CMAP

Ken Cox and John DeHart

WUCS-94-21

Version 3.0 (ARL-94-08)

Previous Versions (WUCS-92-01, ARL-89-06) authored by:

John DeHart
Mike Gaddis
Rick Bubenik

July, 1994

Department of Computer Science
Washington Univeristy
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

Connection Management Access Protocol (CMAP) Specification

Ken Cox and John DeHart

Version 3.0
November 10, 1994

Applied Research Laboratory Working Note —**ARL-94-08**
Department of Computer Science Technical Report —**WUCS-94-21**

Applied Research Laboratory
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, Missouri 63130-4899
Telephone: 314-935-6160
FAX: 314-935-7302
Email: {jdd, kcc}@arl.wustl.edu

Abstract

This document specifies a *Connection Management Access Protocol* (CMAP) for call management in high-speed packet switched networks. We target CMAP to networks employing the *Asynchronous Transfer Mode* (ATM) communication standard. CMAP specifies the access procedures exercised by network *clients* to manipulate multipoint calls; it is thus a *User-Network Interface* (UNI) signalling protocol. We define a *multipoint call* as a group of *multipoint connections*. A *multipoint connection* is a communication channel between two or more clients or *endpoints* of the network, where all data sent by one client is received by all other clients who have elected to receive. A *point-to-point connection* is a special case of a multipoint connection involving only two clients. CMAP provides facilities to create, modify, and delete calls, connections, and endpoints. Once a connection is established, clients exchange data using ATM data-transfer protocols that are specified separately from CMAP.

Hypertext Linkage

Previous Link

Master Index
Document Index

Title | Contents | Figures | Quit
Tables | Body | Index

Author: **Ken Cox and John DeHart**
Organization: **Applied Research Laboratory**
Project: **Zeus Project**
File: **/project/gbn_sw/switch/documentation/CM/CMAP/CMAP_hdr.frm**
Created: **February 26, 1992**
Modified: **July 8, 1994 4:41 pm by kcc**

Version Information

Version 1.0: Initial document containing CMAP message formats and little else.

Version 2.0: Added introductory sections on ATM networks, the call model and the protocol stack (these sections were derived mostly from papers published by the authors). Changed many of the CMAP message formats and added some new messages. Added detailed explanations where deemed necessary to highlight obscure CMAP features. Added implementation and future direction sections (incomplete).

Version 2.1: Major editing rewrite of Sections 1 through 8.5 from version 2.0 (now organized as Sections 1 through 6). More editing to come.

Version 2.1.1: Minor editing changes from version 2.1. More editing to come (temporarily on hold).

Version 3.0: Major editing involving restructuring of document, removal of redundacies, and general modifications/clarifications preparatory to implementation. More editing expected as implementation proceeds. Summary of major changes:

- Restructuring of document as a FrameMaker "book" with table of contents, lists of figures and tables, and index.
- Collection of several sections from earlier versions into a single chapter describing the environment in which CMAP operates.
- Expansion and clarification of the call model, including: separation of ownership and root functions; owner now need not be a participant in call; more complete listing of call, connection, *etc*. parameters; clarification of the distinction between clients and endpoints and between connections and UNI parameters, and more consistent use of the terms throughout document; modification of the system for connection and endpoint mappings.
- Collection of information about messages into a single chapter, including: explanation of use of messages; definitions of ACKs, NACKs, *etc*.; redefinition of message objects to support multidrop/surrogate signalling and other changes to model; complete definitions of all message fields.
- Addition of commands to support client/network interfacing (*e.g*., **status**, **alert**).
- Renaming of many commands to better reflect their use (*e.g*., all notifications now begin with **announce_**).
- Rewriting of command explanations, including: explanation of message traffic produced by command; parameter negotiation in command; state machines modified to reflect separation of call ownership from participation in call and other changes to model.
- Expansion of examples, and linkage to actual applications.
- Wrote a preliminary version of the "Future Directions" section.
- Modification of appendices to reflect changes elsewhere in document.
- Unification of the state machines for the various operation types.

Copyright Notification

Authors (Version 3.0 © 1994):
Ken Cox and John DeHart

Previous Authors:
John DeHart, Mike Gaddis, and Rick Bubenik

Table of Contents

List of Figures

List of Tables

1. Introduction

This document describes a *call* model for multipoint connections in a switched Asynchronous Transfer Mode (ATM) networks and specifies the Connection Management Access Protocol (CMAP) that allows network *clients* to create, manipulate and delete multipoint, multiconnection communication channels, which we term *calls*. A *multipoint call* is a call involving two or more clients; a *point-to-point* call is a special case of a multipoint call involving only two clients. Data sent over a connection by one participant in a call is received by all other participants electing to receive on this connection, although reliable delivery is not guaranteed by the network. Calls are allowed to change dynamically during their lifetime, in terms of the number of participants, the number of connections and the reserved bandwidth of the connections.

CMAP defines the interface between clients and the network used to create, manipulate and delete calls. As such, CMAP is an ATM *User Network Interface* (UNI) signalling protocol [3, 14]. It is layered over a reliable substrate, which we term CTL (for CMAP Transport Layer). We do not specify the CTL protocol. Rather, we list the requirements for CTL, which are generally met by several existing transport protocols (for example, TCP/IP over type 4 AAL). In this way, CMAP implementors can choose the most suitable CTL for their implementation environment.

Although CMAP and the associated call model provide a rich set of operations and capabilities, CMAP implementations are not required to be complete: CMAP implementations can omit certain operations and capabilities, while still adhering to the general message layouts and request/acknowledgment handshake procedures. A certain set of core capabilities are required, which we term *minimal* CMAP (Section 3.6). All other capabilities are optional.

CMAP also exhibits flexibility in the areas of addressing and data transfer. CMAP does not dictate any one addressing scheme. Rather, CMAP supports multiple addressing disciplines and multiple routing protocols. Currently-defined addressing schemes include IP (Internet Protocol) addressing [18], public network E.164 addressing [13], and OSI NSAP addressing [17]. An implementation of CMAP may support any or all of these schemes, as well as others. Similarly, CMAP does not require clients to use a particular data transfer protocol (*e.g.*, AAL-5) on the connections it creates. Instead, CMAP supplies raw ATM connections on which clients may layer any protocol they find appropriate.

The remainder of this document is organized as shown in Table 1.

Table 1. Document Structure

Section and Title	Description
Section 1: **Introduction**	Call model and CMAP overview.
Section 2: **Switched ATM Networks**	Introduction to switched, connection-oriented ATM networks.
Section 3: **CMAP Network Environment**	Description of CMAP layering over ATM networks.
Section 4: **Call Model**	Detailed description of the call model.
Section 5: **CMAP Messages**	General information on message use, formats and transmission.
Section 6: **CMAP Operations**	Detailed description of all CMAP operations.
Section 7: **Examples**	Examples of CMAP operations, including several types of call setup.
Section 8: **Future Directions in CMAP**	List of CMAP enhancements being considered.
Appendix A: **References**	List of related references.
Appendix B: **Acronym List**	List of acronyms used in document.
Appendix C: **CMAP Message Field Values**	Numerical values for CMAP message parameters.
Appendix D: **CMAP Status Codes**	Numerical values and actions for CMAP error codes.
Appendix E: **Endpoint Mappings**	Tabular list of all possible endpoint receive/transmit mappings.
Appendix F: **Parameter Negotiation**	Summary of parameter negotiation.

2. Switched ATM Networks

This section presents an overview of the switched ATM network architectures, the ATM standard, ATM network connections, and the two types of ATM connections (*Virtual Path* and *Virtual Channel*).

2.1 Network Architecture

An example ATM network is shown in Figure 1. The network consists of *clients*, *exterior nodes* (nodes that interface to clients), and *interior nodes* (nodes that interface to other nodes only), all interconnected by *fiber optic links*. Clients signal the network to set up connections to other clients by signalling exterior nodes using a *User Network Interface* (UNI) signalling protocol [3, 14]. The exterior nodes perform the requested operation, communicating if necessary with other nodes (both interior and exterior) using a *Network Node Interface* (NNI) signalling protocol.

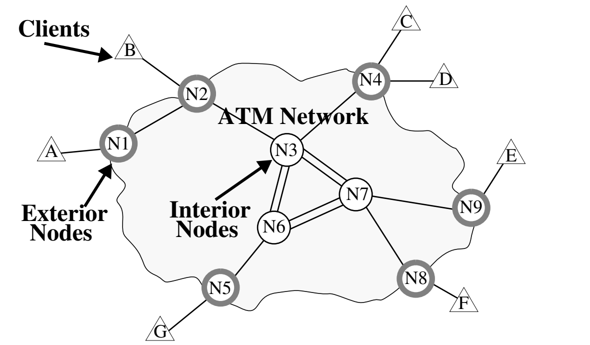


Figure 1. Example ATM Network

Clients may send ATM cells (packets) to the network and receive cells from the network along their fiber links. Each node in the network contains one or more ATM *switches* [29, 33, 34, 36, 37, 59, 62]. The switches route each ATM cell to the desired destination link(s) based upon header fields in the cell (Section 2.3). In order to keep up with line speeds, the switches perform all routing in hardware. Since the time interval within which each cell must be routed is very small, tables in the switches are preconfigured with routing information. This makes ATM networks more suitable for connection-oriented traffic, where the switch tables can be configured during connection setup. Connectionless traffic can be accommodated via *overlay* networks utilizing special-purpose routers or datagram processors [5, 9].

Figure 2 shows the architecture of one ATM switch, Turner's *Broadcast Packet Switch* [59, 62]. This switch contains a *Copy Network* (CN) concatenated to a *Routing Network* (RN). ATM cells (packets) enter the switch on the left. Multicast cells, destined for several locations, are replicated by the CN, then routed to the appropriate destination by the RN. Point-to-point cells follow an arbitrary path through the CN (following the path of "least resistance"), then are routed by the RN. Cells leave the switch on the right, where they traverse fiber optic links to other switches or clients.

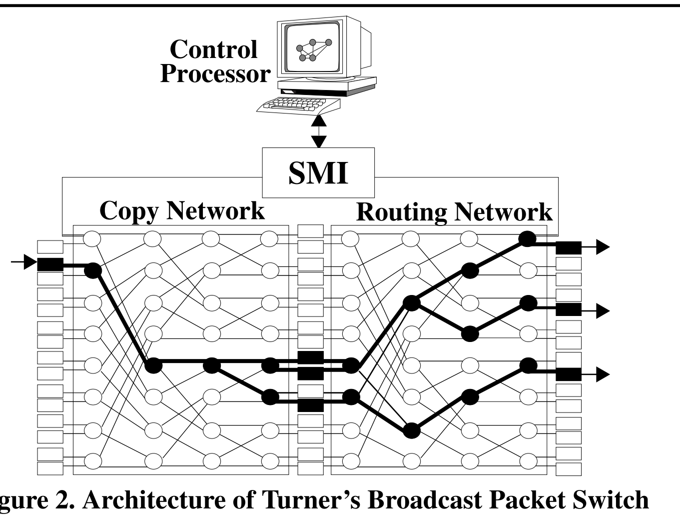


Figure 2. Architecture of Turner's Broadcast Packet Switch

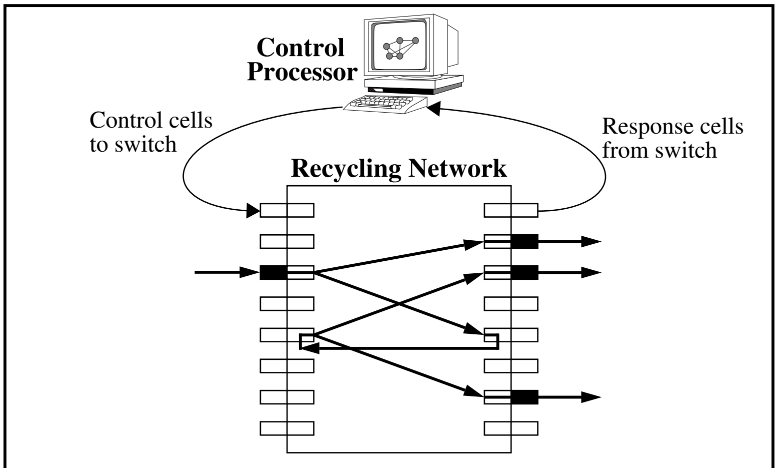


Figure 3. Architecture of Turner's Gigabit Recycling Switch

The broadcast packet switch is controlled by the *Control Processor* (CP) connected to the switch via an Ethernet-based *Switch Module Interface* (SMI). The CP configures the switch hardware to route incoming cells to the appropriate outgoing links by modifying tables within the switch, thus establishing connections.

Figure 3 shows the architecture of another ATM switch, Turner's *Gigabit Recycling Switch* [59, 62]. The internal structure of this switch differs considerably from that of the Broadcast Packet Switch—on each pass through the network at most two copies of a cell are produced, with larger numbers of copies obtained by recycling cells through the network—but the external behavior is the same. ATM cells enter the switch on the left, are replicated and routed as specified by internal tables, and leave the switch on the right. The recycling switch is also controlled by a CP, but instead of using an SMI the CP sends control information over the fiber links in the form of special control cells.

Figure 4 illustrates our concept of a *node*, where more than one ATM switch is under the control of a single CP. The CP manages routing for all the switches in the node, sending commands to update the tables in each switch. We assume that the CP is directly connected to (at least) one *main* switch. Its connections to the other, *satellite* switches may be indirect, for example, it may have to send commands in the form of ATM cells over the fiber links connecting the switches within the node. Note that, from the "outside", a node can be viewed as a single large switch through which connections may be routed, with the routing inside the node hidden.

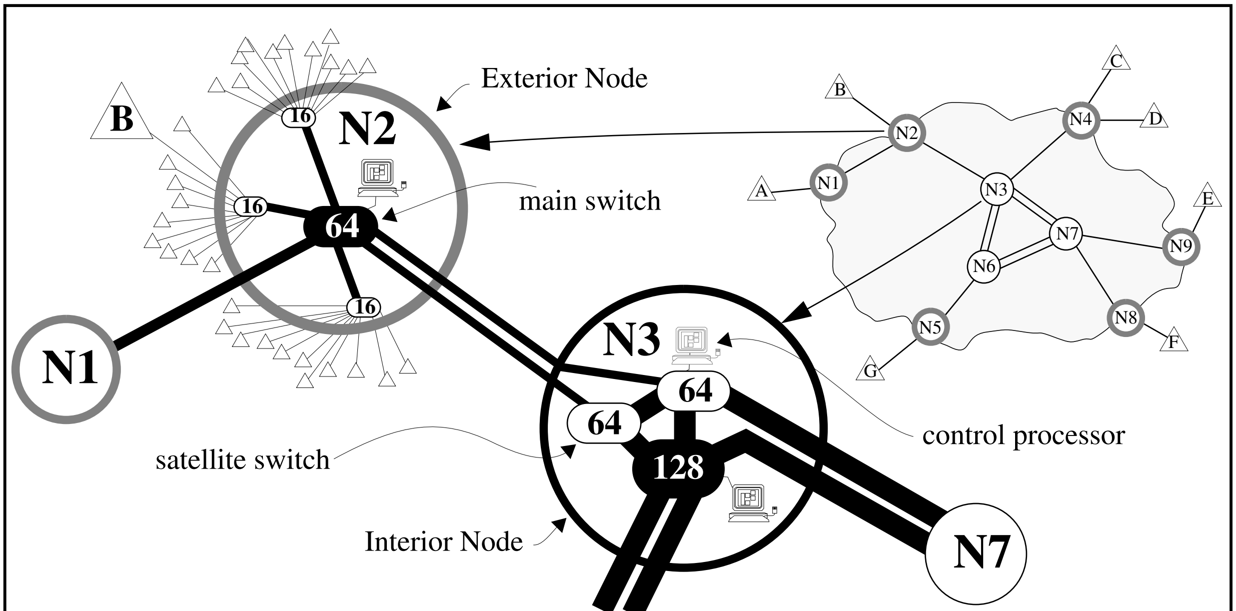


Figure 4. Generalized Node Architecture for Interior and Exterior Network Nodes

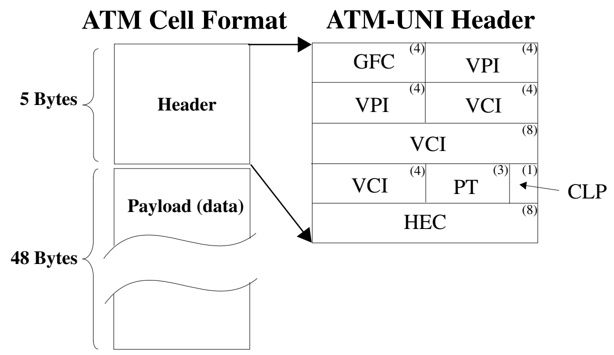


Figure 5. ATM UNI Cell Format.

2.2 The ATM Standard

The emerging ATM standard [3, 14] specifies link-level cell formats for two interfaces: 1) the User Network Interface (UNI), for communication between the client and the network, and 2) the Network Node Interface (NNI), for communication between network nodes. The ATM UNI cell format is shown in Figure 5. It consists of a 48-byte payload (data) field and a five-byte header. The header has six fields: a Global Flow Control (GFC, 4 bits), a Virtual Path Identifier (VPI, 8 bits), a Virtual Channel Identifier (VCI, 16 bits), a Payload Type (PT, 3 bits), a Cell Loss Priority (CLP, 1 bit) and a Header Error Check (HEC, 8 bits).

Use of the GFC has not yet been standardized, although the intent is to use this field for arbitration on shared media access links (such as DQDB). The VPI and VCI fields are used to route cells, as described in the next section. The CLP bit is used to mark low priority cells, where CLP=1 indicates low priority. This bit may be set either by clients or the network. The last header field, the HEC, is a cyclic redundancy check (CRC) on the header.

The three-bit PT field is used to distinguish data cells from other cells. There are four types of client data cells, all with the most-significant PT bit set to 0. The other two bits are the network congestion bit (the middle bit) and the tagged data bits (the least-significant bit). The network congestion bit is used by the network to inform clients receiving a cell that congestion was encountered somewhere in the network. The tagged data PT marking can be used by the client to differentiate cells. This marking is preserved by the network. One potential use is in delineating segmented frames, where the cell containing the last fragment of the frame is tagged and all other frames are untagged [9, 42]. An example of this is shown in Figure 6 for IP datagram frames. The remaining four PT values (those with the most-significant bit equal to 1) are reserved for network control and resource management functions.

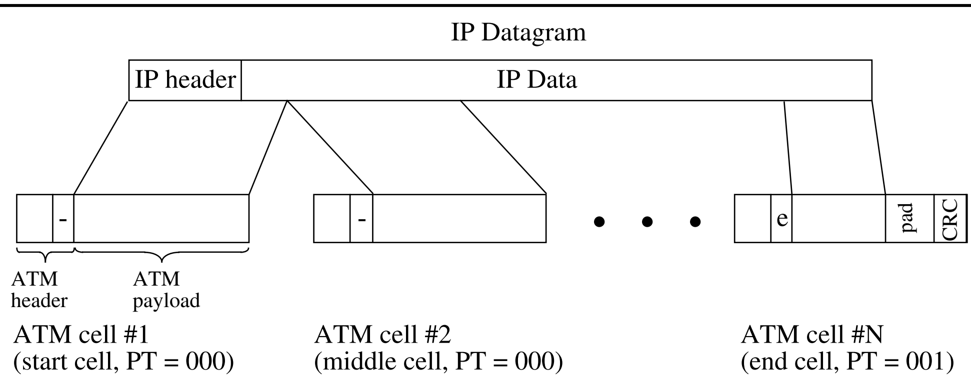


Figure 6. IP Segmentation/Reassembly Scheme.

2.3 Virtual Path and Virtual Channel Connections

The VPI and VCI header fields are used to route cells. The ATM standard provides for two types of routing: *Virtual Path* (VP) and *Virtual Channel* (VC). Both types of routing are based on an abstraction of ATM networks which we call *cell pipes* or *connections*. Clients of ATM networks communicate over cell pipes by sending and receiving

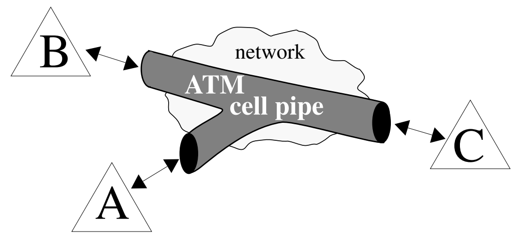


Figure 7. Three-way Cell Pipe

ATM cells. In the general case a cell pipe may be *n*-way (any number of clients), bidirectional (a client may both send and receive data on the same pipe), and multipoint-to-multipoint (all clients may send data, and all clients connected to the cell pipe will receive the data). Figure 7 shows the conceptual view of a three-endpoint cell pipe between clients A, B and C. All data sent by one client is received by all other clients who have elected to receive on this cell pipe.

Clients access cell pipes using the VPI and VCI fields of the header. To be more precise, clients place particular values in the VPI and VCI fields of each ATM cell that they send. The network uses these values to route the cells to the other clients that are using the same cell pipe. The routing tables in the network switches are configured with mapping information which, for each incoming VPI/VCI combination, indicates to which switch output line(s) the cell should be sent and what VPI/VCI pairs should be written into the headers of the output cells. The routing process thus could remap the VPI/VCI pair at each switch in the network.

In a virtual path connection, the network uses the VPI for routing, possibly remapping this field at every switch within the network. The VCI field is transmitted end-to-end unchanged by the network and is available for use by clients. Thus, in VP connections clients route the cell using the VPI field and may place any value in the VCI field. One anticipated use of the VCI can be used for source discrimination in multipoint connections, where each transmiting client places a unique VCI in all of its outgoing cells.

In a virtual channel connection, the network uses both the VPI and VCI for routing, possibly remapping both fields at every switch within the network. Clients route the cell using both the VPI and VCI fields; neither field is available for other purposes such as multiplexing. VC connections are desirable for connections that do not need source discrimination, and for connections that want to take advantage of rapid setup (where the network is able to reduce connection setup overhead by using preconfigured trunks).

Figure 8 shows an expanded view of the cell pipe from Figure 7. This cell pipe has been set up as a VP connection. Client A accesses the cell using VPI 9, client B uses VPI 3, and client C uses VPI 7. Any cell transmitted by one of these clients on the cell pipe is delivered to the other two with its VCI field unchanged. When a client is receiving from two or more transmitters, the cells from the two sources may be interleaved in an arbitrary fashion. However, the ATM standard guarantees that cells from each transmitter will be received in the order that they were sent.

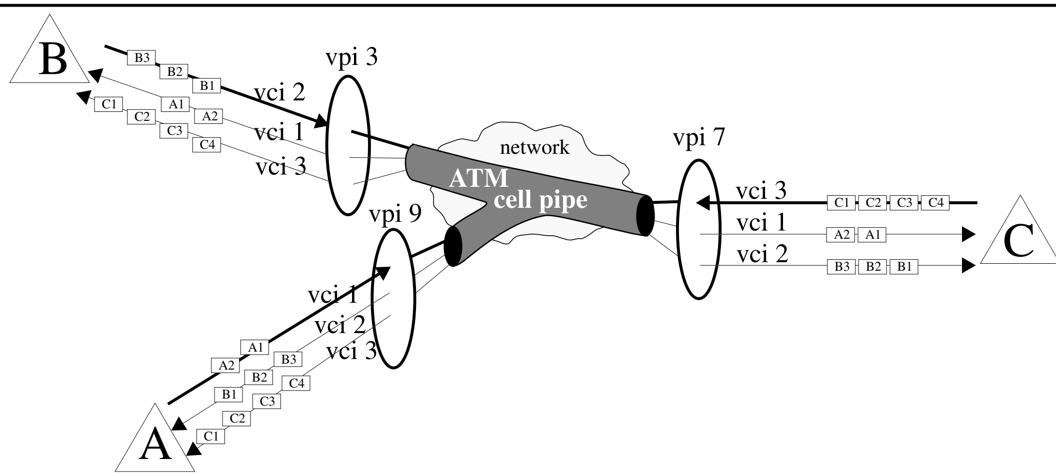


Figure 8. Multipoint VP Connection Using VCI for Source Discrimination

In this example, the clients are using the VCIs for source discrimination. Client A is setting the VCI of cells it transmits to 1, client B to 2, and client C to 3. The receivers can thus distinguish the source of each cell received, even though cells from two or more different transmitters may be interleaved on receipt. This technique would not be possible in a VC connection; therefore, if a VC connection is used for multipoint communication, higher-level protocol information must be embedded in the ATM cell payload for source discrimination.

ATM cell pipes in a network are set up, modified, and torn down under the control of software which arranges that appropriate values be stored in the routing tables of each switch involved in the connection. CMAP forms one component of this software. The next chapter examines CMAP's role in network routing in greater detail.

3. CMAP Network Environment

This section discusses how CMAP is used in a networking environment. Section 3.1 discusses the role of CMAP in network operations. The next three sections (3.2 through 3.5) describe the support provided by the network and by other software components. Finally, Section 3.6 defines complete and minimal CMAP implementations and indicates what facilities are necessary for a minimal implementation.

3.1 CMAP as a Session Management Protocol

Figure 9 shows the network process and protocol architecture used with CMAP. We believe that this abstract view of the network encompasses most actual and proposed network management implementations and is thus not unduely restrictive.

Management of ATM connections is encapsulated in a Connection Management Layer (CML), which provides facilities for building and destroying ATM connections. Clients have no knowledge of or direct access to the CML. Instead, clients interact with the Session Management Layer (SML), which provides a “higher-level” interface to the network resources. The Session Managers (processes of the SML) use the facilities provided by the CML to perform the network operations requested by clients. A network may support several different types of Session Manager, each providing a distinct client interface to the network. Contention for network resources by the Session Managers is resolved at the CML.

CMAP is a client interface protocol based on the creation, manipulation, and deletion of *calls* (Section 4). CMAP clients use a reliable ATM transport protocol (generically called CTL, or CMAP Transport Layer) to send messages or *signals* to the CMAP Session Managers, which then perform the requested network operations. CMAP is designed to support call management in a network supporting dynamic *n*-way multipoint-to-multipoint bidirectional connection-oriented ATM communications. CMAP clients are allowed to select connection types (virtual channel or virtual path), connection bandwidth, and the VPI/VCI pairs that they will use to transmit and receive. A complete CMAP implementation requires that all these facilities be available either from the network or from the other software components. The next three sections describe the support required from each component.

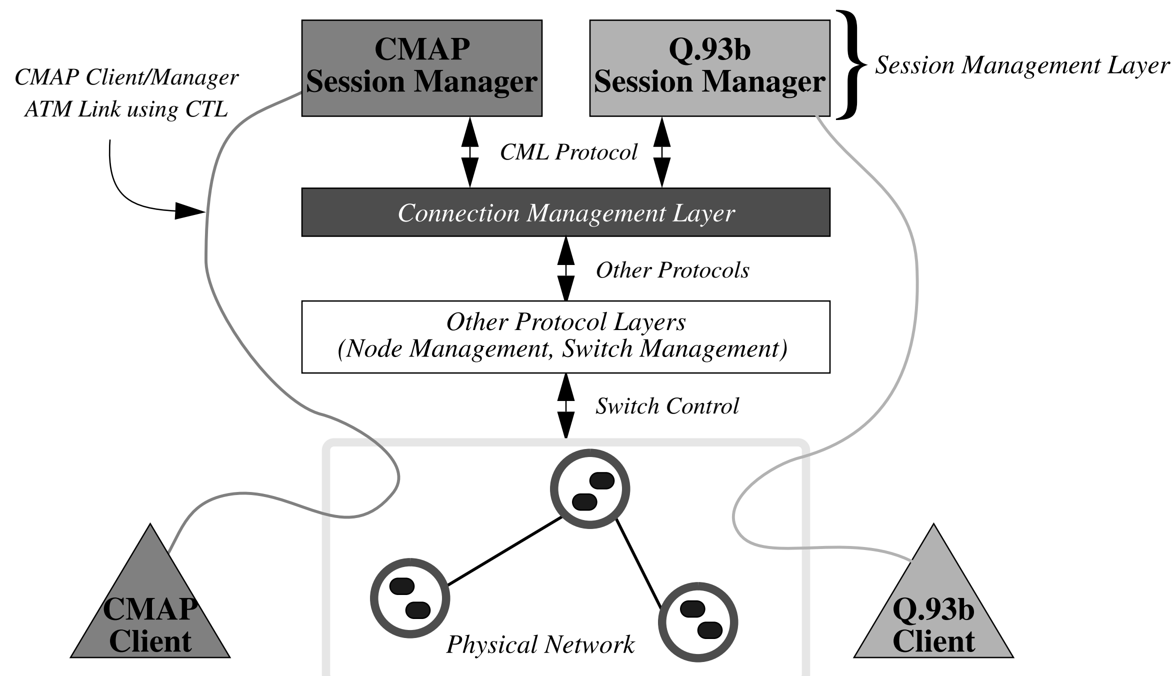


Figure 9. Network Configuration Used with CMAP

3.2 Network Functionality

Despite its central role, the physical network actually has to supply very little direct support for CMAP. This is primarily due to the fact that CMAP is situated at a high level on the protocol stack and is thoroughly insulated from the network by the other software components. The network facilities are primarily those needed to support the types of calls and connections that CMAP allows. In addition, many of the network requirements (*e.g.*, sequenced delivery of cells with no cell duplication or extraneous cells) are already required by the ATM standard and so need not be repeated here.

The network requirements are summarized in Table 2. The third column indicates whether the facility is required for a minimal CMAP implementation or optional. Most of the entries are self-explanatory, with the possible exception of the last four. By a static connection we mean one that is configured exactly once, when it is created, and cannot be changed thereafter. A dynamic connection can be changed while it is in use, for example to add or remove clients or to change bandwidth. An endpoint mapping (Section 4.5.3) determines whether a client receives and/or transmits data on a connection. A static mapping is one that is configured exactly once and not changed thereafter, while a dynamic mapping may change. As the table indicates, we require that the network allow clients to modify their mappings—for example, a client may indicate that it does not wish to receive data from a connection, and the network must stop delivering data to that client.

Table 2. Required and Optional Network Functionality

Network Functionality	Description	Required/ Optional
VP	Virtual Path connections	Optional
VC	Virtual Channel connections	Required
Point-to-point	Connections between exactly two clients	Required
Point-to-multipoint	Connections between two or more clients, only one transmitter	Optional
Multipoint-to-multipoint	Connections between two or more clients, any number of transmitters	Optional
Unidirectional	One-way data flow in connections	Required
Bidirectional	Two- or more-way data flow in connections	Optional
Static connections	Connection paths may not be reconfigured once set up	Required
Dynamic connections	Connection paths may be reconfigured once set up	Optional
Static mappings	Endpoint mappings may not be reconfigured once set up	Required
Dynamic mappings	Endpoint mappings may be reconfigured once set up	Optional

3.3 CTL Functionality

Table 3 lists the functions required of the CMAP Transport Layer (CTL), which is the protocol used to transmit CMAP signals over the ATM links between the CMAP clients and Session Managers. We do not specify a particular protocol but allow any ATM-compatible reliable transport protocol to be used. The table lists both required and optional CTL functions. The required functions are needed for correct message transmission and CMAP operation. The optional functions can be used to augment client interfaces and applications. We do not describe these augmentations in this document.

The first five functions deal with the mechanism whereby signals are broken down into ATM cells, transmitted, and reassembled into CMAP signals. As the table indicates, we require that this process guarantee that messages (CTL frames) be delivered to the remote peer without error. The remaining, optional functions may require further explanation. *Flow control* indicates that the CTL protocol will voluntarily refrain from sending messages when the network or host is congested, thus allowing time for the congestion to clear. *Internal ping* provides a mechanism to determine whether the remote peer is still alive. This would allow clients and CMAP Session Managers to periodically check the status of their peer and (if appropriate) attempt to restart the peer when it dies. *Auto synchronization* performs the internal ping automatically within the CTL layer, with notifications to the higher layers when synchronization is lost—again, the higher layer (CMAP client or Session Manager) might attempt a restart of its peer under these conditions.

Multiple connections on a single access link allow for multiple signalling connections to a single client and for multiple clients on an access link. The use of such connections is described in greater detail below (Section 3.4). *Query* allows for client agents and Session Managers to learn the capabilities of the network and adapt to them. For example, an application could query to learn whether the network supports multipoint connections and, if not, it could emulate multipoint connectivity using several point-to-point connections, transparently to the user.

Table 3. Required and Optional CTL Functionality

CTL Functionality	Description	Required/ Optional
SAR	Segmentation and Reassembly of CMAP frames to/from ATM cells	Required
Sequenced	Sequenced delivery of frames	Required
Lossless	No lost frames	Required
Duplicate suppression	No duplicate frames	Required
Error-free	Frames are delivered to higher level without bit errors	Required
Flow control	Throttling mechanism to handle network or host congestion	Optional
Internal ping	Internal CTL mechanism for learning if remote peer is still alive	Optional
Auto synchronization	CTL maintains synchronization and can report loss to CMAP layer	Optional
Multiple connections	Support for multiple CTL connections on a single access link	Optional
Query	Single ATM cell query to report SAR, transport protocol, *etc*.	Optional

3.4 Signalling Connections

At the lowest level, the ATM cells comprising CMAP messages are sent over a bidirectional signalling connection between the CMAP Client and the CMAP Session Manager (Figure 9), using the CTL protocol described above. The setup of these ATM signalling connections from clients to Session Managers, and the setup of any required ATM signalling paths between CMAP Session Managers, falls into the area of network management. We require that facilities for setting up such one connection between each client and its Session Manager be available. The exact mechanisms and protocols used to establish these connections are outside the scope of this document.

Clients may request the creation of additional signalling connections, but are not required to do so (in this way our signalling connection functions as an ATM standard *meta-signalling connection*). As Table 3 indicates, we do not require that the network or CTL support such additional signalling connections; a CMAP implementation must be able to function with a single signalling connection between each client and the network.

3.4.1 Multidrop Signalling

Our model allows for multiple clients on the same access link, as shown in Figure 10. This is one case where additional signalling connections may be desirable, so as to differentiate the multiple clients on the access link. To allow for the case where additional signalling connections cannot be allocated, CMAP messages contain the address of the client to which they are directed. This allows several clients to share a single signalling link, provided the CTL delivers a copy of the data to each of the clients. As noted above, we do not require support for either additional signalling connections nor multiple connections on a single link.

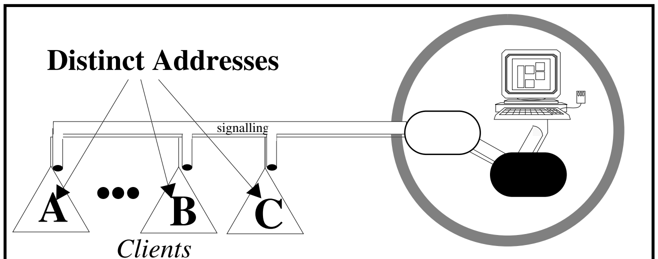


Figure 10. Multidrop Signalling

3.4.2 Surrogate Signalling

Another signalling capability that we allow for is *surrogate signalling*, where one client (called the *surrogate* client) is designated as the signalling entity for another client (called the *mute* client), as shown in Figure 11. The surrogate client originates all signalling messages for the mute client and receives all signalling messages from the network that would otherwise be sent to the mute client. To handle this, CMAP messages contain the address of the client at which the signal is directed. The *surrogate* client can be anywhere in the network and does not have to be local to the *mute* client's node. The configuration of mute and surrogate clients falls into the area of network management, but we do not require the network to provide these facilities.

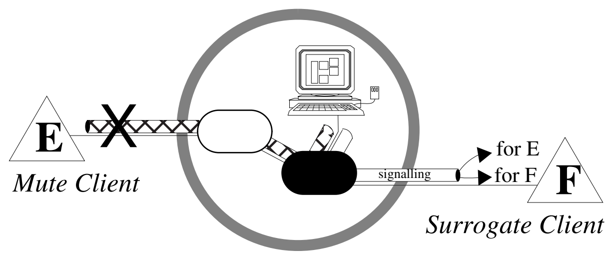


Figure 11. Surrogate Client Signalling at the UNI

An example of where surrogate signalling could be used is shown in Figure 12, where an array of high-resolution displays (used for digitized X-rays or other medical diagnostic displays) are controlled from a nearby workstation. The physician uses the workstation to select which images should be shown on each display and all of the signalling is performed by the workstation, even though each of the displays is a separate CMAP client.

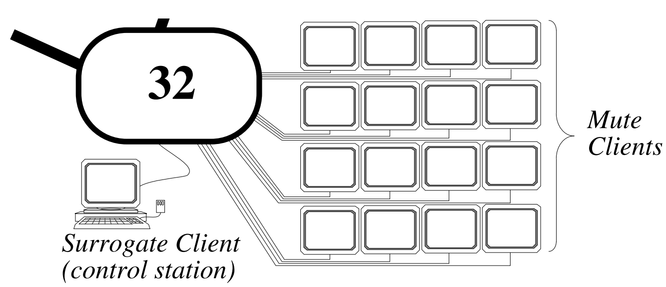


Figure 12. Surrogate Signalling for Medical Applications

3.5 Connection Management Layer Functionality

The Connection Management Layer provides the facilities whereby the CMAP Session Managers build, maintain, and destroy ATM connections, and its requirements thus lie in the area of providing the types of connections that the CMAP model supports. These requirements are summarized in Table 4. Some of these facilities (indicated by "Network" in the third column of the table) overlap with, and depend upon, the facilities made available by the physical network—obviously the CML cannot provide VP connections if the network does not.

The CML also includes a number of resource management facilities which may need explanation. *Prioritization* refers to the ability to assign calls and connections different levels of priority, with mechanisms whereby high-priority calls can take resources from lower-priority calls. *Quality of service* refers to the ability to support different levels of quality—high-quality connections might be able to guarantee no loss of ATM cells, minimal delivery times, and so forth. *Peak bandwidth reservation* refers to the bandwidth management mechanism whereby a client reserves the maximum connection bandwidth that it will ever require. *Bandwidth management* refers to more general schemes, for example involving statistical multiplexing based on average bandwidth and traffic burstiness. *Best-effort connections* are ones without reserved bandwidth. Cells on such connections will be discarded if any of the links are saturated. *Connection holding* refers to the ability to reserve connection resources (bandwidth, VPI/VCI pairs, *etc.*) without actually using the connection. One case where this might arise is when a client wishes to turn off cell reception for a period but still have the connection available; the client would then direct that the connection be held to turn off the cell flow, and when the client later directed that the cell flow be resumed the connection could be rapidly reestablished. Finally, *VPI/VCI pair allocation* refers to the ability of clients to choose which VPI/VCI pairs they will use for receiving and transmitting data. This might be particularly important where specialized hardware is used in data sources or sinks; the hardware might be able to use only a single VPI/VCI pair in the cells it sends or receives.

Table 4. Required and Optional CML Functionality

CML Functionality	Description	Required/ Optional
VP and VC	Virtual Path and Virtual Channel connections	Network
Connection types	Point-to-point, point-to-multipoint, and multipoint-to-multipoint	Network
Directionality	Unidirectional and bidirectional	Network
Connection configuration	Static and dynamic	Network
Endpoint configuration	Static and dynamic	Network
Prioritization	Ability to select among levels of priority	Optional
Quality of service	Ability to select among levels of quality	Optional
Peak bandwidth reservation	Reservation and allocation by peak bandwidth	Required
Bandwidth management	Other types of bandwidth reservation and allocation	Required
Best-effort connections	Connections without reserved bandwidth	Required
Connection holding	Ability to reserve connections without actually using them	Optional
VPI/VCI allocation	Ability to direct use of particular VPI/VCI pairs	Optional

The *Connection Management Network Protocol* (CMNP) is one possible connection management layer. CMNP provides a uniform, connection-oriented view of the ATM network and provides all the facitilities listed in Table 4 (subject, of course, to the network's capabilities). CMNP is described in a separate technical report [***REFERENCE***].

3.6 Minimal CMAP

An implementation of CMAP is distinguished from the general CMAP specification. The design of CMAP (described in Sections 4 through 6) is based on a general model of the ATM protocol and fast packet switching networks and of the supporting software such as the Connection Management Layer. A particular network environment may not be able to support all of the CMAP capabilities described. For this reason we allow an implementation of CMAP to vary from the generalized model. The network is still considered a CMAP network as long as it conforms to at least the minimum specification described here. By making CMAP as general as possible but allowing it to have implementation subsets, we provide a single call management protocol that can be layered over a diverse set of networks.

If all CMAP options are implemented and all abstractions of the call model are supported in a CMAP implementation, then we refer to this as *complete CMAP*. We also defined a *minimal CMAP* that specifies the minimum capabilities that the network must support in order to be considered a CMAP network. The preceding sections (particularly Tables 2, 3, and 4) describe the required support for a minimal CMAP. Table 5 summarizes these sections and describes minimal and complete CMAP. The first column lists each of the functions. The next column indicates whether the function is required or optional in CMAP. Required functions must be provided in both minimal and complete CMAP. Optional functions need not be provided in a minimal CMAP implementation but are provided by complete CMAP. The remaining two columns contrast the minimal and complete CMAP implementations—of course, for required functions the two implementations are identical. In some cases (*e.g.*, bidirectional connections) the function may not be required by the supporting layers but is still required in CMAP. In these cases CMAP is required to emulate the function, possibly using the method suggested in the table.

