the data via unicast and/or multicast. Note that the data flow identifier also communicates the relative priority of the data to MAP. This priority will always be 0 to $N-1$, with 0 being the highest priority (N is the number of priorities that MCS indicated are to be used in the domain). No priority translation between MCS and MAP is required.

Consider Figure A.6-1, which shows a conference consisting of four nodes. Two of the three connections making up the conference are MCs. The third connection, binding Node 3 to Node 2, is a TC through the PSTN protocol stack defined in Recommendation T.123. It is useful to examine

how the MCS at Node 2 reacts to data sourced from each of the four nodes. For this example, assume that non-uniform data is sent on a channel for which there are joined users at every node. Note that this example only discusses the handling of data at Node 2.

1) When data originates on Node 1, it is received by MAP on Node 2. It is then passed to MCS who, in turn, sends the data to the TC that leads to Node 3. MCS will NOT send the data back to MAP.

2) When data originates on Node 2, MCS sends the data to the TC that leads to Node 3. It also sends the data to MAP (once).

3) When data originates on Node 3, it is passed to MCS on Node 2 by the PSTN protocol stack. MCS then passes the data to MAP (once).

4) When data originates on Node 4, it is received by MAP on Node 2. It is then passed to MCS who, in turn, sends the data to the TC that leads to Node 3. MCS will NOT send the data back to MAP.

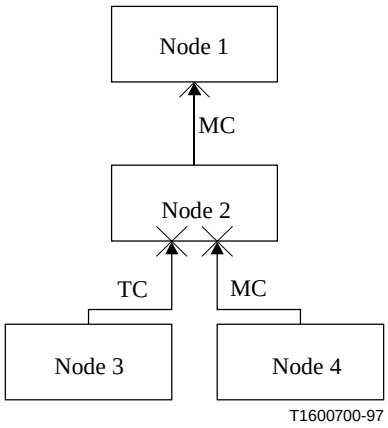

Figure A.6-1/T.125 – Sample 4-way conference

As this example illustrates, MCS trusts MAP to perform the routing for all MCs that are logically grouped (by domain and protocol version). This delegation is necessary, since only MAP knows when multicast is actually being used, and whether or not it is working. The abstraction presented to MCS is that multicast is *always* working.

A.6.3.1 Differing protocol versions

As mentioned above, it is necessary to differentiate data PDUs based on protocol version. This is simply because the format of the data PDUs is different. If half of the MCs in a domain are using one protocol version, while the other half are using a different one, then MCS must perform a translation on any data passing between the two groups. This requirement creates the only case where MCS may need to redirect data back to the same MAP protocol stack from which it came (after translation, of course).

A.6.3.2 Uniform data handling

Because of the global ordering requirement, it is necessary to treat uniform data a little differently than non-uniform data. MAP can determine whether a PDU is uniform or not by examining the data flow identifier.

When MAP receives a uniform data PDU from MCS, it will check to see if there is an upward MC for the specified domain and protocol version. If so, MAP will send the PDU upward via unicast

(since it must be a request). If there is no upward MC for the specified domain and protocol version, MAP will send the PDU downward to all joined recipients (via multicast, if possible).

When MAP receives a uniform data PDU from a downward MC, it will check to see if there is an upward MC for the same domain and protocol version. If so, MAP will forward the PDU upward via unicast (the PDU will NOT be sent to MCS). If there is no upward MC for the same domain and protocol version, MAP will send the PDU to MCS.

When MAP receives a uniform data PDU via multicast or from the upward MC, the PDU will be sent to MCS (if necessary), as well as being forwarded downward to those nodes not receiving data via multicast.

A.6.3.3 Request vs. indication

When sending non-uniform data via MAP, it is possible for the data to arrive at nodes both above AND below the current node. For this reason, it is necessary to specify whether MCS should identify such data PDUs as requests or indications.

If the upward connection is an MC using the same MCS protocol version, then the PDU sent to MAP should be a request. Otherwise, it should be an indication. In other words, data that will ONLY flow downward is an indication, while data that flows upward is a request (even if it also flows downward).

This creates the situation that it is possible for a node to receive a **Send-Data-Request** from a node that is hierarchically above it. MCS needs to recognize that this is not an error condition when the data comes from MAP, by treating the PDU as if it was properly oriented.

Note that this situation does not exist for uniform data, since the requests are unicast upward first. Uniform data will only be multicast from the top, which guarantees that all receivers are getting an indication from a node that is hierarchically above them.

A.6.3.4 Proxy data

When sending non-uniform data, MAP can either multicast the data at the point-of-entry, or it can decide to use a multicast proxy. How the point-of-entry node makes this decision is beyond the scope of this Annex.

Multicasting through a proxy simply means that the data is unicast upward until it reaches the Multicast Group Provider, who then multicasts it downward. This is similar to the handling of uniform data, except that the data do not have to go all the way up to the Top Provider before beginning its downward journey (it only has to go to the Muticast Group Provider). This technique can be used to reduce data flow state information in very large conferences. It can also be used to allow data to be multicast when the point-of-entry node cannot itself transmit via multicast.

The use of proxy data is transparent to MCS. To MCS, it just looks like non-uniform data.

A.6.4 Token management

All Domain MCSPDUs that deal with token management are sent over an MC in exactly the same way they are sent over a TC when using a unicast protocol stack.

A.7 Use of transport protocols

This subclause describes the services that MAP expects to receive from the transport protocols.

A.7.1 Unicast transport protocols

The services required from the unicast protocols are a subset of those defined in Recommendation X.214. MAP does not use exactly the same parameters described in Recommendation X.214, nor does it make use of expedited data. This subclause discusses the services that MAP *does* use.

During connection establishment, MAP will create one or more unicast connections to the adjacent node. These connections are used to carry both control data and user data. If more than one connection is made, they are of varying priorities, allowing MAP to indicate to the transport protocol the relative importance of a given MAPPDU.

For the reliable unicast protocols, MAP assumes that the service is error-free and ordered. Neither of these assumptions is made for the unreliable unicast protocols.

MAP also assumes that all reliable unicast protocols are capable of segmenting MAPPDUs that are too large to transmit as a single unit. These MAPPDUs should be re-assembled on the receive side before they are sent to MAP. To prevent the need to handle arbitrarily large MAPPDUs, it is suggested that a limit of 16 384 bytes be placed on MAP. MAP must ensure that no MAPPDUs exceed this size.

For reliable unicast, MAP assumes a "connection-oriented" service. This implies that both sides of the connection are aware of the other, and that some protocol was involved in the connection establishment process. This further implies that both nodes are maintaining state information about the connection between them, and can automatically sense packet loss and other network problems. When creating a reliable unicast connection, MAP participates by sending protocol of its own (after the transport protocol indicates that it is okay to do so).

For unreliable unicast, MAP assumes a "connection-less" service. This simply means that MAP makes no assumptions about the service that is provided. It does not assume that data will be ordered. It does not assume that packet loss, or any other network problems, will be detected.

The unicast primitives utilized by MAP are summarized in Table A.7-1.

Table A.7-1/T.125 – Unicast transport protocol service definition

Primitive	Parameters	Reliable	Unreliable
T-Connect request	Called address, priority Calling address	X	
T-Connect indication		X	
T-Connect response		X	
T-Connect confirm		X	
T-Data request	User data	X	
T-Data indication	User data	X	
T-Unit-Data request	Destination address, Priority, user data		X
T-Unit-Data indication	Source address, User data		X
T-Disconnect request		X	
T-Disconnect indication	Reason	X	

A.7.1.1 T-Connect

The *T-Connect* primitives are used to create reliable unicast connections to remote nodes.

The calling node will invoke *T-Connect request* to start the creation of a new connection. If the remote node can be found, a *T-Connect indication* is delivered to MAP on that node. MAP will ordinarily accept the connection by invoking *T-Connect response*. This in turn results in a *T-Connect confirm* at the calling node. Once this process is complete, the connection is ready and MAP will continue with its portion of connection establishment.

Parameter definitions

Called Address: This is the network address of the node to be called.

Calling Address: This is the network address of the node placing the call.

Priority: This field is used to specify the priority of this connection (relative to all other connections). The permissible range is 0 through 15, with 0 being the highest priority.

A.7.1.2 T-Data

The *T-Data* primitives are used to transfer MAPPDUs to the remote node via reliable unicast.

When MAP wants to send a MAPPDU via reliable unicast, it will invoke *T-Data request*. This will eventually cause a *T-Data indication* at the remote node.

Parameter definition

User Data: This is a variable length octet string. It is used to carry MAPPDUs.

A.7.1.3 T-Unit-Data

The *T-Unit-Data* primitives are used to transfer MAPPDUs to the remote node via unreliable unicast.

When MAP wants to send a MAPPDU via unreliable unicast, it will invoke *T-Unit-Data request*. This will eventually cause zero or more *T-Unit-Data indications* at the remote node. The order of the requests may not match the order of the indications. Furthermore, there may not be a one-to-one correspondence between requests and indications (some requests may never result in an indication, while other requests may result in multiple indications).

Parameter definitions

Destination Address: This is the network address of the node that is to receive the data.

Source Address: This is the network address of the node that originated the data.

Priority: This field is used to specify the priority of this MAPPDU (relative to all other MAPPDUs). The permissible range is 0 through 15, with 0 being the highest priority.

User Data: This is a variable length octet string. It is used to carry MAPPDUs.

A.7.1.4 T-Disconnect

The *T-Disconnect* primitives are used to shut down existing reliable unicast connections.

One side will typically invoke *T-Disconnect request*, which breaks the connection and causes the other side to receive a *T-Disconnect indication*. However, if the connection is dropped because of network problems, both sides will receive a *T-Disconnect indication*.

Parameter definition

Reason: This contains the reason why a connection was dropped. This is one of: user initiated; provider initiated; or unspecified failure.

A.7.2 Multicast transport protocols

This subclause describes the services required by MAP to perform multicast group management. This includes joining a multicast group prior to data flow, transmission and reception of data via multicast, and leaving a group when multicast services are no longer required.

All data flow in a multicast island is initially unicast. In response to the flow of data, MAP will allocate and distribute multicast group addresses, to be used in optimizing network traffic. At each node in the island, MAP will decide if it is necessary for that node to join an assigned group. If it is, it will call upon the multicast protocols to perform this service.

When MAP decides to join a multicast group, it can specify one of three modes: transmit-only; receive-only; or transmit/receive. Some multicast protocols may be able to optimize their behavior by knowing which combination is required, so this information is passed to them during a group join operation.

For reliable multicast, MAP assumes the service is error-free, possibly after an indeterminate loss of the initial data from each transmitter. This assumption is not made for unreliable multicast.

