



**Communication Manager
(CM) API Reference Manual
V2.0.1**

Copyright © 2002, VIEO Inc. All rights reserved.

Communication Manager (CM) API Reference Manual V2.0.1, May 2002

All product names mentioned herein are the trademarks of their respective owners.

Documentation Conventions

Directories and filenames are presented in *italics*.

Code elements and examples are presented in `Courier New`.

Revision Information

January 2002	V2.0 Release.
May 2002	<p>V2.0.1 Release:</p> <ul style="list-style-type: none">• SID input/output tags have been removed from “Service Registration” on page 17 and “cm_service_reg” on page 35.• Connection data item <code>peer2peer</code> removed; see Table 5 on page 20.• Service data structure bit map values have been revised; see “Service Data Structure” on page 25.• Path data structure bit map values have been revised; see “Path Data Structure” on page 26.• Connection data structure bit map values have been revised; see “Connect Data Structure” on page 28.• SID tag removed from <code>cm_service_del</code>; see page 36.• New function! <code>cm_api_init</code> has been added; see page 41.• <code>rawTraffic</code> path data item added; see Table 2 on page 16.



TABLE OF CONTENTS

<i>TABLE OF CONTENTS</i>	3
Overview	5
Client View	6
Server View	9
Client/Server Interoperation	11
Peer-Peer Interoperation	13
Function Overview	14
Service and Path Query	14
Service Registration	17
Service Delete	18
Callback	18
Connect	19
Listen	22
Disconnect	22
Path Migration	23
CM API Data Structures	25
Service Data Structure	25
Path Data Structure	26
Connect Data Structure	28
CM API Function Definitions	31
cm_service_query	31
cm_path_query	33
cm_service_reg	35
cm_service_del	36
cm_connect	37
cm_listen	38
cm_disconnect	39
cm_load_ap	40
cm_api_init	41
<i>Appendix: Additional CM API Structures and Definitions</i>	43
General Definitions	43
Callback Definitions	46
Resolution Definitions	47
<i>INDEX</i>	51



[This page intentionally left blank.]



Communication Manager (CM) API

Overview

This document describes the application program interface (API) for the VIEO Communication Manager (CM). The library that supports the VIEO CM API can run in user space or kernel space, in both unified and split address space environments.

The VIEO CM API provides the following:

- Functional-level access to the objects exchanged as part of connection establishment¹
- Named services support for registration and querying
- Obtaining path information within the InfiniBand fabric
- Creating and destroying connections
- Resolving unreliable datagram (UD) services

The VIEO CM API assumes that queue pairs (QPs) and End-to-End Contexts (EECs) behave as specified in the Infiniband Architecture Specification. For example, when a teardown is initiated by the connection partner (the remote side), an application may not be able to process the disconnect request (DREQ) for some time. In this situation, the CM will modify the QP or EEC to the error state. This will generate an error for all outstanding work queue elements (WQEs, QP in error), or cause all send operations to fail (EEC in error).²

The VIEO Channel Abstraction Layer (CAL) API extends affiliated asynchronous events such that state changes to QPs or EECs generate an event.³ The VIEO CM API assumes affiliated asynchronous events behave as specified in the Infiniband Architecture Specification. For example, when a connection is being established, it is possible for the CM ready-to-use (RTU) MAD to be dropped by the fabric. In this case, the active side of the connection believes that the connection is established, and the passive side of the connection believes that the connection is pending. The Infiniband Architecture Specification requires the generation of an affiliated asynchronous event when the first packet of a Reliable Connection (RC), Reliable Datagram (RD), or Unreliable Connection (UC) message is received after the QP/EECs state has been modified to ready-to-receive (RTR).⁴ The implication of this is that the active side of the connection must send the first message.

A connection can have a primary and an alternate path. When the hardware switches from the primary path to the alternate path, the original alternate path becomes the new primary path and there is no new alternate path. The CM API provides both manual and automatic capabilities to manage the loading of new alternate paths.

-
1. See Chapter 12 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
 2. See C10-41, C10-42, O10-21 compliance statements in Chapter 10 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
 3. See the "VIEO Channel Abstraction Layer (CAL) API Reference Manual."
 4. See sections 11.6.3.1 and 12.9.6 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."



The CM communicates with an application through inter-process communication (IPC) endpoints, which are represented by the following IPC tags:

- A communication ID (CID) tag is a binding of an IPC endpoint and a CM CID. A CID is unique to a port. The CID tag is used by the CM to route all CM MADs, except requests (REQs), within a node.
- A server ID (SID) tag is used by a server and binds a service ID to an IPC endpoint. An SID is unique within a port. All CM REQ MADs are routed within the node based on this binding.

The VIEO CM API provides an exit that applications can use to examine the contents of CM MADs, allocate resources, and specify subsequent CM protocol actions.

Allocation of resources needed for a connection or UD application is the sole responsibility of the application. ***The CM API does not allocate QP/EEC or related resources.***

Client View

From the client point of view, the VIEO CM API provides a way to locate a server instance on the InfiniBand fabric, select paths to that server instance, and establish a connection. For managing UDs, the CM API also provides a way to locate a QP number associated with a service.

To locate a service, the client must know the service ID or the service name. Using either or both of these, the client can query entities within an InfiniBand fabric to locate instances of the server. The Subnet Administrator (SA) is the only component that provides this capability as specified by the InfiniBand Architecture Specification; there may be additional directory services in the future. The VIEO CM API allows for this expansion of capability through resolution tables.

The SA provides access to service records through MADs.¹ Since multiple service records are permissible, the CM API provides a mechanism to retrieve multiple records. This mechanism is implemented with iterators; that is, a client calls repeatedly the `cm_service_query` function until the `iterator` parameter indicates that there are no more records.

Once a service instance is selected, a path to the service must be obtained. Currently, only the SA provides this service; however, there may be additional suppliers of path information in the future. The path search space can be extended using the same resolution tables that are used for locating services. Given a source and the number of paths requested (n), the SA will return zero to n path records. As with service records, the CM API provides support for multiple records using an iterator. The client can connect to the server once at least one path to the server is known.