Table 5. Minimal CMAP Functionality

Function	Required/ Optional	Effect on Minimal CMAP	Effect on Complete CMAP
VP connections	Optional	Clients requesting a VP connection are told the connection could not be created	All VP's supported by the network are available to clients
VC connections	Required	All VC's supported by the network are available to clients	
Point-to-point	Required	Two-client point-to-point calls must be fully supported	
Point-to-multipoint	Optional	Clients requesting point-to-multipoint are told the connection could not be created	Fully supported
Multipoint-to-multipoint	Optional	Clients requesting multipoint-to-multipoint are told the connection could not be created[1]	Fully supported
Unidirectional	Required	Fully supported	
Bidirectional	Required	If the network does not support bidirectional connections, we require that CMAP emulate the connections by establishing two (or more) unidirectional connections	
Static connections	Required	Fully supported	
Dynamic connections	Optional	Clients attempting to modify a connection are informed that the operation could not be performed	Fully supported
Static mappings	Required	Fully supported	
Dynamic mappings	Required	If dynamic mappings are not supported by the network, the CMAP clients and session managers must set up all connections as receive/transmit. Requests to change mappings affect software data but not hardware. Cells received on a connection that is not set to receive should be silently discarded.	
Error-free CTL	Required	Used for message transmission as described in Section 3.3	
Flow control	Optional	Not used	Used, but invisible to client
Internal ping	Optional	Not used	May be used to restart clients and/or session managers
Auto synchronization	Optional	Not used	May be used to restart clients and/or session managers
Multiple connections	Optional	Not used	May be used to support multi-drop and surrogate signalling
Query	Optional	Not used	May be used to determine network capabilities and assist with emulation activities
Single signalling channel	Required	Configured by network management	
Additional signalling channels	Optional	No facility for allocating additional channels provided to clients	Clients may request any number of additional private signalling channels

Table 5. Minimal CMAP Functionality

Function	Required/ Optional	Effect on Minimal CMAP	Effect on Complete CMAP
Multidrop signalling	Optional	Multidrop configurations may not be used	Where clients share a single link, must ensure each gets a copy of all signals sent
Surrogate signalling	Optional	Surrogate signalling may not be used	Fully supported
Prioritization	Required	CMAP Session Managers keep track of call priorities and abort lower-level calls when their resources are needed by a higher-level call.	
Quality of Service	Optional	All CMAP QOS levels map to the same network QOS	Each CMAP QOS level maps to a different network QOS
Peak bandwidth reservation	Required	Fully supported; sum of peak bandwidths on any link does not exceed the link's capacity	
Bandwidth management	Optional	Session managers do not monitor traffic or attempt to do statistical multiplexing	Session managers may monitor traffic to collect statistics, set up hardware to enforce the connection BW behavior, and perform statistical multiplexing
Best-effort connections	Required	No bandwidth reserved for best-effort connections; cells transmitted on connections have CLP = 1, thus indicating that they should be discarded when congestion is encountered	
Connection holding	Required	Connection holding may be emulated by keeping the connection open and discarding cells (see dynamic mappings above)	
VPI/VCI allocation	Optional	Clients attempting to allocate a specific pair are informed that the pair is unavailable	Fully supported

[1] If the network supports point-to-multipoint but not multipoint-to-multipoint , we strongly encourage the emulation of multipoint-to-multipoint by the use of multiple point-to-multipoint connections, one for each transmitter.

4. Call Model

CMAP is a protocol whereby network *clients* request the creation, modification, and deletion of *calls*, which are distributed entities containing *connections* and *endpoints*. This section defines the attributes of CMAP clients, calls, connections, and endpoints. The attributes of CMAP entities and the operations that may be performed on them (described in Section 6) together define the CMAP *call model*. Section 4.1 first describes the basic concepts of CMAP. Sections 4.2 through 4.5 then describe the attributes of the various entities. Finally, Section 4.6 summarizes the identifiers that serve to uniquely identify each entity in the CMAP context.

4.1 Basic Concepts

A CMAP *client* is a user of the network, in the most general sense. Our model views all clients as equal. We draw no distinction, for example, between network gateways, multimedia workstations, or video phones. All clients are constrained to signal the network through our CMAP protocol and communicate through ATM cell pipes. We believe that our call model provides the necessary core functionality so that other services can be layered over CMAP, perhaps presenting a more sophisticated network model to users. Additionally, we feel that CMAP's interconnection services are suitable for both local and wide area networks.

Network clients signal other clients by issuing CMAP requests that create and manipulate *calls*. A call is a distributed object maintained by the network that describes the communication paths that interconnect clients. Two of the call's parameters are its *owner* and *root*. The owner is a CMAP client which is responsible for managing the call. It need not be a participant in the call (for example, it might be managing a mute video server). The root is another CMAP client which is participating in the call. The root provides a known point in the network toward which routing can be directed. In most cases, the root and the owner will be the same client.

A call has one or more communication channels, which we term *connections*. A connection is simply an ATM end-to-end cell pipe (Section 2.3). Participants in a call can add connections to the call, subject to approval by the owner. Multiple connections within a call are useful for applications such as video conferences, where one connection carries video and another audio. Figure 13 shows an example of such a multimedia call at a client, with separate connections within the call carrying video, voice, and data. The protocols used by clients to send data over connections are outside the scope of CMAP.

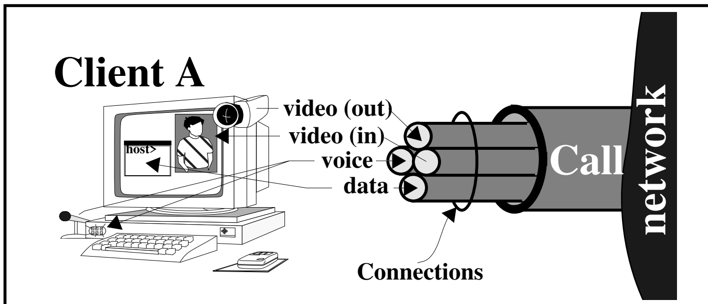


Figure 13. Example Multiple-Connection Multimedia Call

A call also has one or more *endpoints*, which are the interfaces between clients and calls. A client may join in a call several times, resulting in several distinct endpoints associated with that client. When a call is first created, an endpoint for the root and optionally one additional endpoint (for either the root or for another client) are added to the call. Additional endpoints may be added by: 1) invitation from the owner, where the invited party has the option of refusing the invitation, 2) request from a client not currently in the call to be added, where the owner has the option of denying the request, or 3) request from a third party, not necessarily in the call, to add a client, where both the owner and the client being added have the option to refuse.

Each endpoint has separate parameters for each connection in the call. We use the term *UNI* (user network interface) to refer to these per-connection endpoint properties. One of the most important of the UNI properties is the endpoint *mapping*, which indicates whether the endpoint receives, transmits, and/or echoes data on the connection. Figure 14 shows six common ways (of the twelve possible) that an endpoint may map a connection to itself at its access point (refer to Table 12 on page 153 for a complete list of possible mappings). During the lifetime of the call, the client

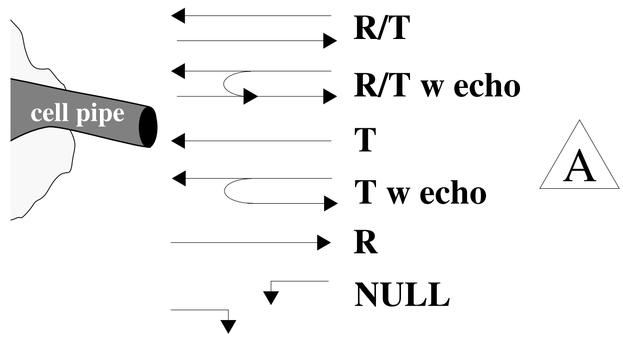


Figure 14. Sample Connection Mappings at the UNI

may dynamically change its mappings for each connection in the call. Such a feature might, for example, be useful in a video teleconferencing system in which only one person can transmit at a time; clients would normally be mapped to the video connection as receivers, but would change to transmit with echo when the client is the speaker.

4.2 Clients

A CMAP client is any user of the network. Clients have only one parameter, an address, which serves to uniquely identify the client for the entire network.

4.2.1 Client Address

Each client of the network is identified by a unique address. These addresses are used in CMAP operations to indicate the client to be added, modified or dropped from a call or connection, and to indicate the client performing the operation. Internally, the network may use the addresses for signal routing and connection setup.

The CMAP protocol does not specify a particular addressing scheme. Rather, CMAP implementations can choose to support one or more addressing schemes (for example, an implementation might support several well-known addressing schemes, such as IP or NSAP, and one or two experimental addressing schemes). The format of a CMAP client address is shown in Figure 15. The address size is a constant 24 bytes, regardless of the addressing scheme used. The first four bytes comprise a *type* field that tells how the remainder of the address should be interpreted. The remainder is partitioned into three additional fields whose sizes vary depending on the type of addressing used: a *network address field*, *local address field* and an *unused* field. The network address field contains the only portion of the address that the network uses in client identification and routing. The local address field is passed end-to-end in CMAP operations for use by the clients and is not interpreted by the network. The remainder of the address is unused.

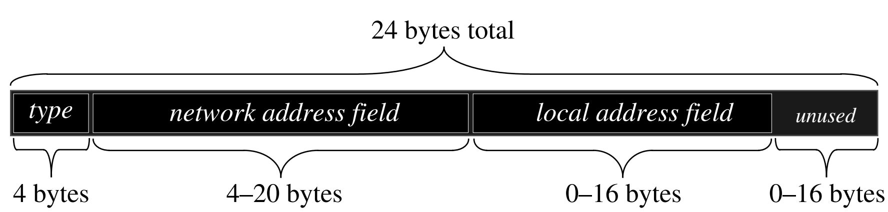


Figure 15. CMAP Client Address Format

Figure 16 shows several example partitioning schemes. The upper portion of the figure shows the partitioning used for IP addresses [18]. The network address field is 4 bytes and holds the IP address. The local address field is 2 bytes and contains the higher level port number (for example, the TCP or UDP port). The remaining 14 bytes of the address field are unused. The middle portion of the figure shows the partitioning for the CCITT E.164 addresses [13] used in the public telecommunications networks. The network address field is 8 bytes and encodes the 15 decimal digit E.164 address. The local address field is 4 bytes and holds the E.164 subaddress field. The remaining 8 bytes are unused. The bottom portion of the figure shows the partitioning for OSI NSAP (Network Service Access Point) addresses

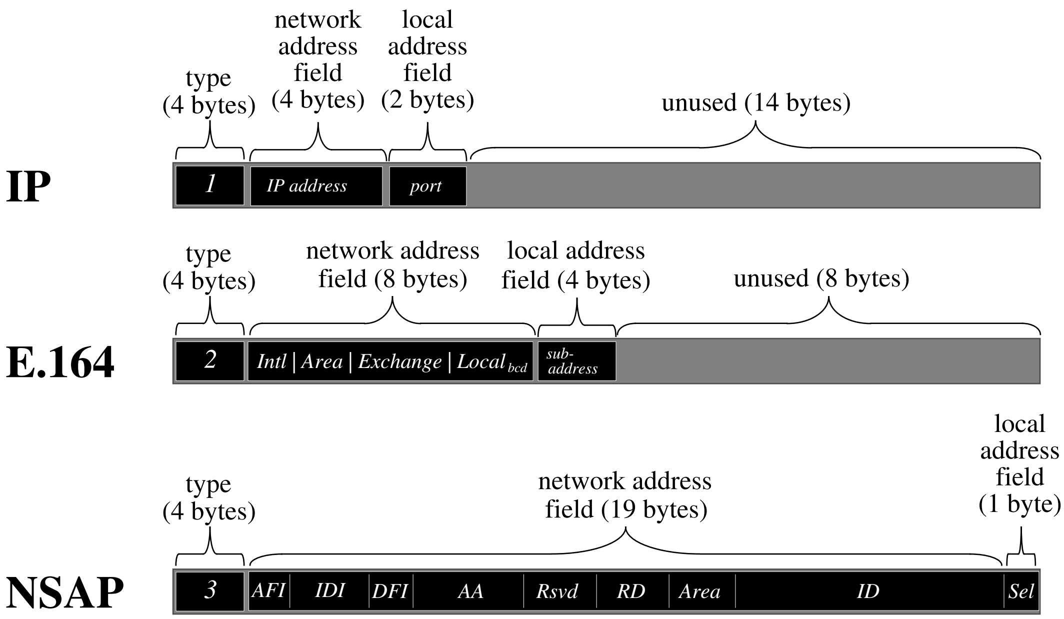


Figure 16. Examples of Client Address Partitioning

[17]. The network address field is 19 bytes and contains all fields of the NSAP address except for the SEL (selector) field, which is contained in the 1-byte local address field. Since NSAP addresses are 20 bytes in total length, there is no unused portion.

4.3 Calls

The call is the primary object manipulated by clients. When a client creates a call it becomes the owner of the call. When a client is added to a call it becomes an endpoint in the call and gains access to all the connections of the call. Calls have a number of parameters that describe the call and indicate how clients may access and modify the call. These parameters are described in the following subsections and summarized in Table 6.

Table 6. Call Parameters

Parameter	Description
Owner	The client that manages this call
Root	The client toward which routing is done
Local Identifier	Root-wide unique identifier for this call
Type	Multipoint or point-to-point
Accessibility	Ability of clients to join the call
Modifiability	Whether a nonowner client may add connections
Traceability	Whether a nonowner client may obtain parameters
Monitoring	Level of notification of client joins and drops
Priority	Call level priorities, for call preemption
User Type	User description of call
Connection List	Current connections in call
Endpoint List	Current endpoints in call

4.3.1 Call Owner

The owner is the address of the client which manages the call. Management operations include activities such as modifying call, connection, or endpoint parameters, closing the call, and approving the addition of endpoints. The owner is initially the client who creates the call.

4.3.2 Call Root

The root is the address of a client which is participating in the call. The root may be used by the network routing algorithms when adding new endpoints to the call. The root is initially chosen by the creator of the call.

4.3.3 Call Local Identifier

The local identifier is a small integer which is unique for all calls having the same root. Together, the root address and the local call identifier form the *call identifier*, a network-wide unique identifier for the call. The local identifier is initially chosen by the creator of the call (with the approval of the network, which ensures that it is not already in use) and subsequently changes only if the root of the call changes.

4.3.4 Call Type

The type designates whether the call is *multipoint* or restricted to *point-to-point*:

Type ∈ { **MULTIPOINT**, **POINT-TO-POINT** }

We use **MULTIPOINT** to refer to both point-to-multipoint and multipoint-to-multipoint calls. While the call model supports multicast calls as the general object, some calls are intrinsically point-to-point and, if designated as such, can be serviced more efficiently by the network (for example, by routing the call over internal trunks).

4.3.5 Call Accessibility

The accessibility parameter is a 2-tuple that controls whether clients are allowed to add endpoints:

Accessibility ∈ { **OPEN**, **VERIFY**, **CLOSED** }

If the call is **OPEN**, any client in the network may join the call without the owner's permission. A **CLOSED** call restricts the call such that only the owner may add clients. If the call has the **VERIFY** accessibility, then any client may attempt to join the call, but he operation will be verified with the owner before it is allowed to complete.

4.3.6 Call Modifiability

The modifiability parameter controls whether non-owners have the permission to add connections to the call

Modifiability ∈ { **ON**, **OFF** }

If this parameter is **ON**, any participant in the call may add connections. If it is **OFF**, only the owner may add connections. Non-participants are never allowed to add connections. One example of where this option is useful is in multiparty conference calls, where each endpoint adds a one-to-all connection for his feed when joining the call.

4.3.7 Call Traceability

The traceability parameter is a 2-tuple that controls whether clients have the permission to obtain information about the call):

Traceability ∈ { **OPEN**, **MEMBERS**, **CLOSED** }

A value of **OPEN** indicates any client may perform traces. A value of **MEMBERS** indicates that only clients who are participating in the call may perform traces. **CLOSED** indicates only the owner may perform a trace. The owner of the call is always allowed to perform traces.

4.3.8 Call Monitoring

The monitoring parameter is a 3-tuple designating whether endpoints of the call are notified when endpoints join or drop out of the call or modify their parameters:

Monitor = < **owner** ∈ { **ON**, **OFF** },
transmitters ∈ { **ON**, **OFF** },
all ∈ { **ON**, **OFF** } >

If the *owner* field is set to **ON**, all endpoint joins and drops are reported to the owner. Likewise, if the *transmitters* field is set to **ON**, all transmitters are notified when these changes occur, and if the *all* field is set to **ON**, all participants in the call are notified. When used with an accessibility of **OPEN**, an owner modifiability setting of **ON** allows the owner to keep track of endpoint joins and drops without being burdened with explicitly verifying every such change.

4.3.9 Call Priority

The priority parameter is a 1-tuple that allows some calls to take precedence over others:

$$\textbf{Priority} \in \{\ \textbf{NORMAL}, \textbf{PREEMPT}, \textbf{OVERRIDE}\ \}$$

NORMAL is the lowest priority, followed by **PREEMPT**, followed by **OVERRIDE**. If the network does not have the resources to support a call and the required resources are currently allocated to lower-priority calls, those resources are reclaimed (aborting the lower priority calls) so that the higher priority call can be supported. The priority affects how the CMAP Session Managers process the call and whether the call should be blocked if resources are unavailable. ATM cell-level priorities are implemented separately as parameters of connections (Sections 4.4.2 and 4.4.3).

4.3.10 Call User Type

The user type is a value chosen by the call creator. This parameter is not manipulated by the network but is simply passed end-to-end so that clients may examine it.

4.3.11 Call's Connection List

The connection list describes the connections of the call. The parameters of connections are described below. The list is dynamic since connections can be added to, modified, or removed from the call at any time.

4.3.12 Call's Endpoint List

The endpoint list describes the endpoints participating in the call. The parameters of endpoints are described below. The list is dynamic since endpoints can be added to, modified, or removed from the call at any time.

4.4 Connections

The connection is the primary information-carrying component of a call. The parameters associated with connections are described in the following subsections and summarized in Table 7.

Table 7. Connection Parameters

Parameter	Description
Identifier	Call-wide unique identifier for this connection
Type	Three tuple of VP/VC, dynamic/static bandwidth and QOS
Bandwidth	Reserved bandwidth for the connection
Defaults	Defaults to be offered to endpoints joining the connection
Permissions	Permissions to be offered to endpoints joining the connection
User Type	User description of connection

4.4.1 Connection Identifier

The identifier is a small integer which uniquely identifies the connection within the call. This value may be determined by the client which adds the connection to the call, with the network ensuring that it is unique, or the client may leave the field blank and permit the network to select a unique identifier.

4.4.2 Connection Type

The type is a 3-tuple:

$$\begin{aligned}\textbf{Type} = \langle\ & \textbf{channel\_type} \in \{\ \textbf{VP}, \textbf{VC}\ \},\\ & \textbf{BW\_type} \in \{\ \textbf{STATIC}, \textbf{DYNAMIC}\ \},\\ & \textbf{QOS} \in \{\ \textbf{HIGH}, \textbf{MEDIUM}, \textbf{LOW}\ \}\ \rangle\end{aligned}$$

The first component specifies the type of channel that the connection requires and must be one of two values: *virtual path* (**VP**) or *virtual channel* (**VC**) (Section 2.3). The second component specifies whether the connection is **DYNAMIC** or **STATIC**. If a connection is static then the bandwidth is fixed throughout the life of the call, which makes the connection more predictable and allows for more efficient use of network resources (such as routing over internal trunks). Dynamic connections can have their bandwidth modified during the life of the call. The last component specifies the desired *quality-of-service* (QOS) and may have the values of **HIGH**, **MEDIUM** or **LOW**. QOS relates to options for cell loss behavior that may vary from network to network. QOS, therefore, is left intentionally vague. If the network can implement different cell loss behavior strategies then the network control software will group these into the categories of **HIGH**, **MEDIUM** and **LOW**.

4.4.3 Connection Bandwidth

We support two types of bandwidth specification for connections: 1) *reserved*, where the network guarantees and enforces the requested bandwidth, and 2) *best-effort*, where the network neither guarantees nor enforces. Reserved bandwidth connections are treated preferentially by the network, whereas best-effort connections compete with one another for the remaining bandwidth not currently used by reserved connections.

The connection's bandwidth is specified as a 3-tuple:

Type = < peak, average, peak_burst_length >

The *peak* and *average* parameters are expressed in cells per second. The *peak burst length* is measured in cells and indicates the maximum number of cells that the endpoint can send during a burst at peak rate. Best-effort connections are indicated by all zeros in the bandwidth specification.

For reserved bandwidth connections, the figures given are for aggregate bandwidth, as seen at any receiver in the connection (this takes into account the increased loading that results when the transmissions from multiple transmitters combine and flow over links traversed by the connection). Figure 17 illustrates how we use the bandwidth specification to characterize the endpoint's transmit behavior on a connection at the UNI. The *peak* value tells us a percentage of the bandwidth used by the endpoint when transmitting a burst of data. The user may not arbitrarily send cells back-to-back for the duration of the *peak burst length*, but must pace them according to the *peak* value as shown. The *peak burst length* tells the network how long the connection will remain transmitting at *peak* rates. The example in the figure shows a *peak* value of 87,000 cells per second (in this case, roughly 25% of a 155Mbps link) and a *peak burst length* of 1000 cells. The *average* value is used over a longer period of time (for example, several seconds) to enforce the long term average utilization of the connection.

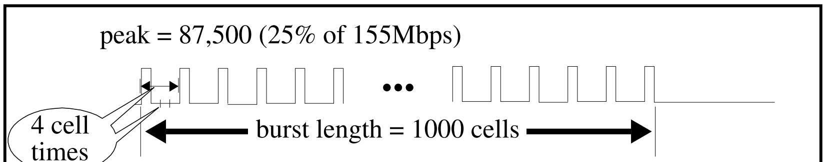


Figure 17. Cell Pacing and Peak and Burst Length at the UNI

For best-effort connections, the endpoint is not restricted in any way. Instead, all cells are forcibly marked at the UNI by setting the header's CLP bit. Inside the network, as buffers fill, the network discards marked cells if there is no room in the buffers for unmarked (reserved) cells.

We chose to provide both reserved and best-effort bandwidth connections for several reasons. Some applications clearly need bandwidth guarantees, such as live video, where cell loss can result in incomprehensible transmissions. However, many other applications do not need these guarantees. For example, data transfer applications can usually tolerate lossy networks and network congestion using schemes such as retransmission and backoff, as in TCP/IP [18]. Additionally, many applications cannot easily predict their bandwidth requirements, and if forced to do so would produce specifications that are too low or too high, each with negative consequences. Finally, when all applications are forced to use reserved, guaranteed connections, network utilization is sometimes low (depending on the connection mix) since the level of allowed multiplexing must be limited to provide the guarantees. By introducing the best-effort bandwidth class, we accommodate those connections that do not need the guarantees and cannot easily estimate requirements, and we provide a means whereby the excess capacity of the network can be utilized.

4.4.4 Connection Defaults

The defaults is the value of the endpoint defaults field (and also of the initial endpoint mapping) offered to endpoints joining the call. The defaults may be overridden by the call owner so that individual endpoints can be assigned different defaults. The defaults is a 3-tuple:

$$\textbf{Defaults} = < \textbf{receive} \in \{ \textbf{ON}, \textbf{HOLD}, \textbf{OFF} \},\ \textbf{transmit} \in \{ \textbf{ON}, \textbf{HOLD}, \textbf{OFF} \},\ \textbf{echo} \in \{ \textbf{ON}, \textbf{OFF} \} >$$

Description of these fields is deferred until the discussion of the endpoint mapping (Section 4.5.3).

4.4.5 Connection Permissions

The permissions is the permissions given to endpoints joining the call. The permission may be overridden by the call owner so that individual endpoints can be assigned different permissions. The permissions is a 3-tuple:

$$\textbf{Permissions} = < \textbf{receive} \in \{ \textbf{ON}, \textbf{VERIFY}, \textbf{OFF} \},\ \textbf{transmit} \in \{ \textbf{ON}, \textbf{VERIFY}, \textbf{OFF} \},\ \textbf{echo} \in \{ \textbf{ON}, \textbf{OFF} \} >$$

Description of these fields is deferred until the discussion of the endpoint permissions (Section 4.5.3).

4.4.6 Connection User Type

The user type is a value chosen by the connection creator. This parameter is not manipulated by the network but is simply passed end-to-end so that clients may examine it.

4.5 Endpoints

The *endpoint* is the interface of a client with a call. The parameters associated with endpoints are described in the following subsections and summarized in Table 8. Except for the endpoint's address and local identifier, all of these parameters are UNI parameters assigned on a per-connection basis — that is, the endpoint has a separate mapping for each connection, a separate defaults, and so forth.

Table 8. Endpoint Parameters

Parameter	Description
Address	Client with which this endpoint is associated
Local Identifier	Client-wide unique identifier for this endpoint
Mapping	Current access of endpoint to connection
Defaults	Default value of the mapping
Permissions	Whether the endpoint may modify its mapping
Transmit Pair	VPI/VCI pair for ATM transmit
Receive Pair	VPI/VCI pair for ATM receive

4.5.1 Endpoint Address

The address is the address of the client with which the endpoint is associated.

4.5.2 Endpoint Local Identifier

The local identifier is a small integer which is unique for the endpoint with respect to the client. Together, the endpoint address and local identifier form the *endpoint identifier* which uniquely identifies the endpoint within the client. The local identifier may be determined by the client which adds the endpoint to the call, with the network ensuring that it is unique, or the client may leave the field blank and permit the network to select a unique identifier.

4.5.3 Endpoint Mapping

Endpoints have separate receive, transmit, and echo mappings for each connection in the call. The mapping is a dynamic field which may be changed by the call owner or by the endpoint itself. Initially the mapping is set to the endpoint *defaults*. For each connection in a call, the associated endpoint mapping is a 3-tuple:

$$\textbf{Mapping} = < \textbf{receive} \in \{\ \textbf{ON}, \textbf{HOLD}, \textbf{OFF}\ \}, \textbf{transmit} \in \{\ \textbf{ON}, \textbf{HOLD}, \textbf{OFF}\ \}, \textbf{echo} \in \{\ \textbf{ON}, \textbf{OFF}\ \} >$$

For the *receive* mapping, a value of **ON** indicates that the endpoint is receiving on the connection, a value of **OFF** indicates that the endpoint is not receiving on the connection, and a value of **HOLD** indicates that the endpoint is not currently receiving but that bandwidth should be reserved within the network up to the endpoint's access node (for use when the endpoint's receive permission is changed to **ON** at some later time). The *transmit* mapping is defined analogously to the receive mapping.

The *echo* mapping is used in conjunction with transmit to allow endpoints to view their own transmissions. When set to **ON**, the endpoint's transmissions will be echoed, and when set to **OFF** they will not be echoed.

4.5.4 Endpoint Defaults

The defaults indicates the default value of the endpoint's mapping, as distinct from the endpoint mapping itself which is the current value of the mapping. The defaults is a 3-tuple whose components and values are the same as those listed above for the mapping. Again, the endpoint has a separate default for each connection in the call. When an endpoint is added its defaults are taken from the connection defaults (Section 4.4.4), or the call owner may override the connection defaults and assign any desired endpoint defaults.

4.5.5 Endpoint Permissions

The permissions indicates the manner in which individual endpoints may change their mappings. Endpoints have separate permissions for each connection in the call and, as with mappings, the permissions for each connection may be completely unrelated to those of the other connections. For each connection in a call, the associated endpoint permission is a 3-tuple:

$$\textbf{Permissions} = < \textbf{receive} \in \{\ \textbf{ON}, \textbf{VERIFY}, \textbf{OFF}\ \}, \textbf{transmit} \in \{\ \textbf{ON}, \textbf{VERIFY}, \textbf{OFF}\ \}, \textbf{echo} \in \{\ \textbf{ON}, \textbf{OFF}\ \} >$$

Each field affects the corresponding field of the mapping. If the permission field is **OFF**, the endpoint is not permitted to change its endpoint mapping and it must equal the endpoint's default mapping for the connection (with one exception: if the default field is **ON** the endpoint may also use a mapping of **HOLD**). If the permission field is **ON**, the endpoint may choose any mapping. In this case, the default mapping is not enforced and is merely viewed as a suggestion to the endpoint. If the permission is **VERIFY**, the endpoint can alter his mapping, but the owner will first be queried to see if the new mapping is acceptable (with one exception, related to the previous: if the default field is **ON** the endpoint may change its mapping from **ON** to **HOLD** or vice-versa without verification). When an endpoint is added its permissions are taken from the connection permissions (Section 4.4.5), or the call owner may override the connection permissions and assign any desired endpoint permissions.

4.5.6 Endpoint Transmit Pair

The transmit pair is the ATM VPI/VCI pair which the endpoint uses to transmit data. When the endpoint is added, it may suggest a pair which (if it is available) will be used by the network. The pair may also be left blank, in which case the network will choose an available pair.

4.5.7 Endpoint Receive Pair

The receive pair is the ATM VPI/VCI pair which the endpoint uses to receive data. When the endpoint is added, it may suggest a pair which (if it is available) will be used by the network. The pair may also be left blank, in which case the network will choose an available pair.

4.6 Summary of Identifiers

Clients, calls, connections and the endpoints participating in calls all require some form of identification at the CMAP level so that these entities can be referenced by clients.

Client addresses are unique labels assigned to clients so that clients can identify one another. Client addresses are also used by the network to locate clients and route messages internally.

Call identifiers are unique labels assigned to calls so that each call can be referenced by clients (for example, for joining and modifying calls). They are a combination of the call's *root address* and a *local identifier* (a small integer). Clients can assign well-known call identifiers to calls that are to be globally known, for example, for broadcast video distributions.

Connection identifiers are unique labels for the connections within a call so that clients can individually reference the connections (for example, for changing ones receive/transmit mapping on a given connection). They are small integers. Clients can assign well-known connection identifiers to connections within a call, allowing the clients to easily distinguish among them.

Endpoint identifiers are labels assigned to each interface between a client and a call. They are a combination of the client's *address* and a *local identifier* (a small integer). A client is allowed to join a call more than once and the local endpoint identifier is used to distinguish the separate instances of client participation.

5. CMAP Messages

This section describes the use, format and transmission of CMAP messages. Section 5.1 describes the way in which CMAP messages implement CMAP operations. (Section 5.2) outlines the notation conventions used to diagram CMAP operations. Sections 5.3 through 5.9 describe the parameters passed in CMAP operations and the grouping of these parameters into CMAP *message objects*. Finally, Section 5.10 describes CMAP message transmission.

5.1 Use of Messages

A CMAP client and the network (or, more technically, a CMAP Session Manager operating on the network) communicate by exchanging messages which describe the operations that are to be performed. A CMAP operation involves up to three separate messages or *phases*. The phase of a message is identified by a field in the message header as described below (Section 5.3). The first CMAP phase is the **REQUEST** message, which asks that the operation be performed. This **REQUEST** may be transmitted from the client to the network or from the network to the client. The second phase is a **RESPONSE** message transmitted by the entity which received the **REQUEST**. The third phase is a **CONFIRMATION** message transmitted by the entity which received the **RESPONSE**. Depending on the operation type and the contents of the message, a CMAP operation may involve one, two, or all three phases. **REQUEST**, **RESPONSE**, and **CONFIRMATION** are sometimes abbreviated to **REQ**, **RES**, and **CONF**.

Messages have a status field which is used to indicate the success or failure of an operation. We use the terms **ACK** (acknowledgement), **NACK** (negative acknowledgement), and **NEG** (negotiation) to characterize **RESPONSE** messages based on the message status field. An **ACK** is a positive response. A **NACK** is a negative response. A **NEG** or negotiation response requests further interaction, *i.e.*, the sending of a **CONFIRMATION** message. Similarly, the terms **COM** (commit) and **ABORT** (abort) are used to characterize **CONFIRMATION** messages, with **COM** being a positive **CONFIRMATION** and **ABORT** being a negative **CONFIRMATION**. These terms are discussed in greater detail in Section 5.3.8.

The messages for each phase of a particular operation share a single message identifier. This allows any number of operations to be performed in parallel, with the clients and network using the message identifiers to correctly match the messages involved. Each currently-active operation has a message identifier which is unique to the entity (network or specific client) which initiated the operation. Clients and the network are free to reuse message identifiers once an operation is complete.

5.2 Notational Conventions

Figure 18 shows a diagram of the **open_call REQUEST**. The example demonstrates how CMAP messages are presented in this document and the notational conventions used. CMAP message diagrams are laid out in eight-byte wide columns. Each field is represented by a space proportional to the size of the field. The name of the field appears in the space and the size of the field in bytes is in parentheses next to the name.

Related parameters within CMAP requests are grouped into *message objects*. The **open_call REQUEST** shown in Figure 18 contains six of the different types of message objects (some of which appear multiple times). The six objects are: 1) the Header Object, 2) the Call Object, 3) the Connection Object, 4) the Endpoint Object, 5) the UNI Object, and 6) the Trailer Object. The seventh type of message object, not used in the **open_call**, is the Operation Object. These objects and their fields are described in detail in Sections 5.3 through 5.9.

The ordering of both message objects and values not contained in message objects is determined by the message's operation type and phase (the first two components of the Header Object). For this reason, in all diagrams showing message formats these two fields (***op_type*** and ***phase***) are filled in with the actual binary bit pattern appropriate to the message, as in this example where ***op_type*** = **00000001** (**open_call**) and ***phase*** = **00000000** (**REQUEST**).

The structure of the message may further vary due to the presence of *repeated objects*. These are sections of the message consisting of one or more occurrences of a particular type of message object. The number of occurrences is determined by a parameter found elsewhere in the message. In Figure 18 and throughout this document such repeated groups are delineated by a thin black line — and terminated by a three-dot separator, with the number of repetitions written in the separator. In the **open_call** example, the Connection Object is repeated ***num_cons*** times, the UNI Object is repeated ***num_cons*** times, and the group consisting of the Endpoint Object and the repeated UNI Objects is itself repeated ***num_ep*** + 1 times.

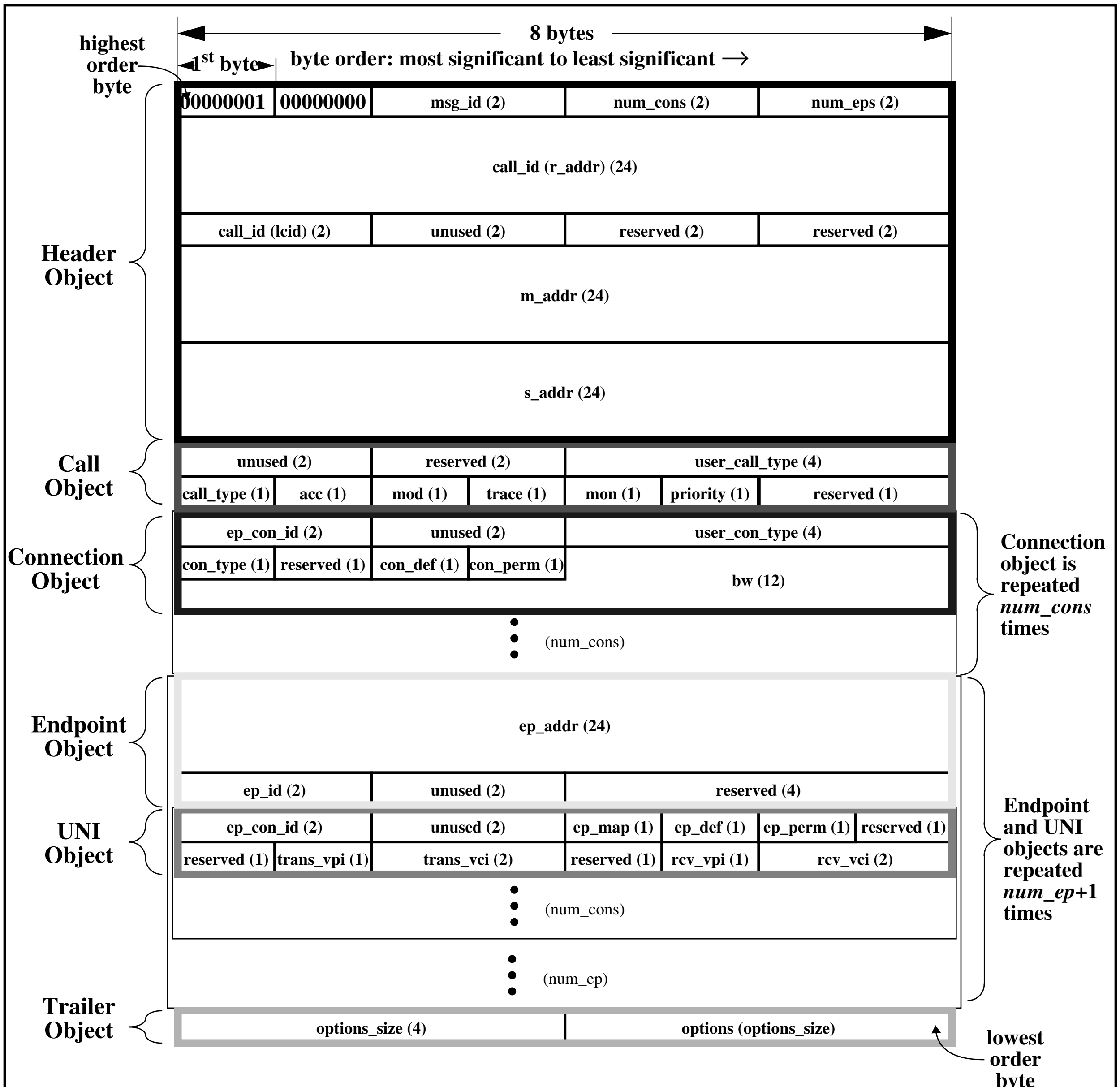


Figure 18. Common Message Format Characteristics (example open_call REQUEST)

Some fields are specified as ***unused*** during one or more phases of a CMAP operation. This means that the field has no context to the phase but may have context to a different phase of that operation or to a different operation that utilizes the information in the field. For example, the Connection Object has a connection status (***con_status***) field that the network uses to inform a client about the status of the connection. The ***con_status*** field is used, therefore, in a CMAP command response phase but is unused in the command's request phase. An ***unused*** field should not be confused with a ***reserved*** field. Fields marked ***reserved*** have no current definition or use, whereas an unused field has a definition but is not used in the operation or phase under discussion.

The following additional rules are used in structuring a CMAP message:

- The Header Object is always the first object, and the Trailer Object is always the last object. The ***op_type*** field of the Header Object is thus the first field in every CMAP message.
- Every CMAP field is either 1 byte in length or an even number of bytes.

- A 1-byte field may start on any byte boundary (byte boundaries are numbered from the beginning of the message, with the byte boundary before the ***op_type*** field of the Header Object being numbered 0).
- A 2-byte field must start on a 2-byte byte boundary (one whose number is a multiple of two).
- A field of more than 2 bytes must start on a 4-byte boundary (one whose number is a multiple of four).
- Multiple-byte fields are ordered in the CMAP message from the highest-order byte to the lowest-order byte.

Figure 19 illustrates some of these structuring rules.

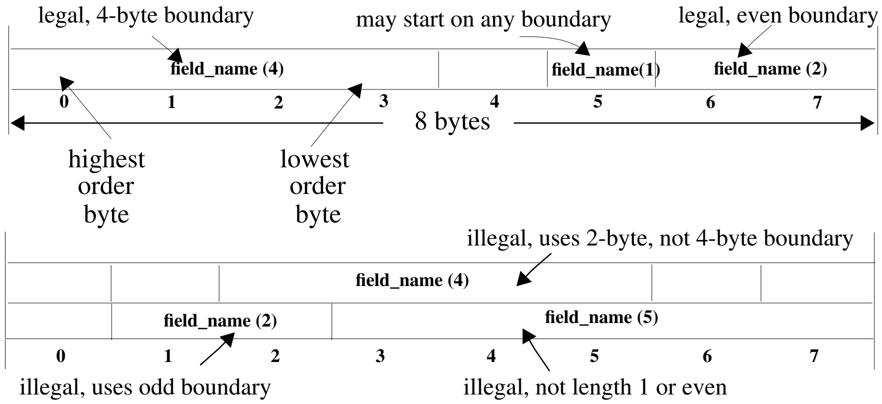


Figure 19. CMAP Message Structuring Rules

5.3 Header Object

The CMAP Header Object (Figure 20) is present in all messages and is always the first object in the message. The fields of the Header Object include parameters which are used by all (or nearly all) CMAP operations. The Header Object is distinguished in CMAP messages by a heavy line with a ▬ pen pattern.

op_type (1)	**phase (1)**	**msg_id (2)**	**num_cons (2)**	**num_eps (2)**
call_id (r_addr) (24)				
call_id (lcid) (2)		**op_status (2)**	**reserved (2)**	**reserved (2)**
m_addr (24)				
s_addr (24)				

Figure 20. The CMAP Header Object

5.3.1 op_type

The ***op_type*** field contains a symbolic constant indicating the operation type, *i.e.*, what CMAP operation is associated with the message. See Appendix C for definitions of the values that may appear in this field. In all message templates, this field is filled in with the bit-pattern appropriate to the operation type.