MAP never assumes that multicast data will be ordered. For reliable multicast, however, MAP does assume that all gaps caused by out-of-order data will be filled. If a new node begins transmitting to a multicast group, the reliable multicast protocol should *not* begin forwarding that data to MAP until reception from that node has stabilized (i.e. all gaps in reception will be filled). This means that the received data stream may begin after different initial losses at different receivers.

It is not necessary for the multicast protocols to support segmentation. This is because **MAPData** is the only MAPPDU ever sent via multicast, and it will be segmented to the proper size by MAP before being sent to the multicast protocol.

The multicast primitives utilized by MAP are summarized in Table A.7-2.

Table A.7-2/T.125 – Multicast transport protocol service definition

Primitive	Parameters
MT-Join request MT-Join confirm	Multicast group address, priority, tx/rx flags
MT-Data request MT-Data indication	User data User data
MT-Leave request MT-Leave indication	 Reason

A.7.2.1 MT-Join

The *MT-Join* primitives are used to join a multicast group.

There are flags in the *MT-Join request* that allow MAP to specify whether it intends to transmit, receive, or both. The state of these flags can be changed by issuing additional *MT-Join request*s, with no need to invoke *MT-Leave request*.

The *MT-Join confirm* tells MAP whether or not the pending *MT-Join request* was successful.

Parameter definitions

Multicast Group Address: This is the address of the multicast group to be joined. A NULL value indicates that the transport protocol should create a new group, and return the address to MAP for distribution.

Priority: This field is used to specify the priority of this group (relative to all other groups). The permissible range is 0 through 15, with 0 being the highest priority.

TX/RX Flags: These flags allow MAP to tell the multicast protocol which of three modes it requires: transmit-only; receive-only; or transmit/receive.

A.7.2.2 MT-Data

The *MT-Data* primitives are used to transfer MAPPDUs to all nodes in the group via multicast.

These primitives work the same way for both reliable and unreliable protocols. When MAP wants to send a MAPPDU via multicast , it will invoke *MT-Data request*. This will eventually cause a *MT-Data indication* at all remote nodes that need to receive the data. The order of the requests may not match the order of the indications. For unreliable multicast, there may not be a one-to-one correspondence between requests and indications (some requests may never result in an indication, while other requests may result in multiple indications).

Parameter definition

User Data: This is a variable length octet string. It is used to carry MAPPDUs.

A.7.2.3 MT-Leave

The *MT-Leave* primitives are used to leave a multicast group. *MT-Leave indication* is used to tell MAP that its membership in the group was prematurely terminated.

When MAP decides that it no longer needs to transmit *or* receive data on a given multicast group, it will invoke *MT-Leave request*. The multicast protocol should immediately stop forwarding data from the group to MAP. There is no confirm for this primitive.

MT-Leave indication only occurs if the multicast protocol needs to tell MAP that it is no longer joined (even though MAP did not request a leave). This indication is *not* caused by other nodes leaving a group.

Parameter definition

Reason: This contains the reason why the multicast group was left. This is one of: provider initiated; or unspecified failure.

A.7.3 Local information

For MAP to handle many transport protocols simultaneously, with no prior knowledge of their capabilities, there will have to be local configuration information. Each transport protocol is described to MAP in a generic fashion. This information is used by MAP during the transport protocol arbitration process, and includes the following parameters:

- *Transport Protocol ID*: This is used to uniquely identify the transport protocol during the protocol arbitration process.
- *Transport Protocol Type*: Each transport protocol must be identified as one of the following: reliable unicast; unreliable unicast; reliable multicast; or unreliable multicast.
- *Preference Weighting*: This relative weighting is used to determine which transport protocols are preferred over others.
- *Maximum Number of Connections*: This is only relevant for unicast protocols. This is the maximum number of connections that the unicast protocol recommends *per MAP connection*.
- *Maximum Payload Size*: This is the maximum payload size that the transport protocol can handle. Note that for reliable unicast, the transport protocol *must* be able to handle up to

16 384 bytes (with segmentation, if necessary), so this is just a recommended maximum size.

- *Configuration Data*: When MAP is ready to arbitrate protocols with a remote node, it gets a block of configuration data from each candidate protocol (which can be NULL). This information is passed to the remote node during protocol arbitration, where it is passed to the equivalent protocol on that node (if it exists). MAP on the called node then gets the configuration data to be returned from its local stack, in order to complete the protocol arbitration process. This configuration data should be separately retrieved for each MAP connection that is arbitrated.

  The information contained in the configuration data block is separately defined for each transport protocol. The format of these blocks can either be standardized (along with transport protocol profiles), or can be proprietary. An example of how it might be used would be to communicate a dynamic IP port number back to the caller.

A.8 Protocol specification

This subclause contains a step-by-step description of how MAP works, which includes several PDU timing diagrams. Figure A.8-1 illustrates the domain hierarchy that was used to generate these diagrams.

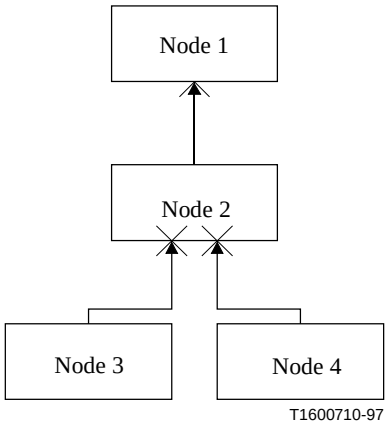

Figure A.8-1/T.125 – Sample domain referenced by subsequent examples

A.8.1 Establishing the initial reliable unicast connection

During processing of an MCS Connect Request, the first thing that must be done is the creation of a reliable unicast connection. The connection is made using the protocol specified by MCS, or the default protocol for the network (if no protocol is specified). This connection is used to transmit all control PDUs (both MCS and MAP). The first MAPPDU sent across the new connection by the calling node is **MAPConnectRequest**. This MAPPDU contains the following fields:

- A protocol version number identifying which version of MAP the calling node wishes to use.
- A MAP Service Access Point (MAPSAP) identifying who on the called node should accept the call (this is specified by MCS as part of the network address).
- A Domain Reference ID that can bind this connection to a particular domain (for the initial connection this field is set to zero).

- Low and high priority values that indicate the range of priorities to be sent on this connection (for the initial connection these fields will both be set to zero).

If the called node wishes to accept the call, it will transmit a **MAPConnectConfirm** back to the calling node. This MAPPDU contains the following fields:

- A protocol version number identifying which version of MAP *will* be used (this must be less than or equal to the value specified by the caller).
- The MAPSAP on the called node that actually accepted the incoming connection.

If the called node does not wish to accept the incoming call, it will transmit a **MAPDisconnectRequest** back to the calling node. This MAPPDU contains the reason for the rejection. It also contains a field indicating that no confirm is required, so the connection is dropped immediately.

Figure A.8-2 contains a timing diagram that shows a successful call set-up.

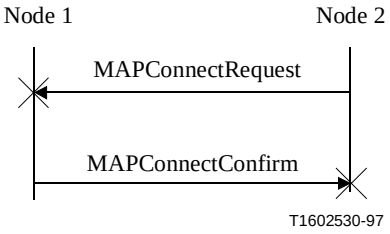

Figure A.8-2/T.125 – Timing of the connect MAPPDUs

A.8.2 Arbitrating transport protocols

Once the initial reliable unicast connection is established, MCS will proceed with its connection set-up. When MCS has completed its set-up, it will pass the following information to MAP: what domain the connection is bound to; what protocol version MCS will be using on the connection; and how many priorities and reliability levels will be used in the domain (this is provided to MAP when the domain parameters are first locked). Once MAP has this information, it can proceed with the process of transport protocol arbitration.

The calling node will issue a **MAPArbitrateProtocolsRequest** that contains a list of the transport protocols that it is willing to use (both unicast and multicast). As a minimum, this list will include the reliable unicast protocol used to create the existing connection. Support for other transport protocols is optional. After comparing this list to its own capabilities, the called node will respond with a **MAPArbitrateProtocolsConfirm** that indicates which protocols are to be used.

If no unreliable protocols are selected, then unreliable data is sent using the reliable protocols (for unicast, unreliable data is sent through the same connections as reliable data). If MCS has indicated that there will be no unreliable data in the domain, then it is unnecessary to include unreliable protocols in either of the arbitration MAPPDUs. If no multicast protocols are selected, then all data is sent via unicast (between these two nodes).

The confirm will always include exactly one reliable unicast protocol. It will optionally include one unreliable unicast protocol. For multicast, it will include all protocols that both nodes have agreed can be used (there can be more than one for each reliability level).

For the reliable unicast protocol, the nodes also arbitrate the number of transport connections that will make up this one MAP connection. If there is more than one transport connection, then unicast data is split between them according to priority. It is perfectly okay for different MAP connections to

be made up of differing numbers of transport connections, even in the same multicast island. The process of creating additional transport connections is detailed in the next subclauses.

The arbitration MAPPDUs also contain a Domain Reference ID. This value is later used by the sender to indicate which domain a MAPPDU is associated with, in the case that the domain cannot be implicitly inferred. Specifically, this value is used when creating additional reliable unicast connections, and when transmitting unreliable data.

Any node can force a re-arbitration of transport protocols with a neighboring node simply by issuing a **MAPArbitrateProtocolsRequest**. To ensure that no connections are lost, however, it is required that the previously selected reliable unicast protocol be available in the protocol menu. If the re-arbitration results in a change to the unicast protocols, it is necessary to make the transition immediately (this process is detailed in a later subclause). Changes to the set of multicast protocols have no direct effect on existing multicast groups. These changes are reflected only in subsequent group allocations (this process is detailed in the subclause on allocating and distributing multicast groups, see A.8.3).

It is possible for adjacent nodes to encounter an "arbitration collision". This occurs when both nodes issue a **MAPArbitrateProtocolsRequest** at the same time. This situation is detectable at both nodes because they will both receive an arbitration request when they were expecting a confirm. When this occurs, the request from the downward node is ignored (and the downward node stops waiting for a confirm). The request from the upward node is processed normally.

Figure A.8-3 illustrates the arbitration process when Node 2 calls Node 1.

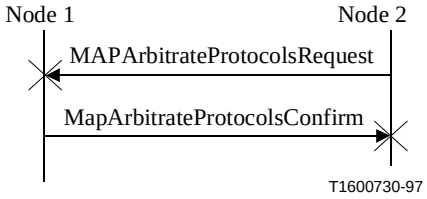

Figure A.8-3/T.125 – Timing of the arbitration MAPPDUs

A.8.2.1 Creating additional reliable unicast connections

As mentioned above, nodes also arbitrate the number of reliable unicast connections that will make up a single MAP connection. If they decide to use more than one, then it is necessary to create the additional connections. This subclause uses the term *connection* to refer to a reliable unicast connection.