The VIEO CM API provides the `cm_connect` function to initiate the connection process; see “`cm_connect`” on page 37.² The `cm_connect` function informs the CM to start a connection and associate all CM protocol events to an IPC

1. For more information, refer to Chapter 15 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
2. The protocol used to establish this connection is detailed in Chapter 12 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."



endpoint. The IPC endpoint is bound to a local communication ID (CID) that is allocated by the CM. This IPC endpoint is represented by an IPC tag and is referred to as the CID tag; see Table 5, “Connection Data,” on page 20. The `cm_connect` function blocks until the callback function (see page 18) returns a value signalling the unblock or time expires.

The parameters of the connection are specified in data passed to the `cm_connect` function; see Table 5, “Connection Data,” on page 20. This data contains all the information required to start a connection. As the connection progresses, data items are added. This additional information is obtained from the CM MADs that are exchanged in the connection establishment protocol. After failure or success, `cm_connect` returns, and the connection state is reflected in the connect data.

The following diagrams, though not exhaustive, illustrate the process of client connect/disconnect:

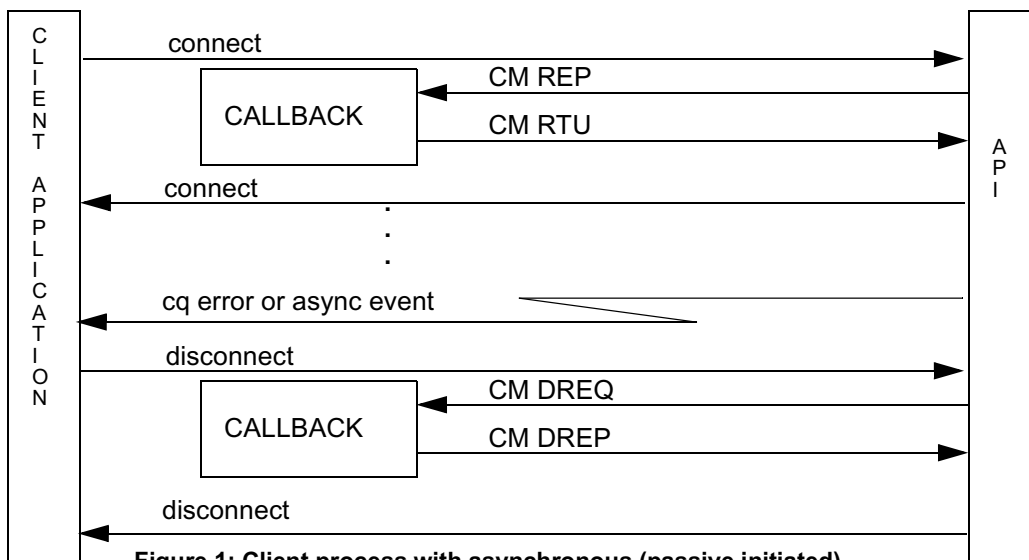


Figure 1: Client process with asynchronous (passive initiated) disconnect (CM set QP state to error) resulting in the call to disconnect.

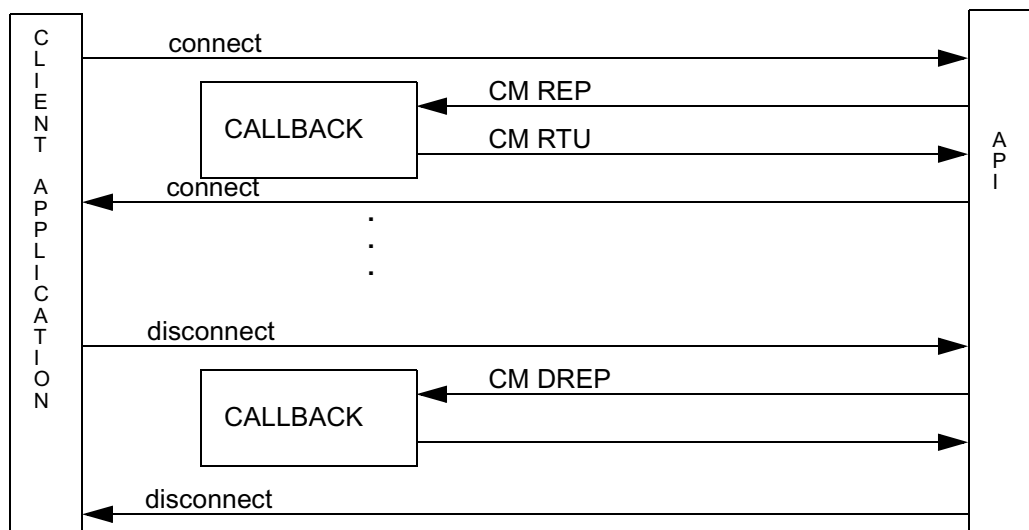


Figure 2: Client process with synchronous (active initiated) disconnect.



Unreliable datagrams (UDs) do not have a connection. However, the CM provides the capability of locating a UD QP associated with a service. As with the connection-oriented transport protocols, the client issues a connect call, but only the service ID and primary path are needed for UD. On successful completion, the connect data contains the QP number and queue key (Q_Key) associated with the service.

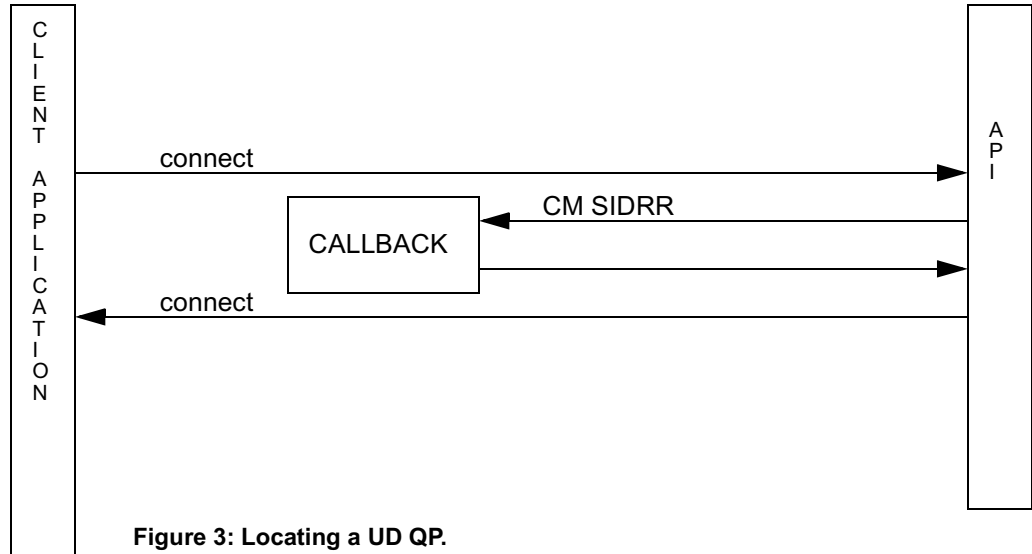


Figure 3: Locating a UD QP.