5.3.2 phase

The ***phase*** field contains a symbolic constant indicating the operation phase. See Appendix C for definitions of the values that may appear in this field. In all message templates, this field is filled in with the bit-pattern appropriate to the operation type. Together, the ***op_type*** and ***phase*** determine which template is used to interpret the message.

5.3.3 msg_id

The ***msg_id*** serves to associate different phases of the same operation. The ***msg_id*** is generated by the entity that initiates an operation and must be unique to that entity at the time of creation. It serves to associate the messages corresponding to the different phases of the operation., thereby allowing parallel CMAP operations to be sent to the network (rather that having to serialize the requests). A ***msg_id*** generated by a client must have the highest order bit set to 0. A ***msg_id*** generated by the network must have the highest order bit set to 1 (Figure 21).

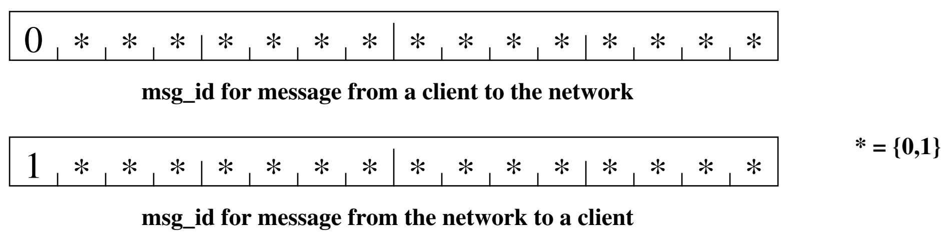


Figure 21. msg_id format

5.3.4 num_cons

The ***num_cons*** field is a two-byte unsigned integer which indicates the number of Connection Objects and/or the number of UNI Objects in the message. Recall from Section 4.5 that, except for the identifier, all information associated with an endpoint is on a per-connection basis. Data about endpoints is thus conveyed by giving an Endpoint Object followed by ***num_cons*** UNI Objects, one for each connection that is affected by the operation.

5.3.5 num_eps

The ***num_eps*** field is a two-byte unsigned integer which indicates the number of Endpoint Objects in the message.

5.3.6 call_id (r_addr)

The ***call_id (r_addr)*** is the twenty-four-byte address of the call root (Section 4.3.2). Together, the ***r_addr*** and ***lcid*** uniquely identify the call on which the operation is to be performed. Most operations manipulate a specific call instantiation, although the field is unused in some operations.

5.3.7 call_id (lcid)

The ***call_id (lcid)*** is the two-byte local call identifier portion of the call identifier (Section 4.3.3). The ***lcid*** may be generated by the client creating the call, with the network verifying that it is unique to the root address. Alternatively, the client creating the call may leave the ***lcid*** field blank and the network will generate a unique identifier and return it in the **open_call RESPONSE**. The value 0xFFFF (hexadecimal 0xFFFF, or all 1 bits) is blank.

5.3.8 op_status

The ***op_status*** field is used by the network in **RESPONSEs** to indicate whether the operation was successful and, if not, to indicate what error occurred. The field is divided into five subfields (Figure 22). The high-order four bits are used to indicate whether there were errors in the call, connection, endpoint, or UNI specifications. If any of these bits are set, the client should check the corresponding message objects to determine the specific error that occurred. The low order twelve bits (***status*** subfield) are used to indicate specific errors in the execution of the command as a whole. The values that may appear in the ***status*** subfield are listed and explained in Appendix D.

An **ACK** (acknowledgement) is a positive **RESPONSE**, indicated by an ***op_status*** field in which the four high-order bits (call, connection, endpoint address, and endpoint error bits) are all 0 and the remaining 12 low-order bits (the ***status*** subfield) are equal to the predefined value **OK**. A **NACK** (negative acknowledgement) is a negative **RE-**

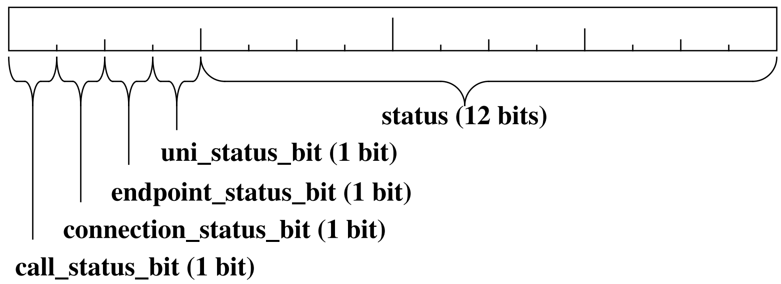


Figure 22. op_status format (16 bits)

SPONSE, one with at least one of the four high-order error bits of ***op_status*** set or with a ***status*** field not equal to **OK** or **NEGOTIATING**. A **NEG** (negotiation) is a **RESPONSE** in which the four high-order bits of the ***op_status*** field are equal to 0 and the remaining 12 low-order bits are equal to the predefined value **NEGOTIATING**. A **COM** (commit) is a positive **CONFIRMATION**, indicated by an ***op_status*** field in which the four high-order bits (call, connection, endpoint address, and endpoint error bits) are all 0 and the remaining 12 low-order bits (the ***status*** subfield) are equal to the predefined value **OK**. An **ABORT** (negative confirmation) is a negative **CONFIRMATION**, one with at least one of the four high-order error bits of ***op_status*** set or with a ***status*** field not equal to **OK**.

5.3.9 m_addr

In messages from clients to the network, ***m_addr*** is the twenty-four-byte address of the client that produced the CMAP message. In messages from the network to clients, ***m_addr*** is the twenty-four-byte address of the client that should receive the CMAP message. This field supports multidrop signalling arrangements where several clients must use the same VPI/VCI pair for network signalling operations.

5.3.10 s_addr

In messages from clients to the network, ***s_addr*** is the twenty-four-byte address of the client that produced the CMAP signal. In messages from the network to clients, ***s_addr*** is the twenty-four-byte address of the client that should receive the CMAP signal. This field supports surrogate signalling operations, where the client that produces (or receives) a message is not necessarily the one that should be considered to be signalling (or being signalled by) the network.

5.4 Trailer Object

The CMAP Trailer Object (Figure 23) is present in all CMAP messages and is the last object in the message. The Trailer Object is provided to allow implementors to establish extensions to CMAP that may be put into a standard CMAP message. The Trailer Object is distinguished in CMAP messages by a heavy line with a pen pattern.

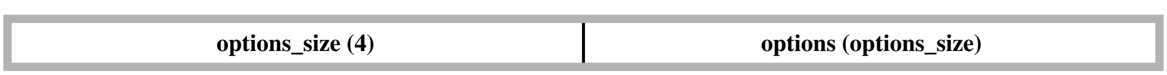


Figure 23. The CMAP Trailer Object

5.4.1 options_size

The ***options_size*** field is a four-byte unsigned integer indicating the length of the ***options*** field in the trailer object. If ***options_size*** is 0, there is no ***options*** field and the least significant byte of the ***options_size*** field is the least significant byte of the whole CMAP message.

5.4.2 options

The ***options*** field is a sequence of ***options_size*** bytes, with the most significant byte being the first and the least significant being the last (and also the last byte of the CMAP message, if the field is present).

5.5 Call Object

The CMAP Call Object (Figure 24) groups most of the parameters of a call (Section 4.3) into one object. (The owner, root, and identifier parameters are omitted because they either appear in the Header Object or are not applicable to all commands.) The Call Object is distinguished in CMAP messages by a heavy line with a pen pattern.

<table>
<tr><th colspan="2">call_status (2)</th><th colspan="2">reserved (2)</th><th colspan="4">user_call_type (4)</th></tr>
<tr><td>call_type (1)</td><td>acc (1)</td><td>mod (1)</td><td>trace (1)</td><td>mon (1)</td><td>priority (1)</td><td colspan="2">reserved (2)</td></tr>
</table>

Figure 24. The CMAP Call Object

5.5.1 call_status

The ***call_status*** field contains a symbolic value indicating whether any errors occurred in the specification of the call. See Appendix D for an explanation of the values that can appear in this field.

5.5.2 user_call_type

The ***user_call_type*** field is a four-byte unsigned integer representing the call's *user type* (Section 4.3.10). It is not examined by the network but is delivered end-to-end without modification. It may be used to communicate to potential endpoints some additional, application-level information about the type of the call. For instance, the owner of a data call might want to indicate the intent of the data transfer (*e.g.*, **FILE_TRANSFER**, **ELECTRONIC_MAIL**, or **IP_DATAGRAM**).

5.5.3 call_type

The ***call_type*** field contains a symbolic constant representing the call's *type* (Section 4.3.4). See Appendix C for definitions of the values that may appear in this field.

5.5.4 acc

The ***acc*** field contains a symbolic constant representing the call's *accessibility* (Section 4.3.5). See Appendix C for definitions of the values that may appear in this field.

5.5.5 mod

The ***mod*** field contains a symbolic constant representing the call's *modifiability* (Section 4.3.6). See Appendix C for definitions of the values that may appear in this field.

5.5.6 trace

The ***trace*** field contains a symbolic constant representing the call's *traceability* (Section 4.3.7). See Appendix C for definitions of the values that may appear in the sub-fields.

5.5.7 mon

The ***mon*** field represents the call's *monitoring* parameter (Section 4.3.8). The low-order three bits of this field indicate whether the owner, transmitters, or all endpoints receive notification of endpoint changes (Figure 25). See Appendix C for definitions of the values that may appear in the sub-fields.

*	*	*	*	*	O	T	A

O = owner
T = transmitter
A = all

*** = unused (0)**

Figure 25. mon format (8 bits)

5.5.8 priority

The ***priority*** field contains a symbolic constant representing the call's *priority* (Section 4.3.9). See Appendix C for definitions of the values that may appear in this field.

5.6 Connection Object

The Connection Object (Figure 26) groups the parameters of a single connection. The Connection Object is distinguished in CMAP messages by a heavy line with a pen pattern.

<table>
<tr><td colspan="2">con_id (2)</td><td colspan="2">con_status (2)</td><td>user_con_type (4)</td></tr>
<tr><td>con_type (1)</td><td>reserved (1)</td><td>con_def (1)</td><td>con_perm (1)</td><td rowspan="2">bw (12)</td></tr>
<tr><td colspan="4"></td></tr>
</table>

Figure 26. The CMAP Connection Object

5.6.1 con_id

The ***con_id*** field is an unsigned two-byte integer containing the connection *identifier* (Section 4.4.1). The identifier may be generated by the client creating the connection, in which case the network will verify that the ***con_id*** is unique for the call. Alternatively, the client can have the network generate the identifier by leaving the ***con_id*** field blank (value 0xFFFF), and the value will be returned in the network **RESPONSE**.

5.6.2 con_status

The ***con_status*** field contains a symbolic value indicating whether any errors occurred in the specification of the connection. It is used only in **RESPONSEs** from the network. See Appendix D for an explanation of the values that can appear in this field.

5.6.3 user_con_type

The ***user_con_type*** field is a four-byte unsigned integer containing the connection's *user type* (Section 4.4.6). It is not examined by the network but is delivered end-to-end without modification. It may be used to communicate to potential endpoints some additional, application-level information about the type of the connection. For instance, the creator of a video connection might want to indicate the type of video format (*e.g.*, **NTSC**, **HDTV**, **MPEG**) that is being used.

5.6.4 con_type

The ***con_type*** field is a one-byte field containing the connection *type* (Section 4.4.2).The subfields of the connection type 3-tuple are packed into one byte as shown here (Figure 27). The possible values for each subfield are given in Appendix C.

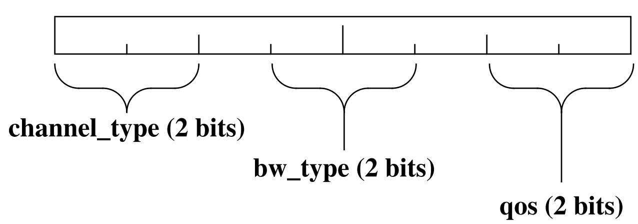


Figure 27. con_type format (8 bits)

5.6.5 con_def

The ***con_def*** field is a one-byte field containing the connection *defaults* (Section 4.4.4). The subfields of the mapping are packed as shown here (Figure 28). The possible values for each subfield are given in Appendix C. The value 0xFF is blank.

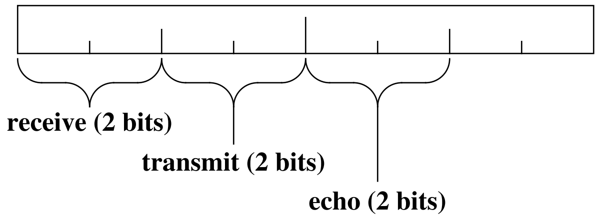


Figure 28. con_def format (8 bits)

5.6.6 con_perm

The ***con_perm*** field is a one-byte field containing the connection *permissions* (Section 4.4.5). The subfields of the permissions are packed in the same manner as the ***con_def*** field (Figure 28). The possible values for each subfield are given in Appendix C. The value 0xFF is blank.

5.6.7 bw

The ***bw*** field is a twelve-byte field containing the connection *bandwidth* (Section 4.4.3). Each of the three components of the bandwith (peak rate in cells/second, average rate in cells/second, and peak burst length in cells) is represented by a four-byte unsigned integer. These integers are arranged in the ***bw*** field as shown here (Figure 29). A best-effort connection is indicated by all three of these values being 0.

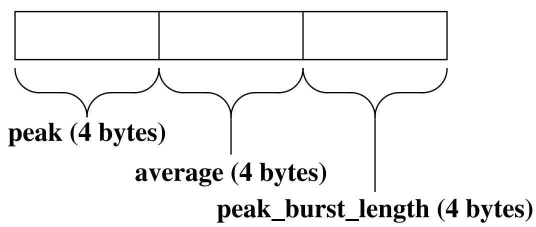


Figure 29. bw format (12 bytes)

5.7 Endpoint Object

The Endpoint Object (Figure 30) identifies a particular endpoint (call/client interface). The Endpoint Object is distinguished in CMAP messages by a heavy line with a pen pattern.

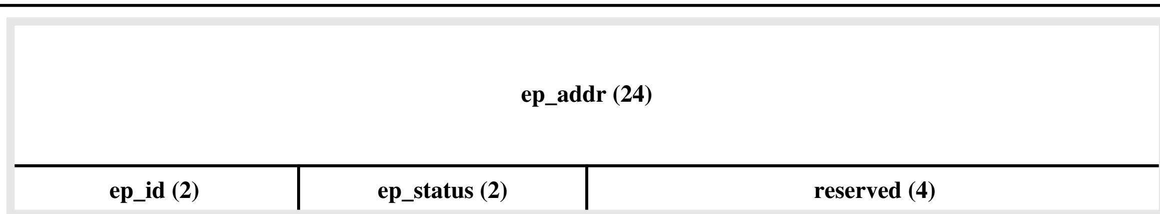


Figure 30. The CMAP Endpoint Object

5.7.1 ep_addr

The ***ep_addr*** field is the twenty-four-byte *endpoint address*, that is, the address of the client at which the endpoint is located (Section 4.5.1).

5.7.2 ep_id

The ***ep_id*** field is the endpoint's *local identifier* (Section 4.5.2). This field is an unsigned two-byte integer. It may be generated by the client, in which case it is checked by the network to ensure that it is unique for the client. Alternatively, the client may leave the identifier field blank (value 0xFFFF) and the network will generate and return a unique value.

5.7.3 ep_status

The ***ep_status*** field contains a symbolic value indicating whether any errors occurred in the specification of the endpoint address. It is used only in **RESPONSEs fr**om the network. See Appendix D for an explanation of the values that can appear in this field.

5.8 UNI Object

The UNI Object (Figure 31) groups the per-connection UNI parameters of an endpoint for a single connection. The UNI Object is distinguished in CMAP messages by a heavy line with a pen pattern.

<table>
<tr><td colspan="2">ep_con_id (2)</td><td>uni_status (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>reserved (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td>trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
</table>

Figure 31. The CMAP UNI Object

5.8.1 ep_con_id

The ***ep_con_id*** field is an unsigned two-byte integer containing the connection *identifier* (Section 4.4.1). Recall from Section 4.5 that, except for the endpoint identifier, all endpoint properties are on a per-connection basis. This field indicates with which connection the remaining object fields are associated. The value 0xFFFF is blank.

5.8.2 uni_status

The ***uni_status*** field contains a symbolic value indicating whether any errors occurred in the specification of the endpoint. See Appendix D for an explanation of the values that can appear in this field.

5.8.3 ep_map

The ***ep_map*** field is a one-byte field containing the current endpoint *mapping* (Section 4.5.3). The subfields of the mapping are packed in the same manner as the ***con_def*** field (Figure 28). The possible values for each subfield are given in Appendix C. The value 0xFF is blank.

5.8.4 ep_def

The ***ep_def*** field is a one-byte field containing the endpoint *defaults* (Section 4.5.4). The subfields of the mapping are packed in the same manner as the ***con_def*** field (Figure 28). The possible values for each subfield are given in Appendix C. The value 0xFF is blank.

5.8.5 ep_perm

The ***ep_perm*** field is a one-byte field containing the endpoint *permissions* (Section 4.5.5). The subfields of the mapping are packed in the same manner as the ***con_def*** field (Figure 28). The possible values for each subfield are given in Appendix C. The value 0xFF is blank.

5.8.6 trans_vpi, trans_vci, rcv_vpi, rcv_vci

These four fields are all unsigned integers of the indicated sizes. ***trans_vpi*** and ***trans_vci*** contain the endpoint *transmit pair* (Section 4.5.6), and ***rcv_vpi*** and ***rcv_vci*** contain the endpoint *receive pair* (Section 4.5.7). A client adding an endpoint may propose values for these pairs, and the network will check that they are available. Alternatively, the client may leave a pair blank (both fields contain the value 0) and the network will allocate and return pairs.

5.9 Operation Object

The Operation Object (Figure 32) is used to request or signal the status of CMAP operations. The Operation Object is distinguished in CMAP messages by a heavy line with a pen pattern.

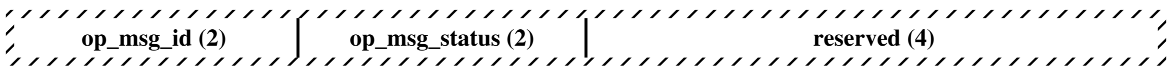


Figure 32. The CMAP Operation Object

5.9.1 op_msg_id

The ***op_msg_id*** field is the two-byte message identifier (Section 5.3.3) for the operation to which this object refers.

5.9.2 op_msg_status

The ***op_msg_status*** field contains a symbolic value giving the status of the operation. See Appendix D for an explanation of the values that can appear in this field.

5.10 CMAP Message Byte and Bit Order of Transmission

Figure 33 describes the CMAP message byte order of transmission. The CMAP message is always transmitted from the highest ordered byte (which is always the ***op_type*** field of the CMAP message header object) to the lowest ordered byte (which is always the final byte of the trailer object). Multiple-byte fields are stored and transmitted in order from highest-order byte to lowest-order byte. All CMAP bytes are bit ordered from the highest ordered bit to the lowest and the bits are transmitted in this order. When transmitting a CMAP message any ***unused*** or ***reserved*** fields should contain all 0 bits.

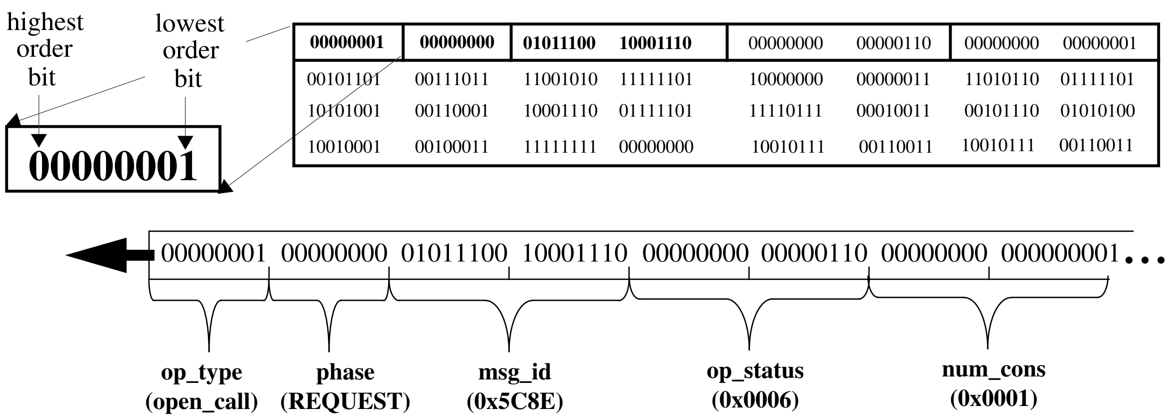


Figure 33. Byte/Bit Order of Transmission

6. CMAP Operations

6.1 Overview

CMAP operations may be divided into two broad classes: *call* operations and *maintenance* operations. Call operations implement the CMAP call model, while maintenance operations support the session manager/client interface. All operations may involve multiple phases and various types of responses and confirmations; see Sections 5.3.2 and 5.3.8 for explanations of phases, responses, and confirmations.

Call operations are divided into four classes: *commands*, *prompts*, *queries*, and *notifications*. *Commands* are two-phase operations that a client uses to request a service from the network. The client issues a **REQUEST** to the network, which replies with an **ACK** or a **NACK**. As part of the execution of a command, the network may send *prompts*, *queries* and *notifications* to other clients; these types of operations do not arise in any other way. A *prompt* is a two- or three-phase operation that the network uses to request an action by a client. The network issues a **REQUEST** to the client, which responds with an **ACK**, **NACK**, or **NEG**. The network may then respond with a **COM** or an **ABORT**. A *query* is a two-phase operation that the network uses to request verification from the owner on whether to proceed with an operation. The network issues a **REQUEST** to the owner, which responds with an **ACK** or **NACK**. A *notification* is a one-phase operation that the network uses to inform a client of a change in state. The network sends a **REQUEST**; the client does not send any response.

Table 9 lists the CMAP commands together with the prompts, queries, and notifications that may arise during execution of the command (N/A indicates no such operation will arise). Note that the names of all prompts begin with **invite**, the names of all queries with **verify**, and the names of all notifications with **announce**.

Table 9. CMAP Call Operations

Command	Prompts	Queries	Notifications	Description
open_call	**invite_add_ep**	**verify_mod_ep**	**announce_add_ep**	create a call.
mod_call	**N/A**	**N/A**	**announce_mod_call**	change call attributes.
close_call	**N/A**	**N/A**	**announce_close_call**	terminate a call.
add_con	**invite_add_con**	**verify_mod_ep**	**announce_add_con**	add a connection to a call.
mod_con	N/A	N/A	**announce_mod_con**	change connection attributes.
drop_con	N/A	N/A	**announce_drop_con, announce_close_call**	drop a connection.
add_ep	**invite_add_ep**	**verify_add_ep, verify_mod_ep**	**announce_add_ep**	add an endpoint to a call.
mod_ep	**invite_mod_ep**	**verify_mod_ep**	**announce_mod_ep**	change endpoint attributes.
drop_ep	N/A	N/A	**announce_drop_ep**	drop an endpoint.
trace_call	N/A	N/A	N/A	report call and connection attributes, and the endpoints participating in the call.
trace_ep	N/A	N/A	N/A	report endpoint's attributes.
change_owner	**invite_change_owner**	N/A	**announce_change_owner**	pass ownership responsibility to another client.
change_root	N/A	N/A	**announce_change_root**	change root client of the call.

As a simple example of the relationship between commands, prompts, queries, and notifications, consider the case where a client ***A*** wishes to add endpoint ***B*** to a call managed by client ***C*** with an accessibility of **VERIFY**. Client ***A*** sends an **add_ep REQUEST** to ask that client ***B*** be added to the call. The network sends a **verify_add_ep REQUEST** query to ***C*** to determine if ***B*** may be added. Assuming that ***C*** approves of the addition, it will send a **verify_add_ep ACK** back to the network. The network then prompts ***B*** with an **invite_add_ep REQUEST** to see if ***B*** wants to be added. ***B*** may agree to be added by responding with an **invite_add_ep ACK**. The network then adds ***B***, sending an **add_ep ACK** to ***A*** and possibly an **announce_add_ep REQUEST** notification to other participants in the call, depending on the call's monitoring parameter (Section 4.3.8). The original **add_ep** command thus led to a series of CMAP operations.

Table 10 lists the CMAP maintenance operations. In this table, the "Number of Phases" column refers to the **REQUEST**, **RESPONSE**, and **CONFIRMATION** messages exchanged in the operation (Section 5.1). For example, the **status** operation has two phases, in which a client (or the network) first sends a **REQUEST** to obtain information about an operation, and the network (or a client) then sends a **RESPONSE** containing the requested information.

Table 10. CMAP Maintenance Operations

Operation	Number of Phases	Description
status	2	request information about status of an operation.
alert	1	inform client or network about status of an operation.
client_reset	2	inform network that a client has been reset.
network_reset	1	inform clients that the network has reset.
error_report	1	inform client or network of serious message errors.

6.1.1 Client State Machines

The call and maintenance operations are processed by the client according to the state machines shown in the figures below. These state machines are not intended to completely capture the state of the client, but only to indicate its state with respect to the operation that is being performed. The client must store additional call-model state information for each call, connection, and endpoint to describe the parameters of these entities.

Figure 34 shows the state machine for commands and the **status** and **client_reset** maintenance operations. These two-phase operations are initiated by the client, which sends a **REQUEST** message to the network. At some later time the network returns a **RESPONSE** to the client, which completes the operation. The client uses the contents of the **RESPONSE** message to update its call-model information.

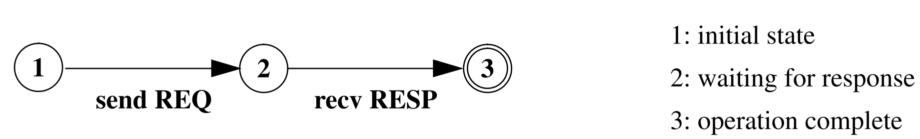


Figure 34. State Machine for a Command, status, or client_reset

Figure 35 shows the state machine for prompts. These two-phase operations are initiated by the network, which sends a **REQUEST** message to the client. The client must return a **RESPONSE**, which may be an **ACK**, **NACK**, or **NEG**. If the response is a **NACK** the operation is complete. If it is an **ACK** or **NEG**, the network will return a **CONFIRMATION** to complete the operation. The client uses the contents of the **CONFIRMATION** message to update its call-model information.

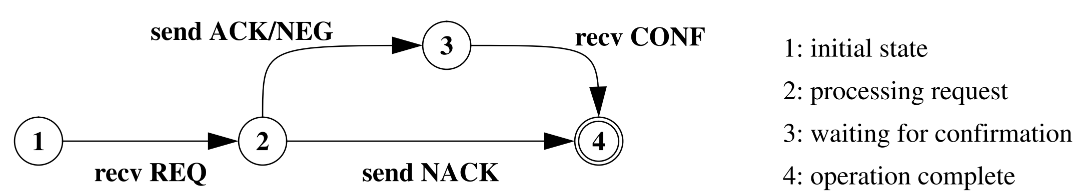


Figure 35. State Machine for a Prompt

Figure 36 shows the state machine for queries and **status** operations. These two-phase operations are initiated by the network, which sends a **REQUEST** message to the client. The client must return a **RESPONSE**, which may be an **ACK** or **NACK**, to complete the operation. The client will not normally update its call-model information from a query, since the contents of the message represents tentative information that may not become true (*e.g.*, in a **verify_add_ep** the endpoint which is being added may eventually **NACK** the operation).

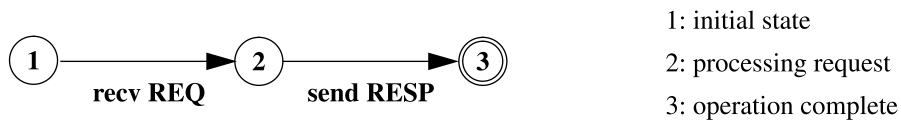


Figure 36. State Machine for a Query or status

Figure 37 shows the state machine for notifications and the **alert** and **network_reset** maintenance operations. These one-phase operations are initiated by the network, which sends a **REQUEST** message to the client. The client does not need to send a response. The client will update its call-model information from the **REQUEST** message.

Figure 37. State Machine for a Notification, alert, or network_reset

Two maintenance operations are not covered by the above state machines. An **alert** initiated by the client is a one-phase operation in which the client sends a **REQUEST**; the network makes no response. Its state machine is thus similar to that of Figure 37, except that the transition would be made on sending the **REQUEST**. The **error_report** is handled similarly to an **alert**, except that it also has an effect on the state machine for some other operation. This is further explained in the description of the **error_report** operation (Section 6.35).

Because CMAP permits operations to be performed concurrently, the client must maintain a separate state machine for each currently-active operation. This is the case even if the client does not ever perform two commands concurrently, since commands may cause other operations (prompts, queries, and notifications). For example, if a client starts an **open_call** operation, it may have to participate in **invite_add_ep**, **verify_mod_ep**, and **announce_add_ep** operations (all produced by the **open_call**) before the **open_call** is completed. In addition, of course, other clients may make requests which cause prompts, queries, or notifications to be sent to the client.

6.1.2 Operation Descriptions

The remainder of this section defines the individual CMAP operations. The commands are described first, followed by the prompts, queries, notifications, and finally the maintenance operations. Each operation definition contains five parts: *Synopsis*, *Message Traffic*, *Message Format*, *Parameter Negotiation*, and *Client Operation*.

The *Synopsis* section gives a brief description of the operation.

The *Message Traffic* section describes the messages that may be sent as a result of the operation. This includes not only the **RESPONSE** and **CONFIRMATION** phases for the original **REQUEST**, but also operations that are caused by the original operation.

The *Message Format* section provides the template message formats for each operation phase and a brief description of the use of the individual fields of the message. If the use of a field does not substantially differ from that described in Sections 5.3 through 5.9, it is omitted.

The *Parameter Negotiation* section describes how the client and network mutually agree on any unspecified parameters of the operation. A summary of parameter negotiation appears in Appendix F.

The *Operation* section notes any special actions that the client that sent or received the operation's **REQUEST** should perform. This section is omitted from some descriptions.

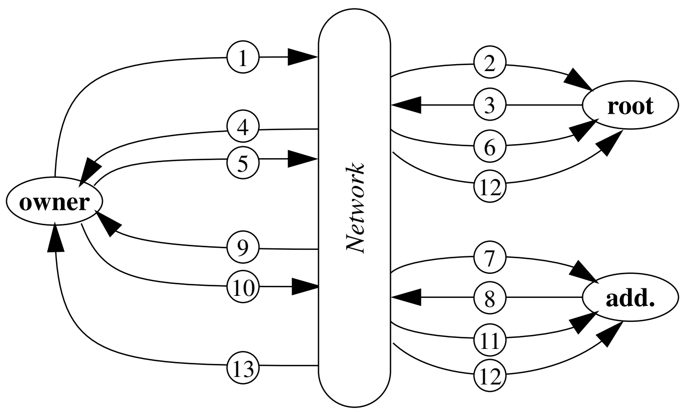
6.2 open_call Command

6.2.1 Synopsis

This command requests that a new call be created. It may be performed by any CMAP client, which (if the operation is successful) becomes the owner of the new call. An initial set of connections are established, and one or two endpoints are added to the call.

6.2.2 Message Traffic

Up to three distinct clients may be involved in an **open_call** operation: the client requesting the operation (henceforth the owner), the root client, and the optional additional client. It is possible that a single client may act in all three roles. Figure 38 shows the traffic pattern for the operation.



Message list:
1: open_call REQUEST
2: invite_add_ep REQUEST
3: invite_add_ep ACK/NACK/NEG
4: verify_mod_ep REQUEST
5: verify_mod_ep ACK/NACK
6: invite_add_ep COM/ABORT
7: invite_add_ep REQUEST
8: invite_add_ep ACK/NACK/NEG
9: verify_mod_ep REQUEST
10: verify_mod_ep ACK/NACK
11: invite_add_ep COM/ABORT
12: announce_add_ep REQUEST
13: open_call ACK/NACK

Client list:
owner: requester of open_call
root: root of call
add.: optional additional endpoint

Figure 38. Message Traffic for open_call

Initiation. The owner initiates the operation by sending an **open_call REQUEST** (1). The network checks the message for errors. If errors are found, an **open_call NACK** (13) is sent to the owner and the operation is complete. If there are no errors, the operation continues with the invitation of the root and the additional endpoint (in parallel).

Invitation of root. An **invite_add_ep REQUEST** (2) is sent to the root, which must respond (3) with an **invite_add_ep ACK**, **NACK**, or **NEG**. If the root sends an **invite_add_ep NEG**, verification with the owner may be required (specifically, if the root attempts to change its mapping and the corresponding permission is set to **VERIFY**). The owner is sent a **verify_mod_ep REQ** (4) to which it responds (5) with a **verify_mod_ep ACK** or **NACK**. If the owner sends a **verify_add_ep NACK**, or if the requested changes were not acceptable, the root is sent an **invite ABORT** (6), the owner is sent an **open_call NACK** (13), and the operation is then complete.

Invitation of additional endpoint. If the the owner did not specify an additional endpoint in the **open_call REQUEST** (1), it is sent an **open_call ACK** (13) and the operation is complete. If an endpoint was specified an **invite_add_ep/verify_mod_ep** sequence (7-11) similar to that of the root (2-6) is performed with the additional endpoint.

Final confirmation. If two endpoints were specified and both were successfully added, **invite_add_ep CONFs** (6, 11) are sent to the endpoints and an **open_call ACK** (13) to the owner; if required by the call's monitoring parameter, **announce_add_ep REQUESTs** (12) are sent to each endpoint informing it that the other has joined. If either endpoint was not added (due to the endpoint sending a **NACK** (3, 8) or to the owner refusing a verification (5, 10)), **invite_add_ep ABORTs** (6, 11) are sent to the endpoints and an **open_call NACK** (13) to the owner. If only one endpoint was specified, an **invite_add_ep COM** or **ABORT** (6) is sent to the endpoint and an **open_call ACK** or **NACK** (13) to the owner, depending on whether the endpoint was successfully added or not.

Note: The network will skip an **invite_add_ep** operation (2,7) in the **open_call** if it is directed to the owner of the call and all parameters of the endpoint, including VPI/VCI pairs, are specified in the **open_call REQUEST** and are legal.

6.2.3 Message Formats

open_call REQUEST

00000001	00000000	msg_id (2)		num_cons (2)		num_eps (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		unused (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
unused (2)		reserved (2)		user_call_type (4)			
call_type (1)	acc (1)	mod (1)	trace (1)	mon (1)	priority (1)	reserved (2)	
con_id (2)		unused (2)		user_con_type (4)			
con_type (1)	reserved (1)	con_def (1)	con_perm (1)	bw (12)			
(num_cons)							
ep_addr (24)							
ep_id (2)		unused (2)		reserved (4)			
ep_con_id (2)		unused (2)		ep_map (1)	ep_def (1)	ep_perm (1)	unused (1)
reserved (1)	trans_vpi (1)	trans_vci (2)		reserved (1)	rcv_vpi (1)	rcv_vci (2)	
(num_cons)							
(num_eps)							
options_size (4)				options (options_size)			

Data:

- ***call_id -*** call identifier of the call to be created. The requesting client must specify the ***r_addr*** portion of this field, which is the address of the root client; the ***lcid*** portion may be left blank.
- ***num_cons*** - the number of connections to be created in the call; equal to the number of Connection Objects and the number of UNI Objects associated with each Endpoint Object. Must be at least 1.

- ***num_eps*** - the number of endpoints to be added to the call; equal to the number of repetitions of the Endpoint Object/UNI Objects combination. Must be either 1 or 2.
- ***call_type***, ***acc***, ***mod***, ***trace***, ***mon*** - must be fully specified.

For each of the ***num_cons*** connections, the message has a Connection Object specifying the initial values of the connection's parameters.

- ***con_id*** - may be left blank. All non-blank ***con_ids*** must be distinct.
- ***con_type***, ***con_def***, ***con_perm***, ***bw*** - must be fully specified.

For each of the ***num_eps*** endpoints, the message has an Endpoint Object and ***num_cons*** UNI Objects specifying the endpoint and the initial values of the endpoint attributes. If ***num_eps*** is 1, the first (and only) group of Endpoint and UNI Objects is that associated with the root. If ***num_eps*** is 2, the first group is that associated with the root, and the second is that associated with the additional endpoint.

- ***ep_addr*** - that associated with the root must be equal to the ***r_addr*** portion of the ***call_id***.
- ***ep_id*** - may be left blank.
- ***ep_con_id*** - each ***ep_con_id*** must be the same as the ***con_id*** in the corresponding position in the list of Connection Objects.
- ***ep_map***, ***ep_def***, ***ep_perm*** - may be left blank.
- ***trans_vpi***, ***trans_vci***, ***rcv_vpi***, ***rcv_vpi*** - may be left blank.

open_call RESPONSE

<table>
<tr><td>00000001</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">num_eps (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">call_status (2)</td><td colspan="2">reserved (2)</td><td colspan="4">user_call_type (4)</td></tr>
<tr><td>call_type (1)</td><td>acc (1)</td><td>mod (1)</td><td>trace (1)</td><td>mon (1)</td><td>priority (1)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="2">con_id (2)</td><td colspan="2">con_status (2)</td><td colspan="4">user_con_type (4)</td></tr>
<tr><td>con_type (1)</td><td>reserved (1)</td><td>con_def (1)</td><td>con_perm (1)</td><td colspan="4">bw (12)</td></tr>
<tr><td colspan="8">(num_cons)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">ep_status (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">uni_status (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>reserved (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">(num_cons)</td></tr>
<tr><td colspan="8">(num_eps)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***call_id*** - call identifier of the call that was created.
- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **BAD_NUM_EPS**, **BAD_CALL_ID_ADDR**, **DUP_CALL_ID**, **VERIFY_REFUSED**, **EP_REFUSED**, **INSUFF_BW**, **TIMEOUT** }

- ***call_status*** - this field may take on the following values:

  call_status ∈ { **OK**, **BAD_CALL_TYPE**, **BAD_ACC**, **BAD_MOD**, **BAD_TRACE**, **BAD_MON**, **BAD_PRIORITY** }

- ***con_status*** - this field may take on the following values:

  con_status ∈ { **OK**, **DUP_CON_ID**, **BAD_CON_TYPE**, **BAD_CON_DEF**, **BAD_CON_PERM**, **BAD_BW** }

- ***ep_status*** - this field may take on the following values:

  ep_status ∈ { **OK**, **BAD_EP_ADDR**, **DUP_EP_ID**, **BAD_EP_ADDR_NOT_ROOT** }

- ***uni_status*** - this field may take on the following values:

  uni_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID**, **BAD_EP_MAP**, **BAD_EP_DEF**, **BAD_EP_PERM**, **NO_AVAIL_VPI**, **NO_AVAIL_VCI**, **TRANS_VPI_IN_USE**, **TRANS_VPI_RESERVED**, **TRANS_VPI_NOT_SUPPORTED**, **TRANS_VCI_IN_USE**, **TRANS_VCI_RESERVED**, **TRANS_VCI_NOT_SUPPORTED**, **RCV_VPI_IN_USE**, **RCV_VPI_RESERVED**, **RCV_VPI_NOT_SUPPORTED**, **RCV_VCI_IN_USE**, **RCV_VCI_RESERVED**, **RCV_VCI_NOT_SUPPORTED** }

6.2.4 Parameter Negotiation

r_addr. Theroot address may differ between the **REQUEST** and the **RESPONSE** — for example, the address supplied in the **REQUEST** may deliver the message to a server on a specific machine, which uses the user call type to start a process that becomes the actual root. The address in the **RESPONSE** is the root address (and call identifier address) for the call. If an additional endpoint is specified, the final root address is sent to it in the **invite_add_ep COM**.

lcid. If the owner leaves this field blank, the network will select a value before performing the invitations; if the owner provides a value, the network will check that it is not in use before performing the invitiations. During the invitations the root has the option of suggesting a different value. If the owner or the root provides a value which is already in use, the network will return a **NACK** or **ABORT** (respectively) with ***status*** = **DUP_CALL_ID**; otherwise it will use the owner-supplied value.

con_ids. The network will assign values to any blank ***con_ids***. Assignment will be by consecutive integers starting with 1 and increasing through the Connection Objects in the order given, skipping any identifiers already in use.

ep_addrs. Treated in the same way as the ***r_addr***.

ep_ids. Treated in the same way as the ***lcid***.

ep_con_ids. The non-blank ***ep_con_ids*** are first matched with the non-blank ***con_ids*** (the two groups must contain the same identifiers). The network then assigns values to any blank ***ep_con_ids***. Assignment will be by consecutive integers starting with 1 and increasing through the Connection Objects in the order given, skipping any identifiers already in use. This will produce the same set of identifiers as for the ***con_ids***.

ep_map, ***ep_def***, ***ep_perm***. If any of these fields are left blank, the network assigns the corresponding connection value (***con_def*** for ***ep_map*** and ***ep_def***, ***con_perm*** for ***ep_perm***) to the field before offering it to the endpoints. The endpoint may attempt to negotiate ***ep_map***, if the permissions allow, but not ***ep_def*** or ***ep_perm***.

trans_vpi/trans_vci, ***rcv_vpi/rcv_vci***. If any of these pairs are blank, the network will select a value before offering it to the endpoints. The client may accept these values or propose different ones in its response. Thus, both the owner and the endpoint client have an opportunity to specify the VPI/VCI pairs.