This process is initiated by the calling node, who calls upon the selected reliable unicast protocol to create the appropriate number of additional connections. As each connection becomes ready for use, the calling node will issue a **MAPConnectRequest** over it. This MAPPDU will contain the following fields:

- The protocol version number is the same one already selected for the initial connection.
- The MAPSAP is the same one used to create the initial connection.
- The Domain Reference ID is the one that was sent in the **MAPArbitrateProtocolsRequest** on the initial connection.
- The low and high priority values will indicate the range of priorities that will be sent over this connection (the required priorities, as specified by MCS, are typically spread out evenly over the number of connections).

The called node should accept each of the additional connections by issuing a **MAPConnectConfirm** over each one. This MAPPDU will contain an exact copy of the first two fields of the request above.

It is a protocol error to refuse one of the additional connections. If one of the two nodes cannot create the agreed upon number of connections, then *all* connections corresponding to this MAP connection must be shut down, severing the lower node from the domain.

It is important that all required priorities be mapped to a connection when this process completes. As each additional connection is brought up, it is mapped to a range of priorities. All required priorities that are not explicitly mapped to an additional connection are handled by the initial connection. These leftover priorities *must* be contiguous with priority zero (which is already mapped to the initial connection).

A.8.2.2 Changing reliable unicast connections

After a protocol re-arbitration has occurred, it may be necessary for two nodes to dynamically switch to a different set of reliable unicast connections. This will occur if any of the following conditions exist: the nodes have chosen to switch to a different reliable unicast protocol; the nodes have chosen to use a different number of transport connections; the nodes have elected to change the maximum payload size of the reliable unicast connections (this cannot be done while multicast groups are still active); or the nodes have exchanged new configuration data for the reliable unicast protocol. Note that a dynamic transition cannot occur until the set-up of the previous set of reliable unicast connections is complete.

To dynamically transition to a different set of connections, it is necessary to perform the following steps:

1) The initiator of the arbitration must create the agreed upon number of new connections to the remote node using the selected protocol. This node will now be referred to as the "calling node".

2) The calling node will issue a **MAPConnectRequest** on each new connection as it becomes ready. This MAPPDU will contain the following fields:
   - The protocol version number is the same one already selected for the previous connections.
   - The MAPSAP is the same one used to create the previous connections.
   - The Domain Reference ID is the one that was sent in the **MAPArbitrateProtocolsRequest** on the previous initial connection.
   - The low and high priority values will indicate the range of priorities that will be sent over this connection (the required priorities, as specified by MCS, are typically spread out evenly over the number of connections).

3) The called node will respond by sending a **MAPConnectConfirm** on each new connection when the request is received. The protocol version number and the MAPSAP is set the same as in the request.

4) As each confirm is received, the calling node will keep track of which new connections are ready for use. By comparing the priority range of the previous connections with the priority range of the confirmed new connections, the calling node can determine when a previous connection can be shut down (because all of its priorities are covered by new connections). When this occurs, the calling node will issue a **MAPDisconnectRequest** to the previous connection. At this point, the calling node will begin sending all reliable unicast traffic in the priority range of the previous connection on the appropriate new connection(s).

5) Upon receiving a disconnect request, the called node will issue a **MAPDisconnectConfirm** on the previous connection. At this point, the called node will begin sending all reliable unicast traffic in the priority range of the previous connection on the appropriate new connection(s). The called node must also keep track of which previous connections have been disconnected. By comparing the priority range of the new connections with the priority range of the previous connections, the called node can determine when it is safe to begin processing data received from a new connection (this occurs when all previous connections that carried data in the priority range of the new connection have been disconnected).

6) When the calling node receives a disconnect confirm, it can be assumed that all pending data on the previous connection has been flushed, and that connection can be shut down. By comparing the priority range of the new connections with the priority range of the previous connections, the calling node can determine when it is safe to begin processing data received from a new connection (this occurs when all previous connections that carried data in the priority range of the new connection have been disconnected).

A.8.3 Allocating and distributing multicast groups

Once a set of transport protocols have been arbitrated, MAP can begin allocating and distributing multicast group addresses. In order to minimize the number of active multicast groups, MAP should not assign a multicast group address to a metachannel until data traffic is detected that could use the group. No data will be lost, since all data is initially transmitted via unicast.

When MAP detects data flow for which there is no multicast group, a multicast group address may be assigned. This process begins at each node in the domain that either has no upward MAP connection, or has an upward MAP connection but no common multicast protocols with the node above. These nodes become Multicast Group Providers. They will examine the lists of multicast protocols supported by the nodes below them, and attempt to choose the protocol(s) that will result in the fewest number of multicast islands. Ideally, it will be possible to select a single protocol that is supported by *all* downward nodes (allowing everyone below to be in the same multicast island).

Once a multicast protocol is selected for each downward node, MAP will obtain a multicast group address for each different protocol to be used. Depending on a local set of policies, the group address to be assigned may already be in use for another metachannel, or it may be newly allocated. How this allocation is performed is outside the scope of this protocol. Once a multicast group address is decided upon, it is propagated downward using **MAPAddGroupRequest**.

MAPAddGroupRequest tells the recipient what channel, reliability level, and priority range the assignment corresponds to. It also tells the recipient which multicast protocol and group address to use. As described in a previous subclause, the act of selecting multicast protocols for the downward nodes results in the definition of multicast island boundaries. All nodes that are told to use the same multicast protocol and group address are, by definition, in the same multicast island.

When a node receives a **MAPAddGroupRequest**, it will add the group to its list of active multicast groups. If it has any downward MAP connections, then it must select a multicast protocol for each downward node. If possible, it should use the same protocol that was selected for it by the node above. If this is not possible for some downward nodes, then it should try to select protocols that will result in the fewest, largest multicast islands. Once the selections are made, the node will obtain a multicast group address for each different protocol to be used, and propagate the information downward using **MAPAddGroupRequest**.

The process of propagating multicast group information downward is depicted in Figure A.8-4.

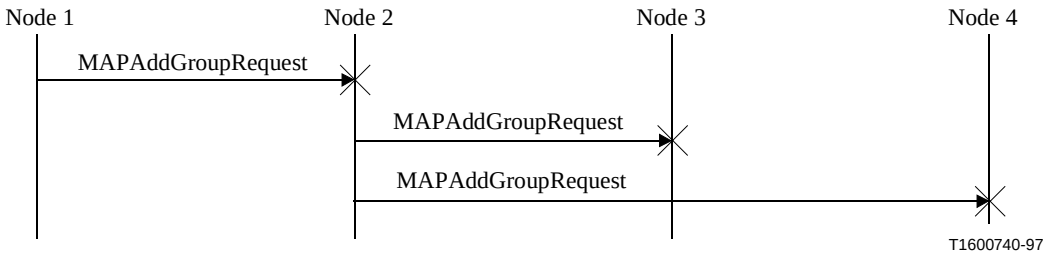

Figure A.8-4/T.125 – Timing of the add group procedure

A.8.4 Managing multicast data traffic (self-tuning)

Once a multicast group address has been distributed, the nodes in the multicast island can begin to use it. It is not acceptable, however, for the nodes to assume that multicast is working immediately. There are many things that can cause multicast to fail. First of all, not all networks support multicast. This can result in multicast data not reaching one or more nodes in the multicast island. Even if the network does support multicast, other factors, such as Time-To-Live (TTL), can cause multicast data to fall short of its destination.

For these reasons, among others, MAP employs a self-tuning strategy that allows it to depend on multicast only when multicast is working. This strategy works by sending all data via both unicast AND multicast, and then dynamically turning off unicast on those connections that lead to nodes who are successfully receiving the data via multicast.

Unicast data within a multicast island is controlled independently for each "data flow". A data flow is uniquely identified by the following five fields: sender ID; channel ID; reliability level; priority; and data type. By default, all data are sent via unicast to all nodes in the multicast island that need to receive it. The data are also sent to the multicast group by a single node in the multicast island (the multicast sender). When a node detects that it is receiving a given data flow via multicast, it will send a **MAPDisableUnicastRequest** to its neighboring node in the direction of the multicast sender. This will cause the neighboring node to stop forwarding that data flow via unicast.

The identity of the multicast sender for a given data flow is determined by the data type, as follows:

- For non-uniform data, the multicast sender is simply the data flow's original point-of-entry node.
- For uniform data, all data are unicast upward to the Top Provider before being eligible for multicast. The data are sent to the multicast group by the Multicast Group Provider on its way back down (the multicast sender is *always* the Multicast Group Provider). When sending a **MAPDisableUnicastRequest** for uniform data, it is always sent upward. Furthermore, sender ID shall be ignored when uniquely identifying uniform data flows.
- For proxy data, all data are unicast upward to the Multicast Group Provider. The Multicast Group Provider will forward the data upward via unicast (if necessary), while simultaneously sending the data back downward (via both unicast and multicast). As with uniform data, the multicast sender is *always* the Multicast Group Provider. When sending a **MAPDisableUnicastRequest** for proxy data, it is always sent upward. Furthermore, sender ID shall be ignored when uniquely identifying proxy data flows.

The proxy data type provides an alternative way of sending non-uniform data. Proxy data **cannot** be used to transmit uniform data. Using proxy data reduces the amount of data flow state information, since all proxy data senders for a given channel ID/reliability level/priority are merged into one data flow. Proxy data is also useful on networks that do not allow nodes to originate multicast information. Knowing when to use proxy data is a local implementation issue. However, once a node

begins sending proxy data for a given sender ID/channel ID/reliability level/priority, it cannot later decide to use the non-uniform data type (the reverse is also true). The decision to use proxy data is strictly local, and different nodes may handle local senders differently.

There is one other important consideration when using proxy data. Each node on the path from the originator to the Multicast Group Provider will see the data twice (once on its way up, and again on its way back down). These nodes should only send the data to MCS once. It does not matter whether this is done as the data flows up, or as it comes back down (but it should be done consistently). Furthermore, the originating node should *never* forward the data back to the user attachment that sent it.

Before a node tells its neighbor to disable unicast for a data flow, it must be certain that it can transition to multicast with no data loss. To ensure this, it must receive at least one data packet via unicast, so that it knows what the next sequence number should be (note that sequence numbers are unique to each data flow, and are assigned by the originator of the data flow). Either before or after receiving this unicast packet, the node must receive a data packet via multicast with a sequence number that is less than or equal to the next expected sequence number. This indicates an overlap in the two data streams, allowing unicast to be disabled.

When data is first transmitted on a new data flow, it is sent via unicast AND multicast throughout the island. Overtime unicast is disabled wherever possible, resulting in optimal network traffic.