Server View

A server may register itself such that clients can locate it on the InfiniBand fabric. Currently, the CM in the node where the server resides and the subnet SA are the only two entities that will accept registration from a server, as specified by the InfiniBand Architecture Specification. Servers should normally register with both. The registration with the local CM is needed so that the CM will know the service ID and can route incoming REQs to the registrant. In the future, there may be additional entities to accept registration; the VIEO CM API allows for this extension with the addition of resolution tables.

If both the CM and the SA will be accepting registration from the server, the server must first register with the local CM. This prevents the SA from informing a client of the existence of the server prior to the registration with the local CM. When the service is registered with the CM, an IPC endpoint is created. This IPC endpoint is represented by an SID tag. The `cm_listen` function takes an SID IPC tag, the default connection data, and a pointer to the callback function as input and returns the new connection data created by the callback function as output.

External CM REQ MADs are routed to the SID IPC endpoint based on the SID in the CM REQ MAD. A newly-created local CID is associated with each CM REQ MAD. A new IPC endpoint is created and bound to this local CID. When the listen callback is executed, the `CIDipc` data item in the connect data is set to this new tag. It is the responsibility of the callback function to allocate storage for the new connection data that will be returned to `cm_listen`. The new connect data will be written by the CM API, with the exception of the context data item.



NOTE: Multiple threads within the server's address space can call the `cm_listen` function. ***Multithreaded servers cannot rely on any ordering of execution of the multiple listen calls.***



The following diagrams, though not exhaustive, illustrate the listen process:

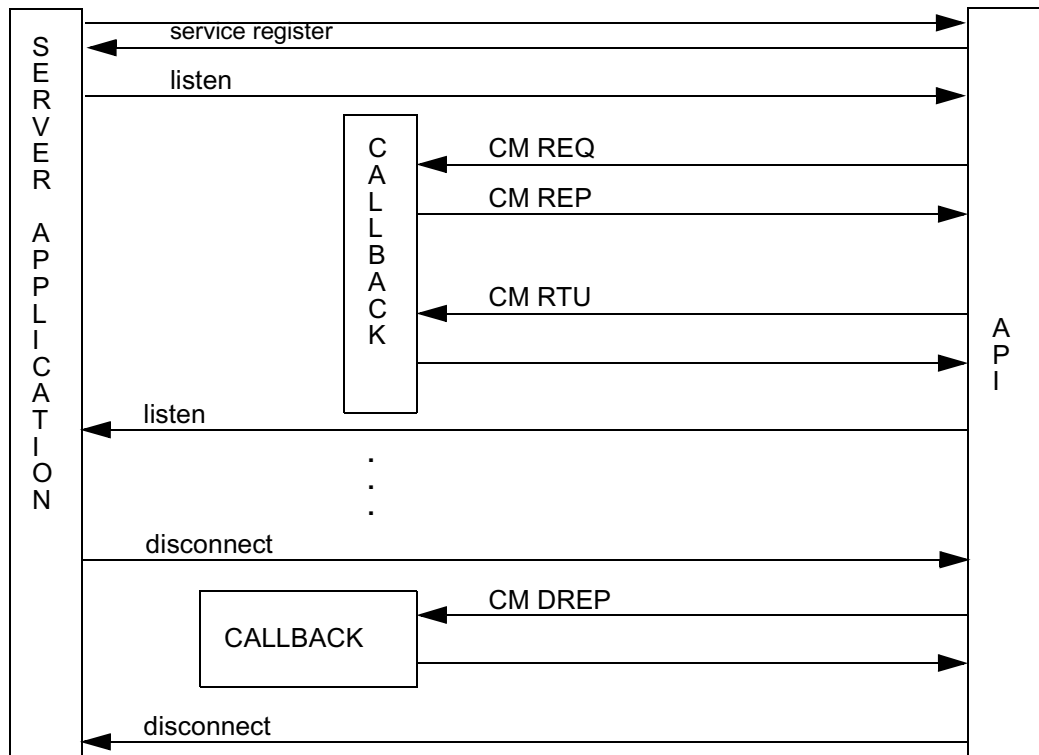


Figure 4: Server process with synchronous (server initiated)

