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

<table>
<thead>
<tr>
<th>00011000</th>
<th>0000000</th>
<th>msg_id (2)</th>
<th>num_cons (2)</th>
<th>unused (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>call_id (r_addr) (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>call_id (clid) (2)</td>
<td>unused (2)</td>
<td>reserved (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>reserved (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m_addr (24)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s_addr (24)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>con_id (2)</td>
<td>unused (2)</td>
<td>user_con_type (4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>con_type (1)</td>
<td>reserved (1)</td>
<td>con_def (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>con_perm (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bw (12)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>00011001</th>
<th>00000000</th>
<th>msg_id (2)</th>
<th>num_cons (2)</th>
<th>unused (2)</th>
</tr>
</thead>
</table>

  call_id (r_addr) (24)

<table>
<thead>
<tr>
<th>call_id (clid) (2)</th>
<th>unused (2)</th>
<th>reserved (2)</th>
<th>reserved (2)</th>
</tr>
</thead>
</table>

  m_addr (24)

<table>
<thead>
<tr>
<th>s_addr (24)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ep_addr (24)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ep_id (2)</th>
<th>unused (2)</th>
<th>reserved (4)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ep_con_id (2)</th>
<th>unused (2)</th>
<th>ep_map (1)</th>
<th>ep_def (1)</th>
<th>ep_perm (1)</th>
<th>reserved (1)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>reserved (1)</th>
<th>trans_vpi (1)</th>
<th>trans_vci (2)</th>
<th>reserved (1)</th>
<th>rcv_vpi (1)</th>
<th>rcv_vci (2)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>(num_cons)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>options_size (4)</th>
<th>options (options_size)</th>
</tr>
</thead>
</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

<table>
<thead>
<tr>
<th>00011010</th>
<th>00000000</th>
<th>msg_id (2)</th>
<th>num_cons (2)</th>
<th>unused (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>call_id (r_addr) (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>call_id (lcid) (2)</td>
<td>unused (2)</td>
<td>reserved (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s_addr (24)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| ep_addr (24) |

<table>
<thead>
<tr>
<th>ep_id (2)</th>
<th>unused (2)</th>
<th>reserved (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ep_con_id (2)</td>
<td>unused (2)</td>
<td>ep_map (1)</td>
</tr>
<tr>
<td>reserved (1)</td>
<td>trans_vpi (1)</td>
<td>trans_vcl (2)</td>
</tr>
</tbody>
</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.

6.27.4 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>
<thead>
<tr>
<th>00011011</th>
<th>00000000</th>
<th>msg_id (2)</th>
<th>unused (2)</th>
<th>unused (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>call_id (r_addr) (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call_id (leid) (2)</td>
<td>unused (2)</td>
<td>reserved (2)</td>
<td>reserved (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>m_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ep_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ep_id (2)</td>
<td>unused (2)</td>
<td>reserved (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>options_size (4)</td>
<td>options (options_size)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00011100</td>
<td>00000000</td>
<td>msg_id (2)</td>
<td>unused (2)</td>
<td>unused (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>call_id (r_addr) (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>call_id (lclid) (2)</td>
<td>unused (2)</td>
<td>reserved (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m_addr (24)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s_addr (24)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>new_owner (24)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>options_size (4)</td>
</tr>
</tbody>
</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>
<thead>
<tr>
<th>00011101</th>
<th>00000000</th>
<th>msg_id (2)</th>
<th>unused (2)</th>
<th>unused (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>call_id (r_addr) (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call_id (clid) (2)</td>
<td>unused (2)</td>
<td>reserved (2)</td>
<td>reserved (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>m_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>new_call_id (r_addr) (24)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new_call_id (clid) (2)</td>
<td>reserved (2)</td>
<td>reserved (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>options_size (4)</td>
<td>options (options_size)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Data:

- *new_call_id (r_addr), new_call_id (clid)* - 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.

![Diagram 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.

![Diagram 60] Message Traffic for Client-Initiated status
6.31.3 Message Formats

**status REQUEST**

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000 00000000</td>
<td></td>
</tr>
<tr>
<td>msg_id (2)</td>
<td></td>
</tr>
<tr>
<td>unused (2)</td>
<td></td>
</tr>
<tr>
<td>unused (2)</td>
<td></td>
</tr>
<tr>
<td>call_id (r_addr) (24)</td>
<td></td>
</tr>
<tr>
<td>call_id (lcid) (2)</td>
<td></td>
</tr>
<tr>
<td>unused (2)</td>
<td></td>
</tr>
<tr>
<td>reserved (2)</td>
<td></td>
</tr>
<tr>
<td>reserved (2)</td>
<td></td>
</tr>
<tr>
<td>m_addr (24)</td>
<td></td>
</tr>
<tr>
<td>s_addr (24)</td>
<td></td>
</tr>
<tr>
<td>op_msg_id (2)</td>
<td></td>
</tr>
<tr>
<td>unused (2)</td>
<td></td>
</tr>
<tr>
<td>reserved (4)</td>
<td></td>
</tr>
<tr>
<td>options_size (4)</td>
<td></td>
</tr>
<tr>
<td>options (options_size)</td>
<td></td>
</tr>
</tbody>
</table>