Figure A.8-5 shows an example of this self-tuning strategy (refer back to Figure A.8-1 for domain hierarchy). This example tracks the transmission of three data packets from a previously unknown data flow, as follows:

1) A user application attached to Node 1 transmits a data packet. Node 1 transmits the data to the multicast group. It also transmits the data to Node 2 via unicast, since Node 2 has not yet disabled unicast for this data flow. Node 4 has not yet finished joining the group, so the multicast data packet only arrives at Nodes 2 and 3. Nodes 2 and 3 will hold the data since they do not yet know what sequence number is valid. When Node 2 receives the data via unicast, it redirects the data to Nodes 3 and 4 (via unicast). Nodes 2 and 3 now recognize that the multicast packet is the same as the unicast packet, so they each issue a **MAPDisableUnicastRequest** in the direction of the sender.

2) The user application at Node 1 transmits another data packet on the same data flow. As before, Node 1 transmits the data to the multicast group. This time Node 1 does *not* transmit the packet to Node 2 via unicast, since Node 2 has already disabled unicast for this data flow. This time the multicast transmission arrives at all three downstream nodes (including Node 4). Node 2 transmits the data via unicast to Node 4, but not to Node 3 (since Node 3 has disabled unicast for this data flow). Node 4 now knows that it is synchronized to the multicast data stream, so it issues a **MAPDisableUnicastRequest** upward.

3) The user application at Node 1 transmits a third data packet on the same data flow. As before, Node 1 transmits the data to the multicast group. This time, however, no one will unicast the data, since all nodes in the island have disabled unicast for this data flow.

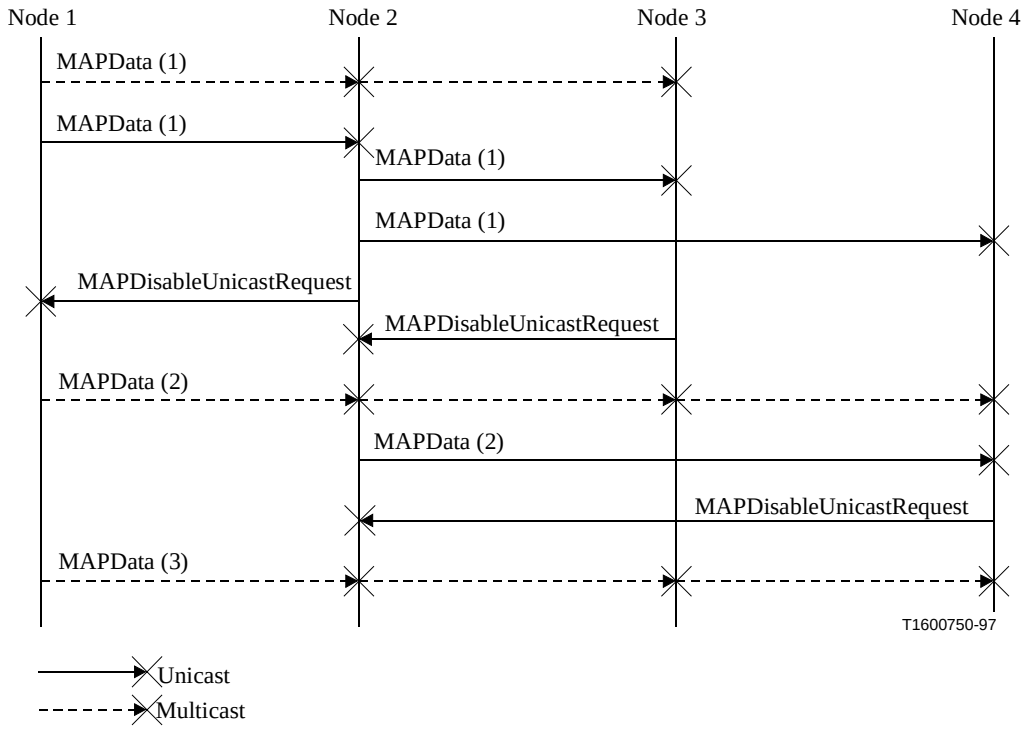

Figure A.8-5/T.125 – Example timing of the disable unicast procedure for a given data flow

A.8.5 Giving up on multicast

The previous subclause focused on what happens when multicast is working. This subclause discusses what a node can do to optimize its transmissions in the case that multicast does not seem to be working.

When a node first begins transmitting data for a non-uniform data flow, it will do so via both unicast and multicast. If the adjacent nodes receive the multicast data, they will disable the unicast transmissions as described in the previous subclause. If the adjacent nodes do not receive the data via multicast, this disabling of unicast will never occur. This could leave the sending node in the position of transmitting every data PDU more times than necessary (once for each adjacent node, plus once for multicast). This problem is most pronounced on a leaf node, who could end up using twice the required bandwidth. This may be unacceptable (especially if the node is attached using a low-speed connection). Unfortunately, the node cannot simply stop transmitting data via multicast. It is possible that a node somewhere in the island (not adjacent) is receiving the multicast data, and may have already disabled unicast.

Each **MAPData** PDU contains a flag that can be used by a node to resolve this problem. This flag is called the Unicast Forward Flag. This flag forces all intermediate nodes in the multicast island to once again forward the data flow via unicast to all receivers (even if they have previously disabled unicast). When a node decides that it wants to stop sending a given data flow via multicast, it simply sends all future **MAPData** PDUs (for that data flow) via unicast only, with the Unicast Forward Flag set. This makes it safe for the node to stop transmitting via multicast.

A.8.6 Removing multicast groups

When a Multicast Group Provider senses that a multicast group is no longer needed, that group can be removed from the island. For reliable data, it is very important that the group be removed in a manner that guarantees no loss of data.

To ensure this, the Multicast Group Provider will first disable multicast reception for the group. This is done by sending a **MAPDisableMulticastRequest** downward to all nodes in the island (this request is automatically forwarded by each receiving node to those nodes that lie below them). This MAPPDU causes each node to begin the process of re-enabling unicast for all data flows on the group for which unicast was previously disabled. This is done by issuing a **MAPEnableUnicastRequest** in the direction of the sender for each affected data flow. This MAPPDU also causes the group to be marked as disabled, which prohibits any further disabling of unicast for the group. Note that this does NOT cause the nodes to stop transmitting data to the multicast group. Furthermore, the nodes must stay joined as receivers to the multicast group at least until unicast has been successfully enabled for all affected data flows.

When the **MAPDisableMulticastRequest** reaches the bottom nodes in the multicast island, they can respond with a **MAPDisableMulticastConfirm**. Before doing this, they must ensure that all data flows for which unicast was previously disabled on the group are completely re-enabled. To ensure this, each node must have received a **MAPEnableUnicastConfirm** for each outstanding request. Furthermore, each node must examine the sequence number in the confirm to ensure that all in-flight multicast data has been received (if not, it must wait for it to arrive). Once this is done, the **MAPDisableMulticastConfirm** is sent to the node above.

Intermediate nodes in the island must wait until they have received **MAPDisableMulticastConfirm** PDUs from all nodes below them in the island. They must also wait until all previously disabled unicast data flows are re-enabled, as described above. Then they can issue a **MAPDisableMulticastConfirm** to the node above.

Once the Multicast Group Provider has received **MAPDisableMulticastConfirm** PDUs from all downward nodes, and ensured that its unicast data flows are re-enabled, it is known that the group is no longer in use. At this point, the Multicast Group Provider can issue a **MAPRemoveGroupRequest** downward. This MAPPDU will propagate to all nodes in the island, causing each node to leave the group (if it is joined), and remove it from their list of active groups.

Figure A.8-6 illustrates the process of group removal for the sample domain shown in Figure A.8-1. This is a continuation of the example in Figure A.8-5, in which unicast has been disabled for a single sender attached to Node 1.

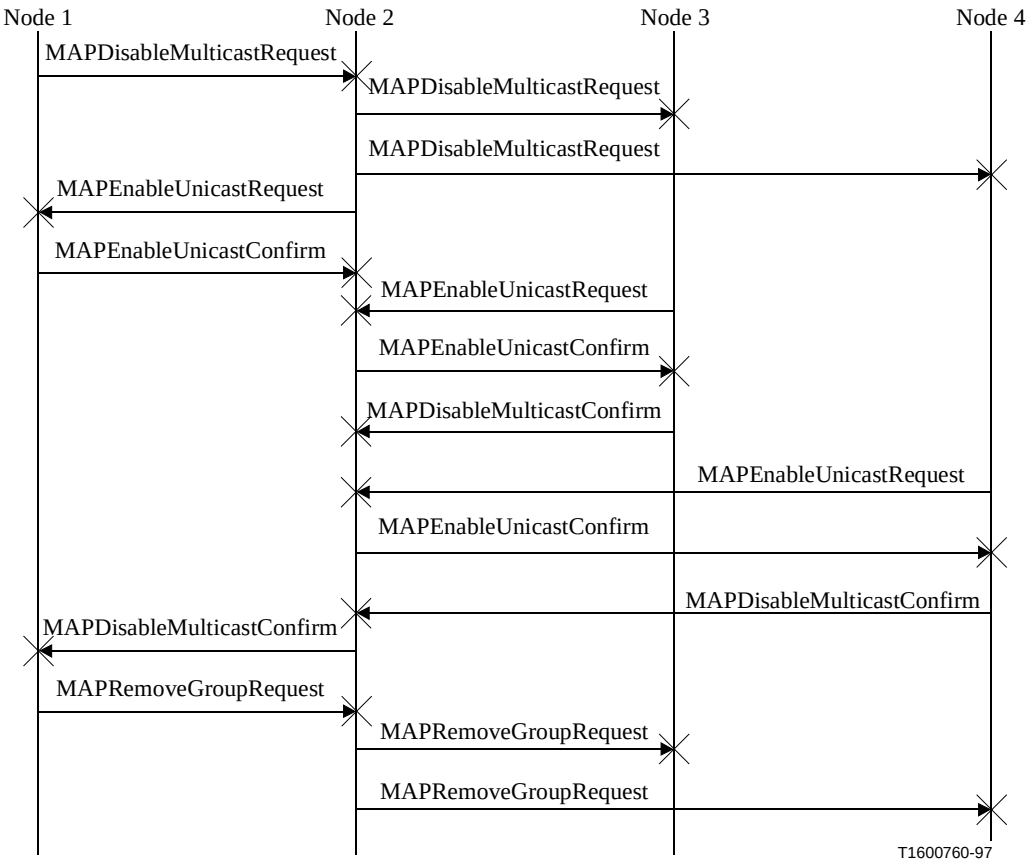

Figure A.8-6/T.125 – Timing of the group removal procedure

A.9 MAPPDU descriptions

This subclause describes the PDUs that are used by MAP. Each MAPPDU begins with a brief description of what it is used for. This is followed by a table showing the fields that are encoded into the MAPPDU. Finally, there is a description of what each field in the MAPPDU is.