6.2.5 Operation

Execution of the **open_call** command is made somewhat more complex by the possibility that the owner will receive **invite_add_ep REQUESTs** or **verify_mod_ep REQUESTs** for this call while in the *open_call_pending* state. The network issues an **invite_add_ep REQUEST** to each of the initial endpoints, so if the owner is an endpoint of the call it will receive such a request. A similar case arises in surrogate signalling; if, for example, the root is a mute client that is being managed by the owner, the owner will receive an **invite_add_ep REQUEST** for the root. The owner may receive a **verify_mod_ep REQUEST** when a remote client is invited to join the call and proposes modifications to its endpoint mapping for which the corresponding endpoint permission is set to **VERIFY**. The network will then check with the owner to determine if the modification is allowed.

These messages add complexity in the following way. Normally, on receiving an **invite_add_ep REQUEST** a client will examine its internal list of calls to determine if it is already participating in the call given in the message header. If the client finds the call, it will add (or not add, as the case may be) an endpoint to the data structure associated with that call. If the client does not find the call, it will create a new call object and add (or not add) the endpoint to the new call. Similarly, on receiving a **verify_mod_ep REQUEST** a client examines its internal list of calls that it is managing in order to find the appropriate call data structure; if it does not find the call, it signals an error. The adjustment to the message-handling is thus a simple one: on receipt of an **invite_add_ep** or **verify_mod_ep REQUEST**, instead of simply examining the lists of calls in which it is participating or managing the client should also examine the list of calls which it is in the process of creating.

6.3 mod_call Command

6.3.1 Synopsis

This command requests a change in the general call parameters. It may only be requested by the owner of the call. The call's user type, accessibility, modifiability, traceability, owner notification, default notification and priority may be modified by means of this command.

6.3.2 Message Traffic

Figure 39 shows the message traffic that may result from this command. The owner initiates the operation by sending a **mod_call REQUEST** (1). If the network detects any errors a **mod_call NACK** (2) is sent to the owner and the operation is complete. Otherwise the network sends a **mod_call ACK** (2) to the owner and an **announce_mod_call REQUEST** (3) to each endpoint describing the modifications.

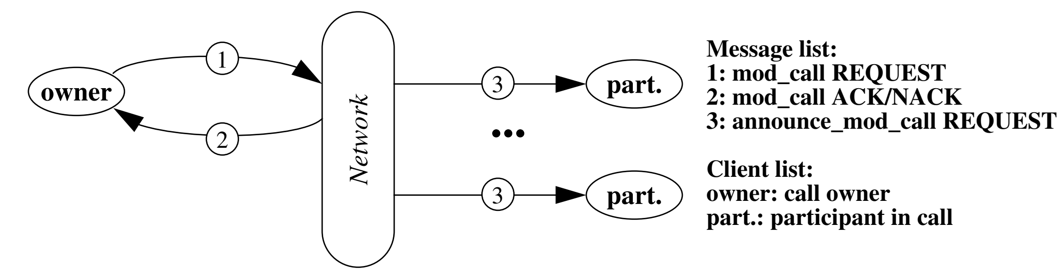


Figure 39. Message Traffic for mod_call

6.3.3 Message Formats

mod_call REQUEST

<table>
<tr><td>00000010</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="4">user_call_type (4)</td></tr>
<tr><td>call_type (1)</td><td>acc (1)</td><td>mod (1)</td><td>trace (1)</td><td>mon (1)</td><td>priority (1)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***call_id*** - call identifier of call to be modified.
- ***user_call_type***, ***acc***, ***mon***, ***trace***, ***mon***, ***priority*** - new values to be assigned to the parameters. Must be fully specified.

mod_call RESPONSE

<table>
<tr><td>00000010</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">call_status (2)</td><td colspan="2">reserved (2)</td><td colspan="4">user_call_type (4)</td></tr>
<tr><td>call_type (1)</td><td>acc (1)</td><td>mon (1)</td><td>trace (1)</td><td>own_not (1)</td><td>def_not (1)</td><td>priority (1)</td><td>reserved (1)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:
  status ∈ { **OK**, **UNKNOWN_CALL**, **NOT_OWNER**, **TIMEOUT** }
- ***call_status*** - this field may take on the following values:
  call_status ∈ { **OK**, **BAD_CALL_TYPE**, **BAD_ACC**, **BAD_MOD**, **BAD_TRACE**, **BAD_MON**, **BAD_PRIORITY** }

6.3.4 Parameter Negotiation

There is no parameter negotiation in this command.

6.4 close_call Command

6.4.1 Synopsis

This command requests that an existing call be terminated and may only be requested by the owner of the call.

6.4.2 Message Traffic

Figure 40 shows the message traffic that may result. The owner initiates the operation by sending a **close_call REQUEST** (1). The network checks the message for errors. If errors are found a **close_call NACK** (2) is sent to the owner and the operation is complete. If no errors are found, the network sends a **close_call ACK** (2) to the owner and a **announce_close_call REQUEST** (3) to each participant. The operation is complete for the owner upon receipt of the **close_call ACK**, and for each participant upon receipt of the **announce_close_call REQUEST**.

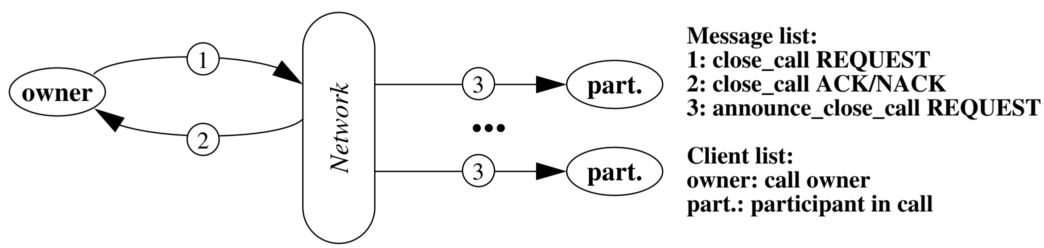


Figure 40. Message Traffic for close_call

6.4.3 Message Formats

close_call REQUEST

00000011	**00000000**	**msg_id (2)**	**unused (2)**	**unused (2)**
call_id (r_addr) (24)				
call_id (lcid) (2)		**unused (2)**	**reserved (2)**	**reserved (2)**
m_addr (24)				
s_addr (24)				
options_size (4)			**options (options_size)**	

Data:

- ***call_id -*** call identifier of the call to be closed.

close_call RESPONSE

00000011	**00000001**	**msg_id (2)**	**unused (2)**	**unused (2)**
call_id (r_addr) (24)				
call_id (lcid) (2)		**op_status (2)**	**reserved (2)**	**reserved (2)**
m_addr (24)				
s_addr (24)				
options_size (4)			**options (options_size)**	

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:
  status ∈ { **OK**, **UNKNOWN_CALL**, **NOT_OWNER**, **TIMEOUT** }

6.4.4 Parameter Negotiation

There is no parameter negotiation in this command.

6.4.5 Operation

When the owner provides a valid **close_call REQUEST**, the call is terminated with respect to the owner. If there is a failure in the network, or if the network has problems tearing down all the connections, the **close_call** will still be acknowledged as successful (*i.e.*, the owner will not receive an error indication). It is up to the connection management layer to correct any such problems. The clients are not burdened by connections that remain up after the **close_call REQUEST**, nor should they be billed for any resources not properly freed after the **close_call**.

6.5 add_con Command

6.5.1 Synopsis

This operation requests that one or more new connections be opened within an existing call. This command may be requested by any participant in the call, subject to the call's *modifiability* parameter (Section 4.3.6). The requester may specify the parameters for one endpoint (already in the call) in the **add_con**. If the requester is the owner, this endpoint may belong to any client; if the requester is not the owner, the endpoint must belong to the requester. The mappings and defaults for the endpoint may differ from those for the new connections.

6.5.2 Message Traffic

Each participant in the call receives messages as a result of a successful **add_con** as shown in Figure 41. Each endpoint already in the call is forced to join the new connection, although it may do so with a NULL mapping; see **invite_add_con** for more details.

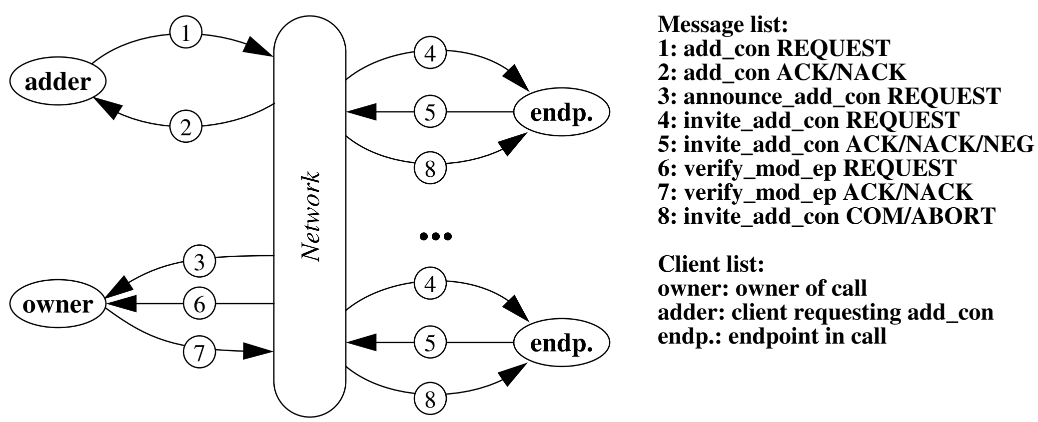


Figure 41. Message Traffic for add_con

Initiation. The requesting client initiates the operation by sending an **add_con REQUEST** (1). The network checks the message for errors. If errors are found an **add_con NACK** (4) is sent to the requester and the operation is complete. If no errors are found, the operation continues with the connection verification.

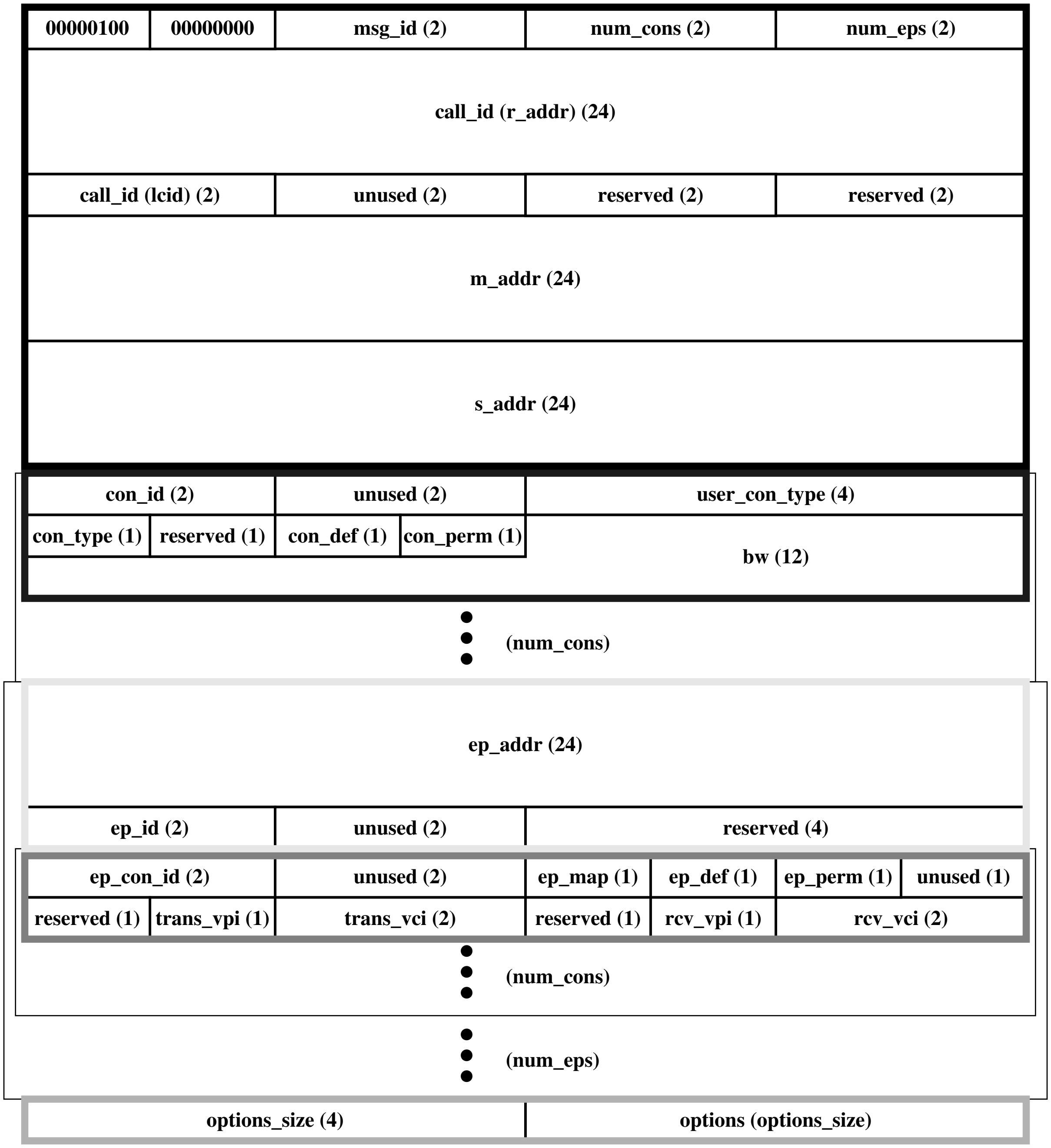
Connection verification. If the requester is the owner or if the call's *modifiability* is OPEN an **add_con ACK** (4) is sent to the requester and the operation continues with the invitations. If the requester is not the owner and the call's *modifiability* is CLOSED, an **add_con NACK** (4) is sent to the requester and the operation is complete.

Invitations. If the requester is not the owner of the call, an **announce_add_con REQUEST** (3) is sent to the owner to inform it of the new connection. An **invite_add_con REQUEST** (4) is sent to each endpoint in the call. These prompts cause each of the endpoints to join the new connection. The endpoint may return an **invite_add_con ACK** (5), joining the connection with the suggested parameters and completing the operation for that endpoint. It may return an **invite_add_con NACK** (5); in this case the endpoint still joins the connection, but it will receive a NULL mapping for the new connection (*i.e.*, it can neither transmit nor receive). It may return an **invite_add_con NEG** (5), requesting a modification in some of its endpoint parameters. If verification with the owner is required, the owner is sent a **verify_mod_ep REQ** (6) to which it responds (7) with a **verify_mod_ep ACK** or **NACK**. If the owner sends a **verify_add_ep ACK** (7), or if the requested changes were acceptable without verification, the endpoint is sent an **invite_add_ep COM** (8) and the endpoint is added to the connection. If the owner sends a **verify_add_ep NACK** (7), or if the requested changes were not acceptable, the endpoint is sent an **invite ABORT** (8) and the endpoint is added to the connection with a NULL mapping.

Note: The network will skip the **invite_add_con** operation (4) for the endpoint specified in the **add_con** if it is directed to the requester of the **add_con** and all parameters of the connection and endpoint, including connection identifier and VPI/VCI pairs, are specified in the **add_con REQUEST** and are legal. If any are illegal, the operation will fail with a (2) **add_con NACK**; otherwise the legal values for the endpoint will be returned in the **add_con ACK**.

6.5.3 Message Formats

add_con REQUEST



Data:

- ***num_cons*** - the number of connections to be added to the call; equal to the number of Connection Objects and (if present) UNI Objects. Must be at least 1.
- ***num_eps*** - the number of endpoint mappings specified; equal to the number of repetitions of the Endpoint Object/UNI Objects combination. Must be 0 or 1.

For each of the ***num_cons*** connections, the message has a Connection Object specifying the new parameters.

- ***con_id*** - may be left blank.

- ***con_type***, ***con_def***, ***con_perm***, ***bw*** - must be fully specified.

If ***num_eps*** = 1, the message contains an Endpoint Address Object and ***num_cons*** UNI Objects which specify the parameters of one endpoint for the new connections. If ***num_eps*** = 0 these objects are not present.

- ***ep_con_id*** - the number of blank ***ep_con_ids*** must be the same as the number of blank ***con_ids***; each non-blank ***ep_con_id*** must equal some non-blank ***con_id*** and all must be distinct.
- ***ep_map***, ***ep_def***, ***ep_perm***, ***trans_vpi***, ***trans_vci***, ***rcv_vpi***, ***rcv_vci*** - may be blank.

add_con RESPONSE

00000100	**00000001**	**msg_id (2)**		**num_cons (2)**		**num_eps (2)**	
call_id (r_addr) (24)							
call_id (lcid) (2)		**op_status (2)**		**reserved (2)**		**reserved (2)**	
m_addr (24)							
s_addr (24)							

con_id (2)		**con_status (2)**		**user_con_type (4)**
con_type (1)	**reserved (1)**	**con_def (1)**	**con_perm (1)**	**bw (12)**

(num_cons)

ep_addr (24)		
ep_id (2)	**ep_status (2)**	**reserved (4)**

ep_con_id (2)		**uni_status (2)**		**ep_map (1)**	**ep_def (1)**	**ep_perm (1)**	**unused (1)**
reserved (1)	**trans_vpi (1)**	**trans_vci (2)**		**reserved (1)**	**rcv_vpi (1)**	**rcv_vci (2)**	

(num_cons)

(num_eps)

options_size (4)	**options (options_size)**

Data:

- ***num_cons***, ***num_eps*** - these will have the same values as in the **REQUEST**.

If ***num_eps*** = 1, the message contains an Endpoint Address Object and ***num_cons*** UNI Objects which specify the final parameters of the endpoint for the new connections. If ***num_eps*** = 0 these objects are not present.

- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **UNKNOWN_CALL**, **ILL_REQUEST**, **INSUFF_BW**, **TIMEOUT** }

- ***con_status*** - this field may take on the following values:

  con_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID**, **BAD_CON_TYPE**, **BAD_CON_DEF**, **BAD_CON_PERM**, **BAD_BW** }

- ***ep_status*** - this field may take on the following values:

  ep_status ∈ { **OK**, **BAD_EP_ADDR** }

- ***uni_status*** - this field may take on the following values:

  uni_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID**, **BAD_EP_MAP**, **ILL_EP_MAP**, **BAD_EP_DEF**, **BAD_EP_PERM**, **NO_AVAIL_VPI**, **NO_AVAIL_VCI**, **TRANS_VPI_IN_USE**, **TRANS_VPI_RESERVED**, **TRANS_VPI_NOT_SUPPORTED**, **TRANS_VCI_IN_USE**, **TRANS_VCI_RESERVED**, **TRANS_VCI_NOT_SUPPORTED**, **RCV_VPI_IN_USE**, **RCV_VPI_RESERVED**, **RCV_VPI_NOT_SUPPORTED**, **RCV_VCI_IN_USE**, **RCV_VCI_RESERVED**, **RCV_VCI_NOT_SUPPORTED** }

6.5.4 Parameter Negotiation

con_ids. The network will assign values to any blank ***con_ids***. Assignment will be by consecutive integers starting with 1 and increasing through the Connection Objects in the order given, skipping any identifiers already in use in the call. This assignment is performed before any other operations (such as the **invite_add_cons**).

ep_con_ids. The non-blank ***ep_con_ids*** are first matched with the non-blank ***con_ids*** (the two groups must contain the same identifiers). The network then assigns values to any blank ***ep_con_ids***. Assignment will be by consecutive integers starting with 1 and increasing through the Connection Objects in the order given, skipping any identifiers already in use. This will produce the same set of identifiers as for the ***con_ids***.

ep_map, ***ep_def***, ***ep_perm***. If any of these fields are left blank, the network assigns the corresponding connection value (***con_def*** for ***ep_map*** and ***ep_def***, ***con_perm*** for ***ep_perm***) to the field before offering it to the endpoint in the **invite_add_con**. The endpoint may negotiate ***ep_map*** but not ***ep_def*** or ***ep_perm***.

trans_vpi/trans_vci, ***rcv_vpi/rcv_vci***. If any of these pairs are blank, the network will select a value before offering it to the endpoint in the **invite_add_con**. The client may accept these values or propose different ones in its response. Thus, both the requester and the endpoint client have an opportunity to specify the VPI/VCI pairs.

6.6 mod_con Command

6.6.1 Synopsis

This operation requests that the parameters of one or more connections be changed. The type, defaults, permissions, bandwidth, and user type may be changed. This command may only be requested by the owner. Changing the parameters of a connection has no effect on the values of endpoint mappings, defaults, and permissions.

6.6.2 Message Traffic

Each participant in the call receives messages as a result of a successful **mod_con** as shown in Figure 42. The owner initiates the operation by sending a **mod_con REQUEST** (1). The network checks the message for errors, and if any are found returns a **mod_con NACK** (2); this completes the operation. If no errors are found, a **mod_con ACK** (2) is sent to the owner and an **announce_mod_con REQUEST** (3) to each endpoint in the call.

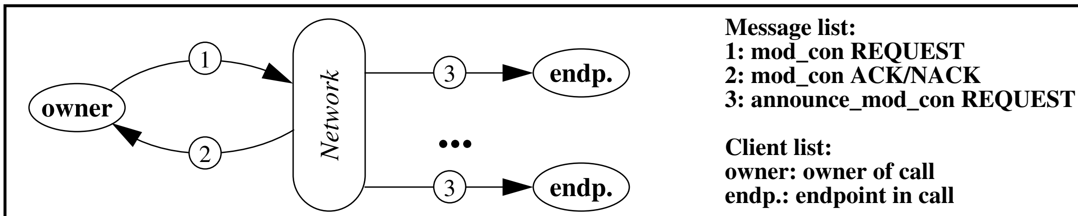


Figure 42. Message Traffic for mod_con

6.6.3 Message Formats

mod_con REQUEST

00000101	00000000	msg_id (2)		num_cons (2)		unused (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		unused (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
con_id (2)		unused (2)		user_con_type (4)			
con_type (1)	reserved (1)	con_def (1)	con_perm (1)	bw (12)			
(num_cons)							
options_size (4)				options (options_size)			

Data:

- ***num_cons*** - the number of Connection Objects. Must be at least 1.

For each of the ***num_cons*** connections, the message has a Connection Object containing the new values of the connection parameters.

- ***con_id*** - must be specified, and must equal the identifier of an existing connection. All ***con_ids*** must be distinct.
- ***con_type***, ***con_def***, ***con_perm***, ***bw*** - must be fully specified.

mod_con RESPONSE

<table>
<tr><td>00000101</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">con_id (2)</td><td colspan="2">con_status (2)</td><td colspan="4">user_con_type (4)</td></tr>
<tr><td>con_type (1)</td><td>reserved (1)</td><td>con_def (1)</td><td>con_perm (1)</td><td colspan="4">bw (12)</td></tr>
<tr><td colspan="8">(num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **UNKNOWN_CALL**, **NOT_OWNER**, **INSUFF_BW**, **TIMEOUT** }

- ***con_status*** - this field may take on the following values:

  con_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID, BAD_CON_TYPE**, **BAD_CON_DEF**, **BAD_CON_PERM**, **BAD_BW** }

6.6.4 Parameter Negotiation

There is no parameter negotiation in this command.

6.7 drop_con Command

6.7.1 Synopsis

This operation requests that one or more connections be closed. It may only be performed by the call's owner.

6.7.2 Message Traffic

Each participant in the call receives messages as a result of a successful **drop_con** as shown in Figure 43. The owner initiates the operation by sending a **drop_con REQUEST** (1). The network checks the message for errors, and if any are found returns a **drop_con NACK** (2); this completes the operation. If no errors are found, a **drop_con ACK** (2) is sent to the owner. If one or more connections will remain open after the **drop_con**, an **announce_drop_con REQUEST** (3) is sent to each endpoint. If no connections will remain open after the **drop_con**, the call is closed and an **announce_close_call REQUEST** (4) is sent to the owner and to each endpoint.

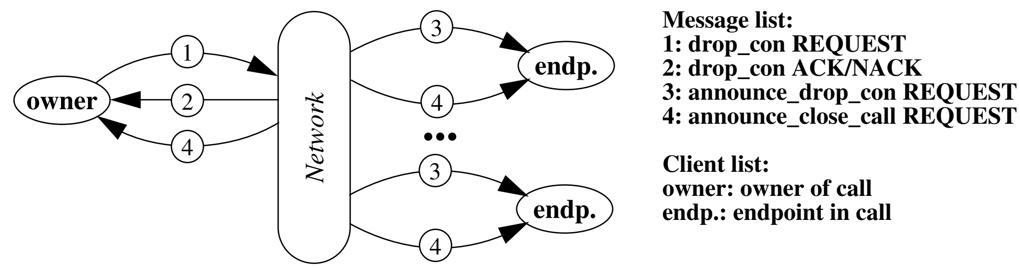


Figure 43. Message Traffic for drop_con

6.7.3 Message Formats

drop_con REQUEST

<table>
<tr><td>00000110</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">con_id (2)</td><td colspan="2">con_status (2)</td><td colspan="4">user_con_type (4)</td></tr>
<tr><td>con_type (1)</td><td>reserved (1)</td><td>con_def (1)</td><td>con_perm (1)</td><td colspan="4" rowspan="2">bw (12)</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="8">• • • (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - the number of connections to be dropped; equal to the number of ***con_ids***. Must be at least 1.
- ***con_id*** - identifier of the connection to be dropped**.**

drop_con RESPONSE

00000110	00000001	msg_id (2)		num_cons (2)	unused (2)
call_id (r_addr) (24)					
call_id (lcid) (2)		unused (2)		reserved (2)	reserved (2)
m_addr (24)					
s_addr (24)					
con_id (2)		con_status (2)		user_con_type (4)	
con_type (1)	reserved (1)	con_def (1)	con_perm (1)	bw (12)	
(num_cons)					
options_size (4)				options (options_size)	

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **UNKNOWN_CALL**, **NOT_OWNER**, **INSUFF_BW**, **TIMEOUT** }
- ***con_status*** - this field may take on the following values:

  con_status ∈ { **OK**, **BAD_CON_ID** }

6.7.4 Parameter Negotiation

There is no parameter negotiation in this command.

6.7.5 Operation

When a client provides a valid **drop_con REQUEST**, the connection is closed. If there is a failure in the network, or if the network has problems tearing down all the connections, the **drop_con** will still be acknowledged as successful (*i.e.*, the owner will not receive an error indication). It is up to the connection management layer to correct any such problems. The clients are not burdened by connections that remain up after the **drop_con REQUEST**, nor should they be billed for any resources not properly freed after the **drop_con**.

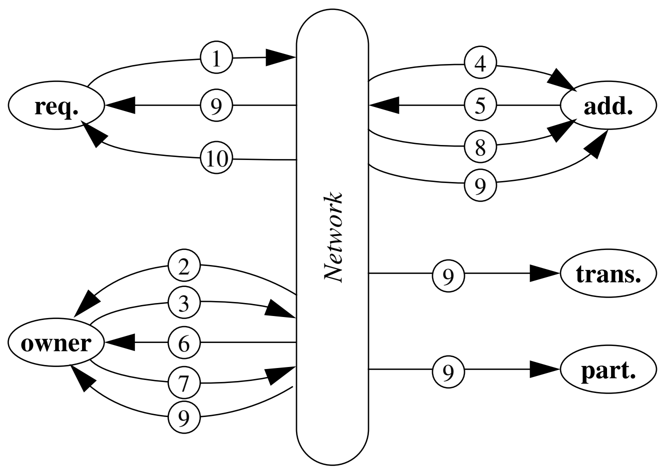
6.8 add_ep Command

6.8.1 Synopsis

This operation requests that a new endpoint be added to an existing call. An **add_ep** may be requested by the owner of the call, the client to be added to the call, or a third client (who may or may not be part of the call).

6.8.2 Message Traffic

Three primary clients may be involved in an **add_ep** operation: the owner of the call, the client requesting the addition of an endpoint, and the client which is to be added. Of course, a single client may play two or even all three of these roles. In addition to these clients, other participants in the call may be notified that the endpoint has been added, based on the call's *monitoring* parameter (Section 4.3.8). Figure 44 shows the message traffic involved.



Message list:
1: add_ep REQUEST
2: verify_add_ep REQUEST
3: verify_add_ep ACK/NACK
4: invite_add_ep REQUEST
5: invite_add_ep ACK/NACK/NEG
6: verify_mod_ep REQUEST
7: verify_mod_ep ACK/NACK
8: invite_add_ep COM/ABORT
9: announce_add_ep REQUEST
10: add_ep ACK/NACK

Client list:
req.: client requesting operation
add.: client to be added
owner: owner of call
trans.: any transmitter in call
part.: other participant in call

Figure 44. Message Traffic for add_ep

Initiation. The requesting client initiates the operation by sending an **add_ep REQUEST** (1). The network performs a number of parameter checks. If errors are found an **add_ep NACK** (10) is sent to the requester and the operation is complete. If no errors are found the operation continues with the verification of the addition.

Verification of addition. If the requester is not the owner and the call's *accessibility* (Section 4.3.5) is set to **CLOSED**, an **add_ep NACK** (10) is sent the the requester and the operation is complete. If the requester is not the owner and the call's accessiblility is sent to **VERIFY**, a **verify_add_ep REQUEST** (2) is sent to the owner which must respond (3) with a **verify_add_ep ACK** or **NACK**. If the owner responds with a **verify_add_ep ACK**, the operation continues with the invitation of the endpoint. If the owner responds with a **verify_add_ep NACK**, an **add_ep NACK** (10) is sent to the requester and the operation is complete.

Invitiation of endpoint. If the requester is not the client that is being added, or if the requester omitted any endpoint parameters in the **REQUEST**, the network sends an **invite_add_ep REQUEST** (4) to the client that is being added. The client must respond (5) with an **invite_add_ep ACK**, **NACK**, or **NEG**. If it responds with an **invite_add_ep ACK**, the operation continues with the addition of the endpoint. If it responds with an **invite_add_ep NACK**, an **add_ep NACK** (10) is sent to the requester and the operation is complete. If the client responds with an **invite_add_ep NEG**, verification by the owner may be required (if the client changed its mapping and the corresponding permissions are **VERIFY**). In this case a **verify_mod_ep REQUEST** (6) is sent to the owner of the call which responds (7) with a **verify_mod_ep ACK** or **NACK**. If the owner sends a **verify_mod_ep ACK**, an **invite_add_ep COM** (8) is sent to the client being added and the operation continues with the addition of the endpoint. If the owner sends a **verify_mod_ep NACK**, an **invite_add_ep ABORT** (8) is sent to the client being added, an **add_ep NACK** (10) to the requester, and the operation is complete.

Addition of endpoint. When the endpoint has been successfully added, **announce_add_ep REQUESTs** (9) are sent to all appropriate parties as determined by the call's monitoring parameter. An **add_ep ACK** (10) is sent to the requester and the operation is complete.

6.8.3 Message Formats

add_ep REQUEST

00000111	00000000	msg_id (2)		num_cons (2)		unused (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		unused (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
unused (2)		reserved (2)		user_call_type (4)			
call_type (1)	acc (1)	mod (1)	trace (1)	mon (1)	priority (1)	reserved (2)	
con_id (2)		unused (2)		user_con_type (4)			
con_type (1)	reserved (1)	con_def (1)	con_perm (1)	bw (12)			
(num_cons)							
ep_addr (24)							
ep_id (2)		unused (2)		reserved (4)			
ep_con_id (2)		unused (2)		ep_map (1)	ep_def (1)	ep_perm (1)	unused (1)
reserved (1)	trans_vpi (1)	trans_vci (2)		reserved (1)	rcv_vpi (1)	rcv_vci (2)	
(num_cons)							
options_size (4)				options (options_size)			

Data:

- ***num_cons*** - the number of Connection and UNI Objects specified in the call; must equal the number of connections actually in the call.

The Call and Connection Objects describe the call to which the endpoint is to be added and must match the parameters of the call. All parameters of these objects must be specified.

- ***ep_addr*** - address of the client to be added.
- ***ep_id*** - local identifier of the endpoint; may be left blank.

- ***ep_con_id*** - must be specified. ***E***ach ***ep_con_id*** must be the same as the ***con_id*** in the corresponding position in the list of Connection Objects.
- ***ep_map***, ***ep_def***, ***ep_perm***, ***trans_vpi***, ***trans_vci***, ***rcv_vpi***, ***rcv_vpi*** - may be left blank.

add_ep RESPONSE

<table>
<tr><td>00000111</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">call_status (2)</td><td colspan="2">reserved (2)</td><td colspan="4">user_call_type (4)</td></tr>
<tr><td>call_type (1)</td><td>acc (1)</td><td>mod (1)</td><td>trace (1)</td><td>mon (1)</td><td>priority (1)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="2">con_id (2)</td><td colspan="2">con_status (2)</td><td colspan="4">user_con_type (4)</td></tr>
<tr><td>con_type (1)</td><td>reserved (1)</td><td>con_def (1)</td><td>con_perm (1)</td><td colspan="4" rowspan="2">bw (12)</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="8">• • • (num_cons)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">ep_status (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">uni_status (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">• • • (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - the number of connections in the call, and also the number of Connection and Uni Objects in the message.
- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **UNKNOWN_CALL**, **ILL_REQUEST**, **VERIFY_REFUSED**, **EP_REFUSED**, **INSUFF_BW**, **TIMEOUT** }

- ***call_status*** - this field may take on the following values:

  call_status ∈ { **OK**, **BAD_CALL_TYPE**, **BAD_ACC**, **BAD_MOD**, **BAD_TRACE**, **BAD_MON**, **BAD_PRIORITY** }

- ***con_status*** - this field may take on the following values:

  con_status ∈ { **OK**, **DUP_CON_ID**, **BAD_CON_TYPE**, **BAD_CON_DEF**, **BAD_CON_PERM**, **BAD_BW** }

- ***ep_status*** - this field may take on the following values:

  ep_status ∈ { **OK**, **BAD_EP_ADDR**, **DUP_EP_ID** }

- ***uni_status*** - this field may take on the following values:

  uni_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID**, **BAD_EP_MAP**, **ILL_EP_MAP**, **BAD_EP_DEF**, **BAD_EP_PERM**, **NO_AVAIL_VPI**, **NO_AVAIL_VCI**, **TRANS_VPI_IN_USE**, **TRANS_VPI_RESERVED**, **TRANS_VPI_NOT_SUPPORTED**, **TRANS_VCI_IN_USE**, **TRANS_VCI_RESERVED**, **TRANS_VCI_NOT_SUPPORTED**, **RCV_VPI_IN_USE**, **RCV_VPI_RESERVED**, **RCV_VPI_NOT_SUPPORTED**, **RCV_VCI_IN_USE**, **RCV_VCI_RESERVED**, **RCV_VCI_NOT_SUPPORTED** }

In an **ACK**, the UNI objects contain the values of the endpoint's UNI parameters for each connection in the call.

6.8.4 Parameter Negotiation

ep_addr. The local address field of the address may differ between the **REQUEST** and the **RESPONSE** — for example, the address supplied in the **REQUEST** may deliver the message to a server on a specific machine, which uses the user call type to start a process that becomes the actual client. The address in the **RESPONSE** is the actual address of the client.

ep_id. The network will pass this field to the endpoint client in the **invite_add_ep REQUEST**. Thus, if the requester leaves it blank the endpoint client will have the opportunity to select a value. If the client also leaves it blank, the network will select an appropriate value.

ep_map, ***ep_def***, ***ep_perm***. If any of these fields are left blank, the network assigns the corresponding connection value (***con_def*** for ***ep_map*** and ***ep_def***, ***con_perm*** for ***ep_perm***) to the field before offering it to the endpoint client in the **invite_add_ep REQUEST**.

trans_vpi/trans_vci, ***rcv_vpi/rcv_vci***. If any of these pairs are blank, the network will select a value before offering it to the endpoint client. The endpoint client may accept these values or propose different ones in its response. Thus, both the requester and the endpoint client have an opportunity to specify the VPI/VCI pairs.

6.8.5 Operation

If the call and connection parameters supplied in the **REQUEST** do not correspond to the actual parameters of the call, a **NACK** will be returned with one or more of the status fields set to indicate the problem.

As with the **open_call** command, execution of **add_ep** by the owner for another client is made more complex by the possibility that the owner will receive a **verify_mod_ep REQUEST** while waiting for the response to the **add_ep REQUEST**. This is handled in the same way as in the **open_call** operation.

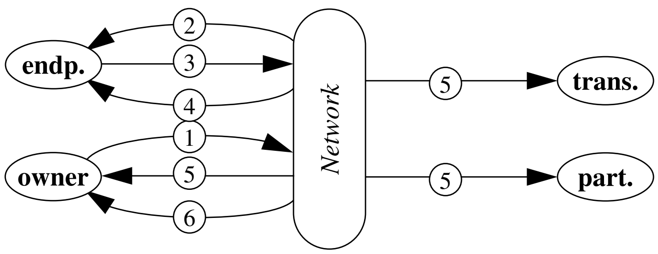
6.9 mod_ep Command

6.9.1 Synopsis

This operation requests a change in the parameters of an existing endpoint. The operation may be requested by the owner or by the client whose endpoint parameters are being changed. The owner may change the endpoints mapping, defaults, permissions, and VPI/VCI pairs; the client may only change its mapping and VPI/VCI pairs. When the owner changes a client's parameters, the client is not permitted to refuse the change.

6.9.2 Message Traffic

Figure 45 shows the message traffic for the case where the owner initiates the operation. The owner begins by sending a **mod_ep REQUEST** (1). If there are any errors in the request, the network responds with a **mod_ep NACK** (6). Otherwise, the endpoint is invited to accept the new parameters with an **invite_mod_ep REQUEST** (2), to which it replies with an **ACK**, **NACK**, or **NEG** (3); the operation may then be confirmed with a **COM** or **ABORT**. The network sends a **mod_ep ACK** or **NACK** (6) to the owner and **announce_mod_ep REQUESTs** (5) to all interested parties (as determined by the call's monitoring parameter).

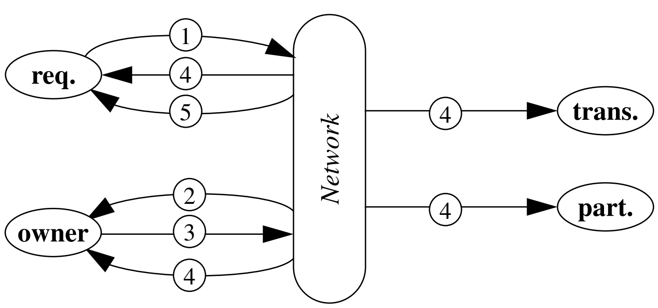


Message list:
1: mod_ep REQUEST
2: invite_mod_ep REQUEST
3: invite_mod_ep ACK/NACK/NEG
4: invite_mod_ep COM/ABORT
5: announce_mod_ep REQUEST
6: mod_ep ACK/NACK

Client list:
endp.: client being modified
owner: owner of call
trans.: any transmitter in call
part.: other participant in call

Figure 45. Message Traffic for mod_ep by Owner

Figure 46 shows the traffic in the endpoint-initated case. The requesting client initiates the operation by sending a **mod_ep REQUEST** (1). The network checks this message, and if errors are found (including mapping changes forbidden by the endpoint's permissions) it sends a **mod_ep NACK** (5) to the requester and the operation is complete. If no errors are found and the requested changes are allowed by the endpoint's permissions, the network sends a **mod_ep ACK** (5) to the requester and **announce_mod_ep REQUESTs** (4) to all interested parties. If the permission corresponding to any of the requested changes is **VERIFY**, the network queries the owner with a **verify_mod_ep REQUEST** (2), to which the owner must respond with a **verify_mod_ep ACK** or **NACK** (3). If the owner responds with an **ACK**, the network sends a **mod_ep ACK** (5) to the requester and **announce_mod_ep REQUESTs** (4) to all interested parties. If the owner responds with a **NACK**, the network sends a **mod_ep NACK** (5) to the requester.



Message list:
1: mod_ep REQUEST
2: verify_mod_ep REQUEST
3: verify_mod_ep ACK/NACK
4: announce_mod_ep REQUEST
5: mod_ep ACK/NACK

Client list:
req.: client requesting operation
owner: owner of call
trans.: any transmitter in call
part.: other participant in call

Figure 46. Message Traffic for mod_ep by Non-Owner

6.9.3 Message Formats

mod_ep REQUEST

00001000	00000000	msg_id (2)		num_cons (2)		unused (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		unused (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
ep_addr (24)							
ep_id (2)		unused (2)		reserved (4)			
ep_con_id (2)		unused (2)		ep_map (1)	ep_def (1)	ep_perm (1)	unused (1)
reserved (1)	trans_vpi (1)	trans_vci (2)		reserved (1)	rcv_vpi (1)	rcv_vci (2)	
(num_cons)							
options_size (4)				options (options_size)			

Data:

- ***num_cons*** - the number of UNI Objects specified in the call. Must be at least 1.
- ***ep_addr***, ***ep_id*** - identifier of the endpoint to be modified.

The UNI objects contain the new values of the mapping.