**Data:**

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

**status RESPONSE**

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000 00000001</td>
<td></td>
</tr>
<tr>
<td>msg_id (2)</td>
<td></td>
</tr>
<tr>
<td>unused (2)</td>
<td></td>
</tr>
<tr>
<td>unused (2)</td>
<td></td>
</tr>
<tr>
<td>call_id (r_addr) (24)</td>
<td></td>
</tr>
<tr>
<td>call_id (lcid) (2)</td>
<td></td>
</tr>
<tr>
<td>unused (2)</td>
<td></td>
</tr>
<tr>
<td>reserved (2)</td>
<td></td>
</tr>
<tr>
<td>reserved (2)</td>
<td></td>
</tr>
<tr>
<td>m_addr (24)</td>
<td></td>
</tr>
<tr>
<td>s_addr (24)</td>
<td></td>
</tr>
<tr>
<td>op_msg_id (2)</td>
<td></td>
</tr>
<tr>
<td>op_msg_status (2)</td>
<td></td>
</tr>
<tr>
<td>reserved (4)</td>
<td></td>
</tr>
<tr>
<td>options_size (4)</td>
<td></td>
</tr>
<tr>
<td>options (options_size)</td>
<td></td>
</tr>
</tbody>
</table>

**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:
  
  \[
  \text{status} \in \{ \text{OK, UNKNOWN\_CALL, TIMOUT} \} \]
• op_msg_status - this field may take on the following values:
  \[ \text{op_msg_status} \in \{ \text{OK\_RESPONSE, OK\_CONFIRMATION, NO\_SUCH\_OPERATION} \} \]

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

```
10000001 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)  op_msg_status (2)  reserved (4)

   options_size (4)  options (options_size)
```

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 information 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>
<thead>
<tr>
<th>10000010</th>
<th>00000000</th>
<th>msg_id (2)</th>
<th>unused (2)</th>
<th>unused (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>unused (24)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unused (2)</th>
<th>unused (2)</th>
<th>reserved (2)</th>
<th>reserved (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>m_addr (24)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>s_addr (24)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>options_size (4)</th>
<th>options (options_size)</th>
</tr>
</thead>
</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**

<table>
<thead>
<tr>
<th>10000010</th>
<th>0000001</th>
<th>msg_id (2)</th>
<th>unused (2)</th>
<th>unused (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>unused (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unused (2)</td>
<td>op_status (2)</td>
<td>reserved (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>options_size (4)</td>
<td></td>
<td>options (options_size)</td>
</tr>
</tbody>
</table>

**Data:**

- **op_status** - the status portion of this field may take on the following values:
  
  \[ \text{status} \in \{ \text{OK, TIMEOUT} \} \]

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>
<thead>
<tr>
<th>10000011</th>
<th>00000000</th>
<th>msg_id (2)</th>
<th>unused (2)</th>
<th>unused (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>unused (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>unused (2)</td>
<td>unused (2)</td>
<td>reserved (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>s_addr (24)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>options_size (4)</td>
<td>options (options_size)</td>
<td></td>
</tr>
</tbody>
</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>
<thead>
<tr>
<th>1111111</th>
<th>0000000</th>
<th>0xffff</th>
<th>unused (2)</th>
<th>unused (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>unused (24)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>op_status (2)</td>
<td>reserved (2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>m_addr (24)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>s_addr (24)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>message_length (4)</td>
<td>reserved (4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>options_size (4)</td>
<td>options (options_size)</td>
</tr>
</tbody>
</table>

Data:

The msg_id field for an error_report is always 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:
  \[ status \in \{ \text{BAD\_OP\_TYPE}, \text{BAD\_PHASE\_EXP\_REQUEST}, \text{BAD\_PHASE\_EXP\_RESPONSE}, \text{BAD\_PHASE\_EXP\_CONFIRMATION}, \text{DUP\_MSG\_ID}, \text{BAD\_MSG\_ID}, \text{BAD\_MADDR}, \text{BAD\_SADDR}, \text{FORMAT\_ERROR} \]  

- 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.

![Network Diagram](image)

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.

![Diagram showing the sequence of operations for call setup]

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.
• \( ep_id = 0 \); client A is selecting an endpoint identifier.

• \( ep_con_id = \text{blank} \); this is required, since the \( con_id \) is also blank.

• \( ep_map = ep_def = \langle \text{OFF, ON, OFF} \rangle \); client A is overriding the connection defaults and assigning itself a transmit-only mapping.