Note that the next subclause contains the ASN.1 definition of each MAPPDU.

A.9.1 MAPConnectRequest

When creating a new connection between two nodes for a given domain, the first step is to open a reliable unicast connection. **MAPConnectRequest** is the first MAPPDU sent across that connection. The calling node sends this MAPPDU to tell the called node what protocol version it wishes to use. This is just a suggested value. It is up to the called node to make the final decision and communicate it back to the calling node by sending a **MAPConnectConfirm**.

This MAPPDU includes a MAP Service Access Point (MAPSAP) field which allows the calling node to specify who on the called node should receive the incoming call.

This MAPPDU also contains a Domain Reference ID, which allows the calling node to associate this connection with a particular domain. This is used when transport protocol arbitration results in the creation of additional connections, or the transition to a different set of connections.

The following table summarizes the **MAPConnectRequest** contents.

Version Number
Connection MAPSAP
Domain Reference ID
Priority Range

Version Number: This field indicates the version of this protocol that the calling node wishes to use. This consists of both a Major and Minor version number. The value of this field for the protocol defined in this Annex is 1.0 (Major version 1, Minor version 0). Note that this is the suggested value. The called node will determine the actual protocol version to be used.

Connection MAPSAP: This field tells the receiver which MAP Service Access Point is to handle the incoming connection. The value zero (0) indicates that the caller wishes to access the default MAPSAP. All other values are assigned and obtained out-of-band to this protocol.

Domain Reference ID: This field allows the calling node to associate this connection with a particular domain on the called node. When creating the initial connection to a remote node, this field should be set to zero (0). When creating additional connections, or transitioning to a new set of connections, this field should be set to the value that was sent in the **MAPArbitrateProtocolsRequest** on the initial connection. This value, in conjunction with the calling address, allows the called node to know which domain the in-bound connection is to be bound to.

Priority Range: This is the priority range of the data that will be carried on this connection. The initial connection to a remote node will always carry *only* priority zero (zero is the highest priority).

A.9.2 MAPConnectConfirm

This MAPPDU is used to accept a new connection from another node. It informs the calling node of what protocol version will be used on the connection. It also informs the calling node of which remote MAPSAP accepted the call.

The following table summarizes the **MAPConnectConfirm** contents.

Version Number
Connection MAPSAP

Version Number: This field indicates the version of this protocol that will be used for this connection. This consists of both a Major and Minor version number. The value of this field for the protocol defined in this Annex is 1.0 (Major version 1, Minor version 0). If the calling node cannot handle (or does not want to handle) the version specified by the called node in this MAPPDU, it must disconnect.

Connection MAPSAP: This is the MAPSAP on the called node that actually accepted the call. If the MAPSAP specified in the **MAPConnectRequest** was non-zero, then this field should be set the same. If the MAPSAP in the **MAPConnectRequest** was zero, then this field is set to the valid value of the MAPSAP that accepted the call (which may also be zero).

A.9.3 MAPDisconnectRequest

This MAPPDU is used to break an existing transport connection between two nodes, or to reject an incoming connection if the called node does not wish to participate in the conference. Once this MAPPDU is sent, the connection cannot be used for the transmission of any other data.

If this MAPPDU is being sent as part of a transition to a new set of reliable unicast connections, the sender should set the Confirm Required flag. This informs the receiver that it should respond by issuing a **MAPDisconnectConfirm** on the same connection.

If the Confirm Required flag is set, then the sending node must wait for the **MAPDisconnectConfirm**, before dropping the connection. This is done to ensure that all data is flushed from the connection before shutting it down.

The following table summarizes the **MAPDisconnectRequest** contents.

Reason
Confirm Required

Reason: This field indicates to the recipient why the connection is being broken or rejected.

Confirm Required: This is a Boolean flag that indicates whether or not the receiver should respond with a **MAPDisconnectConfirm**: This flag is set if the disconnect is occurring as part of a transition to a different set of reliable unicast connections. Waiting for the confirm allows both sides to know that all MAPPDUs transmitted on this connection have been processed.

A.9.4 MAPDisconnectConfirm

This MAPPDU is sent in response to a **MAPDisconnectRequest** that has the Confirm Required flag set. By sending this confirm, both sides can be assured that all data has been flushed from the connection before they shut it down and begin using a new one.

Upon receiving this MAPPDU, the node can safely shut down the connection upon which it arrived. Depending on the state of the replacement connections, it may be possible for the node to begin processing data on one or more of those connections.

This MAPPDU has no fields associated with it.

A.9.5 MAPArbitrateProtocolsRequest

Once a connection has been established for a given domain between two nodes, this MAPPDU can be used to arbitrate which transport protocols are to be used for data delivery. This MAPPDU *must* be sent by the calling node when the initial connection is made. It may also be sent by any node if it becomes desirable to re-arbitrate the transport protocols at any time.

This MAPPDU tells the receiver which transport protocols are supported by the sender. This includes both unicast and multicast protocols, for both reliability levels. Note that it is *mandatory* that the reliable unicast protocol used for the existing connection be represented in this MAPPDU (this allows the receiver to reject a reliable unicast transition simply by selecting the same protocol already in use).

Each transport protocol listed in this MAPPDU can be assigned a preference weighting by the sender. For the multicast protocols, there is also a field to be used by a downward node to indicate how many nodes in its subtree support the protocol. The upward node can use this information to more wisely select multicast protocols for a given metachannel.

This MAPPDU is also used to communicate a domain reference ID to the receiver. This value, in conjunction with the sender's network address, can subsequently be used to uniquely identify a particular domain to the receiver. This is necessary for those cases in which the domain cannot be implied. Specifically, this value is used when creating additional reliable unicast connections, or when sending unreliable data PDUs.

If adjacent nodes both try to originate a re-arbitration at the same time, the request from the downward node is ignored. This situation is detectable at both nodes because they will both receive an arbitration request when they were expecting a confirm. The upward node responds to this by throwing away the request (and continuing to wait for the confirm). The downward node responds by processing the request (and ceasing to wait for the confirm). If the downward node still wishes to perform re-arbitration, it can issue another request (after sending the confirm).

The following table summarizes the **MAPArbitrateProtocolsRequest** contents.

Domain Reference ID
More to Come Flag
Transport Protocol Menu

Domain Reference ID: This is a locally-assigned value used by the sender to reference a particular domain. After the arbitration process is complete, the sender can use this value to indicate which domain a **MAPConnectRequest** PDU is associated with. It is also used to indicate which domain an unreliable **MAPData** PDU is associated with.

More to Come Flag: When set, this flag indicates that there are too many transport protocols to fit in one MAPPDU, so the receiver should wait for subsequent **MAPArbitrateProtocolsRequest** PDUs before processing them. The last MAPPDU in the sequence should have this flag reset.

Transport Protocol Menu: This field contains a set of Transport Protocol Request Entries. This is a list of the transport protocols that the sender can use in this conference. This should be a complete list of the protocols supported by the sender (unreliable protocols are optional). It is mandatory that the reliable unicast protocol used to transport this MAPPDU be included in the list. There is a field in each entry allowing the caller to specify which protocols it prefers to use. For a given protocol type, the sender should also list the entries in the order of preference. It is the job of the receiver to select which protocols can actually be used for the connection. For the unicast protocols, the receiver should try to select the preferred protocols if possible. For the multicast protocols, the receiver should select all protocols that are common to both sender and receiver. It is okay if there are no common multicast protocols at a given reliability level (all data will be unicast at that reliability level). It is also okay if there is no common unreliable unicast protocol (unreliable data will be sent through the reliable connections).

A.9.5.1 Transport Protocol Request Entry

The menu is just a set of Transport Protocol Request Entries, the format of which is defined in the following table.

Transport Protocol ID
Transport Protocol Type
Network Address
Maximum Payload Fixed Flag
Maximum Payload Size
Preference Weighting (optional)
Node Count (optional)
Number of Connections (optional)
Configuration Data (optional)

Transport Protocol ID: This field uniquely identifies the transport protocol. Values for this field will have to be standardized, although it will also be possible to specify proprietary values.

Transport Protocol Type: This field is set to one of four values: reliable unicast; unreliable unicast; reliable multicast; or unreliable multicast.

Network Address: This is the address at which the originator can be reached using this protocol.

Maximum Payload Fixed Flag: This is a Boolean flag that indicates whether or not the Maximum Payload Size is negotiable for this Transport Protocol Entry. See the description of Maximum Payload Size for more detail.

Maximum Payload Size: This field specifies the maximum payload size supported by this transport protocol. If the Maximum Payload Fixed Flag is reset, then this is the maximum value supported by the local implementation (which can be negotiated down). If the Maximum Payload Fixed Flag is set, then this is the exact value that must be used in order to accept this Transport Protocol Entry. MAP will use the negotiated values to determine the maximum size of a data frame (data frames are discussed in more detail in the section on **MAPData**). The smallest Maximum Payload Size supported by MAP is 128.

Preference Weighting: This optional field is used to indicate to the receiver which protocols are preferred. A higher value indicates a protocol that is preferred over one with a lower value. If the field is not included, then a preference weighting of zero is assumed. This field is not used for multicast protocols when the MAPPDU is being sent downward.

Node Count: This optional field is only present for the multicast protocols (reliable and unreliable), and only when the MAPPDU is being sent upward. It indicates how many nodes at or below the sender support this multicast protocol entry. This number can be used by the upward node to make better decisions about which multicast protocol to use when mapping a new metachannel to multicast. For this number to remain accurate, a re-arbitration will have to occur whenever a new MAP capable node joins or leaves the conference. This re-arbitration will only occur on the branch of the domain tree that leads to where the node joined or left. In order to reduce network traffic, re-arbitration should not occur *every* time a node joins or leaves once the conference exceeds a certain size. This field can also be used to prevent the use of multicast when there are fewer than *N* nodes that will use it (*N* = 3?).

Number of Connections: This optional field is only present for the reliable unicast protocol. It indicates the number of transport connections that the sender wishes to create for this one MAP connection (if this protocol is selected). If the receiver does select this protocol, it should choose a number that is less than or equal to the number specified by the sender.