- ***ep_con_id***, ***ep_map*** - must be specified.
- ***ep_map***, ***ep_def***, ***ep_perm***, ***trans_vpi***, ***trans_vci***, ***rcv_vpi***, ***rcv_vpi*** - may be left blank.

mod_ep RESPONSE

00001000	00000001	msg_id (2)		num_cons (2)		unused (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		op_status (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
ep_addr (24)							
ep_id (2)		ep_status (2)		reserved (4)			
ep_con_id (2)		uni_status (2)		ep_map (1)	ep_def (1)	ep_perm (1)	unused (1)
reserved (1)	trans_vpi (1)	trans_vci (2)		reserved (1)	rcv_vpi (1)	rcv_vci (2)	
⋮ (num_cons)							
options_size (4)				options (options_size)			

Data:

- ***num_cons*** - the number of Endpoint Objects in the message; equal to the number of connections.
- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **UNKNOWN_CALL**, **NOT_OWNER**, **VERIFY_REFUSED**, **EP_REFUSED**, **INSUFF_BW**, **TIMEOUT** }
- ***ep_status*** - this field may take on the following values:

  ep_status ∈ { **OK**, **BAD_EP_ADDR** }
- ***uni_status*** - this field may take on the following values:

  uni_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID**, **BAD_EP_MAP**, **BAD_EP_DEF**, **ILL_EP_MAP**, **BAD_EP_PERM**, **NO_AVAIL_VPI**, **NO_AVAIL_VCI**, **TRANS_VPI_IN_USE**, **TRANS_VPI_RESERVED**, **TRANS_VPI_NOT_SUPPORTED**, **TRANS_VCI_IN_USE**, **TRANS_VCI_RESERVED**, **TRANS_VCI_NOT_SUPPORTED**, **RCV_VPI_IN_USE**, **RCV_VPI_RESERVED**, **RCV_VPI_NOT_SUPPORTED**, **RCV_VCI_IN_USE**, **RCV_VCI_RESERVED**, **RCV_VCI_NOT_SUPPORTED** }

The UNI objects contain the new values of the endpoint's UNI parameters.

6.9.4 Parameter Negotiation

There is no parameter negotiation in this operation.

6.10 drop_ep Command

6.10.1 Synopsis

This operation requests that an endpoint or a client be dropped from a call. This may be requested by the owner of the call or the client to be dropped. The last endpoint of the root may not be dropped.

6.10.2 Message Traffic

The message traffic of the **drop_ep** operation differs slightly, depending on whether the owner or the endpoint itself initiates the operation. Figure 47 shows the message traffic in the owner-inititated case. The owner initiates the operation by sending a **drop_ep REQUEST** (1), which may specify either a specific endpoint or all endpoints at a client. The network performs a number of parameter checks. If errors are found a **drop_ep NACK** (4) is sent to the owner and the operation is complete. If no errors are found, an **announce_drop_ep REQUEST** (2) is sent to the endpoint being dropped, **announce_drop_ep REQUESTs** (3) are sent to all appropriate endpoints (as determined by the call's monitoring parameter (Section 4.3.8)), a **drop_ep ACK** (4) is sent to the owner, and the operation is complete.

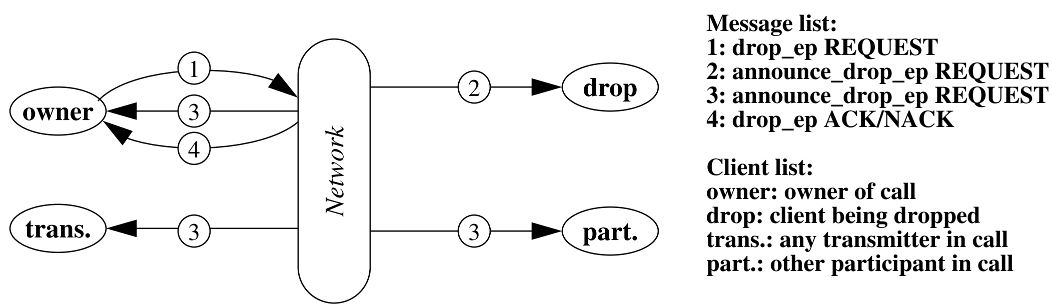


Figure 47. Message Traffic for drop_ep by Owner

Figure 48 shows the message traffic in the endpoint-initiated case. The endpoint initiates the operation by sending a **drop_ep REQUEST** (1), which may specify either a specific endpoint or all endpoints at the client. The network performs a number of parameter checks. If errors are found a **drop_ep NACK** (3) is sent to the endpoint and the operation is complete. If no errors are found, **announce_drop_ep REQUESTs** (2) are sent to all appropriate endpoints, a **drop_ep ACK** (3) is sent to the endpoint and the operation is complete.

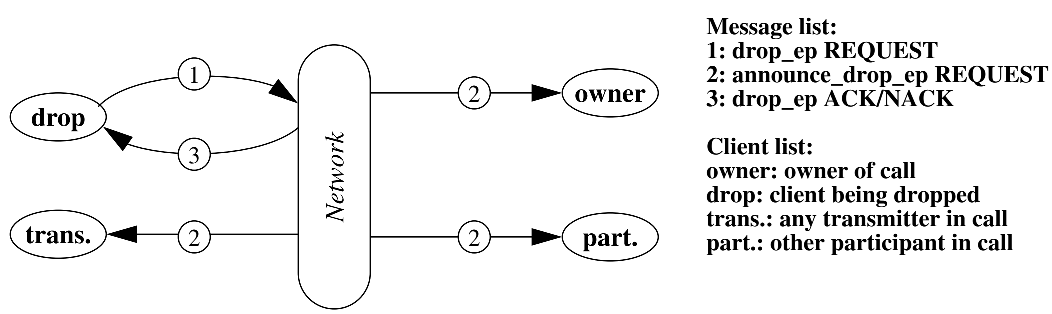


Figure 48. Message Traffic for drop_ep by Non-Owner

6.10.3 Message Formats

drop_ep REQUEST

00001001	00000000	msg_id (2)	unused (2)	unused (2)
call_id (r_addr) (24)				
call_id (lcid) (2)		unused (2)	reserved (2)	reserved (2)
m_addr (24)				
s_addr (24)				
ep_addr (24)				
ep_id (2)		unused (2)	reserved (4)	
options_size (4)			options (options_size)	

Data:

- ***ep_addr*** - address of the client to be dropped.
- ***ep_id*** - identifier of the endpoint to be dropped. If left blank, specifies that all endpoints at the client are to be dropped.

drop_ep RESPONSE

<table>
<tr><td>00001001</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">ep_status (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:
  status ∈ { **OK**, **UNKNOWN_CALL**, **NOT_OWNER**, **ILL_DROP_ROOT**, **TIMEOUT** }
- ***ep_status*** - this field may take on the following values:
  ep_status ∈ { **OK**, **BAD_EP_ADDR** }

6.10.4 Parameter Negotiation

There is no parameter negotiation in this command.

6.10.5 Operation

The owner is permitted to drop all its own endpoints, but it must still manage the call. If the owner or root attempts to drop the last endpoint of the root, it will be refused (a ***status*** field of **ILL_DROP_ROOT** in the returned **drop_ep RESPONSE**). If there are network errors, the **drop_ep** is acknowledged as successful and it is up to the network to clean up any problems it may have had.

When a client provides a valid **drop_ep REQUEST**, the endpoint is closed. If there is a failure in the network, or if the network has problems tearing down all the connections, the **drop_ep** will still be acknowledged as successful (*i.e.*, the owner will not receive an error indication). It is up to the connection management layer to correct any such problems. The clients are not burdened by connections that remain up after the **drop_ep REQUEST**, nor should they be billed for any resources not properly freed after the **drop_ep**.

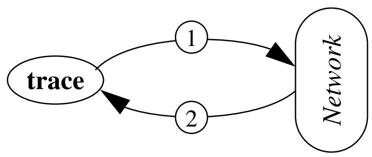
6.11 trace_call Command

6.11.1 Synopsis

This operation requests that a call be traced, *i.e.*, that information about the call be supplied to the client. It may be requested by any client, subject to the call's *traceability* (Section 4.3.7).

6.11.2 Message Traffic

Figure 49 shows the message traffic for **trace_call**. The client sends a **trace_call REQUEST** (1). The network performs a number of parameter checks, and in particular checks the call traceability to see if the operation is permitted. If errors are found a **trace_call NACK** (2) is sent to the requester. If no errors are found, a **trace_call ACK** (2) is sent to the requester.



Message list:
1: trace_call REQUEST
2: trace_call ACK/NACK

Client list:
trace: client performing trace

Figure 49. Message Traffic for trace_call

6.11.3 Message Formats

trace_call REQUEST

00001010	00000000	msg_id (2)	unused (2)	unused (2)
call_id (r_addr) (24)				
call_id (lcid) (2)		unused (2)	reserved (2)	reserved (2)
m_addr (24)				
s_addr (24)				
options_size (4)			options (options_size)	

Data:

- ***call_id*** - identifier of the call to be traced.

trace_call RESPONSE

00001010	00000001	msg_id (2)		num_cons (2)		num_eps (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		op_status (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
owner_addr (24)							
unused (2)		reserved (2)		user_call_type (4)			
call_type (1)	acc (1)	mod (1)	trace (1)	mon (1)	priority (1)	reserved (2)	
con_id (2)		unused (2)		user_con_type (4)			
con_type (1)	reserved (1)	con_def (1)	con_perm (1)	bw (12)			
⋮ (num_cons)							
ep_addr (24)							
ep_id (2)		unused (2)		reserved (4)			
⋮ (num_eps)							
options_size (4)				options (options_size)			

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:
  status $\in$ { **OK**, **UNKNOWN_CALL**, **ILL_REQUEST**, **TIMEOUT** }
- ***num_cons*** - the number of connections in the call, and the number of Connection Objects in the message.
- ***num_eps*** - the number of endpoints in the call, and the number of Endpoint Objects in the message.
- ***owner_addr*** - the address of the owner/manager of the call.

The Call Object contains the call parameters. The Connection Objects contain the parameters of the connections. The Endpoint Objects contain the addresses of the endpoints participating in the connection (parameters of individual endpoints may be obtained with the **trace_ep** command).

In a **NACK**, ***num_cons*** = ***num_eps*** = 0 and the message contains no ***owner_addr*** or Call, Connection, or Endpoint Objects. The Trailer Object immediately follows the Header Object.

6.11.4 Parameter Negotiation

There is no parameter negotiation in this command.

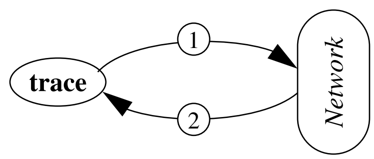
6.12 trace_ep Command

6.12.1 Synopsis

This operation requests that an endpoint be traced, *i.e.*, that information about the endpoint be supplied to the client. It may be requested by any client, subject to the call's *traceability* (Section 4.3.7).

6.12.2 Message Traffic

Figure 50 shows the message traffic for **trace_ep**. The client sends a **trace_ep REQUEST** (1). The network performs a number of parameter checks, and in particular checks the call traceability to see if the operation is permitted. If errors are found a **trace_ep NACK** (2) is sent to the requester. If no errors are found, a **trace_ep ACK** (2) is sent to the requester.



Message list:
1: trace_ep REQUEST
2: trace_ep CK/NACK

Client list:
trace: client performing trace

Figure 50. Message Traffic for trace_ep

6.12.3 Message Formats

trace_ep REQUEST

<table>
<tr><td>00001011</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***ep_addr***, ***ep_id*** - identifier of the endpoint to be traced.

trace_ep RESPONSE

<table>
<tr><td>00001011</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">ep_status (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">unused (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">⋮ (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:
  status $\in$ { **OK**, **UNKNOWN_CALL**, **ILL_REQUEST**, **TIMEOUT** }
- ***num_cons*** - the number of connections in the call, and the number of UNI Objects in the message
- ***ep_status*** - this field may take on the following values:
  ep_status $\in$ { **OK**, **BAD_EP_ADDR** }

The UNI Objects contain the per-connection parameters of the endpoint. In a **NACK** ***num_cons*** = 0 and the UNI Objects are omitted.

6.12.4 Parameter Negotiation

There is no parameter negotiation in this command.

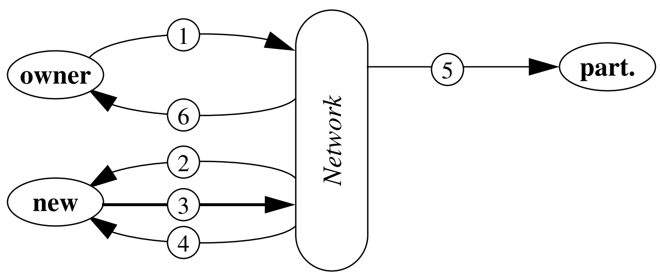
6.13 change_owner Command

6.13.1 Synopsis

This operation requests that the owner of a call be changed. It may only be requested by the owner of the call. The new owner need not be a participant in the call.

6.13.2 Message Traffic

Figure 51 shows the message traffic. The owner initiates the operation by sending a **change_owner REQUEST** (1). If the message contains any errors, a **change_owner NACK** (6) is sent to the owner and the operation is complete. Otherwise, the network sends an i**nvite_change_owner REQUEST** (2) to the candidate new owner, which must respond (3) with an **invite_change_owner ACK** or **NACK**. If it sends an **ACK** and the operation can be completed, the call owner is changed, the old owner is sent a **change_owner ACK** (5), the new owner is sent an **invite_change_owner COM** (4) and all participants in the call are sent **announce_change_owner REQUESTs** (4). If it sends an ACK and the operation cannot be competed, the call owner is unchanged, the candidate new owner is sent an **invite_change_owner ABORT** (4), and the owner is sent a **change_owner NACK** (5). If the candidate owner sends a **NACK**, the call owner is unchanged and the owner is sent a **change_owner NACK** (5).



Message list:
1: change_owner REQUEST
2: invite_change_owner REQUEST
3: invite_change_owner ACK/NACK
4: invite_change_owner COM/ABORT
5: announce_change_owner REQUEST
6: change_owner ACK/NACK

Client list:
owner: owner of call
new: new owner of call
part.: participant in call

Figure 51. Message Traffic for change_owner

6.13.3 Message Formats

change_owner REQUEST

<table>
<tr><td>00001100</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">new_owner (24)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***new_owner*** - address of the candidate new owner.

change_owner RESPONSE

<table>
<tr><td>00001100</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">new_owner (24)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status $\in$ { **OK**, **UNKNOWN_CALL**, **NOT_OWNER**,
  EP_REFUSED, **BAD_OWNER_ADDR**, **INSUFF_BW**, **TIMEOUT** }

- ***new_owner*** - address of the new owner.

6.13.4 Parameter Negotiation

new_owner. The address of the new owner may differ between the **REQUEST** and the **RESPONSE.** The address in the **RESPONSE** is the correct address of the new owner.

6.13.5 Operation

The **change_owner** command has no effect on any endpoints that the old or new owner has in the call.

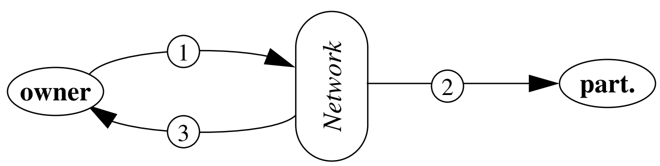
6.14 change_root Command

6.14.1 Synopsis

This operation requests that the root of a call (and hence the call identifier) be changed. It may only be requested by the owner of the call. The new root must be a participant (*i.e.*, have an endpoint) in the call.

6.14.2 Message Traffic

Figure 52 shows the message traffic. The owner initiates the operation by sending a **change_root REQUEST** (1). The network checks the message for errors, and if any are found a **change_root NACK** (3) is sent to the owner and the operation is complete. If no errors are found, the call root is changed, the owner is sent a **change_root ACK** (3), and all participants in the call are sent **announce_change_root REQUESTs** (2).



Message list:
1: change_owner REQUEST
2: announce_change_owner REQUEST
3: change_owner ACK/NACK

Client list:
owner: owner of call
part.: participant in call

Figure 52. Message Traffic for change_root

6.14.3 Message Formats

change_root REQUEST

00001101	00000000	msg_id (2)	unused (2)	unused (2)
call_id (r_addr) (24)				
call_id (lcid) (2)		unused (2)	reserved (2)	reserved (2)
m_addr (24)				
s_addr (24)				
new_call_id (r_addr) (24)				
new_call_id (lcid) (2)		reserved (2)	reserved (2)	reserved (2)
options_size (4)			options (options_size)	

Data:

- ***new_call_id (r_addr)*** - address of the new root. This must be a client with an endpoint in the call.
- ***new_call_id (lcid)*** - suggested new local call identifier. May be left blank.

change_root RESPONSE

00001101	**00000001**	**msg_id (2)**	**unused (2)**	**unused (2)**
call_id (r_addr) (24)				
call_id (lcid) (2)		**unused (2)**	**reserved (2)**	**reserved (2)**
m_addr (24)				
s_addr (24)				
new_call_id (r_addr) (24)				
new_call_id (lcid) (2)		**reserved (2)**	**reserved (2)**	**reserved (2)**
options_size (4)			**options (options_size)**	

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:
  status ∈ { **OK**, **BAD_CALL_ID_ADDR**, **DUP_CALL_ID**,
  UNKNOWN_CALL, **NOT_OWNER**, **INSUFF_BW**, **TIMEOUT** }
- ***new_call_id*** - new identifier (root address and local id) of the call.

6.14.4 Parameter Negotiation

lcid. If the owner does not supply a value for the new local call identifier, the network will select one, with preference being given to the existing local call identifier. If the owner supplies a value, the network will check that it is not already in use; if it is in use, the network will return a **NACK** (***status*** = **DUP_CALL_ID**), otherwise it will use the owner-supplied value.

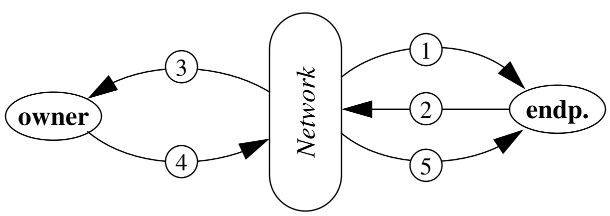
6.15 invite_add_con Prompt

6.15.1 Synopsis

This operation invites an endpoint to join one or more connections which have been added to a call in which the endpoint is participating. The endpoint is forced to join the new connections, although it may do so with a NULL mapping. The **invite_add_con** prompt is triggered by the **add_con** command.

6.15.2 Message Traffic

Figure 53 shows the message traffic. As part of a successful **add_con**, an **invite_add_con REQUEST** (1) is sent to each endpoint in the call. The endpoint may return an **invite_add_con ACK** (2), joining the connections with the suggested parameters and completing the operation. It may return an **invite_add_con NACK** (2); in this case the endpoint still joins the connections, but it will receive a NULL mapping for each new connection (*i.e.*, it can neither transmit nor receive). It may return an **invite_add_con NEG** (2), requesting a modification in some of its endpoint parameters. If verification with the owner is required (if the endpoint attempts to change its mapping and the corresponding permissions are set to **VERIFY**), the owner is sent a **verify_mod_ep REQ** (3) to which it responds (4) with a **verify_mod_ep ACK** or **NACK**. If the owner sends a **verify_add_ep ACK** (4), or if the requested changes were acceptable without verification, the endpoint is sent an **invite_add_con COM** (5) and the endpoint is added to the connections. If the owner sends a **verify_add_ep NACK** (4), or if the requested changes were not acceptable, the endpoint is sent an **invite_add_con ABORT** (5) and the endpoint is added to the connections with a NULL mapping.



Message list:
1: invite_add_con REQUEST
2: invite_add_con ACK/NACK/NEG
3: verify_mod_ep REQUEST
4: verify_mod_ep ACK/NACK
5: invite_add_con COM/ABORT

Client list:
owner: owner of call
endp.: endpoint receiving prompt

Figure 53. Message Traffic for invite_add_con

6.15.3 Message Formats

invite_add_con REQUEST

00001110	**00000000**	**msg_id (2)**		**num_cons (2)**		**unused (2)**	
call_id (r_addr) (24)							
call_id (lcid) (2)		**unused (2)**		**reserved (2)**		**reserved (2)**	
m_addr (24)							
s_addr (24)							
ep_addr (24)							
ep_id (2)		**ep_status (2)**		**reserved (4)**			
con_id (2)		**unused (2)**		**user_con_type (4)**			
con_type (1)	**reserved (1)**	**con_def (1)**	**con_perm (1)**	**bw (12)**			
ep_con_id (2)		**unused (2)**		**ep_map (1)**	**ep_def (1)**	**ep_perm (1)**	**unused (1)**
reserved (1)	**trans_vpi (1)**	**trans_vci (2)**		**reserved (1)**	**rcv_vpi (1)**	**rcv_vci (2)**	
⋮ **(num_cons)**							
options_size (4)				**options (options_size)**			

Data:

- ***num_cons*** - the number of connections that were added; equal to the number of Connection and UNI Objects.
- ***ep_addr***, ***ep_id*** - identifier of the endpoint which is being invited to join the new connections.

For each of the ***num_cons*** connections, the message has a Connection Object containing the parameters of the connection and a UNI Object containing the (suggested) per-connection endpoint parameters. Within each pair, the ***con_id*** and ***ep_con_id*** will be identical.

invite_add_con RESPONSE

00001110	00000001	msg_id (2)		num_cons (2)		unused (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		op_status (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
ep_con_id (2)		unused (2)		ep_map (1)	ep_def (1)	ep_perm (1)	unused (1)
reserved (1)	trans_vpi (1)	trans_vci (2)		reserved (1)	rcv_vpi (1)	rcv_vci (2)	
⋮ (num_cons)							
options_size (4)				options (options_size)			

Data:

- ***num_cons*** - the number of UNI objects in the message; must be less than or equal to the value of ***num_cons*** in the **REQUEST**.
- ***op_status*** - the client may set the ***status*** portion of this field to the following values (indicating, respectively, an **ACK**, **NACK**, and **NEG**):

  status ∈ { **OK**, **REFUSED**, **NEGOTIATING** }

The UNI Objects describe the per-connection parameters of the endpoint. The client must include a UNI Object for any parameter set it is negotiating. UNI Objects for other connections may be included or omitted.

invite_add_con CONFIRMATION

<table>
<tr><td>00001110</td><td>00000010</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">uni_status (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">⋮ (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - the number of UNI objects in the message; equal to ***num_cons*** in the **REQUEST**.
- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **UNKNOWN_CALL**, **VERIFY_REFUSED**, **INSUFF_BW**, **TIMEOUT** }

The UNI Objects describe the final per-connection parameters of the endpoint.

- ***uni_status*** - this field may take on the following values:

  uni_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID**, **BAD_EP_MAP**, **BAD_EP_DEF**, **ILL_EP_MAP**, **BAD_EP_PERM**, **NO_AVAIL_VPI**, **NO_AVAIL_VCI**, **TRANS_VPI_IN_USE**, **TRANS_VPI_RESERVED**, **TRANS_VPI_NOT_SUPPORTED**, **TRANS_VCI_IN_USE**, **TRANS_VCI_RESERVED**, **TRANS_VCI_NOT_SUPPORTED**, **RCV_VPI_IN_USE**, **RCV_VPI_RESERVED**, **RCV_VPI_NOT_SUPPORTED**, **RCV_VCI_IN_USE**, **RCV_VCI_RESERVED**, **RCV_VCI_NOT_SUPPORTED** }

6.15.4 Parameter Negotiation

ep_map. In the **REQUEST**, this field will equal the connection's ***con_def***. The client may attempt to modify it in the **RESPONSE**. After the usual checks (the ***ep_perm*** field and possibly verification with the owner), the network will return the final values in the **CONFIRMATION**.

trans_vpi, ***trans_vci***, ***rcv_vpi***, ***rcv_vci.*** The network will choose values for these parameters and supply them in the **REQUEST**. The client may suggest new values in the **RESPONSE**. The network returns the final values in the **CONFIRMATION**; if the pair suggested by the client is unacceptable, the network uses the pair it originally chose.

If the operation fails for any reason, including failure in negotiation, the client will receive an **invite_add_con ABORT** with all UNI objects having a NULL ***ep_map*** — even for connections that were not negotiated, or for which the negotiation was successful. The endpoint is then participating in all the new connections, but with NULL mappings. The client may then use the **mod_ep** command to change the mappings as it sees fit.

6.16 invite_add_ep Prompt

6.16.1 Synopsis

This operation invites a client to join a call as a new endpoint. The client has the option of refusing to join the call. The **invite_add_ep** prompt is triggered by the **open_call** and **add_ep** commands.

6.16.2 Message Traffic

Figure 54 shows the message traffic. As part of an **open_con** or **add_ep**, an **invite_add_ep REQUEST** (1) is sent to the client. The client may return an **invite_add_ep NACK** (2); in this case the endpoint does not join the call, and the operation is complete. The client may return an **invite_add_ep ACK** (2), joining the call with the suggested parameters. The network will then return an **invite_add_ep COM** or **ABORT** (5) to complete the operation. Finally, the client may return an **invite_add_ep NEG** (2), requesting a modification in some of its endpoint parameters. If verification with the owner is required (if the endpoint attempts to change its mapping and the corresponding permissions are set to **VERIFY**), the owner is sent a **verify_mod_ep REQ** (3) to which it responds (4) with a **verify_mod_ep ACK** or **NACK**. If the owner sends a **verify_mod_ep ACK** (4), or if the requested changes were acceptable without verification, the endpoint is sent an **invite_add_ep COM** (5) and the endpoint is added to the call. If the owner sends a **verify_mod_ep NACK** (4), or if the requested changes were not acceptable, the endpoint is sent an **invite_add_ep ABORT** (5) and the endpoint is not added to the call.

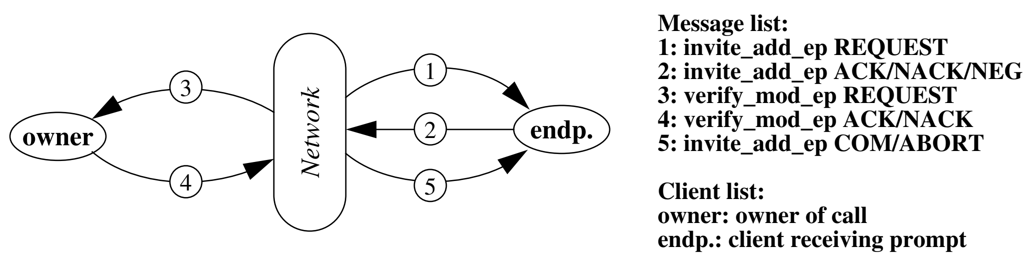


Figure 54. Message Traffic for invite_add_ep

6.16.3 Message Formats

invite_add_ep REQUEST

00001111	00000000	msg_id (2)		num_cons (2)		unused (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		unused (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
unused (2)		reserved (2)		user_call_type (4)			
call_type (1)	acc (1)	mod (1)	trace (1)	mon (1)	priority (1)	reserved (2)	
req_addr (24)							
ep_addr (24)							
ep_id (2)		unused (2)		reserved (4)			
con_id (2)		unused (2)		user_con_type (4)			
con_type (1)	reserved (1)	con_def (1)	con_perm (1)	bw (12)			
ep_con_id (2)		unused (2)		ep_map (1)	ep_def (1)	ep_perm (1)	unused (1)
reserved (1)	trans_vpi (1)	trans_vci (2)		reserved (1)	rcv_vpi (1)	rcv_vci (2)	
⋮ (num_cons)							
options_size (4)				options (options_size)			

Data:

- ***num_cons*** - the number of connections in the call; equal to the number of Connection and UNI Objects.
- ***req_addr*** - the address of the client which requested the endpoint addition.

The Call Object gives the parameters of the call.

The Endpoint Object gives the identifier of the endpoint; the ***ep_id*** field may be blank.

For each of the ***num_cons*** connections, the message has a Connection Object containing the parameters of the connection and a UNI Object containing the (suggested) per-connection endpoint parameters. Within each pair, the ***con_id*** and ***ep_con_id*** will be identical.

invite_add_ep RESPONSE

<table>
<tr><td>00001111</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">unused (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">⋮ (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - the number of UNI objects in the message; must be less than or equal to the value of ***num_cons*** in the **REQUEST**.
- ***op_status*** - the client may set the ***status*** portion of this field to the following values (indicating, respectively, an **ACK**, **NACK**, and **NEG**):

  status ∈ { **OK**, **REFUSED**, **NEGOTIATING** }

The UNI Objects describe the per-connection parameters of the endpoint. The client must include a UNI Object for each parameter set it is negotiating. UNI Objects for other connections may be included or omitted.

invite_add_ep CONFIRMATION

<table>
<tr><td>00001111</td><td>00000010</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">ep_status (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">uni_status (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">⋮ (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - the number of UNI objects in the message; equal to ***num_cons*** in the **REQUEST**.
- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **UNKNOWN_CALL**, **VERIFY_REFUSED**, **INSUFF_BW**, **TIMEOUT** }
- ***ep_status*** - this field may take on the following values:

  ep_status ∈ { **OK**, **BAD_EP_ADDR**, **DUP_EP_ID** }

The UNI Objects describe the final per-connection parameters of the endpoint.

- ***uni_status*** - this field may take on the following values:

  uni_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID**, **BAD_EP_MAP**, **ILL_EP_MAP**, **BAD_EP_DEF**, **BAD_EP_PERM**, **NO_AVAIL_VPI**, **NO_AVAIL_VCI**, **TRANS_VPI_IN_USE**, **TRANS_VPI_RESERVED**, **TRANS_VPI_NOT_SUPPORTED**, **TRANS_VCI_IN_USE**, **TRANS_VCI_RESERVED**, **TRANS_VCI_NOT_SUPPORTED**, **RCV_VPI_IN_USE**, **RCV_VPI_RESERVED**, **RCV_VPI_NOT_SUPPORTED**, **RCV_VCI_IN_USE**, **RCV_VCI_RESERVED**, **RCV_VCI_NOT_SUPPORTED** }

6.16.4 Parameter Negotiation

r_addr and ***lcid.*** The ***r_addr*** field may be equal to the ***ep_addr*** field. If it is, and if the client is not already part of the call, then this **invite_add_ep** must be for the root client as part of an **open_call** operation. The root client may then propose a new ***lcid*** value in its **RESPONSE**. If the value it proposes is already in use, the root will receive an **ABORT** and the owner will receive an **open_call NACK**, with ***status*** = **DUP_CALL_ID** in both cases.

ep_id. The client may propose a new value in its **RESPONSE**. If the value it proposes is invalid, an **ABORT** will be sent as the **CONFIRMATION**. If the field is blank, the client must return a **NEG** whether it chooses a value or not. If the client selects an acceptable value, the network will send a **COM**; if it selects an unacceptable value, the network will send an **ABORT**; and if it leaves the field blank the network will choose a value and send it in the **COM**.

ep_map. In the **REQUEST**, this field will equal the connection's ***con_def***. The client may attempt to modify it in the **RESPONSE**. After the usual checks (the ***ep_perm*** field and possibly verification with the owner), the network will return the final values in the **CONFIRMATION**.

trans_vpi, ***trans_vci***, ***rcv_vpi***, ***rcv_vci.*** The network will choose values for these parameters and supply them in the **REQUEST**. The client may suggest new values in the **RESPONSE**. The network will return the final values in the **CONFIRMATION**; if the pair suggested by the client is unacceptable, the network will use the pair it originally chose.

If the operation fails for any reason, including failure in negotiation, the client will receive an **invite_add_ep ABORT** with ***num_cons*** equal to 0. The endpoint is then not added to the call.

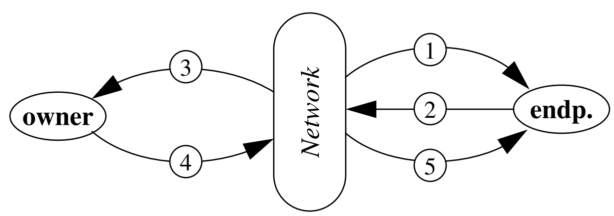
6.17 invite_mod_ep Prompt

6.17.1 Synopsis

This operation invites a client to accept changes to its endpoint mapping. The client has the option of refusing to accept the changes. The **invite_mod_ep** prompt is triggered when the **mod_ep** command is issued by the owner for another client's endpoint.

6.17.2 Message Traffic

Figure 55 shows the message traffic. As part of a **mod_ep** by the call owner, an **invite_mod_ep REQUEST** (1) is sent to the client. The client may return an **invite_mod_ep NACK** (2); in this case the endpoint parameters are not modified, and the operation is complete. The client may return an **invite_mod_ep ACK** (2), accepting the suggested parameters. The network will then return an **invite_mod_ep COM** or **ABORT** (5) to complete the operation. Finally, the client may return an **invite_mod_ep NEG** (2), requesting a modification in some of its endpoint parameters. If verification with the owner is required (if the endpoint attempts to change its mapping and the corresponding permissions are set to **VERIFY**), the owner is sent a **verify_mod_ep REQ** (3) to which it responds (4) with a **verify_mod_ep ACK** or **NACK**. If the owner sends a **verify_mod_ep ACK** (4), or if the requested changes were acceptable without verification, the endpoint is sent an **invite_mod_ep COM** (5) and the endpoint is added to the call. If the owner sends a **verify_mod_ep NACK** (4), or if the requested changes were not acceptable, the endpoint is sent an **invite_mod_ep ABORT** (5) and the endpoint is not added to the call.



Message list:
1: invite_mod_ep REQUEST
2: invite_mod_ep ACK/NACK/NEG
3: verify_mod_ep REQUEST
4: verify_mod_ep ACK/NACK
5: invite_mod_ep COM/ABORT

Client list:
owner: owner of call
endp.: client receiving prompt

Figure 55. Message Traffic for invite_mod_ep

6.17.3 Message Formats

invite_mod_ep REQUEST

<table>
<tr><td>00010000</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">req_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">unused (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">⋮ (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - the number of connections in the call; equal to the number of Connection and UNI Objects.
- ***req_addr*** - the address of the client which requested the endpoint modification.

The Endpoint Object gives the identifier of the endpoint; the ***ep_id*** field may be blank.

For each of the ***num_cons*** connections, the message has a UNI Object containing the (suggested) per-connection endpoint parameters.

invite_mod_ep RESPONSE

<table>
<tr><td>00010000</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">unused (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">⋮ (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - the number of UNI objects in the message; must be less than or equal to the value of ***num_cons*** in the **REQUEST**.
- ***op_status*** - the client may set the ***status*** portion of this field to the following values (indicating, respectively, an **ACK**, **NACK**, and **NEG**):

  status ∈ { **OK**, **REFUSED**, **NEGOTIATING** }

The UNI Objects describe the per-connection parameters of the endpoint. The client must include a UNI Object for each parameter set it is negotiating. UNI Objects for other connections may be included or omitted.

invite_mod_ep CONFIRMATION

<table>
<tr><td>00010000</td><td>00000010</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">ep_status (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">uni_status (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">⋮ (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - the number of UNI objects in the message; equal to ***num_cons*** in the **REQUEST**.
- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **BAD_NUM_CONS**, **UNKNOWN_CALL**, **VERIFY_REFUSED**, **INSUFF_BW**, **TIMEOUT** }
- ***ep_status*** - this field may take on the following values:

  ep_status ∈ { **OK**, **BAD_EP_ADDR**, **DUP_EP_ID** }

The UNI Objects describe the final per-connection parameters of the endpoint.

- ***uni_status*** - this field may take on the following values:

  uni_status ∈ { **OK**, **BAD_CON_ID**, **DUP_CON_ID**, **BAD_EP_MAP**, **ILL_EP_MAP**, **BAD_EP_DEF**, **BAD_EP_PERM**, **NO_AVAIL_VPI**, **NO_AVAIL_VCI**, **TRANS_VPI_IN_USE**, **TRANS_VPI_RESERVED**, **TRANS_VPI_NOT_SUPPORTED**, **TRANS_VCI_IN_USE**, **TRANS_VCI_RESERVED**, **TRANS_VCI_NOT_SUPPORTED**, **RCV_VPI_IN_USE**, **RCV_VPI_RESERVED**, **RCV_VPI_NOT_SUPPORTED**, **RCV_VCI_IN_USE**, **RCV_VCI_RESERVED**, **RCV_VCI_NOT_SUPPORTED** }

6.17.4 Parameter Negotiation

ep_map. The client may attempt to modify the value in the **RESPONSE**. After the usual checks (the ***ep_perm*** field and possibly verification with the owner), the network will return the final values in the **CONFIRMATION**.

trans_vpi, ***trans_vci***, ***rcv_vpi***, ***rcv_vci.*** The network will choose values for these parameters and supply them in the **REQUEST**. The client may suggest new values in the **RESPONSE**. The network will return the final values in the **CONFIRMATION**; if the pair suggested by the client is unacceptable, the network will use the pair it originally chose.

If the operation fails for any reason, including failure in negotiation, the client will receive an **invite_mod_ep ABORT** with ***num_cons*** equal to 0. The endpoint is then not added to the call.

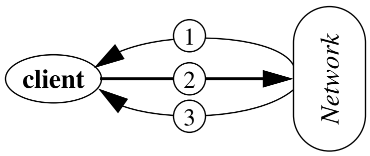
6.18 invite_change_owner Prompt

6.18.1 Synopsis

This operation invites a client to become the new owner of a call. It is triggered by a **change_owner**.

6.18.2 Message Traffic

Figure 56 shows the message traffic for **invite_chagne_owner**. The client receives an **invite_change_owner REQUEST** (1), to which it must respond (2) with an **invite_change_owner ACK** or **NACK**. The network then either confirms or cancels the operation with an **invite_change_owner COM** or **ABORT** (3).



Message list:
1: invite_change_owner REQUEST
2: invite_change_owner ACK/NACK
3: invite_change_owner COM/ABORT

Client list:
client: client receiving prompt

Figure 56. Message Traffic for invite_change_owner

6.18.3 Message Formats

invite_change_owner REQUEST

00010001	00000000	msg_id (2)		num_cons (2)		num_eps (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		op_status (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
owner_addr (24)							
unused (2)		reserved (2)		user_call_type (4)			
call_type (1)	acc (1)	mod (1)	trace (1)	mon (1)	priority (1)	reserved (2)	
con_id (2)		unused (2)		user_con_type (4)			
con_type (1)	reserved (1)	con_def (1)	con_perm (1)	bw (12)			
⋮ (num_cons)							
ep_addr (24)							
ep_id (2)		unused (2)		reserved (4)			
⋮ (num_eps)							
options_size (4)				options (options_size)			

Data:

- ***num_cons*** - the number of connections in the call, and the number of Connection Objects in the message.
- ***num_eps*** - the number of endpoints in the call, and the number of Endpoint Objects in the message.
- ***owner_addr*** - the address of the current owner of the call; also the client that requested the operation.

The Call Object contains the call parameters. The Connection Objects contain the parameters of the connections. The Endpoint Objects contain the addresses of the endpoints.

invite_change_owner RESPONSE

00010001	00000001	msg_id (2)	unused (2)	unused (2)
call_id (r_addr) (24)				
call_id (lcid) (2)		op_status (2)	reserved (2)	reserved (2)
m_addr (24)				
s_addr (24)				
options_size (4)			options (options_size)	

Data:

- ***op_status*** - the client may set the ***status*** portion of this field to the following values (indicating, respectively, an **ACK** and **NACK**):
  status ∈ { **OK**, **REFUSED** }

invite_change_owner CONFIRMATION

00010001	00000010	msg_id (2)	unused (2)	unused (2)
call_id (r_addr) (24)				
call_id (lcid) (2)		op_status (2)	reserved (2)	reserved (2)
m_addr (24)				
s_addr (24)				
options_size (4)			options (options_size)	

Data:

- ***op_status*** - the network uses the ***status*** portion of this field to indicate a commit or abort:
  status ∈ { **OK**, **REFUSED** }

6.18.4 Parameter Negotiation

There is no parameter negotiation in this prompt.

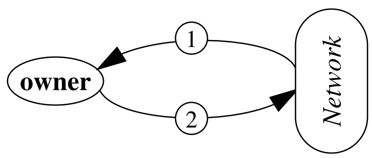
6.19 verify_add_ep Query

6.19.1 Synopsis

This operation queries the call owner to see if a new endpoint should be added to a call. It is triggered when a non-owner performs an **add_ep** operation and either (1) the call's *accessibility* (Section 4.3.5) is **VERIFY** or (2) the non-owner requests an endpoint *mapping* (Section 4.5.3) or *defaults* (Section 4.5.4) that differs from the connection's *defaults* (Section 4.4.4) and one or more of the corresponding connection *permissions* (Section 4.4.5) are **VERIFY**. The verification is performed before the endpoint is invited. See **add_ep**.

6.19.2 Message Traffic

The network traffic is shown in Figure 57. The network initiates the operation by sending a **verify_add_ep REQUEST** to the owner. If the requested addition is acceptable, the owner must respond with a **verify_add_ep ACK**. If it is unacceptable, the owner must respond with a **verify_add_ep NACK**.



Message list:
1: verify_add_ep REQUEST
2: verify_add_ep ACK/NACK

Client list:
owner: owner of call

Figure 57. Message Traffic for verify_add_ep

6.19.3 Message Formats

verify_add_ep REQUEST

<table>
<tr><td>00010010</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">req_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">unused (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">• • • (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - number of UNI Objects in the message; also number of connections in the call.
- ***req_addr*** - address of the client which requested the operation.
- ***ep_addr***, ***ep_id*** - address and identifier of the endpoint to be added.