• \( trans_vpi = trans_vci = rev_vpi = rev_vci = \text{blank} \); 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 \( leid = 0 \), \( con_id = ep_con_id = 1 \), transmit pair = 3/10, and receive pair = 3/11. It then sends 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 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 an open_call ACK to client A. The call, with one connection and endpoint, now exists within the network.
The call setup in Figure 62 then continues when client A sends 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 = 0x00001; 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 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 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 an invite_add_ep ACK to client B.

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 connection 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](image)

Client A first sends 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 an ACK while client B sends a NACK, client A will receive an ABORT to tell it not to join the connection; in its other capacity as owner of the call, client A will also receive 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 the add_ep REQUEST, and client B must be prepared to receive data as soon as it sends 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.

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 management 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](image)

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 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_sps = 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 = MULTIPLEX;` 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 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.
- VP/VC 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.
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 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 an ACK in which m_addr = client C and s_addr = client A. The network signals to client A that the connections to the endpoint were established and signals the owner that the call was 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 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 a verify_add_ep prompt to the owner, client C. Assuming that client C responds 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 an add_ep ACK to client B and (in accord with the call’s monitoring parameter) sends an announce_add_ep REQUEST to client A.

![Figure 69. First Client Joins Video Server Call](image)

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).
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 a drop_ep REQUEST for itself. The network responds with a drop_ep ACK and (in accord with the call's monitoring parameter) sends 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.

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:

- \texttt{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.
**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.

---

**Figure 72. open_call Message for Conference Call Example**
• 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](image-url)
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 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).

![Network Diagram](network_diagram.png)

Figure 74. Example Mappings for Conference Call
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 `announce_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.


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)**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>no operation</td>
</tr>
<tr>
<td>00000001</td>
<td>open_call</td>
</tr>
<tr>
<td>00000010</td>
<td>mod_call</td>
</tr>
<tr>
<td>00000011</td>
<td>close_call</td>
</tr>
<tr>
<td>00000100</td>
<td>add_con</td>
</tr>
<tr>
<td>00000101</td>
<td>mod_con</td>
</tr>
<tr>
<td>00000110</td>
<td>drop_con</td>
</tr>
<tr>
<td>00000111</td>
<td>add_ep</td>
</tr>
<tr>
<td>00001000</td>
<td>mod_ep</td>
</tr>
<tr>
<td>00001001</td>
<td>drop_ep</td>
</tr>
<tr>
<td>00001010</td>
<td>trace_call</td>
</tr>
<tr>
<td>00001011</td>
<td>trace_ep</td>
</tr>
<tr>
<td>00011100</td>
<td>change_owner</td>
</tr>
<tr>
<td>00011101</td>
<td>change_root</td>
</tr>
<tr>
<td>00011110</td>
<td>invite_add_con</td>
</tr>
<tr>
<td>00011111</td>
<td>invite_add_ep</td>
</tr>
<tr>
<td>00100000</td>
<td>invite_mod_ep</td>
</tr>
<tr>
<td>00100001</td>
<td>invite_change_owner</td>
</tr>
<tr>
<td>00100100</td>
<td>verify_add_ep</td>
</tr>
<tr>
<td>00100101</td>
<td>verify_mod_ep</td>
</tr>
<tr>
<td>00101000</td>
<td>announce_mod_call</td>
</tr>
<tr>
<td>00101001</td>
<td>announce_close_call</td>
</tr>
<tr>
<td>00101010</td>
<td>announce_add_con</td>
</tr>
<tr>
<td>00101011</td>
<td>announce_mod_con</td>
</tr>
<tr>
<td>00110000</td>
<td>announce_drop_con</td>
</tr>
<tr>
<td>00110010</td>
<td>announce_add_ep</td>
</tr>
<tr>
<td>00110100</td>
<td>announce_mod_ep</td>
</tr>
<tr>
<td>00110110</td>
<td>announce_drop_ep</td>
</tr>
<tr>
<td>00111000</td>
<td>announce_change_owner</td>
</tr>
<tr>
<td>00111101</td>
<td>announce_change_root</td>
</tr>
<tr>
<td>10000000</td>
<td>status</td>
</tr>
<tr>
<td>10000001</td>
<td>alert</td>
</tr>
<tr>
<td>10000010</td>
<td>client_reset</td>
</tr>
<tr>
<td>10000011</td>
<td>network_reset</td>
</tr>
<tr>
<td>11111111</td>
<td>error_report</td>
</tr>
</tbody>
</table>

**Header Object: phase (value in binary)**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>REQUEST</td>
</tr>
<tr>
<td>00000001</td>
<td>RESPONSE</td>
</tr>
<tr>
<td>00000010</td>
<td>CONFIRMATION</td>
</tr>
</tbody>
</table>
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

\[ \text{call_status_bit} = \text{connection_status_bit} = \text{endpoint_status_bit} = \text{uni_status_bit} = \text{OK} \]