Configuration Data: This optional field is a variable length octet string that is used by the transport protocol stacks to pass configuration information. This field will not be present if the transport stacks do not wish to exchange any configuration data, or if this is a re-arbitration and the transport stacks wish to keep the previously agreed upon value. Note that if this field *is* present during a re-arbitration, and this protocol is selected as a unicast protocol, then it is necessary to transition to a new set of unicast connections (since the configuration data may have changed). The format of this field is specific to each transport protocol. For standardized transport protocols, this field will have to be part of a standard. For proprietary transport protocols, it can be whatever the manufacturer thinks is appropriate. This field can be used to communicate capabilities, and can be modified in the **MAPArbitrateProtocolsConfirm** to arbitrate those capabilities. This field must not be so large that the MAPPDU exceeds 16 384 bytes in size.

A.9.6 MAPArbitrateProtocolsConfirm

This MAPPDU is used to finalize arbitration of transport protocols for a given connection. After receiving a **MAPArbitrateProtocolsRequest**, the receiver should respond by issuing this MAPPDU.

The request MAPPDU contained a list of all transport protocols supported by the sender (including preferences). The receiver should select from this list the protocols that can be used on this connection. This includes *exactly* one reliable unicast protocol (and, optionally, one unreliable unicast protocol). If the sender specified preferences for the unicast protocols, these should be taken into consideration. The receiver should also determine the complete set of multicast protocols that are common to the two nodes. Returning this common set in the confirm helps to ensure that the best multicast protocols are selected during group allocation.

This MAPPDU is also used to communicate a domain reference ID to the receiver. This value, in conjunction with the sender's network address, can subsequently be used to uniquely identify a particular domain to the receiver. This is necessary for those cases in which the domain cannot be implied. Specifically, this value is used when sending unreliable data PDUs.

The following table summarizes the **MAPArbitrateProtocolsConfirm** contents.

Domain Reference ID
More to Come Flag
Transport Protocol Menu

Domain Reference ID: This is a locally-assigned value used by the sender to reference a particular domain. After the arbitration process is complete, the sender can use this value to indicate which domain an unreliable **MAPData** PDU is associated with.

More to Come Flag: When set, this flag indicates that there are too many transport protocols to fit in one MAPPDU, so the receiver should wait for subsequent **MAPArbitrateProtocolsConfirm** PDUs before processing them. The last MAPPDU in the sequence should have this flag reset.

Transport Protocol Menu: This field contains a set of Transport Protocol Confirm Entries. This is a list of the transport protocols that the called node has decided to use on this connection. As a minimum, the called node must indicate a reliable unicast protocol. Optionally, it can indicate an unreliable unicast protocol and protocols for the multicast reliability levels. If an unreliable unicast protocol is not included, then unreliable data will be unicast over the reliable connections. If a given multicast reliability level is not included, then all data at that reliability level will be unicast between these two nodes (multicast islands will be split for that reliability level). If the arbitrated unicast protocols are different than the ones already in use, then a unicast protocol transition must take place.

A.9.6.1 Transport Protocol Confirm Entry

The menu is just a set of Transport Protocol Confirm Entries, the format of which is defined in the following table.

Transport Protocol ID
Transport Protocol Type
Network Address
Maximum Payload Size
Preference Weighting (optional)
Node Count (optional)
Number of Connections (optional)
Configuration Data (optional)

Transport Protocol ID: This field uniquely identifies the transport protocol. Values for this field will have to be standardized, although it will also be possible to specify proprietary values.

Transport Protocol Type: This field is set to one of four values: reliable unicast; unreliable unicast; reliable multicast; or unreliable multicast.

Network Address: This is the address at which the originator can be reached using this protocol.

Maximum Payload Size: This field specifies the maximum payload size that will be used by this transport protocol. Note that the maximum payload size for a given multicast protocol *must* be the same as the maximum payload size for the unicast protocol at the same reliability level. This is due to the fact that data frames at that reliability level may have to be redirected via unicast at some points within the multicast island. The smallest Maximum Payload Size supported by MAP is 128.

Preference Weighting: This optional field is used to indicate to the receiver which protocols are preferred. A higher value indicates a protocol that is preferred over one with a lower value. If the field is not included, then a preference weighting of zero is assumed. This field is only used for multicast protocols when the MAPPDU is being sent upward.

Node Count: This optional field is only present for the multicast protocols (reliable and unreliable), and only when the MAPPDU is being sent upward. It indicates how many nodes at or below the sender support this multicast protocol entry. This number can be used by the upward node to make better decisions about which multicast protocol to use when mapping a new metachannel to multicast. For this number to remain accurate, a re-arbitration will have to occur whenever a new MAP capable node joins or leaves the conference. This re-arbitration will only occur on the branch of the domain tree that leads to where the node joined or left. In order to reduce network traffic, re-arbitration should not occur *every* time a node joins or leaves once the conference exceeds a certain size. This field can also be used to prevent the use of multicast when there are fewer than *N* nodes that will use it (*N* = 3?).

Number of Connections: This optional field is only present for the unicast protocols. It indicates the number of transport connections that will be created for this one MAP connection (at this reliability level). This number should be the lesser of the requested value, and the locally supported value.

Configuration Data: This optional field is a variable length octet string that is used by the transport protocol stacks to pass configuration information. This field will not be present if the transport stacks do not wish to exchange any configuration data, or if this is a re-arbitration and the transport stacks wish to keep the previously agreed upon value. Note that if this field *is* present during a

re-arbitration, and this protocol is selected as a unicast protocol, then it is necessary to transition to a new set of unicast connections (since the configuration data may have changed). The format of this field is specific to each transport protocol. For standardized transport protocols, this field will have to be part of the standard. For proprietary stacks, it can be whatever the manufacturer thinks is appropriate. This field can be used to communicate capabilities, and can be modified to arbitrate those capabilities. This field must not be so large that the MAPPDU exceeds 16 384 bytes in size.

A.9.7 MAPData

This MAPPDU is the workhorse of the protocol. It is used to carry all user data to its eventual destination. Unlike the TCP/IP stack defined in Recommendation T.123, MAP allows for the concatenation of multiple MCSPDUs into a single **MAPData** PDU (to promote efficient use of the network). MAP also allows for the segmentation of a single MCSPDU into multiple **MAPData** PDUs (when the MCSPDU exceeds the Maximum Payload Size).

Each individual data component of a **MAPData** PDU is referred to as a *data frame*. There can be one or many data frames in a single **MAPData** PDU. The maximum size of a data frame is determined during transport protocol arbitration. It is the largest size that will fit into the maximum payload for a given reliability level (after taking overhead into consideration). The maximum data frame size for one reliability level may be different than that for a different reliability level.

MCSPDUs are broken into data frames at the node acting as the original point-of-entry for the data flow. If necessary, they are re-assembled at all receiving nodes.

MAPData supports the notion of header compression. There are several fields in the **MAPData** header that are also included in the MCS data PDU header. If these fields can be extracted from the MCS header of the first data frame in the MAPPDU, then they may not be included in the **MAPData** header (they are optional fields). If the first data frame in the MAPPDU is a segmented MCS data PDU, the sender will decide whether or not enough of the MCS header is present to extract these fields. The receiver must be prepared to extract these fields if they are not explicitly included in the **MAPData** header.

The following table summarizes the **MAPData** contents.

Data Descriptor (optional)
Data Frame Array

Data Descriptor: This optional field is only present when sending MCS data PDUs. When present, the fields within this descriptor describe the data contained in this MAPPDU. If this field is not present, then this MAPPDU contains MCS control PDUs, which are to be forwarded directly to MCS at the receiving node.

Data Frame Array: This is a sequence of Data Frame Entries (as defined below).

A.9.7.1 Data Descriptor

A Data Descriptor is only present in a **MAPData** PDU if it is carrying MCS data PDUs. The following table summarizes the Data Descriptor contents.

Unicast Forward Flag
Starting Sequence Number
Data Flow Identifier (optional)
Domain Reference ID (optional)

Unicast Forward Flag: This flag is used to force the recipient to forward this MAPPDU via unicast. When this flag is set, the recipient must begin transmitting all data on this data flow via unicast (even to neighboring nodes that have previously disabled unicast). The mandatory unicast transmission begins with this MAPPDU. This flag should remain set when the PDU is forwarded. Once a sender chooses to set this flag for a particular data flow, the flag must be set for all subsequent **MAPData** PDUs on the same data flow (to avoid race conditions). If this flag is reset, then this MAPPDU should be treated normally.

Starting Sequence Number: This field contains the unique sequence number assigned to the first data frame in this MAPPDU. Subsequent data frames in this MAPPDU simply increment from this starting sequence number. Sequence numbers are independently assigned to each data frame by the originator of the data flow.

Data Flow Identifier: This optional field must be present if the information cannot be extracted from the first data frame. The fields making up the data flow identifier are those that are required to properly route the data, and to determine which unique data flow the data frames are associated with (which in turn identifies the sequence counter).

Domain Reference ID: This optional field is only present for unreliable data. This is the Domain Reference ID that was transmitted by the sender during protocol arbitration. This field may be present for multicast transmissions (for performance reasons), but should be ignored. This value, combined with the network address of the originator, is sufficient to uniquely determine which domain the data is associated with.

A.9.7.2 Data flow identifier

A data flow identifier is used to identify which data flow a set of data frames is associated with. This information must be explicitly included if it cannot be extracted from the first data frame. The following table summarizes the Data Flow Identifier contents.

Sender ID (optional)
Metachannel ID (optional)
Channel ID
Reliability Level
Priority
Data Type (optional)

Sender ID: This optional field is only present for non-uniform data, or for uniform or proxy data flowing upward via unicast. This field may also be excluded if the information can be extracted from the MCS header of the first data frame. This field indicates the MCS User ID of the user application that originated the data associated with this data flow.

Metachannel ID: This optional field may be excluded if the information can be extracted from the MCS header of the first data frame. This field identifies which metachannel is being referenced, and contains three sub-fields: channel ID; reliability level; and priority.

Data Type: This optional field may be excluded if the information can be extracted from the MCS header of the first data frame. This field indicates what type of data this is. This is one of: non-uniform; uniform; or proxy.

A.9.7.3 Data frame entry

MAPData PDUs contain an array of data frames. The following table summarizes the Data Frame Entry contents.

First Segment Flag
Last Segment Flag
User Data

First Segment Flag: This flag indicates whether or not this data frame is the first segment of a MCSPDU. If this flag is set, then this is the first data frame in a MCSPDU. If this flag is reset, then this is not the first data frame in a MCSPDU.

Last Segment Flag: This flag indicates whether or not this data frame is the last segment of a MCSPDU. If this flag is set, then this is the last data frame in a MCSPDU. If this flag is reset, then this is not the last data frame in a MCSPDU.

User Data: This is the user data from the MCSPDU.