The UNI Objects give the requested UNI parameters for the endpoint for each connection.

verify_add_ep RESPONSE

00010010	**00000001**	**msg_id (2)**	**unused (2)**	**unused (2)**
call_id (r_addr) (24)				
call_id (lcid) (2)		**op_status (2)**	**reserved (2)**	**reserved (2)**
m_addr (24)				
s_addr (24)				
options_size (4)			**options (options_size)**	

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values (indicating, respectively, an **ACK** and a **NACK**):
  status ∈ { **OK**, **REFUSED** }

6.19.4 Parameter Negotiation

There is no parameter negotiation in this command. The owner is permitted to examine the fields of the Endpoint and UNI Objects of the **REQUEST** in making its decision, but is not permitted to modify them (hence they are not returned in the **RESPONSE**). The ***op_status*** field in the **RESPONSE** is used to return an **ACK** (the value **OK**) or **NACK** (any other value, with **VERIFY_REFUSED** indicating that the owner does not accept the addition).

6.20 verify_mod_ep Query

6.20.1 Synopsis

This operation queries the call owner to see if modifications to an endpoint's parameters should be permitted. It is triggered (1) when an endpoint responds to an **invite_add_ep** or **invite_add_con REQ** with a **NEG**, requesting an endpoint *mapping* (Section 4.5.3) or *defaults* (Section 4.5.4) that differs from the connection's *defaults* (Section 4.4.4) and one or more of the corresponding connection *permissions* (Section 4.4.5) are **VERIFY**, or (2) when a non-owner endpoint issues a **mod_ep REQ** and the above conditions apply. See **open_call, invite_add_ep**, **invite_add_con**, and **mod_ep**.

6.20.2 Message Traffic

The message traffic is shown in Figure 58. The network initiates the operation by sending a **verify_mod_ep REQUEST** to the owner. If the requested addition is acceptable, the owner must respond with a **verify_mod_ep ACK**. If it is unacceptable, the owner must respond with a **verify_mod_ep NACK**.

Figure 58. Message Traffic for verify_mod_ep

6.20.3 Message Formats

verify_mod_ep REQUEST

<table>
<tr><td>00010011</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">req_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">unused (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm (1)</td><td>unused (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">• • • (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - number of UNI Objects in the message; also number of connections in the call.
- ***req_addr*** - the address of the client which requested the modification.
- ***ep_addr***, ***ep_id*** - address and identifier of the endpoint to be modified.

The UNI Objects give the requested UNI parameters for the endpoint for each connection.

verify_mod_ep RESPONSE

<table>
<tr><td>00010011</td><td>00000001</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values (indicating, respectively, an **ACK** and a **NACK**):

  status ∈ { **OK**, **REFUSED** }

6.20.4 Parameter Negotiation

There is no parameter negotiation in this command. The owner is permitted to examine the fields of the Endpoint and UNI Objects of the **REQUEST** in making its decision, but is not permitted to modify them (hence they are not returned in the **RESPONSE**). The ***op_status*** field in the **RESPONSE** is used to return an **ACK** (the value **OK**) or **NACK** (any other value, with **VERIFY_REFUSED** indicating that the owner does not accept the addition).

6.21 announce_mod_call Notification

6.21.1 Synopsis

This operation notifies an endpoint that the call parameters have been modified. See **mod_call**.

6.21.2 Message Traffic

No additional traffic results from the **announce_mod_call REQUEST**.

6.21.3 Message Formats

announce_mod_call REQUEST

00010100	00000000	msg_id (2)		unused (2)		unused (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		unused (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
unused (2)		reserved (2)		user_call_type (4)			
call_type (1)	acc (1)	mod (1)	trace (1)	mon (1)	priority (1)	reserved (2)	
options_size (4)				options (options_size)			

Data:

- ***call_type***, ***acc***, ***mod***, ***trace***, ***mon***, ***priority*** - new values of the call parameters.

6.21.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.22 announce_close_call Notification

6.22.1 Synopsis

This operation notifies the owner or an endpoint that a call has been deleted. See **drop_con**, **close_call**.

6.22.2 Message Traffic

No additional traffic results from the **announce_close_call REQUEST**.

6.22.3 Message Formats

announce_close_call REQUEST

<table>
<tr><td>00010101</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

6.22.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.23 announce_add_con Notification

6.23.1 Synopsis

This operation notifies the call owner that one or more connections have been added. See **add_con**.

6.23.2 Message Traffic

No additional traffic results from the **announce_add_con REQUEST**.

6.23.3 Message Formats

announce_add_con REQUEST

<table>
<tr><td>00010110</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">con_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">user_con_type (4)</td></tr>
<tr><td>con_type (1)</td><td>reserved (1)</td><td>con_def (1)</td><td>con_perm (1)</td><td colspan="4">bw (12)</td></tr>
<tr><td colspan="8">•
•
• (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - number of connections that were added, and number of Connection Objects in the message. The Connection Objects contain the descriptions of the connections that were added.

6.23.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.24 announce_mod_con Notification

6.24.1 Synopsis

This operation notifies a participant that one or more connections have been modified. See **mod_con**.

6.24.2 Message Traffic

No additional traffic results from the **announce_mod_con REQUEST**.

6.24.3 Message Formats

announce_mod_con REQUEST

<table>
<tr><td>00010111</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="2">con_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">user_con_type (4)</td></tr>
<tr><td>con_type (1)</td><td>reserved (1)</td><td>con_def (1)</td><td>con_perm (1)</td><td colspan="4">bw (12)</td></tr>
<tr><td colspan="8">• • • (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - number of connections that were modified, and number of Connection Objects in the message. The Connection Objects contain the new parameters of the connections that were modified.

6.24.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.25 announce_drop_con Notification

6.25.1 Synopsis

This operation notifies a participant that one or more connections have been dropped.

6.25.2 Message Traffic

No additional traffic results from the **announce_drop_con REQUEST**.

6.25.3 Message Formats

announce_drop_con REQUEST

00011000	00000000	msg_id (2)		num_cons (2)		unused (2)	
call_id (r_addr) (24)							
call_id (lcid) (2)		unused (2)		reserved (2)		reserved (2)	
m_addr (24)							
s_addr (24)							
con_id (2)		unused (2)		user_con_type (4)			
con_type (1)	reserved (1)	con_def (1)	con_perm (1)	bw (12)			
(num_cons)							
options_size (4)				options (options_size)			

Data:

- ***num_cons*** - number of connections that were dropped, and number of Connection Objects in the message. The Connection Objects contain the last parameters of the connections that were modified.

6.25.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.26 announce_add_ep Notification

6.26.1 Synopsis

This operation notifies the call owner or a participant that an endpoint has been added. See **add_ep**, **open_call**.

6.26.2 Message Traffic

No additional traffic results from the **announce_add_ep REQUEST**.

6.26.3 Message Formats

announce_add_ep REQUEST

<table>
<tr><td>00011001</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">num_cons (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="2">ep_con_id (2)</td><td colspan="2">unused (2)</td><td>ep_map (1)</td><td>ep_def (1)</td><td>ep_perm(1)</td><td>reserved (1)</td></tr>
<tr><td>reserved (1)</td><td>trans_vpi (1)</td><td colspan="2">trans_vci (2)</td><td>reserved (1)</td><td>rcv_vpi (1)</td><td colspan="2">rcv_vci (2)</td></tr>
<tr><td colspan="8">• • • (num_cons)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***num_cons*** - number of UNI Objects in the message; also the number of connections in the call.

The Endpoint Object contains the description of the endpoint that was added. Each of the ***num_cons*** UNI Objects contain the endpoint parameters for the corresponding connection of the call.

6.26.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.27 announce_mod_ep Notification

6.27.1 Synopsis

This operation notifies a client that an endpoint has been modified. See **mod_ep**.

6.27.2 Message Traffic

No additional traffic results from the **announce_mod_ep REQUEST**.

6.27.3 Message Formats

announce_mod_ep REQUEST

00011010	**00000000**	**msg_id (2)**		**num_cons (2)**		**unused (2)**	
call_id (r_addr) (24)							
call_id (lcid) (2)		**unused (2)**		**reserved (2)**		**reserved (2)**	
m_addr (24)							
s_addr (24)							
ep_addr (24)							
ep_id (2)		**unused (2)**		**reserved (4)**			
ep_con_id (2)		**unused (2)**		**ep_map (1)**	**ep_def (1)**	**ep_perm(1)**	**reserved (1)**
reserved (1)	**trans_vpi (1)**	**trans_vci (2)**		**reserved (1)**	**rcv_vpi (1)**	**rcv_vci (2)**	
(num_cons)							
options_size (4)				**options (options_size)**			

Data:

- ***num_cons*** - number of UNI Objects in the message; also the number of connections in the call.

The Endpoint Object contains the description of the endpoint that was added. Each of the ***num_cons*** UNI Objects contain the endpoint parameters for the corresponding connection of the call.

6.27.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.28 announce_drop_ep Notification

6.28.1 Synopsis

This operation notifies the call owner or a participant that an endpoint has been dropped. It is triggered by the successful execution of a **drop_ep** by a client.

6.28.2 Message Traffic

No additional traffic results from the **announce_drop_ep REQUEST**.

6.28.3 Message Formats

announce_drop_ep REQUEST

<table>
<tr><td>00011011</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">ep_addr (24)</td></tr>
<tr><td colspan="2">ep_id (2)</td><td colspan="2">unused (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

The Endpoint Object contains the identifier of the endpoint that was dropped.

6.28.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.29 announce_change_owner Notification

6.29.1 Synopsis

This operation notifies a participant that the call's owner has been changed. See **change_owner**.

6.29.2 Message Traffic

No additional traffic results from the **announce_change_owner REQUEST**.

6.29.3 Message Formats

announce_change_owner REQUEST

<table>
<tr><td>00011100</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">new_owner (24)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***new_owner*** - address of the new owner of the call.

6.29.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.30 announce_change_root Notification

6.30.1 Synopsis

This operation notifies a participant that the call's root has been changed. See **change_root**.

6.30.2 Message Traffic

No additional traffic results from the **announce_change_root REQUEST**.

6.30.3 Message Formats

announce_change_root REQUEST

<table>
<tr><td>00011101</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="8">new_call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">new_call_id (lcid) (2)</td><td colspan="2">reserved (2)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***new_call_id (r_addr)***, ***new_call_id (lcid)*** - new identifier (root address and local identifier) of the call.

6.30.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.30.5 Operation

On receiving an **announce_change_root REQUEST**, the client must update all records associated with the call to change the call identifier.

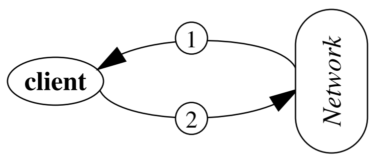
6.31 status Maintenance Operation

6.31.1 Synopsis

This two-phase operation requests the status of another operation, identified by ***msg_id***. It may be initiated by the client or by the network.

6.31.2 Message Traffic

Figure 59 shows the traffic for the network-initiated case. The network sends (1) a **status REQUEST** to the client, which responds (2) with a **status ACK**.

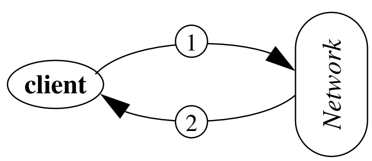


Message list:
1: status REQUEST
2: status ACK/NACK

Client list:
client: client receiving signal

Figure 59. Message Traffic for Network-Initiated status

Figure 60 shows the traffic for the client-initiated case. The client sends (1) a **status REQUEST** to the network, which responds (2) with a **status ACK**.



Message list:
1: status REQUEST
2: status ACK/NACK

Client list:
client: client sending signal

Figure 60. Message Traffic for Client-Initiated status

6.31.3 Message Formats

status REQUEST

10000000	00000000	msg_id (2)	unused (2)	unused (2)
call_id (r_addr) (24)				
call_id (lcid) (2)		unused (2)	reserved (2)	reserved (2)
m_addr (24)				
s_addr (24)				
op_msg_id (2)		unused (2)	reserved (4)	
options_size (4)			options (options_size)	

Data:

- ***op_msg_id*** - operation whose status is being requested.

status RESPONSE

10000000	00000001	msg_id (2)	unused (2)	unused (2)
call_id (r_addr) (24)				
call_id (lcid) (2)		unused (2)	reserved (2)	reserved (2)
m_addr (24)				
s_addr (24)				
op_msg_id (2)		op_msg_status (2)	reserved (4)	
options_size (4)			options (options_size)	

Data:

- ***op_msg_id*** - operation whose status is being returned.
- ***op_status*** - in messages from the network, the ***status*** portion of this field may take on the following values:
  status ∈ { **OK**, **UNKNOWN_CALL**, **TIMEOUT** }

- ***op_msg_status*** - this field may take on the following values:

op_msg_status $\in$ { **OK_RESPONSE**, **OK_CONFIRMATION**, **NO_SUCH_OPERATION** }

6.31.4 Parameter Negotiation

There is no parameter negotiation in this operation.

6.32 alert Maintenance Operation

6.32.1 Synopsis

This one-phase operation is used by either the network or a client to inform the other of the status of an operation (to, for example, inform the other side that the operation is proceeding normally).

6.32.2 Message Traffic

No additional traffic results from the **alert REQUEST**.

6.32.3 Message Formats

alert REQUEST

<table>
<tr><td>10000001</td><td>00000000</td><td>msg_id (2)</td><td>unused (2)</td><td>unused (2)</td></tr>
<tr><td colspan="5">call_id (r_addr) (24)</td></tr>
<tr><td colspan="2">call_id (lcid) (2)</td><td>unused (2)</td><td>reserved (2)</td><td>reserved (2)</td></tr>
<tr><td colspan="5">m_addr (24)</td></tr>
<tr><td colspan="5">s_addr (24)</td></tr>
<tr><td colspan="2">op_msg_id (2)</td><td>op_msg_status (2)</td><td colspan="2">reserved (4)</td></tr>
<tr><td colspan="3">options_size (4)</td><td colspan="2">options (options_size)</td></tr>
</table>

Data:

- ***op_msg_id*** - operation whose status is being reported.
- ***op_msg_status*** - this field may take on the following values:

op_msg_status ∈ { **OK_RESPONSE**, **OK_CONFIRMATION** }

6.32.4 Parameter Negotiation

There is no parameter negotiation in this operation.

6.33 client_reset Maintenance Operation

6.33.1 Synopsis

This two-phase operation informs the network that a client has been reset and all call informarion concerning it has been cleared.

6.33.2 Message Traffic

The client sends a **client_reset REQUEST** to the network, which responds with a **client_reset ACK**.

6.33.3 Message Formats

client_reset REQUEST

<table>
<tr><td>10000010</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">unused (24)</td></tr>
<tr><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***s_addr*** - client which was reset.

The ***call_id*** fields are unused in this message, since it applies to all calls in which the client is involved.

client_reset RESPONSE

10000010	00000001	msg_id (2)	unused (2)	unused (2)
unused (24)				
unused (2)		op_status (2)	reserved (2)	reserved (2)
m_addr (24)				
s_addr (24)				
options_size (4)			options (options_size)	

Data:

- ***op_status*** - the ***status*** portion of this field may take on the following values:

  status ∈ { **OK**, **TIMEOUT** }

6.33.4 Parameter Negotiation

There is no parameter negotiation in this operation.

6.34 network_reset Maintenance Operation

6.34.1 Synopsis

This one-phase operation is used by the network to inform a client that the network has been reset and all call information concerning the client has been lost.

6.34.2 Message Traffic

No additional traffic results from the **network_reset REQUEST**.

6.34.3 Message Formats

network_reset REQUEST

<table>
<tr><td>10000011</td><td>00000000</td><td colspan="2">msg_id (2)</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">unused (24)</td></tr>
<tr><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

- ***s_addr*** - client whose network was reset.

The ***call_id*** fields are unused in this message, since it applies to all calls in which the client is involved.

6.34.4 Parameter Negotiation

There is no parameter negotiation in this notification.

6.34.5 Operation

On receiving a **network_reset REQUEST**, the client disconnects itself from *all* calls in which it was involved (adjusting its internal data structures as necessary). It may then perform whatever actions are necessary to recreate the calls that were lost.

6.35 error_report Maintenance Operation

6.35.1 Synopsis

This one-phase operation is used by the network or by the client to report serious (header or formatting) errors in messages which make the processing of the message impossible.

6.35.2 Message Traffic

No additional traffic results from the **error_report REQUEST**.

6.35.3 Message Formats

error_report REQUEST

<table>
<tr><td>11111111</td><td>00000000</td><td colspan="2">0xffff</td><td colspan="2">unused (2)</td><td colspan="2">unused (2)</td></tr>
<tr><td colspan="8">unused (24)</td></tr>
<tr><td colspan="2">unused (2)</td><td colspan="2">op_status (2)</td><td colspan="2">reserved (2)</td><td colspan="2">reserved (2)</td></tr>
<tr><td colspan="8">m_addr (24)</td></tr>
<tr><td colspan="8">s_addr (24)</td></tr>
<tr><td colspan="4">message_length (4)</td><td colspan="4">reserved (4)</td></tr>
<tr><td colspan="8">message (message_length)</td></tr>
<tr><td colspan="4">options_size (4)</td><td colspan="4">options (options_size)</td></tr>
</table>

Data:

The ***msg_id*** field for an **error_report** is alway 0xffff.

The ***call_id*** fields are unused in this message, since it does not apply to any call in particular.

- ***op_status*** - this field reports on the error that occurred in the original message. The ***status*** portion of this field may take on the following values:

  status ∈ { **BAD_OP_TYPE**, **BAD_PHASE_EXP_REQUEST**, **BAD_PHASE_EXP_RESPONSE**, **BAD_PHASE_EXP_CONFIRMATION**, **DUP_MSG_ID**, **BAD_MSG_ID**, **BAD_MADDR, BAD_SADDR, FORMAT_ERROR** }
- ***message_length*** - a four-byte unsigned integer giving the length of the original message.
- ***message*** - the complete contents of the erroneous message as received by the network (no changes in any fields). The message is end-padded with 0 bytes to equal a multiple of eight bytes (so that ***options_size*** falls on an eight-byte boundary).

6.35.4 Parameter Negotiation

There is no parameter negotiation in this operation.

6.35.5 Operation

When the client receives a message which it cannot process it sends an **error_report** message and remains in the same state.

The situation when a client receives an **error_report** message is somewhat more complex. This message indicates that the client sent an erroneous message. When the client sent that message, it made a state transition; the network has not made a corresponding change of state. The correct action for the client is thus to undo its state transition. On receiving the **error_report** message from the network, the client should examine the original message to determine with which of its state transitions it was associated. It should then return to the previous state, recreate and resend the message (checking the message format to ensure that it is correct), and make the same state transition. If the error occurs again, the client should report it to a network manager and possibly to the programmer of the client application.

NB. In many cases (*e.g*., after sending an **open_call REQUEST**) the receipt of an **error_report** message is functionally equivalent to receiving a **NACK** or **ABORT** in that the client returns to its previous state. However, there is a crucial difference, in that in the **NACK** case the network processed the message and indicated that the requested action was not acceptable, while in the **error_report** case the network was unable to process the message. The two cases must thus be treated differently.

7. Examples

This section describes, through three examples, how common CMAP operations interact by demonstrating how a variety of calls can be established and dynamically change. These examples also illustrate parameter negotiation during operations. The first example is a simple data transfer, such as might be used for interprocess communication. The second example is a video server with a mute transmitter. The final example is a multipoint multiconnection conference call.

All three examples use the simple network shown in Figure 61. This network consists of four nodes ***N1***, ***N2***, ***N3*** and ***N4*** (the first interior, the others exterior) and four clients ***A***, ***B***, ***C***, and ***D***. The nodes and clients are connected by fiber links as indicated; we will use the pen pattern ▬ to represent links without calls and the pen pattern ▬ to represent links with calls throughout this section. The location of the CMAP Session Managers within this network is immaterial, as the configuration of the Session Management Layer (Section 3.1) is hidden from the clients. All four clients might be served by a single Session Manager, or each might have a separate Session Manager (even the two clients ***C*** and ***D***, which connect to the same exterior node, may send requests to different managers). The client simply "knows" that it sends and receives CMAP messages on a particular VPI/VCI pair using the CTL protocol (Section 3.4). The establishment of this configuration is not a part of CMAP's functionality,

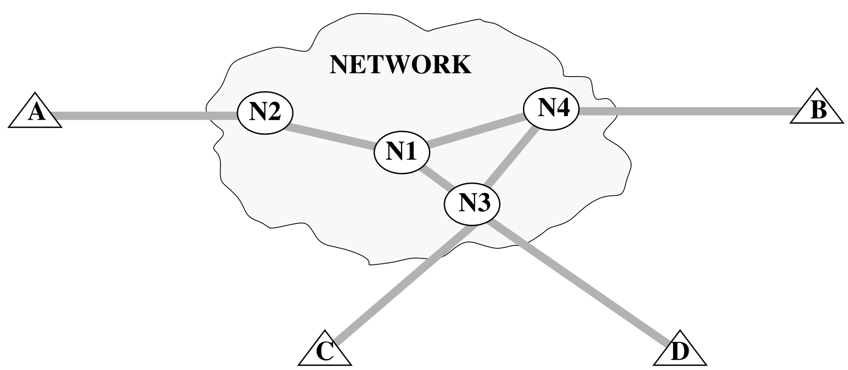


Figure 61. Network Used in Examples

7.1 Data Transfer

The first example is based on a simple data transfer, such as a "remote copy" or file-transfer operation between two computers. One client (***A*** in our sample network) wishes to transmit data to a second client (***B***). Client ***A*** must first set up a data connection between itself and ***B***, then transfer data, and finally close down the call.

7.1.1 Call Setup (Method 1)

In the first method that we illustrate, client ***A*** uses two separate commands (**open_call** and **add_ep**) to establish the data connection between itself and ***B***. One possible sequence of operations is shown in Figure 62.

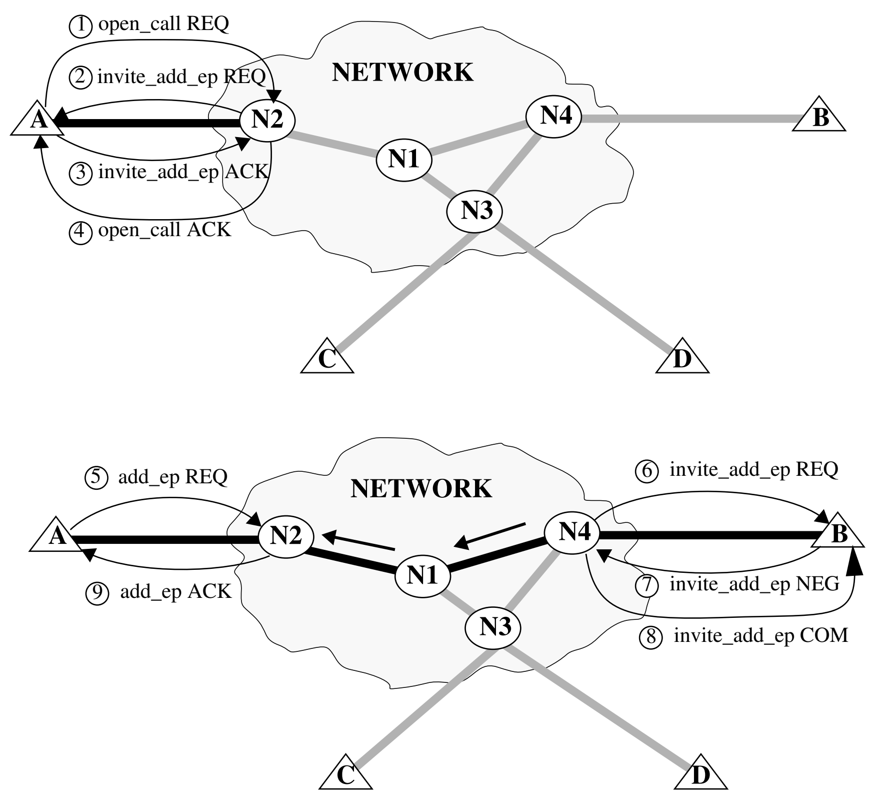


Figure 62. Building a Point-to-Point Call with Two Commands

Client ***A*** first sends 1 an **open_call REQUEST** to the network. The message might look like that in Figure 63. Notable aspects of this message are:

- ***num_cons*** = 1; there is only one connection in this datagram call.
- ***num_eps*** = 1; the **open_call** is only adding the root.
- ***r_addr*** = client ***A***; client ***A*** will be the root.
- ***lcid*** = blank; client ***A*** is willing to allow the network to assign a local call identifier.
- ***user_call_type*** = **IP_DATAGRAM**; this is actually a symbolic constant known to both client ***A*** and ***B***.
- ***call_type*** = **POINT_TO_POINT**; this is a point-to-point call.
- ***mod*** = **CLOSED**; non-owners are not allowed to add or modify endpoints.
- ***con_id*** = blank; client ***A*** is willing to allow the network to assign a connection identifier.
- ***con_type*** = <**VC**, **DYNAMIC**, **HIGH**>; client ***A*** wants a virtual channel connection.
- ***con_def*** = <**ON**, **OFF**, **OFF**>, ***con_perm*** = <**OFF**, **OFF**, **OFF**>; by default, new endpoints can receive but not transmit or echo, and cannot change their mapping.

00000001	00000000	0x0001		1		1	
client A							
blank		0		0		0	
client A							
client A							
0		0		IP_DATAGRAM			
P-TO-P	CLOSED	CLOSED	CLOSED	OF,OF,OF	NORMAL	0	
blank		0		IP_DATAGRAM_CON			
VC,DY,HI	0	ON,OF,OF	ON,ON,ON	0, 0, 0 (BEST-EFFORT)			
client A							
0		0		0			
blank		0		OF,ON,OF	OF,ON,OF	OF,OF,OF	0
0	blank	blank		0	blank	blank	
0							

Figure 63. open_call Message for Data Transfer Example

- ***ep_id*** = 0; client ***A*** is selecting an endpoint identifier.
- ***ep_con_id*** = blank; this is required, since the ***con_id*** is also blank.
- ***ep_map*** = ***ep_def*** = **<OFF**, **ON**, **OFF>**; client ***A*** is overriding the connection defaults and assigning itself a transmit-only mapping.
- ***trans_vpi*** = ***trans_vci*** = ***rcv_vpi*** = ***rcv_vci*** = blank; client ***A*** is willing to allow the network to assign pairs.

When this message is received by the network (CMAP Session Manager), it is checked for correctness (*e.g.*, that the ***ep_id*** is not in use). The network selects values for the blank fields—possibly ***lcid*** = 0, ***con_id*** = ***ep_con_id*** = 1, transmit pair = 3/10, and receive pair = 3/11. It then sends 2 an **invite_add_ep REQUEST** to client ***A*** asking the client to join the call with these parameters. Assuming the parameters are acceptable, client ***A*** responds 3 with an **invite_add_ep ACK**. The network establishes the ATM connections between client ***A*** and its exterior node ***N2*** (solid line in Figure 62 top), although for the time being these connections do not lead anywhere. The network then sends 4 an **open_call ACK** to client ***A***. The call, with one connection and endpoint, now exists within the network.

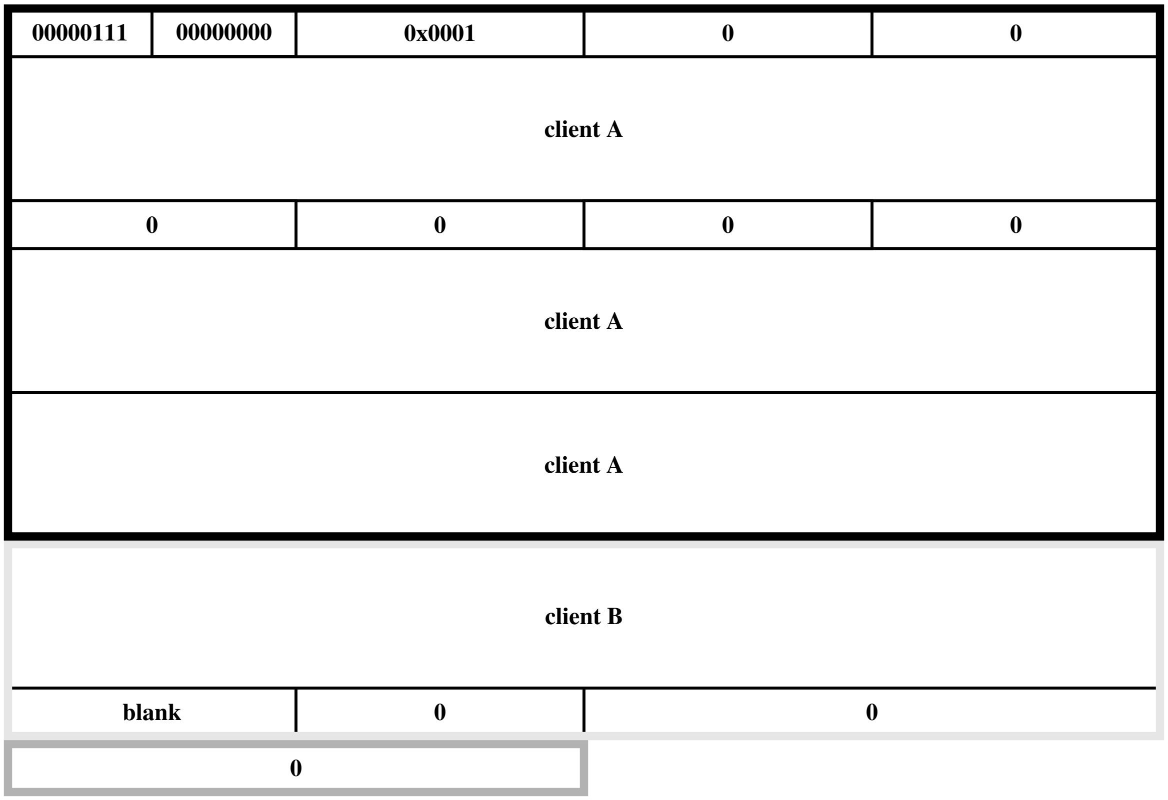


Figure 64. add_ep Message for Data Transfer Example

The call setup in Figure 62 then continues when client ***A*** sends 5 an **add_ep REQUEST** to the network asking that client B be added to the call. The mechanism whereby client ***A*** discovers client ***B***'s address does not fall within the scope of CMAP. One possible method would be a "name-server" client with a well-known CMAP address; client ***A*** could connect to this client and request the address of client ***B***.

The **add_ep REQUEST** message might look like that in Figure 64. Notable aspects of this message are:

- ***msg_id*** = 0x0001; since the **open_call** is complete, client ***A*** is free to re-use this message identifier. (Of course, any unused identifier could have been used in this message.)
- ***num_cons*** = 0; client ***A*** is not supplying any UNI Objects for client ***B***'s parameters.
- ***ep_id*** = blank; client ***A*** is willing to allow client ***B*** or the network to assign this value.

After receiving and checking this message, the network begins reserving bandwidth for the connection. How this is done depends somewhat on the algorithms used in the Connection Management Layer (Section 3.5). One reasonably-efficient algorithm [64] is illustrated by the arrows in the lower portion of Figure 62. The network routes "toward the root", working from ***B***'s exterior node ***N4*** toward ***A***'s node ***N2***, reserving bandwidth on each link and within each node. If there are any problems (*e.g.*, insufficient bandwidth) the network will send an **add_ep NACK** to client ***A***.

Assuming that the needed bandwidth can be reserved, the network next sends 6 an **invite_add_ep REQUEST** to client ***B***. Since client ***A*** did not supply any UNI Objects in the **add_ep REQUEST**, the **invite_add_ep REQUEST** will select values for client ***B***'s UNI parameters for connection 1. Specifically:

- ***ep_map*** = ***ep_def*** = <**ON**, **OFF**, **OFF**>, ***ep_perm*** = <**OFF**, **OFF**, **OFF**>; these parameters are taken from the ***con_def*** and ***con_perm*** of the connection. The ***ep_perm*** value prevents client ***B*** from negotiating its mapping.
- transmit pair = 3/23, receive pair = 3/24; these are chosen by the network and offered to client ***B***, which may negotiate them.

In addition, the network will present the blank ***ep_id*** value to client ***B***. Client ***B*** may select a value for this field, or return a blank value; in either case, it must respond 7 with an **invite_add_ep NEG** (if it selects a value, the network must approve it; if it does not select a value, the network must select one and return it to ***B***). Had client ***A*** provided a value for client ***B***'s ***ep_id***, the network would have presented it to ***B***, which could then accept the value and simply return an **invite_add_ep ACK**. Of course, if in the latter case client ***B*** wished to negotiate new VPI/VCI pairs it would still return an **invite_add_ep NEG**. We may assume that client ***B*** proposes an acceptable value (say, ***ep_id*** = 0) in its response. The network will then finalize the reserved connection and return 8 an **invite_add_ep COM** to client ***B*** and 9 an **add_ep ACK** to client ***A***.

7.1.2 Call Setup (Method 2)

The second method of setting up the call uses the single **open_call** command to establish the data conection between clients ***A*** and ***B***. One possible sequence of operations is shown in Figure 65.

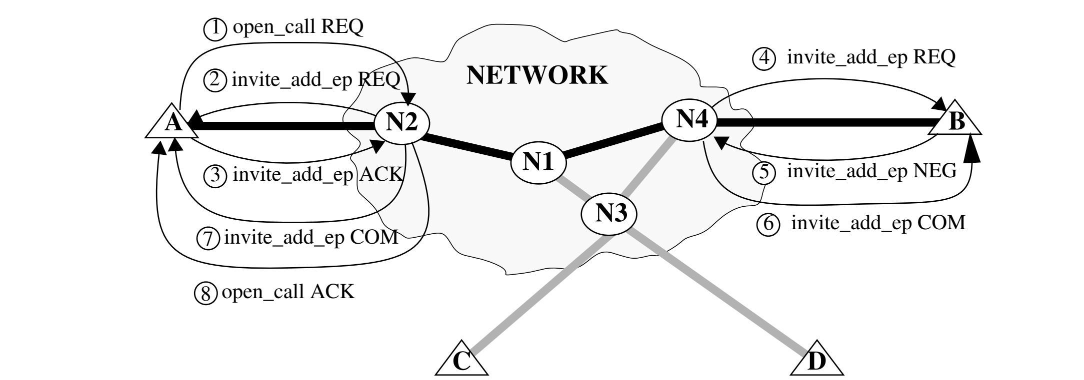


Figure 65. Building a Point-to-Point Call with One Command

Client ***A*** first sends 1 an **open_call REQUEST** to the network. This message differs from the previous one (Figure 63) in that ***num_eps*** = 2 and client ***A*** provides the address of client ***B*** and its UNI Object for the connection in the **open_call REQUEST** (***B***'s UNI Object will still specify a receive-only mapping). This single message thus combines the previous two.

During execution of the command, both clients ***A*** and ***B*** will be sent **invite_add_ep REQUESTs** (2 and 4). The addition of the two endpoints occurs in parallel, and both must succeed for the operation to succeed. If, for example, client ***A*** sends 3 an **ACK** while client ***B*** sends 5 a **NACK**, client ***A*** will receive 7 an **ABORT** to tell it not to join the connection; it its other capacity as owner of the call, client ***A*** will also receive 8 an **open_call NACK**. If both endpoints are successfully added, both will receive **CONFs** and client ***A*** will receive an **open_call ACK**.

7.1.3 Data Transmission

Once the call is established, data transfer may begin. Data transfer protocols are outside the scope of CMAP, which only provides ATM connections between endpoints. Clients are free to use any protocol they wish in sending data. Note that in many cases (for example, an IP protocol, where the data sink detects missing or erroneous packets and requests retransmission from the source) this may involve bidirectional communication between endpoints. The setup for our simple unidirectional connection would have to be modified accordingly. Similarly, careful selection of call and connection parameters (priority, QOS, bandwidth) may be required to support certain protocols.

One aspect of data transmission is general enough to be mentioned here. Assume that both clients in our example can transmit and receive. Neither party should transmit until receiving the last message (8 and 9 in Figure 62) from the network, since the connection is not known to be established until this time. Since these last two messages are independent and can be received in any relative order, either party could begin receiving data before receiving the last message. Thus, client ***A*** must be prepared to receive data as soon as it sends 5 the **add_ep REQUEST**, and client ***B*** must be prepared to receive data as soon as it sends 7 the **invite_add_ep RESPONSE**. Similar precautions apply to the call setup depicted in Figure 65—neither party should transmit until it receives its last message (6 and 8), but either may receive data anytime after sending its last message (3 and 5). This precaution also applies to the other examples in this section; clients cannot reliably transmit until after the message that confirms they have been added to the call, but they may have to receive anytime after the last CMAP message that they send.

7.1.4 Call Closedown

Once data transfer is complete, client ***A*** closes down the call by sending a **close_call REQUEST** which might look like that in Figure 66. This causes an **announce_close_call REQUEST** to be sent to each endpoint (including client ***A***'s) and a **close_call ACK** to be returned to client ***A***. The network also begins tearing down the call and freeing the resources used by the call. The latter operation may take some time and the connections may actually remain valid for a short period, but clients should not send any data after receiving the **announce_close_call REQUESTs**.

<table>
<tr><td>00000011</td><td>00000000</td><td>0x07A1</td><td>0</td><td>0</td></tr>
<tr><td colspan="5">client A</td></tr>
<tr><td colspan="2">0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td colspan="5">client A</td></tr>
<tr><td colspan="5">client A</td></tr>
<tr><td colspan="3">0</td><td></td><td></td></tr>
</table>

Figure 66. close_call Message for Data Transfer Example

7.2 Audio/Video Server

In this example, we assume that one of the clients (***A*** in our sample network) is a mute source of audio and video data—it may be a simple camera, or some more complex piece of equipment. Another client (***C***) is set up as the surrogate (Section 3.4.2) for this mute client and will handle all its signalling. The manner in which network managment establishes such surrogate signalling is outside the scope of CMAP. In this example, we will set up a point-to-multipoint call whereby data from the mute client ***A*** can be distributed to other clients. We will assume that for some reason (security, perhaps) access to this data is to be controlled and monitored by the surrogate ***C***.

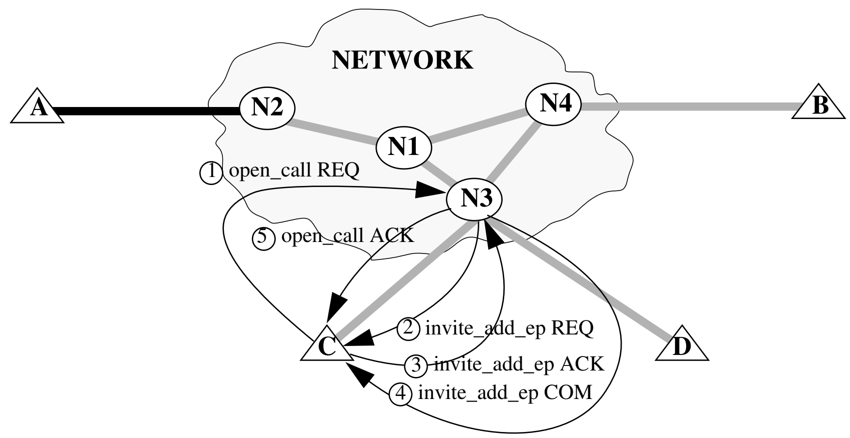


Figure 67. Setup for the Audio/Video Server

7.2.1 Call Setup

The sequence of operations in the call setup is shown in Figure 67. Client ***C*** creates a call rooted at the mute transmitter client ***A*** by sending 1 an **open_call REQUEST** as shown in Figure 68. Note the following:

- ***num_cons*** = 2; there are separate audio and video connections in this call.
- ***num_eps*** = 1; only the root (client ***A***) is in the call initially.
- ***r_addr*** = client ***A***; the root is at client ***A***.
- ***s_addr*** = client ***C***; client ***C*** is the one sending the **open_call** signal, and will thus be the owner of the call. This is discussed in greater detail below.
- ***call_type*** = **MULTIPOINT**; any number of clients may participate in the call, and all will receive data transmitted by any of the other clients.
- ***acc*** = **VERIFY**; attempts by clients to join the call or add other clients will be checked by the owner.
- ***mon*** = **<ON**, **OFF**, **OFF>**; the owner will be notified of all client joins, modifications, and drops.
- ***con_def*** = **<ON**, **OFF**, **OFF>** and ***con_perm*** = **<OFF**, **OFF**, **OFF>** for both connections; by default, new clients are receive-only and cannot change their mapping.
- ***bw*** has very specific values for both connections, reflecting the requirements of the audio and video transmission hardware. Similarly, ***con_type*** requests **STATIC**, **HIGH**-quality connections.
- ***ep_map*** = ***ep_def*** = **<OFF**, **ON**, **OFF>** for both connections; the owner is overriding the connection defaults and making itself a transmit-only endpoint.
- VPI/VCI pairs for client ***A*** are specified. This may be required by the mute client's hardware, for example the signal-to-ATM transcoders may produce cells with fixed headers.

The matter of the call ownership deserves further comment. In our setup, the ***r_addr*** field of the **open_call** must be client ***A***, since the call (and all data transmissions) will be rooted there. Similarly, the ***m_addr*** field must be client ***C***, since ***C*** is the client actually sending the CMAP message. However, the ***s_addr*** field in our example could be either client ***A*** or client ***C***. Whichever is selected becomes the owner of the call.