- 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.

\[ \text{call_status_bit} = \text{ERROR} \]

- 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.

\[ \text{connection_status_bit} = \text{ERROR} \]

- 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.

\[ \text{endpoint_status_bit} = \text{ERROR} \]

- 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.

\[ \text{uni_status_bit} = \text{ERROR} \]

- 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.

\[ \text{status} = \text{OK} \]

- 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.

\[ \text{status} = \text{REFUSED} \]

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

\[ \text{status} = \text{NEGOTIATING} \]

- Description: The client wishes to negotiate operation parameters. Used only in NEG RESPONSEs from clients.
- Recommended Action: none.
\texttt{status = BAD\_OP\_TYPE}

- Description: An undefined value was found in the \textit{op\_type} field. This can only appear in \texttt{error\_report} messages.
- Recommended Action: Generate the message again and resend, checking that the \textit{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.

\texttt{status = NO\_SUCH\_OPERATION}

- Description: The value in the \textit{msg\_id} field does not correspond to an existing operation. This can only appear in \texttt{error\_report} messages.
- Recommended Action: "Back up" the state machine and regenerate and resend the message, checking that the \textit{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.

\texttt{status = BAD\_PHASE}

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

\texttt{status = BAD\_PHASE\_EXP\_REQUEST}

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

\texttt{status = BAD\_PHASE\_EXP\_RESPONSE}

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

\texttt{status = BAD\_PHASE\_EXP\_CONFIRMATION}

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

\texttt{status = DUP\_MSG\_ID}

- Description: The \textit{msg\_id} given in the \texttt{REQUEST} is still active from a previous operation. This can only appear in \texttt{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.

\texttt{status = BAD\_MSG\_ID}

- Description: The \textit{msg\_id} given in the \texttt{REQUEST} is illegal. This can only appear in \texttt{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.

\textit{status = BAD\_MAADDR}

- Description: The \textit{maddr} given in the \textit{REQUEST} is illegal. This can only appear in \textit{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.

\textit{status = BAD\_SAADDR}

- Description: The \textit{msg\_id} given in the \textit{REQUEST} is illegal. This can only appear in \textit{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.

\textit{status = FORMAT\_ERROR}

- Description: The message cannot be matched to the template for its \textit{op\_type} (it may be too short or too long, or the internal fields may make no sense). This can only appear in \textit{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.

\textit{status = BAD\_NUM\_CONS}

- Description: The number of connections specified in the \textit{num\_cons} field was out of range. The range of values is \([0, \text{MAX\_CONS}]\), where \text{MAX\_CONS} is implementation dependent.
- Recommended Action: Redefine the call, possibly splitting the call into multiple calls.

\textit{status = BAD\_NUM\_EPS}

- Description: The number of endpoints specified in the \textit{num\_eps} field was out of range. The range of values is \([0, \text{MAX\_EPS}]\), where \text{MAX\_EPS} is implementation dependent; for particular operations (\textit{e.g.}, \textit{open\_call}) the range may be further restricted.
- Recommended Action: Redefine the call, possibly splitting the call into multiple calls.

\textit{status = BAD\_CALL\_ID\_ADDR}

- Description: The owner address portion of the \textit{call\_id} specified in an \textit{open\_call REQUEST} was not a proper endpoint address or did not match the address of the client that sent the \textit{REQUEST}.
- Recommended Action: Check the address given and correct it. Resend the \textit{open\_call REQUEST} with the correct address.

\textit{status = DUP\_CALL\_ID}

- Description: The \textit{call\_id} specified in an \textit{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 \textit{lcid} and resend the \textit{open\_call REQUEST}.

\textit{status = UNKNOWN\_CALL}

- Description: The \textit{call\_id} specified in an operation does not correspond to any active call.
- Recommended Action: Verify the \textit{call\_id} and resend the \textit{REQUEST}.

\textit{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 (\textit{e.g.}, \textit{mod\_call}), or on its own endpoints (\textit{e.g.}, \textit{drop\_ep}).
• Recommended Action: The client should contact the owner of the call and ask it to perform the operation.

\texttt{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.

\texttt{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.

\texttt{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.

\texttt{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.

\texttt{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.

\texttt{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.

\texttt{status = TIMEOUT}

• Description: A timeout occurred somewhere in the network.

• Recommended Action: Attempt the operation again. If it still fails with this \texttt{status}, the client's network manager should be notified.

### D.2 call\_status Field

\texttt{call\_status = OK}

• Description: No errors were found in this Call Object.

• Recommended Action: None.