A.9.8 MAPAddGroupRequest

This MAPPDU is used to propagate multicast group assignments downward within a domain. This is done whenever MAP detects a data flow for which there is no group assignment. This process is originated by each node in the domain that either has no upward MAP connection, or has an upward MAP connection but no common multicast protocols with the node above. This process is continued by each node that receives a **MAPAddGroupRequest** from the node above it.

Each node should examine the lists of multicast protocols that are supported by the nodes beneath it (these lists are constructed during transport protocol arbitration). Based on this information, each node must tell the nodes beneath it which multicast protocol and group address to use. It should select those protocol(s) that will result in the fewest, largest multicast islands.

When selecting a protocol for each downward node, there are several things that should be considered:

1) *What protocol versions are in use at the downward nodes?* During connection establishment, both MCS and MAP arbitrate protocol versions with each adjacent node. When selecting multicast protocols for the downward nodes, remember that nodes using different protocol versions *cannot* be in the same multicast island (even if they support the same multicast protocol). This is simply because PDU formats are different.

2) *What protocols are supported by the downward nodes?* Obviously, a node can only select from those protocols that the downward nodes have agreed to support. It should attempt to select the protocols that are most broadly supported.

3) *What maximum payload sizes were arbitrated with the downward nodes?* During protocol arbitration, each node should attempt to select the same maximum payload sizes that have already been arbitrated with other connections in the same domain. If different nodes insist on different maximum payload sizes, then they cannot be in the same multicast island (all nodes in an island must agree on the same maximum payload size, for both unicast and multicast).

4) *What protocol was assigned from above?* If a node is performing this selection in response to a **MAPAddGroupRequest** from above, then it should try to use the same protocol that was assigned to it (when possible). This is based on the fact that the node above has broader visibility, and can therefore make better decisions.

5) *How many nodes in each subtree support each protocol?* During protocol arbitration, downward nodes include a node count with each multicast protocol that they agree to use. This number tells the upward node how many nodes in each of its subtrees support that protocol. By summing these numbers for all downward connections, a node can choose the multicast protocol with the broadest support. This gives some visibility beyond one level of the domain hierarchy.

6) *Which protocols are preferred?* During protocol arbitration, downward nodes can also communicate their protocol preferences. This information could be used to finalize a protocol selection if the other factors do not clearly indicate a better choice.

Once a multicast protocol has been selected for each downward node, this information is communicated to those nodes by sending a **MAPAddGroupRequest** to each of them. This will cause each receiving node to add that group to their list of active groups. It will also cause each node to continue propagating the group assignment downward by repeating the above steps for each of their downward connections.

Since **MAPAddGroupRequest** specifies the range of priorities that are to use the multicast group, it is possible for different priorities to use different groups (even for the same channel). When determining which multicast group to send data to, MAP uses channel ID, reliability level, *and* priority. If a given priority is not mapped to multicast, then data at that priority is sent via unicast.

Note that not all nodes in a multicast island will need to join a multicast group immediately. A node will join the group if any of the following conditions are true: if there are local user attachments that need to receive the data; if there are connections through other transport stacks that need to receive the data; or if this node must redirect the data via unicast to any neighbors. Some transport protocols may also require a node to join a group before transmitting data to it (even if that node has no interest in receiving data from the group).

The following table summarizes the **MAPAddGroupRequest** contents.

Metachannel
Transport Protocol ID
Multicast Group Address

Metachannel: This field indicates which metachannel is being mapped to a multicast group.

Transport Protocol ID: This field uniquely identifies the transport protocol to be used for this metachannel.

Multicast Group Address: This is the multicast group address upon which the data for the metachannel will be sent. This field is formatted as an NSAP address.

A.9.8.1 Metachannel

The following table defines the fields that make up a metachannel. This type is used to identify the abstraction that MAP assigns multicast groups to.

Channel ID
Reliability Level
Priority Range

Channel ID: This field indicates which MCS channel is associated with the metachannel.

Reliability Level: This field indicates which reliability level is associated with the metachannel.

Priority Range: This is the priority range that is associated with the metachannel (zero is the highest priority).

A.9.9 MAPRemoveGroupRequest

This MAPPDU is used to remove multicast groups from downward nodes in the multicast island. This process is originated by the Multicast Group Provider when it becomes necessary to remove a multicast group from use. When a node receives a **MAPRemoveGroupRequest**, it must forward the request downward to any nodes beneath it in the multicast island.

Typically, a **MAPRemoveGroupRequest** will not be issued downward until use of the multicast group has already stopped. It is possible to ensure that a multicast group is not being actively used by issuing a **MAPDisableMulticastRequest** downward, and waiting for a **MAPDisableMulticastConfirm** (from all downward nodes in the multicast island). Once this is done, the Multicast Group Provider can be assured that no one is using the group, so it can be removed without fear of data loss.

When a node receives a **MAPRemoveGroupRequest**, it should immediately remove the group and forward the request downward toward all lower nodes in the island.

The following table summarizes the **MAPRemoveGroupRequest** contents.

Metachannel

Metachannel: This field indicates which metachannel the group is being removed from.

A.9.10 MAPDisableUnicastRequest

This MAPPDU is used to disable unicast data from a neighboring node for a specified data flow.

A node should not transmit this MAPPDU until it has verified that it can transition to multicast reception without losing any data. To do this it must have received at least one data frame via unicast, in order to know what the next expected sequence number is. Furthermore, it must receive a data frame via multicast with a sequence number less than or equal to the expected sequence number (so that it knows it is synchronized). Only then will it know that it is okay to shut off the flow of unicast data (for that data flow).

The following table summarizes the **MAPDisableUnicastRequest** contents.

Data Flow Identifier

Data Flow Identifier: This field contains the information necessary to determine which data flow unicast is being disabled for. The data flow identifier is fully defined in the subclause discussing the **MAPData** PDU. Sender ID will be omitted for downward flowing uniform and proxy data flows, but will otherwise be present. Metachannel ID and Data Type will always be present.

A.9.11 MAPEnableUnicastRequest

This MAPPDU is used to re-enable unicast from a neighboring node for a specified data flow.

This MAPPDU would be sent if a node determines that it is no longer receiving data via multicast, or that the error rates for multicast transmission are so high that it is not efficient to use it. There must be some local communication between MAP and the transport protocol to determine when this is necessary. This could also happen in response to **MAPSequenceNumber** PDUs that have revealed a stoppage in multicast data flow.

The following table summarizes the **MAPEnableUnicastRequest** contents.

Data Flow Identifier

Data Flow Identifier: This field contains the information necessary to determine which data flow unicast is being enabled for. The data flow identifier is fully defined in the subclause discussing the **MAPData** PDU. Sender ID will be omitted for downward flowing uniform and proxy data flows, but will otherwise be present. Metachannel ID and Data Type will always be present.

A.9.12 MAPEnableUnicastConfirm

This MAPPDU confirms that unicast has been re-established for the specified data flow. When a node tells a neighboring node to turn unicast back on, it may still be necessary for it to receive more data via multicast. This would happen if it is already too late for the neighboring node to send some data frames via unicast, because they have already passed through. This MAPPDU allows the neighboring node to tell the originator of the **MAPEnableUnicastRequest** what the next sequence number that will be unicast is. That way, the node can continue to use multicast (if possible) to listen for all data frames up through that one. This is necessary in order to ensure that no data are lost during the transition from multicast back to unicast.

For example, a node has received sequence numbers through 20 for a given data flow (but with very high error rates). It may elect to turn on unicast to better deal with the errors. However, the neighboring node may have already processed sequence numbers through 23 for that data flow. If the node immediately begins relying on unicast, it will miss sequence numbers 21 through 23. This confirm tells the originator of the request that the next sequence number it will unicast will be 24, so all sequence numbers through 23 must be received via multicast before it can depend on unicast.

The following table summarizes the **MAPEnableUnicastConfirm** contents.

Data Flow Identifier
Sequence Number

Data Flow Identifier: This field contains the information necessary to determine which data flow unicast is being enabled for. The data flow identifier is fully defined in the subclause discussing the **MAPData** PDU. Sender ID will be omitted for downward flowing uniform and proxy data flows, but will otherwise be present. Metachannel ID and Data Type will always be present.

Sequence Number: This is the next sequence number in the data flow that will travel to the node via unicast. Anything prior to this sequence number must be received via multicast before the node can depend on the unicast data stream.

A.9.13 MAPDisableMulticastRequest

This MAPPDU is transmitted downward in the multicast island to disable the use of multicast prior to group removal.

When a node receives a **MAPDisableMulticastRequest**, it must initiate the process of re-enabling unicast for any data flows for which unicast was previously disabled (for this group). It does this by sending a **MAPEnableUnicastRequest** in the direction of the sender for each affected data flow. The node should also mark the group as being in the disabled state, which prohibits the node from disabling unicast for any data flows on the group. It should NOT, however, stop transmitting to the group.

The **MAPDisableMulticastRequest** should be repeated downward to all nodes that lie beneath the receiver. A node cannot respond with a **MAPDisableMulticastConfirm** until it has received similar confirms from all downward nodes, AND it has successfully re-enabled unicast for all data flows for which unicast was previously disabled (for this group). It cannot assume that unicast is re-enabled for a data flow until it has received a **MAPEnableUnicastConfirm** for each outstanding request (and it must have received all data via multicast with a sequence number less than the one in each confirm).

The following table summarizes the **MAPDisableMulticastRequest** contents.

Metachannel

Metachannel: This is the metachannel associated with the group for which multicast is being disabled. The metachannel type is fully described in the subclause on **MAPRemoveGroupRequest**.

A.9.14 MAPDisableMulticastConfirm

This MAPPDU is used to confirm that all nodes beneath the receiver have stopped using a multicast group.

When a node receives a **MAPDisableMulticastRequest** from the node above, it should respond by sending a **MAPDisableMulticastConfirm** after two conditions have been met: it must have received **MAPDisableMulticastConfirm** PDUs from all nodes beneath it in the multicast island (if any); and it must be sure that unicast is enabled for all data flows on the group.

The first condition is met as a result of forwarding the **MAPDisableMulticastRequest** to the downward nodes. The second condition is met by re-enabling unicast for any data flows that were previously disabled (this is done using the **MAPEnableUnicastRequest** PDU).

The following table summarizes the **MAPDisableMulticastConfirm** contents.

Metachannel

Metachannel: This is the metachannel associated with the group for which multicast has been disabled. The metachannel type is fully described in the subclause on **MAPRemoveGroupRequest**.

A.9.15 MAPEnableMulticastRequest

This MAPPDU is used to re-enable a multicast group that was previously disabled.

When a node receives a **MAPEnableMulticastRequest** for a group that is marked disabled (but still valid), it will re-enable the use of that group. This means that **MAPDisableUnicastRequest** PDUs can again be processed in order to shut off unicast data flow. A re-enabled group takes on the same state it had when first created.