00000001	00000000	0x0001		2		1	
client A							
0		0		0		0	
client C							
client C							
0		0		PAY_PER_VIEW			
MULTIPT	VERIFY	CLOSED	CLOSED	ON,OF,OF	NORMAL	0	
0		0		PAY_PER_VIEW_AUDIO_CON			
VC,ST,HI	0	ON,OF,OF	OF,OF,OF	100, 100, 0			
1		0		PAY_PER_VIEW_VIDEO_CON			
VC,ST,HI	0	ON,OF,OF	OF,OF,OF	25000, 25000, 0			
client A							
0		0		0			
0		0		OF,ON,OF	OF,ON,OF	OF,OF,OF	0
0	5	20		0	5	20	
1		0		OF,ON,OF	OF,ON,OF	OF,OF,OF	0
0	5	21		0	5	21	
0							

Figure 68. open_call Message for Video Server Example

The message in Figure 68 makes client ***C*** the owner, and thus all management operations (*e.g.*, **verify_add_ep**) will be directed to client ***C***. If instead the ***s_addr*** field were client ***A***, client ***A*** would be the owner; however, all messages sent to the mute client ***A*** would be redirected to its surrogate, client ***C***. The overall effect is thus the same in either case, although the specific processing of messages within the network might differ significantly.

As a final note, client ***C*** is only allowed to put the address of client ***A*** into the ***s_addr*** field of the **open_call** because client ***C*** is the surrogate for client ***A***. It is not permissible for client ***C*** to put the address of a client for which it was not the surrogate into the ***s_addr*** field. This applies to all operations, of course; client ***X*** is not allowed to signal for client ***Y*** (putting the address of ***Y*** into the ***s_addr*** field of a message) unless client ***X*** is the surrogate for client ***Y***.

After receiving the request, the network invites 2 client ***A*** to join the call. Because client *A* is mute the invitation is redirected to its surrogate—client ***C***. In this message, ***m_addr*** = client ***C*** (the client to which the *message* is directed), while ***s_addr*** = client ***A*** (the client to which the *signal* is directed). Client ***C*** accepts on behalf of client ***A***, sending 3 an **ACK** in which ***m_addr*** = client ***C*** and ***s_addr*** = client ***A***. The network signals 4 to client ***A*** that the connections to the endpoint were established and signals 3 the owner that the call wass created. Initially the only connection is from client ***A*** to its exterior node. Client ***A*** may be sending data at this point, but the network discards it.

7.2.2 First Client Joins

Assume now that client ***B*** wishes to join the call. The mechanism whereby client ***B*** finds out about the existence of the call, its identifiers, and its parameters is not within the scope of CMAP. One possible method would be for client ***C***, once it has created the call, to "publish" the call identifier in some manner, possibly registering it with some well-known "video directory" client.

The signalling sequence is shown in Figure 69. Client ***B*** sends 1 an **add_ep REQUEST** asking that it be added to the call. For simplicity, we will assume that client ***B*** specifies all its endpoint parameters in this **add_ep** and an **invite_add_ep** prompt is thus not required. Because the call's accessibility parameter is **VERIFY**, the network sends 2 a **verify_add_ep** prompt to the owner, client ***C***. Assuming that client ***C*** responds 3 affirmatively, the network next reserves sufficient resources for the connection from the root ***A*** to the client ***B***. Once the resources are reserved, the network finalizes the reserved connections, sends 4 an **add_ep ACK** to client ***B*** and (in accord with the call's monitoring parameter) sends 5 an **announce_add_ep REQUEST** to client ***A***.

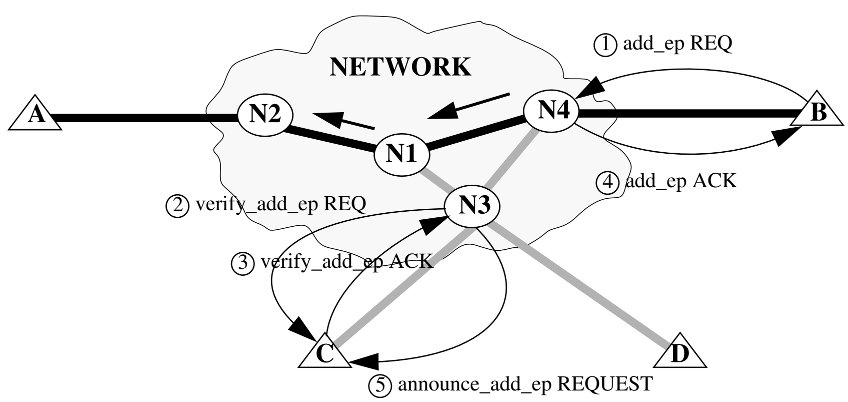


Figure 69. First Client Joins Video Server Call

The above operation could have had several other outcomes. For example, client ***C*** could have refused to permit client ***B*** to join the call (*e.g.*, if client ***B*** was not a subscriber to the service, or did not have the necessary security classification to receive the data). Another possibility might be that client ***B*** proposed improper values (*e.g.*, it asked to map in as a transmitter, which is forbidden by the connection defaults and permissions). Finally, there is always the possibility that the network was unable to support the connections from ***A*** to ***B***, due to existing network traffic. In all of these cases client ***B*** would receive a **NACK** containing the reason it could not join the call (**VERIFY_REFUSED**, **ILL_EP_MAP**, **INSUFF_BW**, *etc.*).

7.2.3 Second Client Joins

Assume now that client ***D*** wishes to join the call. The signalling sequence (shown in Figure 70) is identical to that of Figure 69. Three points are noteworthy:

- The transition from a point-to-point call to a point-to-multipoint call is *seamless*, in that the operation that adds the third endpoint (**add_ep**) is the same operation that added the second endpoint.
- Endpoints may be added while existing endpoints are communicating. The addition of client ***D*** does not interfere with client ***B***'s reception (assuming, of course, that the physical network and the Connection Management Layer can support this type of addition).

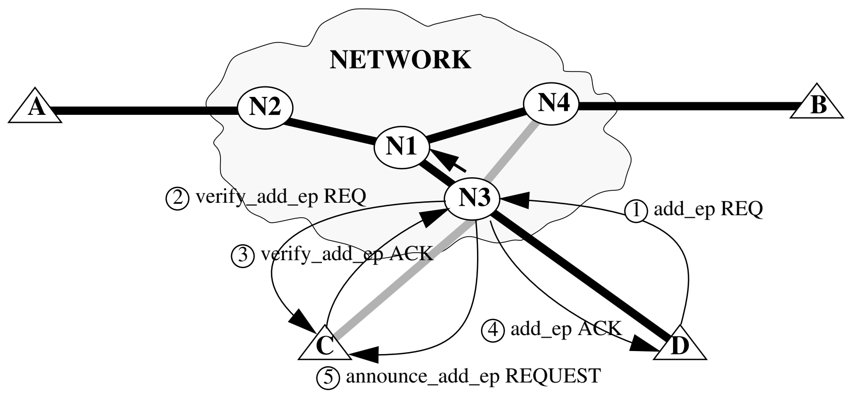


Figure 70. Second Client Joins Video Server Call

- The "toward-the-root" routing algorithm can stop when it reaches any node that is already in the call. In this case, the algorithm reserves bandwidth between ***D*** and ***N3*** and between ***N3*** and ***N1***. Node ***N1*** is already in the call and the algorithm stops. When the new connection is finalized, the Connection Management software only needs to set up ***N1*** so that all cells arriving from the "upstream" (***A***) side are duplicated and one copy sent to each "downstream" (***B***, ***D***) side.

7.2.4 Client Drops Out

Figure 71 illustrates what happens when client ***B*** drops out of the call. Client ***B*** sends 1 a **drop_ep REQUEST** for itself. The network responds 2 with a **drop_ep ACK** and (in accord with the call's monitoring parameter) sends 3 an **announce_drop_ep REQUEST** to the owner. The network also tears down the connection from client ***B*** to client ***A***, leaving any links that are used by other endpoints in the call in place.

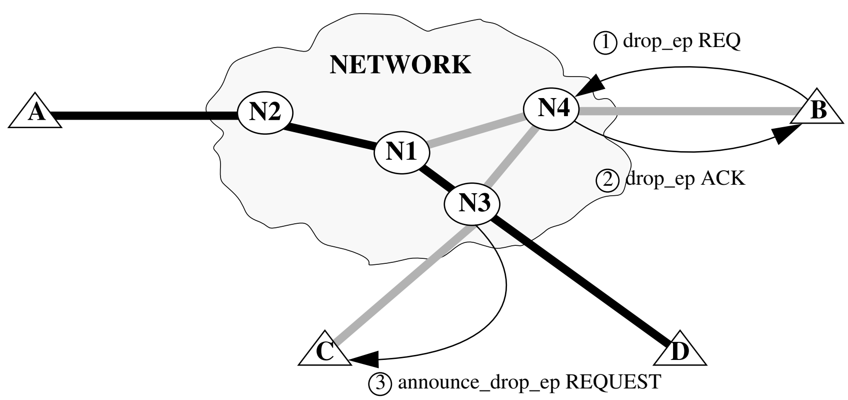


Figure 71. Client Drops Out of Video Server Call

By setting the monitoring parameter for the owner to **ON**, client ***C*** guarantees that it will receive notification of all endpoint adds, modifications, and drops. This facility could be used in a variety of ways. One of the most obvious is in the area of service billing—the owner is able to keep track of when other clients join and leave the call, and can thus charge them for use of the call. Of course, the current facilities are arguably imprecise, in that the service time that the owner computes would have to be based on the times at which it received the notifications, which may not relate closely to the actual times when endpoints were added or dropped. A more complete timestamping facility could be added as an extension to CMAP, using the ***options*** field in the Trailer Object.

7.3 Conference Call

In this example, we assume that the clients wish to engage in a multimedia conference call. Each client might be a workstation equipped with an MMX [50], cameras, microphones, and other hardware which allows the user of the workstation to transmit a compressed audio/video data stream to other users at other workstations. Specialized client software manages the conference call according to the following protocol:

- All users may transmit simultaneously. However, each user may only receive the transmissions from one user at a time. Users are allowed to receive their own transmission (but this shuts out others).
- Any existing user may invite another user to join the call.
- Any user may drop out at any time. If all the users drop out, the call is to be terminated.
- Each workstation has a user-interface process which both handles the receipt and transmission of data and provides controls whereby the user manages his end of the call (by selecting which transmission to view, *etc*.).
- The user interface is to provide a visual indication of what users are in the call and at what transmitter each user is looking.
- The user interface must also provide a means whereby the user can signal that he wishes to speak, and an indication of what users wish to speak.

The purpose of this section is to examine how such a conference call might be implemented in CMAP. We begin with an overview of the way in which the above setup might be mapped to the CMAP call model, then provide examples of the call operations.

7.3.1 Use of CMAP to Support Conference Call

Each of the user interface processes will be a separate CMAP client. For purposes of the conference-call protocol, each user will be internally identified by a small integer (starting with 1). This integer will also be used with the CMAP operations as described below. Each user-interface client has one endpoint in the call.

One of the user interface processes will act as the owner of the CMAP call. This process will create the call and enforce the requirement that users may join the call only if they are invited by a user already in the call. If the user with the owner process drops out of the conference call, the ownership of the call will be transferred to some other user interface process.

The call will contain a *control connection* with identifier 0. This is is used by the user interface processes to communicate any control information they need (*e.g.*, to indicate which user each user is viewing, or to signal that a user wishes to become the speaker). This connection will be multipoint-to-multipoint, with any transmission by an endpoint being received by all other users. The connection type will be VP, allowing clients to use the VCI field for source discrimination (as described in Section 2.3); the unique conference-call identifiers will be used for the VCI values.

The CMAP call will also contain one point-to-multipoint connection for each user in the call. This connection is the one on which that user transmits audio/video data. When a user joins the call, it adds its connection; when the user later drops out, it first removes the connection. The connection identifier will be equal to the user's conference-call identifier (actually, a simple way to assign a unique conference-call identifier is to allow the network to choose a unique connection identifier, then use that value). The mapping for the user's endpoint to its own video connection will be either transmit-only (if the user is viewing some other user's transmission) or transmit-with-echo (if the user is viewing its own transmission)*. The mapping for the user's endpoint to any other video connection will be either receive-only (if the user is viewing that other connection) or receive-hold (if the user is not viewing the other connection). For any user, at most one video-connection mapping will be transmit-with-echo or receive-only at any time, since the user can only view one other user (or itself) at a time.

7.3.2 Call Setup

Assume client ***A*** initiates the conference call. Figure 72 shows the **open_call REQUEST**. Note the following:

- ***num_cons*** = 1; ***A*** is creating only the control connection. Its audio/video connection will be added later.

* Some reduction in network traffic could be attained if clients do not transmit unless at least one other client is receiving them. A user could then have a transmit-hold mapping for its transmission connection.

00000001	00000000	0x0001		1		1	
client A							
0		0		0		0	
client A							
client A							
0		0		MMX_CONFERENCE_CALL			
MULTIPT	VERIFY	OPEN	MEMBERS	ON,OF,ON	NORMAL	0	
0		0		MMX_CONTROL_CON			
VP,DY,HI	0	ON,ON,OF	OF,OF,OF	0, 0, 0			
client A							
0		0		0			
0		0		ON,ON,OF	ON,ON,OF	OF,OF,OF	0
0	blank	blank		0	blank	blank	
0							

Figure 72. open_call Message for Conference Call Example

- ***num_eps*** = 1; ***A*** is only adding itself, as the root.
- ***acc*** = **VERIFY**; the owner will approve endpoint additions. This will be used to enforce the policy that only clients already in the call may invite new clients (the owner can reject any addition request by a non-owner).
- ***mod*** = **OPEN**; participants in the call may add connections. This is needed so each client can add its own audio/video connection as it joins.
- ***trace*** = **MEMBERS**; participants in the call may perform trace operations. This may not be directly required (the control connection can be used by clients to obtain similar information), but it may be useful.
- ***mon*** = <**ON**, **OFF**, **ON**>; the owner and all participants will be informed of endpoint additions, modifications, and drops. Again, this may not be required, as the control connection could perform some of the same functions.
- ***con_type*** = <**VP**, **DYNAMIC**, **HIGH**>; the connection is virtual path. Clients may use the VCI field of the header for source discrimination as described above.

- ***con_id*** = 0 and ***user_con_type*** = **MMX_CONTROL_CON**; this will allow ready identification of the vital control connection.
- ***con_def*** = **<ON**, **ON**, **OFF>**, meaning that all endpoints are able to transmit and receive on the control connection. ***con_perm*** = **<OFF**, **OFF**, **OFF>**, meaning that endpoints can't change the mapping (of course, they probably wouldn't want to change it).

7.3.3 Addition of Connection

Once the call has been created, client ***A*** next adds its audio/video connection to the call. This is accomplished with an **add_con REQUEST** as shown in Figure 73. Each other client will use a similar request to add its own audio/video connection as it joins the call. Note the following:

- ***con_id*** = ***ep_con_id*** = blank; the network will select an unused identifier for the connection and return it in the **add_con RESPONSE**. The client will then use this value as its conference-call unique identifier.
- ***con_def*** = **<OFF**, **OFF**, **OFF>**, ***con_perm*** = **<ON**, **OFF**, **OFF>**; clients will be offered a NULL mapping by default but may change it to a receive-only mapping.
- ***ep_map*** = **<OFF**, **ON**, **ON>**, ***ep_def*** = **<OFF**, **ON**, **ON>**, ***ep_perm*** = **<OFF**, **ON**, **ON>**; for its own endpoint, client ***A*** is given a transmit-with-echo mapping and can turn off transmission or echo.

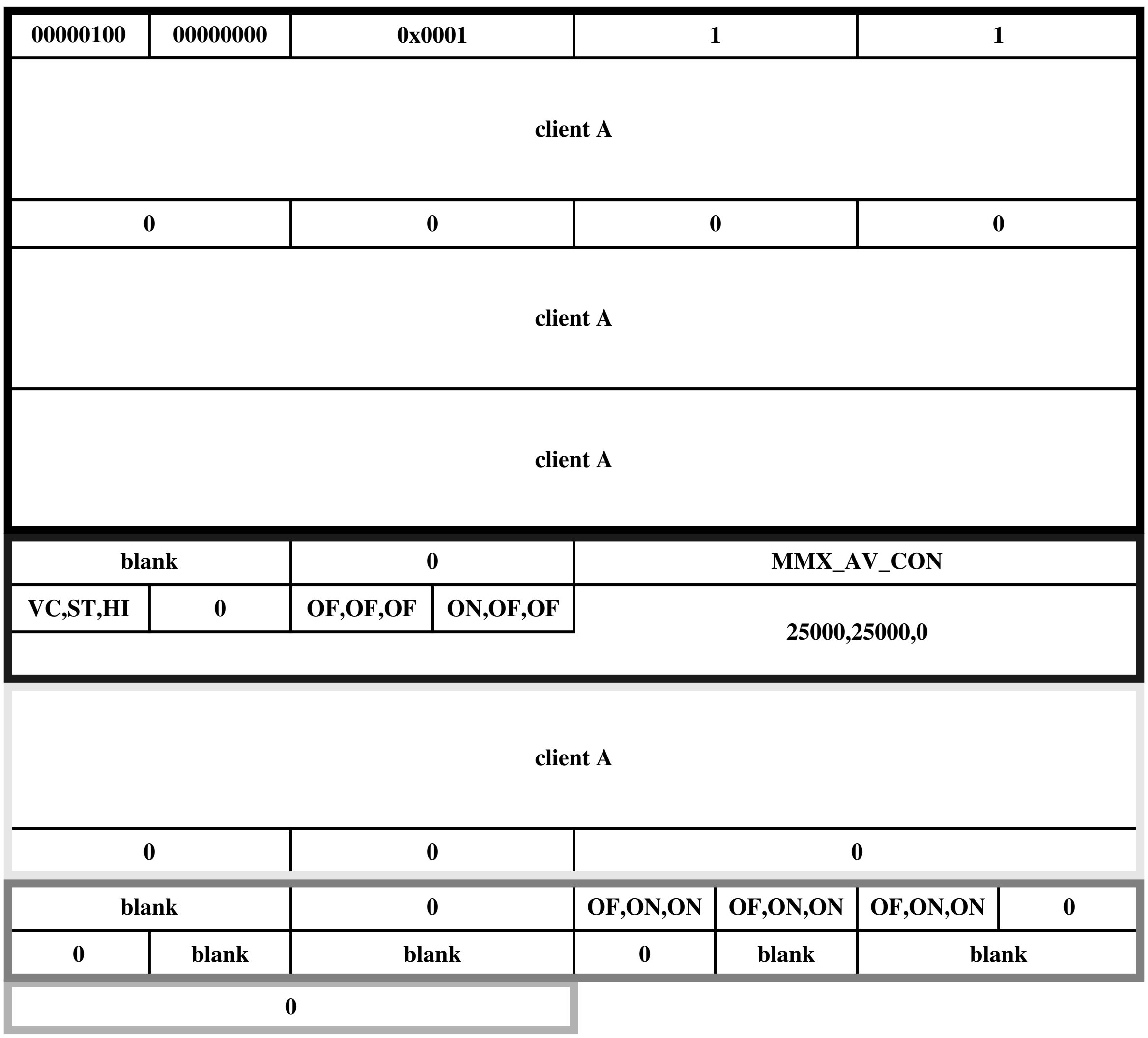


Figure 73. add_con Message for Conference Call Example

7.3.4 Addition of New User to Conference Call

New users may be added to the call by users who are already in the call. Assume that client ***A*** is the owner of the call, client ***B*** is also in the call, and ***B*** wishes to add a new client ***C*** to the call. The sequence of operations is then as follows:

- Client ***B*** sends an **add_ep REQUEST** to the network, naming client ***C***.
- Since the call accessibility is **VERIFY**, the network sends a **verify_add_ep REQUEST** to client ***A***. This **REQUEST** contains both the address of the new client ***C*** and that of the requesting client ***B***.
- ***A*** checks the requesting client. Finding that it is a member of the call, it sends a **verify_add_ep ACK** to approve the addition of client ***C***.
- The network sends an **invite_add_ep REQUEST** to client ***C***, which decides whether to join or not. (We can imagine that upon receiving the **REQUEST** the user interface displays an icon and/or makes a "ringing" noise to get the user's attention. The user would then indicate whether to accept the call.)

If some client not in the call, say ***D***, tried to add itself or another client, the owner ***A*** would detect this and send a **NACK** for the **verify_add_ep REQUEST**. The network would then not invite the client to join the call.

When the new client ***C*** joins the call, it will accept the default NULL mapping for each of the video connections. It will then add its own connection, set to transmit-with-echo (the client initially views its own transmissions). The client may then request and receive information about the call status over the control connection. This data will include the information needed for the visual interface (users in the call, which users are viewing which transmissions, *etc.*) and may include information suggesting a video connection that it should view—reasonable choices would be either that of the client ***B*** that invited the new client, or whichever transmission ***B*** itself was viewing.

7.3.5 User Changes View

Figure 74 depicts a possible state of the conference call with three users ***A***, ***B***, and ***C***. All three users have the receive/transmit mapping for the control connection and the transmit-only mapping for their own video connection (1 for ***A***, 2 for ***B***, and 3 for ***C***). Clients ***A*** and ***B*** are viewing (receive-only mapping) client ***C***, and client ***C*** is viewing client ***B***. (Since no one is viewing client ***A***, it could also use a transmit-hold mapping for its connection.) Assume now that client ***B*** wishes to view client ***A*** instead (*e.g.*, the user at client ***B*** may have "clicked" in the visual display to indicate that client ***A***'s transmission should be displayed).

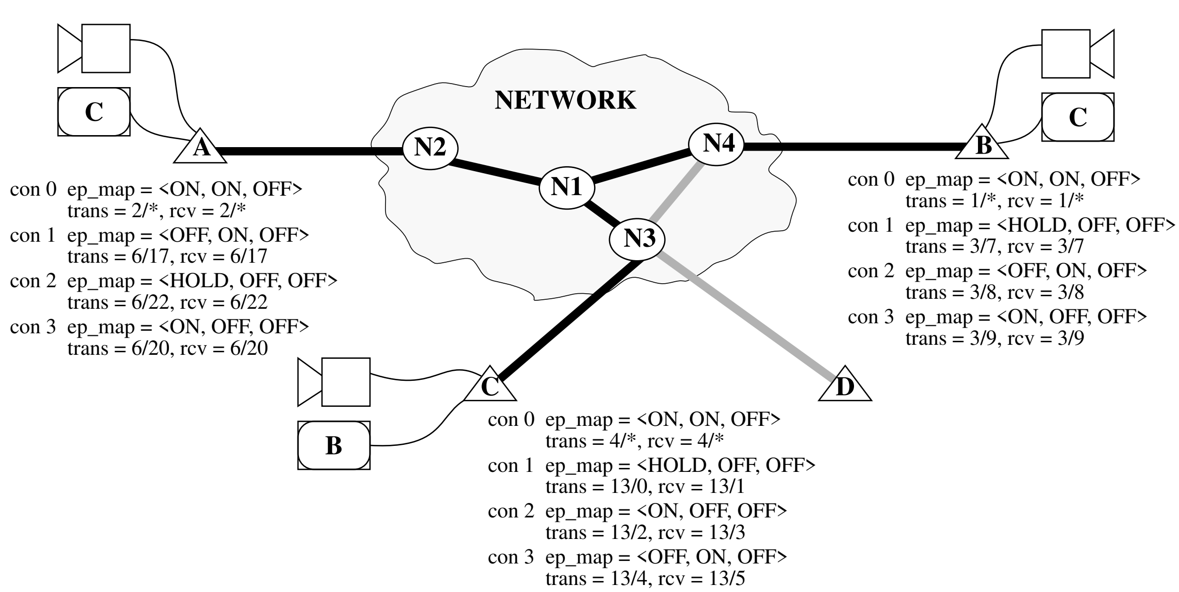


Figure 74. Example Mappings for Conference Call

<table>
<tr><td>00001000</td><td>00000000</td><td colspan="2">0x0001</td><td colspan="2">2</td><td colspan="2">0</td></tr>
<tr><td colspan="8">client A</td></tr>
<tr><td colspan="2">0</td><td colspan="2">0</td><td colspan="2">0</td><td colspan="2">0</td></tr>
<tr><td colspan="8">client B</td></tr>
<tr><td colspan="8">client B</td></tr>
<tr><td colspan="8">client B</td></tr>
<tr><td colspan="2">0</td><td colspan="2">0</td><td colspan="4">0</td></tr>
<tr><td colspan="2">1</td><td colspan="2">0</td><td>ON,OF,OF</td><td>OF,OF,OF</td><td>ON,OF,OF</td><td>0</td></tr>
<tr><td>0</td><td>3</td><td colspan="2">7</td><td>0</td><td>3</td><td colspan="2">7</td></tr>
<tr><td colspan="2">3</td><td colspan="2">0</td><td>HLD,OF,OF</td><td>OF,OF,OF</td><td>ON,OF,OF</td><td>0</td></tr>
<tr><td>0</td><td>3</td><td colspan="2">8</td><td>0</td><td>3</td><td colspan="2">8</td></tr>
<tr><td colspan="4">0</td></tr>
</table>

Figure 75. mod_ep Message for Conference Call Example

Client ***B*** accomplishes this with a **mod_ep REQUEST** as shown in Figure 75. This operation requests that two sets of endpoint mappings be changed. The mapping for connection 1 (that is, the one on which client ***A*** transmits) is to be changed to a receive-only mapping. The mapping for connection 3 (client ***C***'s) is to be changed to a receive-hold mapping. All the other UNI parameters for these connections (defaults, permissions, transmit/receive pairs) are to remain unchanged. Since connections 0 and 2 are not listed in the **REQUEST**, their parameters also remain unchanged.

The network will change the mappings upon receiving the request (no verification is required) and return a **mod_ep ACK**. The network need not change the mappings simultaneously, and the sequence in which the mappings are changed is unspecified. There may thus be a period where both mappings are receive-only or receive-hold. If one of these should be avoided for hardware reasons, two separate **mod_ep REQUESTs** should be sent. In the above example, if overlap must be prevented client ***B*** should first send a **mod_ep** to change connection 3 to receive-only, and after that operation has finished send a **mod_ep** to change connection 1 to receive-only.

Following the execution of the **mod_ep**, client ***B*** will broadcast information on the control connection indicating that it is now viewing client ***A***. On receipt of this information the user interfaces will update their visual displays.

7.3.6 Endpoint Drops Out

When a user wishes to drop out of the conference call, the user-interface client uses this procedure:

- If the user is the last one in the call, the client sends a **close_call** to shut down the entire call. The **close_call** will be legal, since the client must be the owner of the call at this point.

- If the user is not the last one in the call but is the owner of the call, it first selects another client (possibly by negotiation over the control connection) to be the new owner. It then issues a **change_owner** command making that client the new owner.
- The user sends a request to the owner of the call asking that it be dropped. The owner then performs the rest of the procedure.
- If the user is the root of the call, the owner selects a new root endpoint and performs a **change_root**.
- The owner performs a **drop_con** on the user's video connection. All clients receive the **announce_drop_con** notification; clients who are viewing that connection should switch to some other one.
- The owner performs a **drop_ep** on the user's endpoint. All clients, including the user being dropped, will receive the **annouce_drop_ep** notification. The client being dropped will then leave the call, while other clients will update their visual displays.

Some additional coordination of this procedure is required—consider, for example, a two-user call where both users drop out at nearly the same time. The non-owner will send (on the control connection) a request to the owner that the non-owner be dropped, and the owner will send a request to the non-owner that the non-owner take over ownership. These requests could pass one another in the network, causing some coordination problems. Algorithms to handle this are relatively trivial.

7.3.7 Miscellaneous Control Functions

The procedure for closing down the call was covered in the above section—the call is closed when the last user drops out. Most other desired functions, such as maintaining the visual display, are handled by broadcasting information packets over the control connections. Each client transmits on this VP connection setting the VCI equal to its conference-call identifier (video connection identifier). This allows all clients to broadcast simultaneously; at each endpoint, cells from several clients may be interleaved, but the packets can be correctly reassembled using the VCIs for source discrimination.

8. Future Directions in CMAP

This section briefly presents some possible enhancements or extensions to CMAP.

Reducing message size. The current design emphasizes ease of interpretation at the cost of message length. The size of messages could easily be reduced. The many ***reserved*** fields could be eliminated; in many cases, so could the ***unused*** fields. Redundancies could also be eliminated, for example, the separate ***con_id*** and ***ep_con_id*** fields that appear in such messages as the **open_call** and **add_con**.

Transactions. A transaction is defined as a grouping of multiple CMAP operations into a single, larger operation. The operations in the transaction succeed or fail as a group, meaning that if any of the individual operations fail the transaction as a whole fails and all the operations in the transaction are aborted. One mechanism whereby this might be implemented is for the requesting client to issue a command which lists a group of ***msg_ids*** that form a transaction; this would then be followed by the requests for the individual operations. Incorporating this capability may require a reexamination of the phases of the existing operations: some two-phase operations might require a confirmation phase in which the network confirms that the operation should complete, or aborts the operation if other operations in its transaction have failed.

More specific parameters. The current call, connection, and endpoint parameters may not be detailed enough. For example, at the moment call accessibility (indicating whether arbitrary clients may add new endpoints) is **OPEN**, **VERIFY**, or **CLOSED** and applies to all clients in the network. This parameter could be made more specific by allowing the owner to indicate to which clients the parameter applies. This could be done at a group level, for example by having the equivalent of **OPEN/VERIFY/CLOSED** for each of the operations: a client in the call adds a new endpoint for itself; a client in the call adds a new endpoint for another client in the call; a client in the call adds a new endpoint for a client not in the call; a client not in the call adds a new endpoint for itself; and so on. It could also be done on a per-client basis, where the owner could (for example) indicate that client ***A*** may add new endpoints for itself, client ***B*** may add new endpoints itself and add new endpoints for any client with the owner's approval, and client ***C*** may not add new endpoints at all. Obviously this extension would involve major changes to the message formats and require the CMAP Session Managers to maintain a great deal of additional information.

Connection ownership. The current specification allows any member of a call to add new connections, but management of the connection is then taken over by the owner. This was clearly seen in the conference-call example of Section 7.3, where each new client added a connection when it joined the call but had to ask the owner to remove the connection when it dropped out. The simplest way to improve this situation would be to add an ownership parameter to connections. Clients could then modify and drop connections that they owned. Of course, mechanisms for transferring ownership would also have to be added.

Multiple-endpoint operations. Most CMAP operations act on a group of connections but only one (or at most two) endpoints. This arrangement is sensible from the point of view of the network implementation—connections are global across the call, while each endpoint belongs to a specific client. However, there seems no obvious reason why the operations could not be generalized to act simultaneously on several endpoints. For example, the **mod_ep** operation could request changes in multiple endpoints, and for each endpoint request changes to multiple connections. This would obviously require some modification of the message formats. There would also be some control problems, *e.g.*, if an **add_ep** requests that six endpoints be added and two refuse the invitation, what should the status of the operation be and should the four endpoints be added?

Client registration and signalling connections. The current specification assumes that clients simply exist and are known to the network, and that CTL-based signalling connections exist between the clients and the CMAP Session Managers (Section 3.1). These issues are primarily a concern of network management, but some limited CMAP-level support may be appropriate. For example, we might imagine that a new client would have to "register" itself with a session manager by sending a message on a well-known signalling connection. This registration process would also provide an opportunity for the client to request and receive a new signalling connection.

Surrogate configuration. The current specification makes network management responsible for the setup of surrogate signalling. Such facilities could be incorporated into CMAP, in the form of new commands to set up surrogate signalling, transfer surrogacy responsibilities, and so forth. Of course, the use of these commands would have to be carefully safeguarded—clearly we do not wish to allow a client to assign itself as the surrogate for any other client it chooses.

Network query. Facilities whereby clients can query the network and determine what capabilities and resources are available would be a useful enhancement. These queries might allow the client to determine if the network supports point-to-multipoint and multipoint-to-multipoint calls, find out which VPI/VCI pairs are legal and/or available, and determine how much bandwidth is available for the client's use. The first facility is particularly important, since it would allow clients to support multipoint calls in software (by multiple point-to-point calls) even if hardware support is unavailable. The network query function could be easily supported by adding a new CMAP command.

"Yellow Pages" support. CMAP only provides facilities for manipulating calls, and does not provide facilities for determining what CMAP clients or calls are on the network. We envision that such services would be provided by dedicated CMAP client processes with well-known addresses—a client-address server, an active-call server, and so forth. Some additional CMAP support may be required for the implementation of these processes; for example, it might be useful if these clients could query the network for call identifiers, or perform a call trace even if the call parameters do not allow it. This is closely related to the next topic.

Security and privileged clients. At the moment CMAP provides only minimal security, and that largely through the call owner—for example, the owner can set up a call so that outside clients cannot join or trace the call. However, global security concerns are lacking; there is, for example, no means to specify that certain clients should not create calls, or that certain VPI/VCI pairs should not be allocated to clients. There is also no way to indicate that certain clients should have additional privileges, such as the ability to trace arbitrary calls, close other client's calls, or add themselves to a call without the owner's permission. Although these are largely a concern of network management, they will obviously impact the design of the CMAP Session Managers and further study is warranted.

Inter-client signalling. A CMAP operation which allows clients exchange arbitrary data in the form of CMAP signals might be of some use. For example, in the conference call example (Section 7.3) such signals could be used to replace the control connection. The obvious problem with such a facility is in billing; clients could send data with CMAP signals, bypassing normal connections and accounting. However, with appropriate safeguards (such as billing clients for data transmitted on CMAP signals, as well as for data transmitted on connections) the facility might be usable.

Appendix A: References

The reference list contains items that pertain to the area of fast packet switching and broadband networks, possibly of interest to the CMAP user or implementor. Many of the references apply directly to the issue of call management and are cited in this document. Other references are for general background purposes only.

[1] H. Ahmadi, W.E. Denzel, C.A. Murphy, and E. Port. “A High-Performance Switch Fabric for Integrated Circuit and Packet Switching.” In IEEE Infocom ‘88: Proceedings of the Seventh Annual Joint Conference of the IEEE Computer and Communications Societies, pages 9-18, March 1988.

[2] H. Ahmadi and W. E. Denzel. “A Survey of Modern High-Performance Switching Techniques.” In IEEE Journal on Selected Areas in Communications, 7(7):1091-1103, September 1989.

[3] ANSI T1S1 Technical Sub-Committee. Broadband Aspects of ISDN Baseline Document. T1S1.5/90-001, June 1990.

[4] R. Ballart and Y.C. Ching. “SONET: Now It’s the Standard Optical Network.” In IEEE Communications Magazine, 27(3):8-15, March 1989.

[5] Bell Communications Research. Generic System Requirements in Support of Switched Multi-Megabit Data Service. Technical Advisory TA-TSY-000772, Issue 3, October 1989.

[6] P.A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[7] R.G. Bubenik and J.S. Turner. “Performance of a Broadcast Packet Switch.” In IEEE Transactions on Communications, 37(1):60-69, January 1989.

[8] R. G. Bubenik, J. D. DeHart and M. E. Gaddis. “Multipoint Connection Management in High Speed Networks.” In IEEE Infocom ‘91: Proceedings of the Tenth Annual Joint Conference of the IEEE Computer and Communications Societies, pages 59-68, April 1991.

[9] R. G. Bubenik, M. E. Gaddis and J. D. DeHart. “A Strategy for Layering IP over ATM”. Washington University Applied Research Laboratory, Working Note 91-01, Version 1.1, April 1991.

[10] R.G. Bubenik, M.E. Gaddis, and J.D. DeHart. “Virtual Paths and Virtual Channels.” To appear in IEEE Infocom ‘92: Proceedings of the Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies, May 1992.

[11] R. G. Bubenik. “BPN Reliable Datagram Protocol”. Washington University Applied Research Laboratory Working Note 91-11, in progress, June 1991.

[12] J. Burgin and D. Dorman. “Broadband ISDN Resource Management: The Role of Virtual Paths.” In IEEE Communications Magazine, 29(9):44-48, September 1991.

[13] CCITT. Blue Book, volume II, fascicle II.2, “Telephone network and ISDN—Operation, numbering, routing, and mobile service,” Recommendations E.100--E.300, Geneva, Switzerland, 1989.

[14] CCITT. Recommendations Drafted by Working Party XVIII/8 (General B-ISDN Aspects) to be Approved in 1992, Study Group XVIII—Report R 34, December 1991.

[15] CCITT Recommendation Q.931 (I.451), ISDN User-Network Interface Layer 3 Specification, Geneva, 1985.

[16] D.R. Cheriton and W. Zwaenepoel. “Distributed Process Groups in the V Kernel.” In Transactions on Computer Systems, 3(2):77-107, May 1985.

[17] R. Colella, E. Gardner and R. Callon. “Guidelines for OSI NSAP Allocation in the Internet.” INTERNET DRAFT, Networking Group, March 1, 1991.

[18] D. Comer. Internetworking With TCP/IP Principles, Protocols, and Architecture. Prentice Hall, 1988.

[19] J.P. Coudreuse and M Servel. “PRELUDE: An Asynchronous Time-Division Switched Network.” In ICC ‘87: Proceedings of the IEEE International Conference on Communications, pages 69-773, June 1987.

[20] Jr. R. Cox. “Overview of the Washington University Fast Packet Project”. Washington University, Applied Research Laboratory Working Note 89-02, September 1989.

[21] J. R. Cox and J. S. Turner. “Project Zeus Design and Application of Fast Packet Campus Networks”. Washington University, Department of Computer Science Technical Report 91-45, July 1991.

[22] G.E. Daddis, Jr. and H.C. Torng. “A Taxonomy of Broadband Integrated Switching Architectures.” In IEEE Communications Magazine, 27(5):32-42, May 1989.

[23] S.E. Deering. “Multicast Routing in Internetworks and Extended LANs.” In Proceedings of the SIGCOMM ‘88 Symposium: Communications Architectures & Protocols, pages 55-64, August 1988.

[24] K.Y. Eng, M.G. Hluchyj, and Y.S. Yeh. “Multicast and Broadcast Services in a Knockout Packet Switch.” In IEEE Infocom ‘88: Proceedings of the Seventh Annual Joint Conference of the IEEE Computer and Communications Societies, pages 29-34, March 1988.

[25] H.C. Folts. “Procedures for Circuit-Switched Service in Synchronous Public Data Networks.” In IEEE Transactions on Communications, 28(4):489-496, April 1980.

[26] M. E. Gaddis. “ATM-TAP: Patent Disclosure Statement”. Washington University, Applied Research Laboratory Working Note 90-12, Version 1.2, May 1990.

[27] M. E. Gaddis, R.G. Bubenik, and J.D. DeHart. “Connection Management for a Prototype Fast Packet ATM B-ISDN Network.” In Proceedings of the National Communications Forum, vol. 44, pp. 601-608, October 8-10, 1990.

[28] M. E. Gaddis, R.G. Bubenik, and J.D. DeHart. “A Call Model for Multipoint Communications in Switched Networks.” submitted for publication to ICC ‘92, Chicago, Illinois, June 1992.

[29] J.N Giacopelli, W.D. Sincoskie, and M. Littlewood. “Sunshine: A High Performance Self-Routing Broadband Packet Switch Architecture.” In Proceedings of the International Switching Symposium, Volume 3, pages 123-129, May 1990.

[30] W.M. Harman and C.F. Newman. “ISDN Protocols for Connection Control.” In IEEE Journal on Selected Areas in Communications, 7(7):1034-1042, September 1989.

[31] K. Haserodt and J.S. Turner. “An Architecture for Connection Management in a Broadcast Packet Network.” Washington University, Department of Computer Science, Technical Report-WUCS-87-03, 1987.

[32] M.G. Hluchyj and M.J. Karol. “Queueing in Space-Division Packet Switching.” In IEEE Infocom ‘88: Proceedings of the Seventh Annual Joint Conference of the IEEE Computer and Communications Societies, pages 334-343, March 1988.

[33] A. Huang and S. Knauer. “Starlite: a Wideband Digital Switch.” In Proceedings of Globecom 84, pages 121-125, December 1984.

[34] J. Hui. “A Broadband Packet Switch for Multi-Rate Services.” In ICC ‘87: Proceedings of the IEEE International Conference on Communications, pages 782-788, June 1987.

[35] K. Iguchi, H. Takeo, S. Amemiya, and K. Tezuka. “Subscriber Access Scheme for Broadband ISDN.” In ICC ‘90: Proceedings of the IEEE International Conference on Communications, pages 663-669, April 1990.

[36] A.R. Jacob. A Survey of Fast Packet Switches. Computer Communication Review, 20(1):54-64, January 1990.

[37] Y. Kato, T. Shimoe, K. Hajikano, and K. Murakami. “Experimental Broadband ATM Switching System.” In Proceedings of Globecom 88, pages 1288-1292, December 1988.

[38] H.S. Kim and A. Leon-Garcia. “A Self-Routing Multistage Switching Network for Broadband ISDN.” In IEEE Journal on Selected Areas in Communications, 8(3):459-466, April 1990.