\texttt{call\_status = BAD\_CALL\_TYPE}

• Description: An undefined value was found in the \texttt{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 \textit{ep\_status} Field

\texttt{ep\_status = OK}

- Description: No errors were found in this Endpoint Object.
- Recommended Action: None

\texttt{ep\_status = BAD\_EP\_ADDR}

- Description: The client address in \texttt{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.

\texttt{ep\_status = DUP\_EP\_ID}

- Description: The endpoint identifier in \texttt{ep\_id} is already in use for this client.
- Recommended Action: Select a new \texttt{ep\_id} and resend the REQUEST.

\texttt{ep\_status = BAD\_EP\_ADDR\_NOT\_ROOT}

- Description: The endpoint address given for the root in an \texttt{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 \textit{uni\_status} Field

\texttt{uni\_status = OK}

- Description: No errors were found in this UNI Object.
- Recommended Action: None

\texttt{uni\_status = BAD\_CON\_ID}

- Description: The connection identifier in \texttt{con\_id} does not belong to any connection in the call.
- Recommended Action: Select a different connection identifier and resend the REQUEST.

\texttt{uni\_status = DUP\_CON\_ID}

- Description: The connection identifier in \texttt{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.

\texttt{uni\_status = BAD\_EP\_MAP}

- Description: An undefined value was found in the \texttt{ep\_map} field.
- Recommended Action: Correct the value and resend the REQUEST.

\texttt{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.

\texttt{uni\_status = BAD\_EP\_DEF}

- Description: An undefined value was found in the \texttt{ep\_def} field.
- Recommended Action: Correct the value and resend the REQUEST.

\texttt{uni\_status = BAD\_EP\_PERM}

- Description: An undefined value was found in the \texttt{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

<table>
<thead>
<tr>
<th>Transmit = OFF</th>
<th>Receive = OFF</th>
<th>Receive = HOLD</th>
<th>Receive = ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>Echo = OFF</td>
<td>NULL</td>
<td>R-hold</td>
<td>R-only</td>
</tr>
<tr>
<td></td>
<td><img src="null" alt="Diagram" /></td>
<td><img src="r-hold" alt="Diagram" /></td>
<td><img src="r-only" alt="Diagram" /></td>
</tr>
<tr>
<td>Transmit = HOLD</td>
<td>T-hold</td>
<td>R-hold/T-hold</td>
<td>R/T-hold</td>
</tr>
<tr>
<td>Echo = OFF</td>
<td><img src="t-hold" alt="Diagram" /></td>
<td><img src="r-hold/t-hold" alt="Diagram" /></td>
<td><img src="r/t-hold" alt="Diagram" /></td>
</tr>
<tr>
<td>Transmit = ON</td>
<td>T-only</td>
<td>R-hold/T</td>
<td>R/T</td>
</tr>
<tr>
<td>Echo = OFF</td>
<td><img src="t-only" alt="Diagram" /></td>
<td><img src="r-hold/t" alt="Diagram" /></td>
<td><img src="r/t" alt="Diagram" /></td>
</tr>
<tr>
<td>Transmit = ON</td>
<td>T-only w echo</td>
<td>R-hold/T w echo</td>
<td>R/T w echo</td>
</tr>
<tr>
<td>Echo = ON</td>
<td>![Diagram](t-only/w echo)</td>
<td>![Diagram](r-hold/t w echo)</td>
<td>![Diagram](r/t w echo)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Disabling Combinations</th>
</tr>
</thead>
</table>
| ![Diagram](null) | Receive: ep_def ∈ {ON, HOLD}, ep_perm = OFF  
Transmit: ep_def ∈ {ON, HOLD}, ep_perm = OFF  
Echo: ep_def ∈ {ON}, ep_perm = OFF |
| ![Diagram](r-hold) | Receive: ep_def ∈ {OFF}, ep_perm = OFF  
Transmit: ep_def ∈ {ON, HOLD}, ep_perm = OFF  
Echo: ep_def ∈ {ON}, ep_perm = OFF |
| ![Diagram](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)