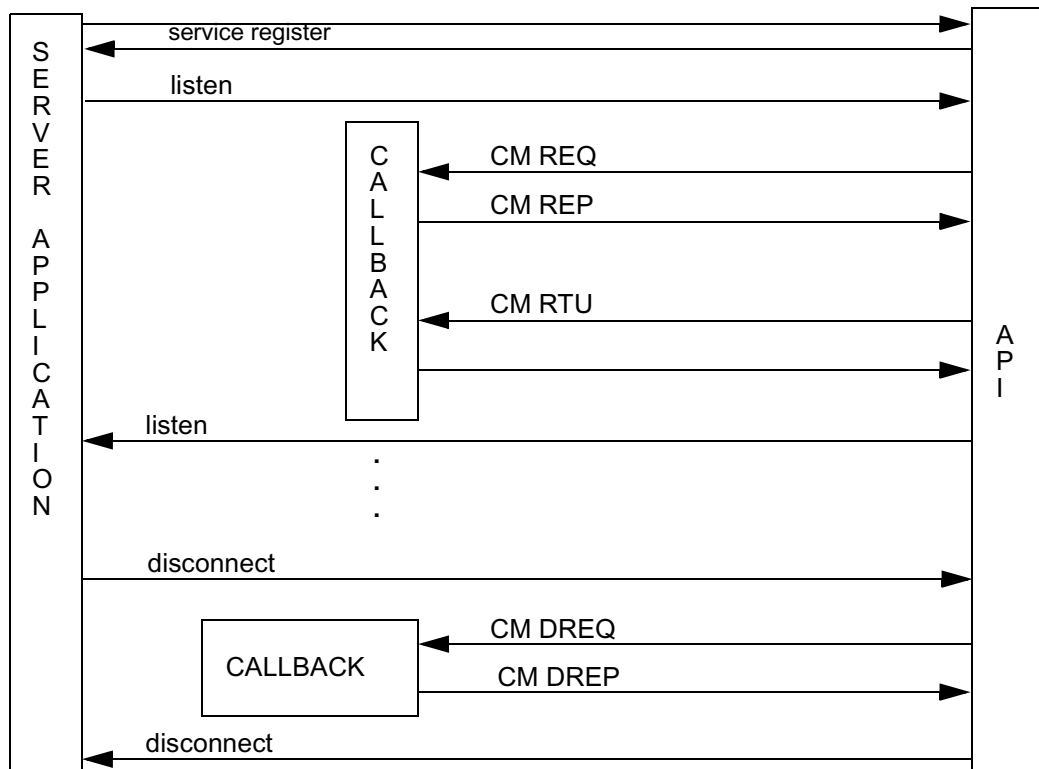


Figure 5: Server process with asynchronous ("passive" initiated) disconnect (CM set QP state to error) resulting in the call to disconnect.



Client/Server Interoperation

The following diagram illustrates the normal sequences that occur when establishing an RC connection for client-server applications:

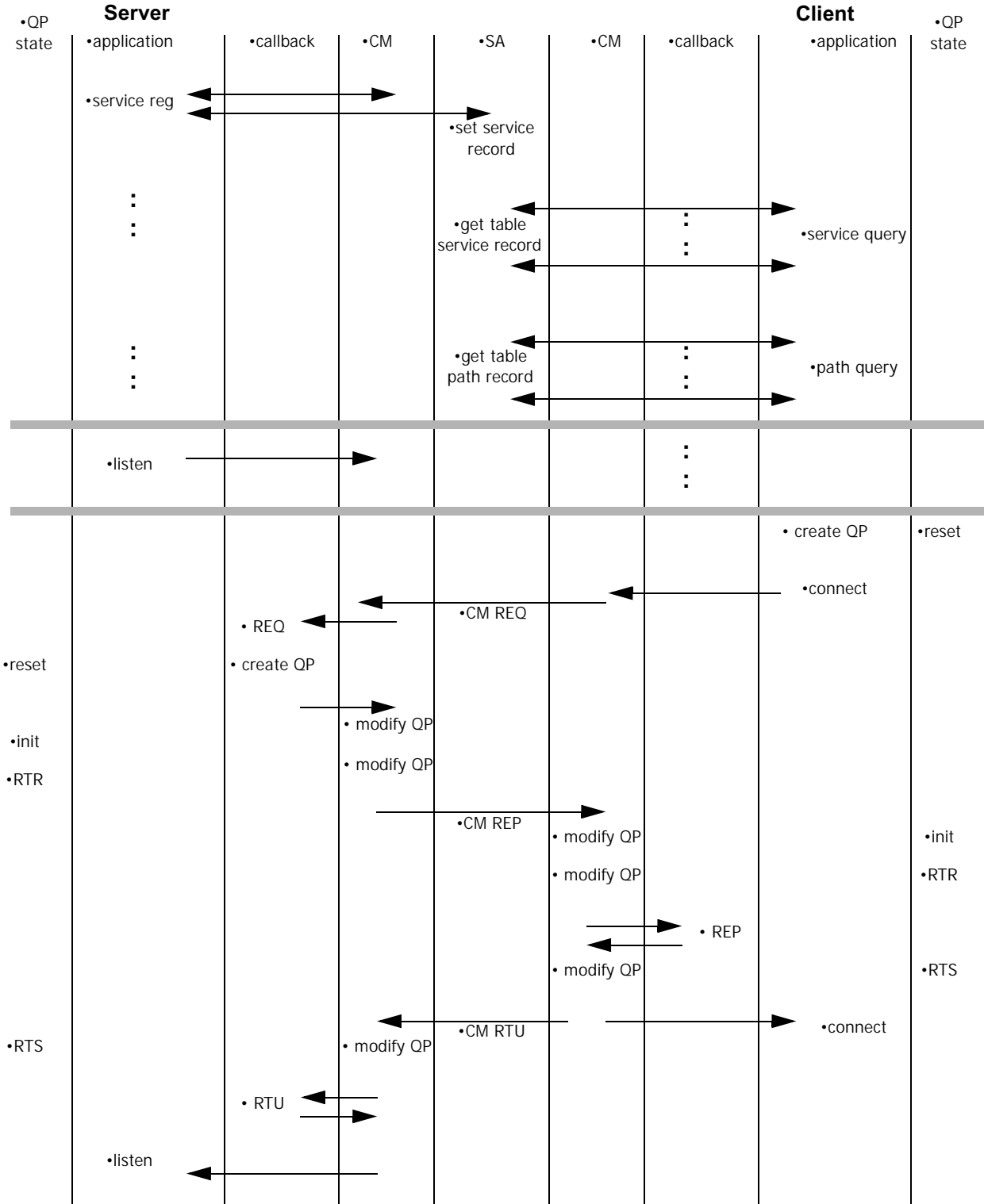


Figure 6: Establishing an RC connection for client-server applications.



The following diagram illustrates the normal sequences that occur when tearing down an RC connection:

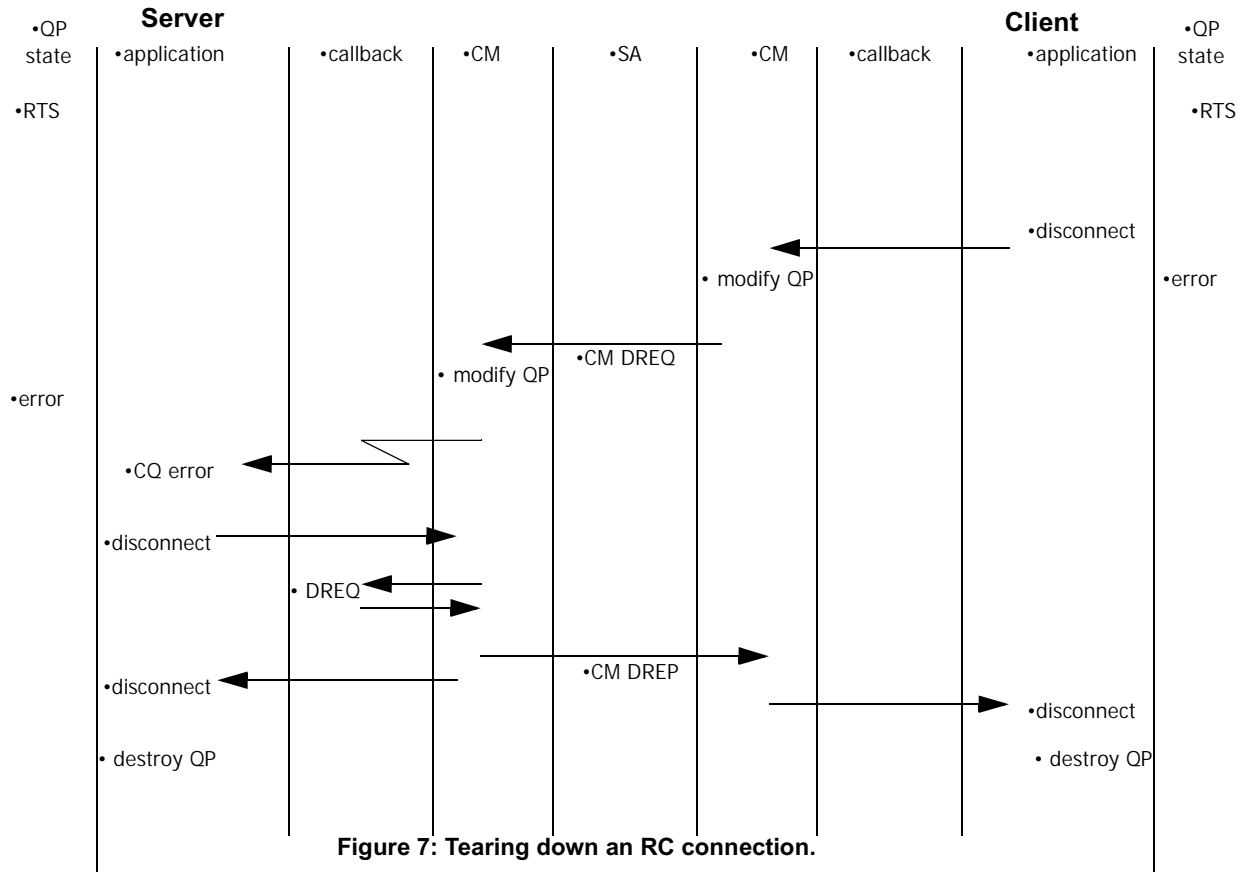


Figure 7: Tearing down an RC connection.



Peer-Peer Interoperation

The following diagram illustrates the normal sequences that occur when establishing an RC connection for peer-peer applications:

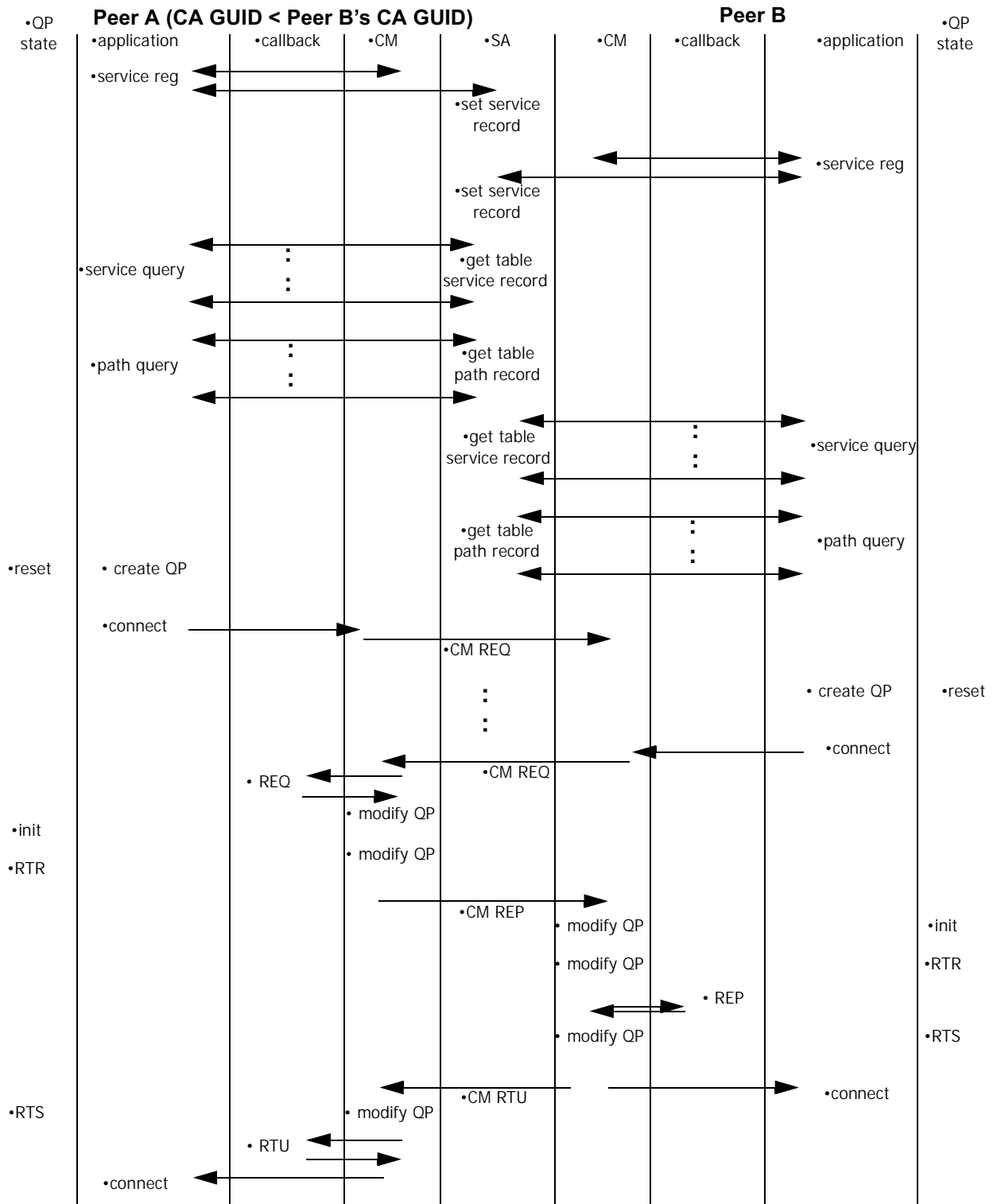


Figure 8: Establishing an RC connection for peer to peer applications.