[39] J.C. Kohli, D.S. Biring, and G.L. Raya. “Emerging Broadband Packet-Switch Technology in Integrated Information Networks.” In IEEE Network, 2(6):37-38,47-51, November 1988.

[40] T.R. La Porta and M. Schwartz. “Architectures, Features, and Implementation of High-Speed Transport Protocols.” In IEEE Network, 4(2):14-22, May 1991.

[41] T.T. Lee, R. Boorstyn, and E. Arthurs. “The Architecture of a Multicast Broadband Packet Switch.” In IEEE Infocom ‘88: Proceedings of the Seventh Annual Joint Conference of the IEEE Computer and Communications Societies, pages 1-8, March 1988.

[42] T. Lyon. “Simple and Efficient Adaptation Layer” ANSI T1S1.5 proposal for type 5 AAL by Sun Microsystems, Inc., August, 12-16, 1991.

[43] S.E. Minzer. “Broadband ISDN and Asynchronous Transfer Mode (ATM).” In IEEE Communications Magazine, 27(9):17-24, September 1989.

[44] S.E. Minzer and D.R. Spears. “New Directions in Signalling for Broadband ISDN.” In IEEE Communications Magazine, 27(2):6-14, February 1989.

[45] S.E. Minzer. “A Signalling Prototype for Complex Multimedia Services.” In IEEE Journal on Selected Areas in Communications, 9(9):1383-1394, December 1991.

[46] J.E.B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, 1985.

[47] C.H. Papadimitriou. The Theory of Concurrency Control. Computer Science Press, 1986.

[48] G.M. Parulkar, J.S. Turner. Towards a Framework for High Speed Communication in a Heterogeneous Networking Environment. In IEEE Infocom ‘89: Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies, pages 655-667, April 1989.

[49] G. M. Parulkar. “The Next Generation of Internetworking”. ACM SIGCOMM Computer Communications Review. vol. 20, no. 1, New York, NY, pp. 18-43, January, 1990.

[50] W.D. Richard, J. R. Cox Jr., A. M. Engebretson, J. Fritts, B. Gottlieb and C. Horn. “The Washington University MultiMedia eXplorer”. Technical Report WUCS-93-40, Department of Computer Science, Washington University in St. Louis, 1993.

[51] F.E. Ross. “An Overview of FDDI: The Fiber Distributed Data Interface.” In IEEE Journal on Selected Areas in Communications, 7(7):1043-1051, September 1989.

[52] A. Rybczynski. “X.25 Interface and End-to-End Virtual Circuit Service Characteristics.” In IEEE Transactions on Communications, 28(4):500-510, April 1980.

[53] R.M. Sanders. The Xpress Transfer Protocol (XTP)—A Tutorial. Computer Networks Laboratory, Department of Computer Science, University of Virginia, January 15, 1990.

[54] J.S. Stacey, T. Pham, and J. Chiou. “Modeling Call Control for Distributed Applications in Telephony.” In IEEE Network, 4(6):14-22, November 1990.

[55] H. Suzuki, H. Nagano, T. Suzuki, T. Takeuchi, and S. Iwasaki. “Output-buffer Switch Architecture for Asynchronous Transfer Mode.” In ICC ‘89: Proceedings of the IEEE International Conference on Communications, pages 99-103, 1989.

[56] H. Suzuki, T. Murase, S. Sato, and T. Takeuchi. “A Burst Traffic Control Strategy for ATM Networks.” Submitted for publication (conference unknown).

[57] A.S. Tanenbaum. Computer Networks. Prentice-Hall, 1981.

[58] S.C. Tu and W.H. Leung. “Multicast Connection-Oriented Packet Switching Networks.” In Proceedings of the International Communications Conference, volume 2, pages 495-501, April 1990.

[59] J. S. Turner, “Fast Packet Switching System”, U.S. Patent 4 494 230, January 15, 1985.

[60] J.S. Turner. “New Directions in Communications.” In IEEE Communications Magazine, 24(10):8-15, October 1986.

[61] J.S. Turner. “Design of an Integrated Services Packet Network.” In IEEE Transactions on Communications, 4(8):1373-1380, November 1986.

[62] J.S. Turner. “Design of a Broadcast Packet Switching Network.” In IEEE Transactions on Communications, 36(6):734-743, June 1988.

[63] J. S. Turner. “A Proposed Management and Congestion Control Scheme for Multicast ATM Networks.” Washington University, Computer and Communication Research Center Technical Report 91-01, May 1991.

[64] Waxman, B. ***A paper on the “toward-the-root” routing algorithm.***

[65] XTP® Protocol Definition, Revision 3.5. Protocol Engines Incorporated, Technical Report PEI 90-120, September 10, 1990.

[66] Y.S. Yeh, M.G. Hluchyj, and A.S. Acampora. "The Knockout Switch: A Simple, Modular Architecture for Performance Packet Switching." In International Switching Symposium, volume 3, pages 801-808, March 1987.

Appendix B: Acronym List

The following acronyms and abbreviations are used within this document in reference to ATM networks, fast packet switches, and our protocols.

ABORT — Abort confirmation message (Section 5.1)

ACK — Acknowledgment response message (Section 5.1)

ATM — Asynchronous Transfer Mode (Section 2)

BPN — Broadcast Packet Network (Section 2.1)

BISDN — Broadband Integrated Services Digital Network

BW — Bandwidth (Section 4.4.3)

CCITT — International Telegraph and Telephone Consultative Committee (Section 4.2.1)

CLP — Cell Loss Priority (ATM header field) (Section 2.2)

CMAP — Connection Management Access Protocol (Section 1)

CML — Connection Management Layer (Section 3.5)

CMNP — Connection Management Network Protocol (Section 3.5)

CN — Copy Network (Section 2.1)

COM — Commit confirmation message (Section 5.1)

CONF — Confirmation message (Section 5.1)

CP — Control Processor (Section 2.1)

CTL — CMAP Transport Layer (Section 3.3)

GFC — Generic Flow Control (ATM header field) (Section 2.2)

HEC — Header Error Check (ATM header field) (Section 2.2)

ISO-OSI — International Standards Organization - Open System Interconnection

PT — Payload Type (ATM header field) (Section 2.2)

QOS — Quality of Service (Section 4.4.2)

NACK — Negative acknowledgment response message (Section 5.1)

NEG — Negotiation response message (Section 5.1)

NNI — Network Node Interface (Section 2.1)

REQ — Request message (Section 5.1)

RES — Response message (Section 5.1)

RN — Routing Network (Section 2.1)

SONET — Synchronous Optical NETwork

SMI — Switch Module Interface (Section 2.1)

UNI — User Network Interface (Section 2.1)

VC — Virtual Channel (Section 2.3)

VCI — Virtual Channel Identifier (ATM header field) (Section 2.2)

VP — Virtual Path (Section 2.3)

VPI — Virtual Path Identifier (ATM header field) (Section 2.2)

Appendix C: CMAP Message Field Values

This appendix contains the values of the symbolic constants used in CMAP messages. The fields are sorted by message object. Values are in hexadecimal (expressed using the “C” programming language “0x” format) except as noted. All unused values (those not appearing below) are reserved for future use.

Header Object: op_type (value in binary)

00000000	no operation
00000001	**open_call**
00000010	**mod_call**
00000011	**close_call**
00000100	**add_con**
00000101	**mod_con**
00000110	**drop_con**
00000111	**add_ep**
00001000	**mod_ep**
00001001	**drop_ep**
00001010	**trace_call**
00001011	**trace_ep**
00001100	**change_owner**
00001101	**change_root**
00001110	**invite_add_con**
00001111	**invite_add_ep**
00010000	**invite_mod_ep**
00010001	**invite_change_owner**
00010010	**verify_add_ep**
00010011	**verify_mod_ep**
00010100	**announce_mod_call**
00010101	**announce_close_call**
00010110	**announce_add_con**
00010111	**announce_mod_con**
00011000	**announce_drop_con**
00011001	**announce_add_ep**
00011010	**announce_mod_ep**
00011011	**announce_drop_ep**
00011100	**announce_change_owner**
00011101	**announce_change_root**
10000000	**status**
10000001	**alert**
10000010	**client_reset**
10000011	**network_reset**
11111111	**error_report**

Header Object: phase (value in binary)

00000000	**REQUEST**
00000001	**RESPONSE**
00000010	**CONFIRMATION**

Header Object: op_status: call_status_bit

0x0	**OK**
0x1	**ERROR**

Header Object: op_status: connection_status_bit

0x0	**OK**
0x1	**ERROR**

Header Object: op_status: endpoint_status_bit

0x0	**OK**
0x1	**ERROR**

Header Object: op_status: uni_status_bit

0x0	**OK**
0x1	**ERROR**

Header Object: op_status: status

0x000	**OK**
0x001	**REFUSED**
0x002	**NEGOTIATING**
0x003	**BAD_OP_TYPE**
0x004	**BAD_PHASE_EXP_REQUEST**
0x005	**BAD_PHASE_EXP_RESPONSE**
0x006	**BAD_PHASE_EXP_CONFIRMATION**
0x007	**DUP_MSG_ID**
0x008	**BAD_MSG_ID**
0x009	**BAD_MADDR**
0x00a	**BAD_SADDR**
0x00b	**FORMAT_ERROR**
0x00c	**BAD_NUM_CONS**
0x00d	**BAD_NUM_EPS**
0x00e	**BAD_CALL_ID_ADDR**
0x00f	**DUP_CALL_ID**
0x010	**UNKNOWN_CALL**
0x011	**NOT_OWNER**
0x012	**ILL_REQUEST**
0x013	**ILL_DROP_ROOT**
0x014	**VERIFY_REFUSED**
0x015	**EP_REFUSED**
0x016	**BAD_OWNER_ADDR**
0x017	**INSUFF_BANDWIDTH**
0x018	**TIMEOUT**

Call Object: call_status

0x0000	**OK**
0x0001	**BAD_CALL_TYPE**
0x0002	**BAD_ACC**
0x0003	**BAD_MOD**
0x0004	**BAD_TRACE**
0x0005	**BAD_MON**
0x0006	**BAD_PRIORITY**

Call Object: call_type

0x0	**MULTIPOINT**
0x1	**POINT_TO_POINT**

Call Object: acc

0x0	**CLOSED**
0x1	**OPEN**
0x2	**VERIFY**

Call Object: mod

0x0	**CLOSED**
0x1	**OPEN**

Call Object: trace

0x0	**CLOSED**
0x1	**OPEN**
0x2	**MEMBERS**

Call Object: mon (owner, transmitters, all)

0x0	**OFF**
0x1	**ON**

Call Object: priority

0x0	**NORMAL**
0x1	**PREEMPT**
0x2	**OVERRIDE**

Connection Object: con_status

0x0000	**OK**
0x0001	**BAD_CON_ID**
0x0002	**DUP_CON_ID**
0x0003	**BAD_CON_TYPE**
0x0004	**BAD_CON_DEF**
0x0005	**BAD_CON_PERM**
0x0006	**BAD_BW**

Connection Object: con_type: channel_type

0x0	**VP**
0x1	**VC**

Connection Object: con_type: bw_type

0x0	**DYNAMIC**
0x1	**STATIC**

Connection Object: con_type: qos

0x0	**HIGH**
0x1	**MEDIUM**
0x2	**LOW**

Connection Object: con_def (receive, transmit, echo)

0x0	**OFF**
0x1	**ON**
0x2	**HOLD**

Connection Object: con_perm (receive, transmit, echo)

0x0	**OFF**
0x1	**ON**
0x2	**VERIFY**

Endpoint Object: ep_status

0x0000	**OK**
0x0001	**BAD_EP_ADDR**
0x0002	**DUP_EP_ID**
0x0003	**BAD_EP_ADDR_NOT_ROOT**

UNI Object: uni_status

0x0000	**OK**
0x0001	**BAD_CON_ID**
0x0002	**DUP_CON_ID**
0x0003	**BAD_EP_MAP**
0x0004	**ILL_EP_MAP**
0x0005	**BAD_EP_DEF**
0x0006	**BAD_EP_PERM**
0x0007	**NO_AVAIL_VPI**
0x0008	**NO_AVAIL_VCI**
0x0009	**TRANS_VPI_IN_USE**
0x000a	**TRANS_VPI_RESERVED**
0x000b	**TRANS_VPI_NOT_SUPPORTED**
0x000c	**TRANS_VCI_IN_USE**
0x000d	**TRANS_VCI_RESERVED**
0x000e	**TRANS_VCI_NOT_SUPPORTED**
0x000f	**RCV_VPI_IN_USE**
0x0010	**RCV_VPI_RESERVED**
0x0011	**RCV_VPI_NOT_SUPPORTED**
0x0012	**RCV_VCI_IN_USE**
0x0013	**RCV_VCI_RESERVED**
0x0014	**RCV_VCI_NOT_SUPPORTED**

UNI Object: ep_map (receive, transmit, echo)

0x0	**OFF**
0x1	**ON**
0x2	**HOLD**

UNI Object: ep_def (receive, transmit, echo)

0x0	**OFF**
0x1	**ON**
0x2	**HOLD**

UNI Object: ep_perm (receive, transmit, echo)

0x0	**OFF**
0x1	**ON**
0x2	**VERIFY**

Operation Object: op_msg_status

0x0	**OK_RESPONSE**
0x1	**OK_CONFIRMATION**
0x2	**NO_SUCH_OPERATION**

client address: addr_type

0x0	undefined
0x1	**IP**
0x2	**ISDN_E_164**
0x3	**NSAP**

Appendix D: CMAP Status Codes

Each of the sections of this Appendix lists all the codes that may appear in the status fields of CMAP messages. An explanation of the meaning of the code is given, together with a list of possible actions that the client may try if it receives the code from the network.

If a message has more than one error, CMAP is only required to report any one of the errors. CMAP is permitted to report several errors simultaneously, if possible—obviously it is impossible to simultaneously report any two distinct errors that use the same field for the error.

D.1 *op_status* Field

call_status_bit = ***connection_status_bit*** = ***endpoint_status_bit*** = ***uni_status_bit*** = **OK**

- Description: No errors occurred in the message objects.
- Recommended Action: Check the ***status*** portion of ***op_status*** to determine if there were errors in the message header or in its execution.

call_status_bit = **ERROR**

- Description: An error occurred in the message Call Object.
- Recommended Action: Check the ***call_status*** field of the Call Object to determine the error, correct the problem, and re-submit the request.

connection_status_bit = **ERROR**

- Description: An error occurred in one of the message Connection Objects.
- Recommended Action: Check the ***con_status*** field of each of the Connection Objects to determine the error(s), correct the problem, and re-submit the request.

endpoint_status_bit = **ERROR**

- Description: An error occurred in one of the message Endpoint Objects.
- Recommended Action: Check the ***ep_status*** field of each of the Endpoint Objects to determine the error(s), correct the problem, and re-submit the request.

uni_status_bit = **ERROR**

- Description: An error occurred in one of the message UNI Objects.
- Recommended Action: Check the ***uni_status*** field of each of the UNI Objects to determine the error(s), correct the problem, and re-submit the request.

status = **OK**

- Description: Provided the high-order bits are all OK, no errors occurred in the header or in the execution of the command and the operation completed successfully.
- Recommended Action: none.

status = **REFUSED**

- Description: The client refuses to perform the operation. Used only in **NACK RESPONSEs** from clients.
- Recommended Action: none.

status = **NEGOTIATING**

- Description: The client wishes to negotiate operation parameters. Used only in **NEG RESPONSEs** from clients.
- Recommended Action: none.

status = **BAD_OP_TYPE**

- Description: An undefined value was found in the ***op_type*** field. This can only appear in **error_report** messages.
- Recommended Action: Generate the message again and resend, checking that the ***op_type*** field contains a legal value. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **NO_SUCH_OPERATION**

- Description: The value in the ***msg_id*** field does not correspond to an existing operation. This can only appear in **error_report** messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the ***msg_id*** field contains a valid operation identifier. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **BAD_PHASE**

- Description: An undefined value was found in the ***phase*** field. This can only appear in **error_report** messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the ***phase*** field contains a valid value. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **BAD_PHASE_EXP_REQUEST**

- Description: An undefined value was found in the ***phase*** field. Based on the ***msg_id*** supplied a **REQUEST** was expected. This can only appear in **error_report** messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the ***phase*** field contains **REQUEST**. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **BAD_PHASE_EXP_RESPONSE**

- Description: An undefined value was found in the ***phase*** field. Based on the ***msg_id*** supplied a **RESPONSE** was expected. This can only appear in **error_report** messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the ***phase*** field contains **RESPONSE**. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **BAD_PHASE_EXP_CONFIRMATION**

- Description: An undefined value was found in the ***phase*** field. Based on the ***msg_id*** supplied a **CONFIRMATION** was expected. This can only appear in **error_report** messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the ***phase*** field contains **CONFIRMATION**. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **DUP_MSG_ID**

- Description: The ***msg_id*** given in the **REQUEST** is still active from a previous operation. This can only appear in **error_report** messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the message identifier is not already in use. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **BAD_MSG_ID**

- Description: The ***msg_id*** given in the **REQUEST** is illegal. This can only appear in **error_report** messages.

- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the message identifier is in the correct format. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **BAD_MADDR**

- Description: The ***maddr*** given in the **REQUEST** is illegal. This can only appear in **error_report** messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the message address is in the correct format. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **BAD_SADDR**

- Description: The ***msg_id*** given in the **REQUEST** is illegal. This can only appear in **error_report** messages. It may indicate that the sender attempted signalling for a client for which it was not the surrogate.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the message identifier is in the correct format. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **FORMAT_ERROR**

- Description: The message cannot be matched to the template for its ***op_type*** (it may be too short or too long, or the internal fields may make no sense). This can only appear in **error_report** messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message. If the same error occurs, report it so that the client's message-generating routines may be checked.

status = **BAD_NUM_CONS**

- Description: The number of connections specified in the ***num_cons*** field was out of range. The range of values is [0, MAX_CONS], where MAX_CONS is implementation dependent.
- Recommended Action: Redefine the call, possibly splitting the call into multiple calls.

status = **BAD_NUM_EPS**

- Description: The number of endpoints specified in the ***num_eps*** field was out of range. The range of values is [0, MAX_EPS], where MAX_EPS is implementation dependent; for particular operations (*e.g.*, **open_call**) the range may be further restricted.
- Recommended Action: Redefine the call, possibly splitting the call into multiple calls.

status = **BAD_CALL_ID_ADDR**

- Description: The owner address portion of the ***call_id*** specified in an **open_call REQUEST** was not a proper endpoint address or did not match the address of the client that sent the **REQUEST**.
- Recommended Action: Check the address given and correct it. Resend the **open_call REQUEST** with the correct address.

status = **DUP_CALL_ID**

- Description: The ***call_id*** specified in an **open_call REQUEST** is already in use. This means that the root address is legitimate, but the local identifier is in use.
- Recommended Action: Select a new ***lcid*** and resend the **open_call REQUEST**.

status = **UNKNOWN_CALL**

- Description: The ***call_id*** specified in an operation does not correspond to any active call.
- Recommended Action: Verify the ***call_id*** and resend the **REQUEST**.

status = **NOT_OWNER**

- Description: The client is not the owner of the call and attempted to perform an operation that can only be performed by an owner (*e.g.*, **mod_call**), or on its own endpoints (*e.g.*, **drop_ep**).

- Recommended Action: The client should contact the owner of the call and ask it to perform the operation.

status = **ILL_REQUEST**

- Description: The client attempted to perform an operation that is forbidden by the call parameters (*e.g.*, an **add_ep** on a call with **CLOSED** accessibility, or changing an endpoint mapping in a way not permitted by the endpoint's defaults and permissions).
- Recommended Action: The client should contact the owner of the call and request that it perform the operation or give the client permission to perform the operation.

status = **ILL_DROP_ROOT**

- Description: The client attempted to perform a **drop_ep** on the last endpoint of the call's root client.
- Recommended Action: If the client is the owner, it should first perform a **change_root** operation then attempt the **drop_ep** again. If the client is not the owner (meaning it is the root), it should contact the owner and request that the owner change the root so the client can drop out.

status = **VERIFY_REFUSED**

- Description: During the operation a query was sent to the owner (*e.g.*, to verify the addition of an endpoint or the modification of a mapping). The owner refused to allow the operation.
- Recommended Action: The client should contact the owner of the call to discuss the operation.

status = **EP_REFUSED**

- Description: An endpoint refused an invitation to join a call or modify its parameters.
- Recommended Action: The client should try again at a later time.

status = **BAD_OWNER_ADDR**

- Description: The new owner address (in **change_owner**) is illegal or unknown.
- Recommended Action: The client should check the address to ensure it is properly formed and is a known address, then resend the **REQUEST**.

status = **INSUFF_BW**

- Description: There was not enough bandwidth in the network to perform the requested operation.
- Recommended Action: The client should check that it has enough bandwidth on its access link. If it does not have the bandwidth to support the new request, it should drop or drop out of some calls to free some bandwidth. If the access link has the bandwidth to support the new request dropping or dropping out of some calls may free enough bandwidth in the network. Otherwise the client should wait and try the request at a later time.

status = **TIMEOUT**

- Description: A timeout occurred somewhere in the network.
- Recommended Action: Attempt the operation again. If it still fails with this ***status***, the client's network manager should be notified.

D.2 *call_status* Field

call_status = **OK**

- Description: No errors were found in this Call Object.
- Recommended Action: None.

call_status = **BAD_CALL_TYPE**

- Description: An undefined value was found in the ***call_type*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

call_status **= BAD_ACC**

- Description: An undefined value was found in the ***acc*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

call_status **= BAD_MOD**

- Description: An undefined value was found in the ***mod*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

call_status **= BAD_TRACE**

- Description: An undefined value was found in the ***trace*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

call_status **= BAD_MON**

- Description: An undefined value was found in the ***mon*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

call_status **= BAD_PRIORITY**

- Description: An undefined value was found in the ***priority*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

D.3 *con_status* Field

con_status **= OK**

- Description: No errors were found in this Connection Object.
- Recommended Action: None

con_status **= BAD_CON_ID**

- Description: The connection identifier in ***con_id*** does not belong to any connection in the call.
- Recommended Action: Select a different connection identifier and resend the **REQUEST**.

con_status **= DUP_CON_ID**

- Description: The connection identifier in ***con_id*** is already in use for this call, or appears twice among the Connection Objects.
- Recommended Action: Select a different connection identifier and resend the **REQUEST**.

con_status **= BAD_CON_TYPE**

- Description: An undefined value was found in the ***con_type*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

con_status **= BAD_CON_DEF**

- Description: An undefined value was found in the ***con_def*** field..
- Recommended Action: Correct the value and resend the **REQUEST**.

con_status **= BAD_CON_PERM**

- Description: An undefined value was found in the ***con_perm*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

con_status **= BAD_BW**

- Description: An illegal or impossible specification was given for the bandwidth for this connection.
- Recommended Action: One possible cause for an illegal bandwidth specification is if the ***average*** is greater than the ***peak***. The client should check that the specification is correct and resend the **REQUEST**.

D.4 *ep_status* Field

ep_status **= OK**

- Description: No errors were found in this Endpoint Object.
- Recommended Action: None

ep_status **= BAD_EP_ADDR**

- Description: The client address in ***ep_addr*** is unknown.
- Recommended Action: Check the address to ensure it is properly formed and is a known address. Correct the address and resend the **REQUEST**.

ep_status **= DUP_EP_ID**

- Description: The endpoint identifier in ***ep_id*** is already in use for this client.
- Recommended Action: Select a new ***ep_id*** and resend the **REQUEST**.

ep_status **= BAD_EP_ADDR_NOT_ROOT**

- Description: The endpoint address given for the root in an **open_call** message is not the same as that of the call identifier's root address.
- Recommended Action: Correct the address and resend the **REQUEST**.

D.5 *uni_status* Field

uni_status **= OK**

- Description: No errors were found in this UNI Object.
- Recommended Action: None

uni_status **= BAD_CON_ID**

- Description: The connection identifier in ***con_id*** does not belong to any connection in the call.
- Recommended Action: Select a different connection identifier and resend the **REQUEST**.

uni_status **= DUP_CON_ID**

- Description: The connection identifier in ***ep_con_id*** is already in use for this call, or appears twice among the UNI Objects.
- Recommended Action: Select a different connection identifier and resend the **REQUEST**.

uni_status **= BAD_EP_MAP**

- Description: An undefined value was found in the ***ep_map*** field..
- Recommended Action: Correct the value and resend the **REQUEST**.

uni_status **= ILL_EP_MAP**

- Description: The requested mapping is not allowed by the endpoint's defaults and permissions.
- Recommended Action: The client should contact the owner of the call and request permission to change its mapping.

uni_status **= BAD_EP_DEF**

- Description: An undefined value was found in the ***ep_def*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

uni_status **= BAD_EP_PERM**

- Description: An undefined value was found in the ***ep_perm*** field.
- Recommended Action: Correct the value and resend the **REQUEST**.

uni_status = **NO_AVAIL_VPI**

- Description: There were no VPIs available to satisfy this **REQUEST**.
- Recommended Action: Wait for a time, then resend the **REQUEST**.

uni_status = **NO_AVAIL_VCI**

- Description: There were no VCIs available to satisfy this **REQUEST**.
- Recommended Action: Wait for a time, then resend the **REQUEST**.

uni_status = **TRANS_VPI_IN_USE**

- Description: The transmit VPI requested is already in use.
- Recommended Action: Select another VPI and resend the **REQUEST**.

uni_status = **TRANS_VPI_RESERVED**

- Description: The transmit VPI requested is reserved.
- Recommended Action: Select another VPI and resend the **REQUEST**. The reserved VPI should be marked as such in the client database so that it is not selected again.

uni_status = **TRANS_VPI_NOT_SUPPORTED**

- Description: The transmit VPI requested is not supported. It is probably out of the range of VPIs supported by the hardware.
- Recommended Action: Check the range of VPIs supported and select a new VPI. Resend the **REQUEST**.

uni_status = **TRANS_VCI_IN_USE**

- Description: The transmit VCI requested is already in use.
- Recommended Action: Select another VCI and resend the **REQUEST**.

uni_status = **TRANS_VCI_RESERVED**

- Description: The transmit VCI requested is reserved.
- Recommended Action: Select another VCI and resend the **REQUEST**. The reserved VCI should be marked as such in the client database so that it is not selected again.

uni_status = **TRANS_VCI_NOT_SUPPORTED**

- Description: The transmit VCI requested is not supported. It is probably out of the range of VCIs supported by the hardware.
- Recommended Action: Check the range of VCIs supported and select a new VCI. Resend the **REQUEST**.

uni_status = **RCV_VPI_IN_USE**

- Description: The receive VPI requested is already in use.
- Recommended Action: Select another VPI and resend the **REQUEST**.

uni_status = **RCV_VPI_RESERVED**

- Description: The receive VPI requested is reserved.
- Recommended Action: Select another VPI and resend the **REQUEST**. The reserved VPI should be marked as such in the client database so that it is not selected again.

uni_status = **RCV_VPI_NOT_SUPPORTED**

- Description: The receive VPI requested is not supported. It is probably out of the range of VPIs supported by the hardware.
- Recommended Action: Check the range of VCIs supported and select a new VPI. Resend the **REQUEST**.

uni_status **= RCV_VCI_IN_USE**

- Description: The receive VCI requested is already in use.
- Recommended Action: Select another VCI and resend the **REQUEST**.

uni_status **= RCV_VCI_RESERVED**

- Description: The receive VCI requested is reserved.
- Recommended Action: Select another VCI and resend the **REQUEST**. The reserved VCI should be marked as such in the client database so that it is not selected again.

uni_status **= RCV_VCI_NOT_SUPPORTED**

- Description: The receive VCI requested is not supported. It is probably out of the range of VCIs supported by the hardware.
- Recommended Action: Check the range of VCIs supported and select a new VCI. Resend the **REQUEST**.

D.6 *op_msg_status* Field

op_msg_status **= OK_RESPONSE**

- Description: The operation is proceeding normally. The next message sent will be a **RESPONSE**.
- Recommended Action: None

op_msg_status **= OK_CONFIRMATION**

- Description: The operation is proceeding normally. The next message sent will be a **CONFIRMATION**.
- Recommended Action: None

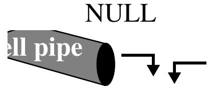
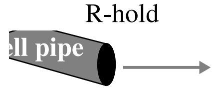
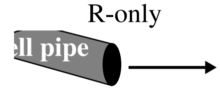
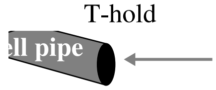
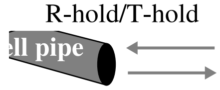
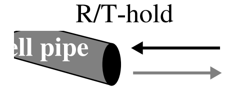
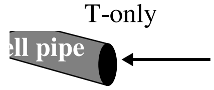
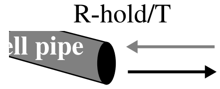
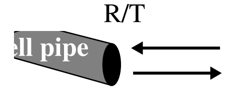
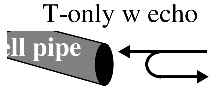
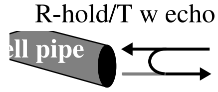
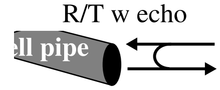
uni_status **= NO_SUCH_OPERATION**

- Description: There is no record of any such operation (message identifier).
- Recommended Action: Return to the initial state, cleaning up local data as required, and resend the **REQUEST**.

Appendix E: Endpoint Mappings

A client has three choices for its receive mapping (**ON**, **OFF**, **HOLD**), three for its transmit mapping (**ON**, **OFF**, **HOLD**), and two for its echo mapping (**ON**, **OFF**). Although this allows 18 different mappings (***ep_map***), only twelve are sensible—in those combinations with echo = **ON** and transmit = **OFF** or **HOLD**, the echo is non-functional since there is no transmission. This is not to say that the latter six combinations are illegal, merely that they are unlikely to arise in practice and so we do not examine them here. Table 11 lists the twelve useful endpoint mappings.

Table 11. The Twelve Endpoint Mappings

	Receive = **OFF**	Receive = **HOLD**	Receive = **ON**
Transmit = **OFF** Echo = **OFF**	NULL 	R-hold 	R-only 
Transmit = **HOLD** Echo = **OFF**	T-hold 	R-hold/T-hold 	R/T-hold 
Transmit = **ON** Echo = **OFF**	T-only 	R-hold/T 	R/T 
Transmit = **ON** Echo = **ON**	T-only w echo 	R-hold/T w echo 	R/T w echo 

For each mapping, certain combinations of ***ep_def*** and ***ep_perm*** would disallow the mapping. Recall the rules for the use of ***ep_def*** and ***ep_perm***, which apply separately to each of the three mappings (receive, transmit, hold):

- If ***ep_perm*** is **ON**, the client may freely choose any value for the mapping.
- If ***ep_perm*** is **OFF**, the client must have a mapping equal to ***ep_def***. (Exception: For receive and transmit, if ***ep_def*** is **ON** the client may have a mapping of either **ON** or **HOLD**.)
- If ***ep_perm*** is **VERIFY**, the client may choose any value for the mapping but the choice will be confirmed with the call owner. (Exception: For receive and transmit, if ***ep_def*** is **ON** the client may change between the mappings **ON** and **HOLD** without owner verification.)

Table 12 lists, for each of the twelve useful mappings, the combinations of ***ep_def*** and ***ep_perm*** that would disallow the mapping. For notational convenience we separate out the receive, transmit, and echo mappings. When a set is used it means that any of the values in the set will disable the mapping. It is sufficient for any one of the conditions to be met for the mapping to be disallowed.

Table 12. Disabling Defaults and Permissions

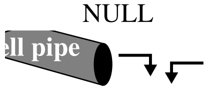
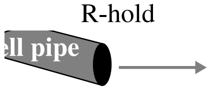
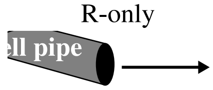
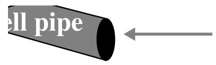
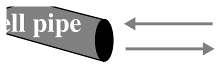
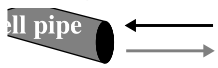
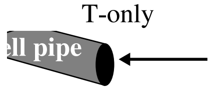
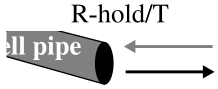
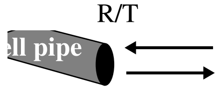
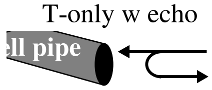
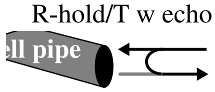
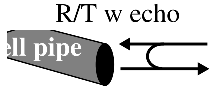
Mapping	**Disabling Combinations**
NULL 	Receive: ***ep_def*** ∈ {**ON**, **HOLD**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**ON**, **HOLD**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**
R-hold 	Receive: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**ON**, **HOLD**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**
R-only 	Receive: ***ep_def*** ∈ {**HOLD**, **OFF**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**ON**, **HOLD**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**

Table 12. Disabling Defaults and Permissions (Continued)

Mapping	Disabling Combinations
T-hold 	Receive: ***ep_def*** ∈ {**ON**, **HOLD**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**
R-hold/T-hold 	Receive: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**
R/T-hold 	Receive: ***ep_def*** ∈ {**HOLD**, **OFF**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**
T-only 	Receive: ***ep_def*** ∈ {**ON**, **HOLD**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**HOLD, OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**
R-hold/T 	Receive: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**HOLD, OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**
R/T 	Receive: ***ep_def*** ∈ {**HOLD**, **OFF**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**HOLD, OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**ON**}, ***ep_perm*** = **OFF**
T-only w echo 	Receive: ***ep_def*** ∈ {**ON**, **HOLD**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**HOLD, OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF**
R-hold/T w echo 	Receive: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**HOLD, OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF**
R/T w echo 	Receive: ***ep_def*** ∈ {**HOLD**, **OFF**}, ***ep_perm*** = **OFF** Transmit: ***ep_def*** ∈ {**HOLD, OFF**}, ***ep_perm*** = **OFF** Echo: ***ep_def*** ∈ {**OFF**}, ***ep_perm*** = **OFF**

One aspect of the table should be noted: Setting ***ep_perm*** = **OFF** for echo severely restricts the mappings available to the client. For this reason it is recommended that the ***ep_perm*** field for echo always be either **ON** or **VERIFY**. This seems a reasonable recommendation, since in most anticipated applications the client itself is best able to determine if it needs a copy of the data it is transmitting.

Appendix F: Parameter Negotiation

Many of the call, connection and endpoint attributes must be agreed upon by clients and the network. These parameters are *negotiable*, in that either the client or the network may choose the values and the other must accept them. This section summarizes the negotiation procedures.

In general, when a client sends a command **REQUEST**, the client may specify values for the negotiable parameters or leave the parameters blank and allow the network to choose. If the client chooses values, the network checks them for correctness and returns a **NACK** in the **RESPONSE** if there are any problems. When a network sends a prompt **REQUEST** to a client, the network will always choose values for the parameters but the client can override the network's choices by sending a **NEG RESPONSE**. If the client's choices are illegal, the network will send an **ABORT**.

F.1 Address Negotiation

We allow for the possibility that client identifiers may differ between the phases of an operation. This type of situation might arise, for example, in an environment where clients are processes and there are multiple clients on each machine connected to the network. Client addresses would then consist of two parts, a machine portion which specifies the machine and a process portion which selects a client on the machine (these might be implemented as the network and local address fields of the CMAP address, see Section 4.2). The operation that adds a client (an **open_call** or **add_ep**) might provide an address which directs the **invite_add_ep** prompt to a server process on the appropriate machine. This server would then use other information, such as the ***user_call_type***, to create a new instance of the appropriate process and pass the message to that process. The client address returned in the response would then be that of the new process, and should be used thereafter. This type of negotiation can arise in the following situations:

r_addr in **open_call** and **invite_add_ep**: The value provided by the owner in the **open_call REQUEST** is transmitted to the root client in the **invite_add_ep REQUEST**. The root client may return a different address in its **invite_add_ep RESPONSE**, which becomes the root address for the call and is returned to the owner in the **open_call RESPONSE**. If the **open_call REQUEST** specified two endpoints, the final value will be provided to the additional endpoint in its **invite_add_ep COM**.

m_addr and ***s_addr*** in **invite_add_ep**: The value in the network's **REQUEST** may differ from that in the client's **RESPONSE**. The network matches the ***msg_ids*** and accepts the new value.

ep_addr in **open_call**, **add_ep**, and **invite_add_ep**: The new-client address supplied by the requesting client in an **open_call** or **add_ep** is provided to the new client in the **invite_add_ep REQUEST**. The value that the new client returns in the **RESPONSE** may differ; this value becomes the client address and is returned to the requesting client in the **open_call** or **add_ep REQUEST**.

new_owner in **change_owner**: The value in the network's **REQUEST** may differ from that in the client's **RESPONSE**. The network matches the ***msg_ids*** and uses the new value as the owner's address.

F.2 Call Identifiers

r_addr in **open_call** and **invite_add_ep**: See the above comments.

lcid in **open_call**, **change_root** and **invite_add_ep**. If the owner leaves the ***lcid*** field in the **open_call REQUEST** blank, the network will select an unused value and send it to the root client in the **invite_add_ep REQUEST**. Otherwise the network sends the owner-proposed value to the root client. The root client then has the option of negotiating a new value. If the owner selects a value that is in use, the network will send the owner an **open_call NACK**; if the value selected by the root in negotiation is already in use, the network will send an **open_call NACK** to the owner and an **invite_add_ep ABORT** to the root. In all these cases the message ***status*** = **DUP_CALL_ID**.

F.3 Connection Identifiers

con_ids in **open_call** and **add_con**. The network assigns values to any **con_id** field left blank by the requester. Identifiers are by consecutive integers, starting with 1 and increasing throught the Connection Objects in the order given, skipping any identifiers already in use or appearing elsewhere in the message. As an example, assume that a call has three connections with identifiers 1, 3, and 31. A client sends an **add_con REQUEST** with three Connection Objects. The first and second objects have a blank **con_id** and the third has a **con_id** of 4. Assuming the operation succeeds, the first blank **con_id** will receive the identifier 2 (1 is already in use) and the second the identifier 5 (3 is already in use, and 4 is assigned to the third additional connection). The Connection Objects will appear in the same order in the **REQUEST** as in the **RESPONSE**, so the first will have ***con_id*** 2, the second ***con_id*** 5, and the third ***con_id*** 4.

ep_con_ids in **open_call** and **add_con**. Both UNI Objects containing ***ep_con_ids*** and Connection Objects containing ***con_ids*** appear in these two commands. We require that each non-blank ***ep_con_id*** equal one of the non-blank ***con_ids*** and that no ***ep_con_ids*** be repeated (in other words, the set of non-blank ***ep_con_ids*** must exactly equal that of the non-blank ***con_ids***). The network will assign values to any blank ***ep_con_ids***. Assignment will be by consecutive integers starting with 1 and increasing through the Connection Objects in the order given, skipping any identifiers already in use. This will produce the same set of identifiers as for the ***con_ids*** as described above.

F.4 Endpoint Identifiers

ep_id in **open_call**, **add_ep**, and **invite_add_ep**. In the **open_call** or **add_ep REQUEST** the requesting client may leave this field blank. The network will select a value and send it to the invited client in the **invite_add_ep REQUEST**. That client may negotiate a new value for the parameter. If the requester selects a value that is in use, the network will send the requester an **open_call** or **add_ep NACK**; if the value selected by the invited client in negotiation is already in use, the network will send an **open_call** or **add_ep NACK** to the owner and an **invite_add_ep ABORT** to the root. In all these cases the Endpoint Object's ***ep_status*** = **DUP_EP_ID**.

F.5 Mappings, Defaults, and Permissions

ep_map in **open_call**, **add_con**, **add_ep**, and **mod_ep**. If the requester leaves the value of ***ep_map*** blank in the command (**open_call**, **add_con**, **add_ep**, or **mod_ep**), the network will offer the value of the connection's ***con_def*** field to the invited endpoint in the prompt (**invite_add_ep** or **invite_add_con**). The invited endpoint may negotiate the value of ***ep_map*** in its **RESPONSE** to the prompt, subject to the restrictions of ep_def and ep_perm. For each of the three subfields (receive, transmit, and echo) of the mapping: if ***ep_perm*** = **OFF** the value of ***ep_map*** must equal that of ***ep_def***; if ***ep_perm*** = **VERIFY** the client may negotiate but verification will be required; and if ***ep_perm*** = **ON** the client may change the value freely.

ep_def in **open_call**, **add_con**, **add_ep**, and **mod_ep**. If the requesting client leaves this field blank the network will use the value of the connection's ***con_def*** field. Negotiation of this value by the client is not permitted.

ep_perm in **open_call**, **add_con**, **add_ep**, and **mod_ep**. If the requesting client leaves this field blank the network will use the value of the connection's ***con_perm*** field. Negotiation of this value by the client is not permitted.

F.6 VPI/VCI Pairs

trans_vpi/trans_vci, ***rcv_vpi/rcv_vci*** in **open_call**, **add_con**, **add_ep**, and **mod_ep**. If any of these pairs is blank in the command, the network will select a value before offering it to the invited client in the prompt. The client may accept these values or negotiate different ones in its response. If the requester selects a value that is in use, reserved, or otherwise unavailable, the network will send the requester an **open_call** or **add_ep NACK**; if the value selected by the invited client in negotiation is unavailable, the network will send an **open_call** or **add_ep NACK** to the owner and an **invite_add_ep ABORT** to the root.

Index