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Disabling Combinations</th>
</tr>
</thead>
</table>
| T-hold      | \(ep\_def \in \{\text{ON, HOLD}\}, ep\_perm = \text{OFF}\)  
             | Transmit: \(ep\_def \in \{\text{OFF}\}, ep\_perm = \text{OFF}\)  
             | Echo: \(ep\_def \in \{\text{ON}\}, ep\_perm = \text{OFF}\) |
| R-hold/T-hold| \(ep\_def \in \{\text{OFF}\}, ep\_perm = \text{OFF}\)  
                | Transmit: \(ep\_def \in \{\text{OFF}\}, ep\_perm = \text{OFF}\)  
                | Echo: \(ep\_def \in \{\text{ON}\}, ep\_perm = \text{OFF}\) |
| R/T-hold    | \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Transmit: \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Echo: \(ep\_def \in \{\text{ON}\}, ep\_perm = \text{OFF}\) |
| T-only      | \(ep\_def \in \{\text{ON, HOLD}\}, ep\_perm = \text{OFF}\)  
                | Transmit: \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Echo: \(ep\_def \in \{\text{ON}\}, ep\_perm = \text{OFF}\) |
| R-hold/T    | \(ep\_def \in \{\text{OFF}\}, ep\_perm = \text{OFF}\)  
                | Transmit: \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Echo: \(ep\_def \in \{\text{ON}\}, ep\_perm = \text{OFF}\) |
| R/T         | \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Transmit: \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Echo: \(ep\_def \in \{\text{ON}\}, ep\_perm = \text{OFF}\) |
| T-only w echo| \(ep\_def \in \{\text{ON, HOLD}\}, ep\_perm = \text{OFF}\)  
                | Transmit: \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Echo: \(ep\_def \in \{\text{ON}\}, ep\_perm = \text{OFF}\) |
| R-hold/T w echo | \(ep\_def \in \{\text{OFF}\}, ep\_perm = \text{OFF}\)  
                | Transmit: \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Echo: \(ep\_def \in \{\text{OFF}\}, ep\_perm = \text{OFF}\) |
| R/T w echo  | \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Transmit: \(ep\_def \in \{\text{HOLD, OFF}\}, ep\_perm = \text{OFF}\)  
                | Echo: \(ep\_def \in \{\text{OFF}\}, ep\_perm = \text{OFF}\) |

One aspect of the table should be noted: Setting \(ep\_perm = \text{OFF}\) 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 \text{ON} or \text{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.

\( leid \) in open_call, change_root and invite_add_ep: If the owner leaves the leid 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 through 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 Ob-
jects. 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 = VERIFIED 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

Bandwidth
  average 19, 30
  best-effort 19
  connection 19
  management 10
  peak 10, 19, 30
  peak burst length 19
  peak burth length 30
  reserved 19
  static and dynamic 19, 29

Best-effort connections 10

Billing 45, 54, 64, 126

Broadcast packet switch 2

Call model 1, 14-22

Call Object 28
  acc 28, 142
  call_status 28, 142, 148-149
  call_type 28, 142
  mod 28, 142
  mon 28, 142
  priority 28, 142
  trace 28, 142
  user_call_type 28

Call operations 33, 36-??

Call parameters 16-18, 133
  accessibility 17, 28
  connection list 18
  endpoint list 18
  identifier 17
  local identifier 17, 26
  modifiability 17, 28
  monitoring 17, 28
  owner 14, 17
  priority 18, 28
  root 14, 17, 26
  traceability 17, 28
  type 17, 28
  user type 18, 28

Calls 7, 16-18
  changing owner of 70
  changing root of 73
  closing 44
  defined 14
  determining parameters of 65
  modifying parameters of 42

ABORT confirmation 23, 33
  defined 27

Accessibility 17, 28

ACK response 23, 33
  defined 26

add_con command 46

add_ep command 55

Addresses
  addr_type 144
    CCITT E.164, represented as CMAP 15
    client 22
    CMAP 1, 15
    determining client 120, 134
    endpoint 20, 30
    IP, represented as CMAP 15
    OSI NSAP, represented as CMAP 15
    types 15, 144

alert maintenance operation 111

announce_add_con notification 100

announce_add_ep notification 103

announce_change_owner notification 106

announce_change_root notification 107

announce_close_call notification 99

announce_drop_con notification 102

announce_drop_ep notification 105

announce_mod_call notification 98

announce_mod_con notification 101

announce_mod_ep notification 104

ATM
  cells 2, 4
  meta-signalling connection 9
  networks 2
  standard 4
  switches 2

Audio/Video Server example 123-126
  call setup 123
  client drops out 126
  client joins 125
  transition to multipoint call 125
Connection Management Access Protocol (CMAP) Specification

multipoint 1, 17
opening 36
point-to-point 1, 17

Cell header fields
Cell Loss Priority (CLP) 4, 19
Global Flow Control (GFC) 4
Header Error Check (HEC) 4
Payload Type (PT) 4
Virtual Channel Identifier (VCI) 4
Virtual Path Identifier (VPI) 4

Cell pipes 4, 14
Cells, ATM 2
  client data 4
  format 4
  header 2, 4
  header fields 2
  network control 4
  payload 4

change_owner command 70
change_root command 73

Client parameters
  address 15

client_reset maintenance operation 112

Clients 2, 7, 15-16
  defined 14
  mute 10
  surrogate 10

close_call command 44

CMAP
  as UNI protocol 1
  CML requirements for 10
  complete 11
  CTL requirements for 8
  environment 7
  implementations 11
  minimal 1, 11
  network requirements for 8