Function Overview

Service and Path Query

Before a connection between two fabric elements can be made, the path(s) between them must be found. Obtaining a path between two fabric elements requires knowing the location of the elements on the fabric. Currently, only the Subnet Administrator (SA) is capable of providing path information; in the future, additional providers of path information may be available. The VIEO CM API allows for this expansion of capability through resolution tables.

The resolution table on page 49 contains an arbitrary number of entries, each of which contains the following information:

- A pointer to the function to execute the resolution protocol (local CM, subnet SA, LDAP, etc.)
- The fabric path to the resolution server

The resolution function on page 48 takes the following parameters:

- The 2-tuple operation type: <query/register/unregister, path/service>
- The path to the resolution server
- An iterator (both input and output) - iterators cannot be cached
- Command
- Data specific to the path or service (both input and output)



NOTE: Currently, there is no InfiniBand definition for registering a path. The 2-tuple <register, path> performs no action.

The VIEO CM API provides resolver functions for CM and SA and a default resolution table. The first entry in the default resolution table is the CM resolver. The second entry in the default resolution table is the SA resolver, with a path to the current master Subnet Manager (SM)/SA. If the resolution table passed is NULL, the default table is used.¹

Iterators are used for queries only and provide the capability of processing multiple data items. When the resolution function is called initially, the command should be set to “initial.” The function then performs the query protocol to its resolution server. If there are multiple records, the iterator is initialized to return the next record when subsequent calls are made (and the command is set to “next”). This continues until the records are exhausted (that is, the iterator is NULL) or the function is passed a command value of “end.” Iterators cannot be cached; that is, an iterator is only valid while the command is “next” and ceases to be valid when it is NULL, or the command is “end.”

The data specific to the query is used as both input and output. When the command is set to “initial,” the data is used as a template specifying the search criteria.² From then on, the data is used as output and contains the current record. The storage needed by the iterator is allocated and freed by the resolution function.

1. For more information about resolution, see “Resolution Definitions” on page 47.

2. For more information, refer to Chapter 15 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."



Service Query

The service query function, `cm_service_query`, is used to locate a specific service. The service specification is provided by the information listed in the following table. This information is returned by `cm_service_query` for each matching record found.

Table 1: Service Data

Name	Description
<code>version</code>	Version of data (this specification is version 1).
<code>magic</code>	Value that uniquely identifies the structure type.
<code>context</code>	Opaque data item for use by application.
<code>caHandle</code>	Specifies the local channel adapter (CA).
<code>port</code>	Specifies the local port as defined in Chapter 2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>SID tag</code>	Endpoint to be associated with the service. If this is NULL, a new endpoint will be created. If it is not NULL, the service is added to the existing endpoint.
<code>componentMask</code>	Bitmap; least significant bit (LSB) is a component (member) following this data item. If bit is set, the the corresponding data item is to be included in the search criteria.
<code>invalidMask</code>	Bitmap with same ordering as <code>componentMask</code> . If there is an error processing the request, this data item specifies the components that caused the error.
<code>servicelease</code>	Amount of time before SA deletes the service record; see section 15.2.5.15 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>partition</code>	P-key needed to access service; see section 10.9 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>sflags</code>	Meaning is service dependent.
<code>gflags</code>	See section 15.2.5.15 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>gid</code>	GID of service.
<code>serviceID</code>	If set to 0, then the SID will be allocated by the API (OS-allocated SID).
<code>serviceName</code>	UTF-8 encoded name, up to 124 characters in length; NULL terminated.



Path Query

The path query function, `cm_path_query`, is used to find paths from a port on a local CA to a port-specific service on the fabric. The information used to specify the source, destination, maximum number of paths to find, and other information is specified in the following table. This information is returned by `cm_path_query` for each path found.¹

Table 2: Path Data

Name	Description
<code>version</code>	Version of data (this specification is version 1).
<code>magic</code>	Value that uniquely identifies the structure type.
<code>context</code>	Opaque data item for use by application.
<code>caHandle</code>	Specifies the local channel adapter (CA).
<code>port</code>	Specifies the local port as defined in Chapter 2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>componentMask</code>	Bitmap; LSB is a component (member) following this data item. If bit is set, the the corresponding data item is to be included in the search criteria.
<code>invalidMask</code>	Bitmap with same ordering as <code>componentMask</code> . If there is an error processing the request, this data item specifies the components that caused the error.
<code>numPaths</code>	Number of path records to return; see section 15.2.5.17 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>serviceID</code>	Service ID from Table 1, "Service Data," on page 15.
<code>pkey</code>	Partition key (P_Key) from service information in Table 1, "Service Data," on page 15; see section 10.9 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>pkeyIndex</code>	P_Key index for source port; see section 10.9.2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>MTUsel</code>	See section 15.2.5.17 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>MTU</code>	MTU of the path; see section 12.7.28 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>rateSel</code>	See section 15.2.5.17 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>rate</code>	1-30 Gb/sec.
<code>lifetimeSel</code>	See section 15.2.5.17 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>lifetime</code>	See section 15.2.5.17 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
<code>serviceLevel</code>	Service level (SL) for path.
<code>injectionRate</code>	Derived from rate.
<code>lclAckTimeout</code>	Derived from lifetime.

1. For more information, refer to section 15.2.5.17 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."



Table 2: Path Data

Name	Description
rawTraffic	Set if raw packet path (no p-key needed).
subnetLocal	Set if source and destination reside in same subnet.
srcLID	LID of source.
srcGID	GID of source.
destLID	LID of destination.
destGID	GID of destination.
trafficClass	See section 5.2.2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
flowLabel	See section 5.2.2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
hopLimit	See section 5.2.2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."

Service Registration

Before a service can be located, the service must register itself using the service registration function, `cm_service_reg`. The service registration function utilizes the following items:

- A resolution table; see “Service and Path Query” on page 14.
- The service data; see Table 1, “Service Data,” on page 15.
- A timeout.