This request should be repeated downward to all nodes that lie beneath the receiver.

The following table summarizes the **MAPEnableMulticastRequest** contents.

Metachannel

Metachannel: This is the metachannel associated with the group for which multicast is being re-enabled. The metachannel type is fully described in the subclause on **MAPRemoveGroupRequest**.

A.9.16 MAPSequenceNumber

This MAPPDU is used to inform downstream receivers of what sequence numbers they should have received up until the current time. This MAPPDU is propagated outward from the originator of the data flow via unicast, and is repeated by all receiving nodes to those nodes that lie farther from the sender (across the upward connection, as well as any downward connections that lead to users joined to the specified channel). In this way, all nodes can be informed of what sequence numbers for the data flow should have been received.

This MAPPDU could be used to determine when a node simply stops receiving data on a multicast group. This could be useful for unreliable protocols where there is no attempt at error correction. It may also be useful for transport protocols that do not cleanly detect the loss of a multicast pathway from a sender to a receiver. The exact use of the MAPPDU is for further study.

The following table summarizes the **MAPSequenceNumber** contents.

Data Flow Identifier
Sequence Number

Data Flow Identifier: This field contains the information necessary to determine which data flow this sequence number is associated with. The data flow identifier is fully defined in the subclause discussing the **MAPData** PDU. Sender ID will be omitted for downward flowing uniform and proxy data flows, but will otherwise be present. Metachannel ID and Data Type will always be present.

Sequence Number: This is the last sequence number that was transmitted for the specified data flow.

A.10 MAPPDU ASN.1 Definition

This subclause contains the actual format of the MAPPDUs, using the ASN.1 notation. The contents of this subclause will also be made available as a separate, compilable, text file.

```
--*****************************************************************************
--*
--*              ASN.1 Definition for MAP PDUs
--*
--*****************************************************************************

MAP-PROTOCOL DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

H221NonStandardIdentifier ::= OCTET STRING (SIZE (4..255))
```

```
Key ::= CHOICE
{
        object                          OBJECT IDENTIFIER,
        h221NonStandard                 H221NonStandardIdentifier
}

NonStandardParameter ::= SEQUENCE
{
        key                             Key,
        data                            OCTET STRING
}

NonStandardPDU ::= SEQUENCE
{
        data                            NonStandardParameter,
        ...
}

VersionNumber ::= SEQUENCE
{
        majorVersionNumber              INTEGER (0 .. 65535),
        minorVersionNumber              INTEGER (0 .. 65535),
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

Priority ::= INTEGER (0 .. 15)

PriorityRange ::= SEQUENCE
{
        highPriority                    Priority,
        lowPriority                     Priority,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPConnectRequestPDU ::= SEQUENCE
{
        versionNumber                   VersionNumber,
        connectionMAPSAP                INTEGER (0 .. 65535),
        domainReferenceID               INTEGER (0 .. 65535),
        priorityRange                   PriorityRange,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPConnectConfirmPDU ::= SEQUENCE
{
        versionNumber                   VersionNumber,
        connectionMAPSAP                INTEGER (0 .. 65535),
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

Reason ::= CHOICE
{
        providerInitiated               NULL,
        userRejected                    NULL,
        userInitiated                   NULL,
        invalidMAPSAP                   NULL,
        invalidDomainReferenceID        NULL,
```

```
        unicastTransition               NULL,
        unspecifiedFailure              NULL,
        nonStandardReason               NonStandardParameter,
        ...
}

MAPDisconnectRequestPDU ::= SEQUENCE
{
        reason                          Reason,
        confirmRequired                 BOOLEAN,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPDisconnectConfirmPDU ::= SEQUENCE
{
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TransportProtocolID ::= CHOICE
{
        objectProtocolID                OBJECT IDENTIFIER,
        h221NonStandardProtocolID       H221NonStandardIdentifier,
        snapProtocolID                  OCTET STRING (SIZE (5)),
        nonStandardProtocolID           NonStandardParameter,
        ...
}

TransportProtocolType ::= CHOICE
{
        reliableUnicast                 NULL,
        unreliableUnicast               NULL,
        reliableMulticast               NULL,
        unreliableMulticast             NULL,
        nonStandardProtocolType         NonStandardParameter,
        ...
}

NetworkAddress ::= SEQUENCE
{
        nsapAddress                     OCTET STRING (SIZE (1..20)),
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

PayloadSize ::= INTEGER (128 .. 65535)

TransportProtocolRequestEntry ::= SEQUENCE
{
        transportProtocolID             TransportProtocolID,
        transportProtocolType           TransportProtocolType,
        networkAddress                  NetworkAddress,
        maximumPayloadFixedFlag         BOOLEAN,
        maximumPayloadSize              PayloadSize,
        preferenceWeighting             INTEGER (0 .. 65535) OPTIONAL,
        nodeCount                       INTEGER (0 .. 65535) OPTIONAL,
        numberOfConnections             INTEGER (0 .. 65535) OPTIONAL,
        configurationData               OCTET STRING OPTIONAL,
```

```
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPArbitrateProtocolsRequestPDU ::= SEQUENCE
{
        domainReferenceID               INTEGER (0 .. 65535),
        moreToComeFlag                  BOOLEAN,
        transportProtocolMenu           SEQUENCE OF TransportProtocolRequestEntry,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

TransportProtocolConfirmEntry ::= SEQUENCE
{
        transportProtocolID             TransportProtocolID,
        transportProtocolType           TransportProtocolType,
        networkAddress                  NetworkAddress,
        maximumPayloadSize              PayloadSize,
        preferenceWeighting             INTEGER (0 .. 65535) OPTIONAL,
        nodeCount                       INTEGER (0 .. 65535) OPTIONAL,
        numberOfConnections             INTEGER (0 .. 65535) OPTIONAL,
        configurationData               OCTET STRING OPTIONAL,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPArbitrateProtocolsConfirmPDU ::= SEQUENCE
{
        domainReferenceID               INTEGER (0 .. 65535),
        moreToComeFlag                  BOOLEAN,
        transportProtocolMenu           SEQUENCE OF TransportProtocolConfirmEntry,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

SenderID ::= INTEGER (1001 .. 65535)

ChannelID ::= INTEGER (1 .. 65535)

ReliabilityLevel ::= CHOICE
{
        reliable                        NULL,
        unreliable                      NULL,
        nonStandardReliabilityLevel     NonStandardParameter,
        ...
}

DataType ::= CHOICE
{
        nonuniform                      NULL,
        uniform                         NULL,
        proxy                           NULL,
        nonStandardDataType             NonStandardParameter,
        ...
}

DataFlowIdentifier ::= SEQUENCE
{
        senderID                        SenderID OPTIONAL,
```

```
        metachannelID SEQUENCE
        {
                channelID                       ChannelID,
                reliabilityLevel                ReliabilityLevel,
                priority                        Priority,
                ...
        } OPTIONAL,

        dataType                        DataType OPTIONAL,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

SequenceNumber ::= INTEGER (0 .. 65535)

DataDescriptor ::= SEQUENCE
{
        unicastForwardFlag              BOOLEAN,
        startingSequenceNumber          SequenceNumber,
        dataFlowIdentifier              DataFlowIdentifier OPTIONAL,
        domainReferenceID               INTEGER (0 .. 65535) OPTIONAL,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

DataFrameEntry ::= SEQUENCE
{
        firstSegmentFlag                BOOLEAN,
        lastSegmentFlag                 BOOLEAN,
        userData                        OCTET STRING,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPDataPDU ::= SEQUENCE
{
        dataDescriptor                  DataDescriptor OPTIONAL,
        dataFrameArray                  SEQUENCE OF DataFrameEntry,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

Metachannel ::= SEQUENCE
{
        channelID                       ChannelID,
        reliabilityLevel                ReliabilityLevel,
        priorityRange                   PriorityRange,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPAddGroupRequestPDU ::= SEQUENCE
{
        metachannel                     Metachannel,
        transportProtocolID             TransportProtocolID,
        multicastGroupAddress           NetworkAddress,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
```

```
MAPRemoveGroupRequestPDU ::= SEQUENCE
{
        metachannel                     Metachannel,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPDisableUnicastRequestPDU ::= SEQUENCE
{
        dataFlowIdentifier              DataFlowIdentifier,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPEnableUnicastRequestPDU ::= SEQUENCE
{
        dataFlowIdentifier              DataFlowIdentifier,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPEnableUnicastConfirmPDU ::= SEQUENCE
{
        dataFlowIdentifier              DataFlowIdentifier,
        sequenceNumber                  SequenceNumber,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPDisableMulticastRequestPDU ::= SEQUENCE
{
        metachannel                     Metachannel,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPDisableMulticastConfirmPDU ::= SEQUENCE
{
        metachannel                     Metachannel,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPEnableMulticastRequestPDU ::= SEQUENCE
{
        metachannel                     Metachannel,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}

MAPSequenceNumberPDU ::= SEQUENCE
{
        dataFlowIdentifier              DataFlowIdentifier,
        sequenceNumber                  SequenceNumber,
        nonStandardParameters           SEQUENCE OF NonStandardParameter OPTIONAL,
        ...
}
```

```
MAPPDU ::= CHOICE
{
        mapConnectRequest                   MAPConnectRequestPDU,
        mapConnectConfirm                   MAPConnectConfirmPDU,
        mapDisconnectRequest                MAPDisconnectRequestPDU,
        mapDisconnectConfirm                MAPDisconnectConfirmPDU,
        mapArbitrateProtocolsRequest        MAPArbitrateProtocolsRequestPDU,
        mapArbitrateProtocolsConfirm        MAPArbitrateProtocolsConfirmPDU,
        mapData                             MAPDataPDU,
        mapAddGroupRequest                  MAPAddGroupRequestPDU,
        mapRemoveGroupRequest               MAPRemoveGroupRequestPDU,
        mapDisableUnicastRequest            MAPDisableUnicastRequestPDU,
        mapEnableUnicastRequest             MAPEnableUnicastRequestPDU,
        mapEnableUnicastConfirm             MAPEnableUnicastConfirmPDU,
        mapDisableMulticastRequest          MAPDisableMulticastRequestPDU,
        mapDisableMulticastConfirm          MAPDisableMulticastConfirmPDU,
        mapEnableMulticastRequest           MAPEnableMulticastRequestPDU,
        mapSequenceNumber                   MAPSequenceNumberPDU,
        nonStandardPDU                      NonStandardPDU,
        ...
}

END

--*****************************************************************************
--*
--*           End of ASN.1 Definition for MAP PDUs
--*
--*****************************************************************************
```

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	**Terminals for telematic services**
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communication
Series Y	Global information infrastructure
Series Z	Programming languages