CMAP Transport Layer (CTL) 1, 7, 8-9
requirements for CMAP 8

CMNP 11
  as Connection Management Layer 11

COM confirmation 23, 33
  defined 27

Complete CMAP
  defined 11

Conference Call example 127-132
addition of connections 129
addition of users 130
call setup 127
changing endpoint mappings 130
control connection 127, 132
dropping user 131
implementation in CMAP 127
user interfaces 127
user-level protocol 127

CONFIRMATION phase 23

Congestion 4, 8, 19

Connection Management Layer (CML) 7, 10-11
CMNP 11
  requirements for CMAP 10

Connection Object 29-30
  bw 30
  con_def 29, 143
  con_id 29
  con_perm 30, 143
  con_status 29, 142, 149
  con_type 29
  con_type (bw_type) 29, 143
  con_type (channel_type) 29, 142
  con_type (qos) 29, 143
  user_con_type 29

Connection parameters 18-20, 133
  bandwidth 19, 30
  defaults 20, 29
  identifier 18, 29, 31
  permissions 20, 30
  type 18, 29
  user type 20, 29

Connections 4, 18-20
  adding to call 46
  best-effort 10, 19
  defined 14
  determining parameters of 65
dropping from call 52
  holding 10
  modifying parameters of 50
  multipoint 8
  multipoint-to-multipoint 8
  point-to-multipoint 8
  point-to-point 8
  Virtual Channel (VC) 5
  Virtual Path (VP) 5

Control Processor (CP) 3
Connection Management Access Protocol (CMAP) Specification

Data Transfer example 118-122
  call closedown 122
  call setup (one operation) 121
  call setup (two operations) 118
  data transmission 121

Data transmission
  interaction with CMAP signals 121
  protocols 1, 121

Defaults
  connection 20, 29
  endpoint 21, 31

drop_con command 52

drop_ep command 62

Echo mapping 21, 153

Endpoint Object 30-31
  ep_addr 30
  ep_id 30
  ep_status 31, 143, 150

Endpoint parameters 20-21, 133
  address 20, 30
  defaults 21, 31
  identifier 20
  local identifier 20, 30
  mapping 14, 21, 31
  permissions 21, 31
  receive pair 21
  transmit pair 21
  UNI parameters 31

Endpoints 20-21
  adding to call 55
  defined 14
  determining members of call 65
  determining parameters of 68
  dropping from call 62
  modifying parameters of 59

error_report maintenance operation 115

Errors
  client recovery from 116
  reporting 115, 145
  status codes 26, 28, 29, 31, 32, 145-152

Examples 117-132
  Audio/Video Server 123
  Conference Call 127
  Data Transfer 118

Extensions to CMAP 133

Exterior nodes 2

Header Object 25-27
  leid 26
  m_addr 27
  msg_id 26
  num_cons 26
  num_eps 26
  op_status 26, 145
  op_status (call_status_bit) 25, 141, 145
  op_status (connection_status_bit) 26, 141, 145
  op_status (endpoint_status_bit) 26, 141, 145
  op_status (status) 141, 145-148
  op_status (uni_status_bit) 26, 141, 145
  op_type 25, 140
  phase 26, 140
  r_addr 26
  s_addr 27

Identifiers
  call 17, 22
  client 22
  connection 18, 22, 29, 31
  endpoint 20, 22
  message 26, 32
  operation 26, 32

Interior nodes 2

invite_add_con prompt 75
invite_add_ep prompt 79
invite_change_owner prompt 89
invite_mod_ep prompt 84

Links 2, 8

Local identifier
  call 17
  endpoint 20, 30

Maintenance operations 33, 34, 108-116

Mappings 14, 153-154
  disabling defaults and permissions 153
  echo 21, 153
  endpoint 21, 31, 153
  example, for Conference Call 130
  NULL 15, 153
  receive 21, 153
  transmit 21, 153

Message objects 23
  Call Object 28
  Connection Object 29

Applied Research Laboratory
Zeus Project
Connection Management Access Protocol (CMAP) Specification

Endpoint Object 30
Header Object 25
Operation Object 32
Trailer Object 27
UNI Object 31

Messages 9, 10, 23-32
  format 23
  identifiers 23, 26, 32
  reserved fields 24, 32
  size of 133
  structuring conventions 24
  transmission 32
  unused fields 24, 32
  use in signalling 23

Minimal CMAP 1, 11
  defined 11
mod_call command 42
mod_con command 50
mod_ep command 59
Modifiability 17, 28
Monitoring 17, 28
Multidrop signalling 9, 27
Multipoint calls 1, 17
Multipoint connections 8
Multipoint-to-multipoint connections 8
Mute clients 10