The SID tag in the service data represents a binding between the service ID and an IPC endpoint. The timeout specifies how long to wait for completion of the registration operation. If the input SID tag pointer is NULL, an IPC endpoint is created and returned as the output SID tag. If the input SID tag is currently active, this service is bound to the input SID tag, and the input SID tag is copied to the output SID tag.

The node registration process is split into two parts; node internal and node external. The VIEO CM API provides an SA registration resolver function.¹ It implements the SA service and path protocols specified in Chapter 15 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."

For each entry in the resolution table, the node registration process external to the node is as follows: call the resolution function with the specified service data and resolver path.



NOTE: When the pending connection (or completed connection) is terminated, the association between the CM IPC tag and the local communication ID (CID) is removed, and the local CID is released.

1. For more information about the SA resolver function, see “CM and SA Resolver Function Prototypes” on page 48.



Service Delete

The service delete function, `cm_service_del`, is the inverse of service registration and is performed in the reverse order; that is, if the registration order is CM, SA, and LDAP, the unregistration order will be LDAP, SA, and CM. The SA record is deleted, all pending or active connections using the service ID are terminated, and all associations between the port, the IPC tag, and the service ID are removed. All resources related to the registration are released.



NOTE: Deleting a service does not affect established connections.

Callback

The callback function is specified through a parameter to the `cm_connect`, `cm_listen`, `cm_disconnect`, and `cm_load_ap` functions. Each of these functions block until a CM MAD is received or time expires. When a MAD is received, the associated callback is invoked (in the context of an application's call). When the callback function returns, and the connection state is established, rejected, or terminated, the CM API call unblocks. Otherwise, when the callback function returns, the CM API call continues to block until either a CM MAD is received or time expires.

The callback function specifies the next CM protocol action. The actions allowed are:

- reject
- message receipt acknowledgment (MRA)
- next

The reject action causes a CM REJ MAD to be sent, and terminates the connection. Before returning, the callback function should set the `rejCode` in the connection data. The callback may also set the `rejARI` data in the connection data before returning.

The MRA action causes a CM MRA MAD to be sent. The purpose of the MRA is to inform the remote CM that extra time is required to process the last request from the remote CM. The MRA must specify the amount of time the remote CM should wait, which is specified with the `serviceTimeout` data item in the connect data (see Table 5 on page 20). Before returning, the `serviceTimeout` must be set. After the CM MRA MAD is sent, the callback is again called with an indication that an MRA has been sent. This allows the callback to continue operation.

The next action causes the next normal CM MAD to be sent. For connection establishment, this is either CM REP (passive side), or CM RTU (active side).



The following tables specify the QP/EEC state relative to the MAD received or sent.¹

Table 3: QP/EEC State relative to CM MAD Received

State prior to MAD	Event	State prior to callback
No QP/EEC, RESET	CM REQ	No QP/EEC, RESET
INIT	CM REP	RTR
RTR	CM RTU	RTS
RTS	CM DREQ	ERROR
ERROR	CM DREP	RESET
Any	CM REJ	RESET
No QP/EEC, RESET	CM SIDR	No QP/EEC, RESET

Table 4: QP/EEC State relative to CM MAD Sent

State after callback	MAD sent	State after MAD sent
INIT	CM REQ	INIT
RESET, INIT, RTR	CM REP	RTR
RTR	CM RTU	RTS
RTS	CM DREQ	ERROR
ERROR	CM DREP	RESET
Any	CM REJ	RESET
RESET, INIT, RTR, RTS	CM SIDRR	RTS

Connect

The connect function, `cm_connect`, initiates a connection or locates a UD QP associated with an SID. The connect function utilizes the following items:

- Connect data; see Table 5, “Connection Data,” on page 20
- A callback function CM; see “Callback” on page 18

The connect function creates an IPC endpoint and notifies the CM to initiate a connection. The CID tag is stored in the connection data. The connect function blocks until the connection state becomes established or rejected or time expires.² The timeout is set by the `lclCMtimeout` and `maxCMretries` connect data items.

When creating a peer to peer connection, the peer that assumes passive mode will have its callback function invoked with the `CMapiCBmaskREQ` bit set.

The connect data items under the “When first available client” column in the following table marked “source,” “connect,” or “QP create” must be valid before calling the connect function (some data items are specific to transport type).

1. The data needed for each transition is specified in sections 11.2.3.2 and 11.2.6.2 of the “InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001.”
2. For more information about callback flags, see “Callback Definitions” on page 46.



Table 5: Connection Data

Name	Description	When first available client	When first available server
version	Version of data (this specification is version 1).	source	source
magic	Value that uniquely identifies the structure type.	source	source
context	Opaque data item for use by application.	source	source
caHandle	Specifies the local channel adapter (CA).	source	source
port	Specifies the local port as defined in Chapter 2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."		
tagCID	IPC tag used for a specific connection.	CM REQ	CM REQ Callback
componentMask	Bitmap; LSB is a component (member) following this data item. If bit is set, the corresponding data item is to be included in the search criteria.	source	source
invalidMask	Bitmap with same ordering as componentMask. If there is an error processing the request, this data item specifies the components that caused the error.	source	source
state	See section 12.9.7 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."	source	source
lclCommID	ID assigned by CM; unique to port only.	CM REQ	CM REQ
rmtCommID	ID assigned by remote CM.	CM REP	CM REQ
adapterGuid	GUID of the node.	Connect call	CM REQ
lclCMqkey	Q_Key for CM.	Connect call	CM REQ
lclStartingPSN	Initial packet sequence number (PSN) for RC, RD, UD connection.	source	source
lclQPN	QP number.	QP create	QP create
QPhandle	Value returned from QP create.	QP create	QP create
lclQkey	Q_Key value for QP (RD).	QP create	QP create
lclEECN	Local End-to-End Context (EEC) number.	EEC create	EEC create
eecHandle	Value returned from EEC create.	EEC create	EEC create
lclRNRretry	Number of times to retry an RNR before failing (RC/RD).	source	source
lclFlowControl	If set, end-to-end flow control is enabled.	source	source
rmtStartingPSN	Starting PSN for connection partner (RC, RD, UC).	CM REP	CM REQ
rmtQPN	QP number of connection partner.	CM REP	CM REQ
rmtQkey	Q_Key of connection partner (RD).	CM REP	CM REQ
rmtEECN	EEC of connection partner.	CM REP	CM REQ