NACK response 23, 33
  defined 26
NEG response 23, 33
  defined 27

Negotiation. See Parameter negotiation

network_reset maintenance operation 114

Network-Node Interface (NNI) 2, 4

Networks
  ATM 2
  management 9, 10
  requirements for CMAP 8

Nodes
  abstraction as large switch 3
  exterior 2
  interior 2

open_call command 36

Operation list
  add_con 46
  add_ep 55
  alert 111
  announce_add_con 100
  announce_add_ep 103
  announce_change_owner 106
  announce_change_root 107
  announce_close_call 99
  announce_drop_con 102
  announce_drop_ep 105
  announce_mod_call 98
  announce_mod_con 101
  announce_mod_ep 104
  change_owner 70
  change_root 73
  client_reset 112
  close_call 44
  drop_con 52
  drop_ep 62
  error_report 115
  invite_add_con 75
  invite_add_ep 79
  invite_change_owner 89
  invite_mod_ep 84
  mod_call 42
  mod_con 50
  mod_ep 59
  network_reset 114
  open_call 36
  status 108
  trace_call 65
  trace_ep 68
  verify_add_ep 92
  verify_mod_ep 95

Operation Object 32
  op_msg_id 32
  op_msg_status 32, 144, 152

Operations 33-116
  commands 33
  identifiers 32
  identifiers 26
  informing peer of status of 111
  notifications 33
  obtaining status of 108
  phases 23, 26
  prompts 33
  queries 33
  trace_ep 68
  type 25

Owner 14, 17
Parameter negotiation 35, 155-156
  in add_con 49, 155, 156
  in add_ep 58, 155, 156
  in change_owner 72, 155
  in change_root 74, 155
  in invite_add_con 78
  in invite_add_ep 82, 155, 156
  in invite_mod_ep 87
  in mod_ep 156
  in open_call 40, 155, 156
    of addresses 155
    of con_id 40, 49, 155
    of ep_addr 40, 58, 155
    of ep_con_id 40, 49, 156
    of ep_def 40, 49, 58, 156
    of ep_id 40, 58, 83, 156
    of ep_map 40, 49, 58, 78, 83, 87, 156
    of ep_perm 40, 49, 58, 156
    of lcid 40, 74, 82, 155
    of m_addr 155
    of new_owner 72, 155
    of r_addr 40, 82, 155
    of s_addr 155
    of VPI/VCI pairs 40, 49, 58, 78, 83, 88, 156

Permissions
  connection 20, 30
  endpoint 21, 31

Phases 23, 26
Point-to-multipoint connections 8
Point-to-point calls 1, 17
Point-to-point connections 8
Priority 10, 28
  CMAP levels 18

Quality of Service
  connection 29
Quality of Service (QOS) 10
  CMAP levels 19

Receive mapping 21, 153
Receive pair 21, 31
REQUEST phase 23
reserved message fields 24, 32
Reset
  of client 112
  of network 114
RESPONSE phase 23
Root 14, 17, 26

Routing 2
  "toward-the-root" algorithm 120, 126
  in broadcast packet switch 3
  in recycling switch 3
  tables 2, 3
  use of VPI/VCI fields 5
Virtual Channel (VC) 4, 5
Virtual Path (VP) 4, 5

Security 134
Session Management Layer (SML) 7
  CMAP as a component of 7
  Session Managers 7, 9

Signalling
  CMAP 14
  connections 9, 133
  inter-client 134
  meta-signalling 9
  multidrop 9, 27
  NNI 2
  surrogate 10, 27, 123, 124, 133
  UNI 2

Source discrimination 5
  use of VCI fields for 6, 127
status maintenance operation 108
Surrogate clients 10
Surrogate signalling 10, 27, 123, 124, 133
Switch Module Interface (SMI) 3
Switches
  ATM 2
    broadcast packet 2
    control processor 3
    gigabit recycling 3

Tagged data 4
trace_call command 65
trace_ep command 68
Traceability 17, 28
Trailer Object 27
  options 27
    options_size 27
Transactions 133
Transmit mapping 21, 153
Transmit pair 21, 31
Type
  call 17, 28
  connection 18, 29
Connection Management Access Protocol (CMAP) Specification

UNI (per-connection) parameters 21, 31
defined 14

UNI Object 31
  ep_con_id 31
  ep_def 31, 144
  ep_map 31, 144
  ep_perm 31, 144
  rcv_vei 31
  rcv_vpi 31
  trans_vei 31
  trans_vpi 31
  uni_status 31, 143, 150-152

unused message fields 24, 32

User type
call 18, 28
connection 20, 29

User-Network Interface (UNI) 1, 2, 4

verify_add_ep query 92
verify_mod_ep query 95

Virtual Channel (VC)
  allocation 11
  connection 5
  connection type 19
  Identifier (VCI) 4
  routing 4, 5

Virtual Path (VP)
  allocation 11
  connection 5
  connection type 19
  Identifier (VPI) 4
  routing 4, 5

VPI/VCI pairs
  allocation 11
  receive 21, 31
  transmit 21, 31