Table 5: Connection Data

Name	Description	When first available client	When first available server
rmtRNRretry	Connection partner's RNR retry count.	CM REP	CM REQ
rmtFlowControl	Connection partner has end-to-end flow control enabled.	CM REP	CM REQ
serviceType	Type of transport (RC, RD, UC, UD).	connect call	CM REQ
partitionKey	P_Key for connection.	connect call	CM REQ
lclRespResources		QP create	QP create
lclInitDepth		QP create	QP create
existRDC	If set, a Reliable Datagram Domain (RDD) already exists.	source	source
lclCMtimeout	Amount of time before local CM issues an MRA or reject.	source	CM REQ
rmtCMtimeout	Amount of time before partner CM issues an MRA or reject.	source	CM REQ
maxCMretries	Number of times CM retries before failing.	connect call	CM REQ
retryCount	Number of times transport will retry before failing.	connect call	CM REQ
rmtRespResources		CM REP	CM REQ
rmtInitDepth		CM REP	CM REQ
targetAckDelay		connect call	CM REQ
failoverAccepted	If set, failover is accepted.	CM APR	
primPath	Primary path data on which data will traverse.	connect call	CM REQ
altPath	Alternate path data on which data will traverse.	connect call	CM REQ
autoLAP	Enables auto-LAP capability (default).	connect call	not used
apStatus	Status after loading alternate path.	CM APR	CM APR
sidrCode	Status response after a service ID resolution (SIDR).	CM SIDRR	not used
serviceTimeout	If MRA received, amount of time to wait for next protocol message.	CM MRA	CM MRA
privateData	Private data associated with current MAD.	all CM MAD	all CM MAD
rejCode	Rejection code if connection is rejected; see "Rejection Codes" on page 44.	CM REJ	CM REJ
rejARI	Additional data associated with reject.	CM REJ	CM REJ



Listen

The listen function, `cm_listen`, waits for a connection request or a service ID resolution (SIDR) request. The listen function utilizes the following items:

- The SID IPC tag created by the service register API function
- A pointer to the callback function
- Connect data; see Table 5, “Connection Data,” on page 20
- The CID IPC tag for the new connection and the connection data created by the callback function are returned by the listen function as output

The listen function blocks until the connection state is established or rejected, or time expires.¹ The timeout is set by the `lclCMtimeout` and `maxCMretries` connect data items.



NOTE: More than one thread can call the listen function with the same callback and SID IPC tag.

Disconnect

The disconnect function, `cm_disconnect`, is used to initiate a teardown of an established connection, or continue the teardown of a terminating connection. This function utilizes the following items:

- A pointer to the callback function
- Connect data; see Table 5, “Connection Data,” on page 20

The disconnect function blocks until the state is terminated or time expires.¹ The timeout is set by the `lclCMtimeout` and `maxCMretries` connect data items.

The disconnect function examines the current QP and connection state and determines if there is a pending disconnect request (DREQ) from the remote CM. If there is a pending DREQ, the callback function is called. Otherwise, the disconnect function initiates the connection teardown by sending a DREQ.

1. For more information about callback flags, see “Callback Definitions” on page 46.



Path Migration

When a connection is created, the connection can specify both a primary and an alternate path. Initially, the transport data is transferred using the primary data path. Should the primary path fail, the fabric hardware automatically switches to the alternate path. This is referred to as path migration.¹

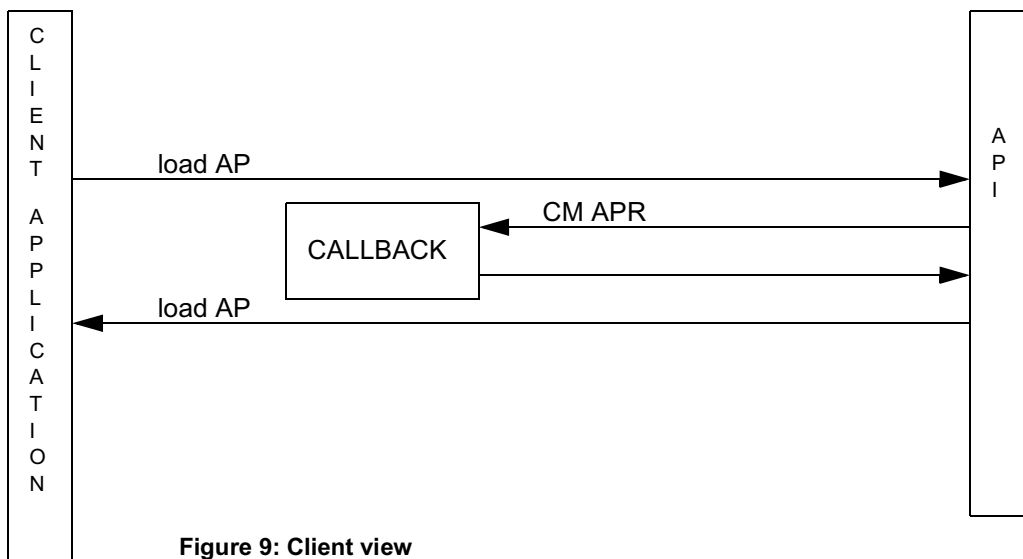
When a path migrates from the primary path to the alternate path, an affiliated asynchronous event is generated. This event can be used by the VIEO CM API to trigger an SA query for a new alternate path, and then load the new alternate path using the load alternate path/alternate path response (LAP/APR) CM protocol; this capability is called auto-LAP.²

The VIEO CM API auto-LAP capability is enabled by the `autoLap` connect data item (see Table 5 on page 20). The algorithm for auto-LAP path selection is as follows:

1. Create an initial query template using the SGID/DGID in the primary path.
2. Query the SA for the maximum number of paths (127).
3. Select the first path that does not match the alternate path. If no path is found and there are multiple GIDs for the port, select the next GID as the SGID and go to Step 2. If all GUIDs for the port are exhausted, exit.
4. Mark the alternate path as the new primary path; mark the newly found path as the alternate path and execute the LAP protocol .

If an application chooses, it can disable the auto-LAP capability by clearing the auto-LAP enable flag in the connect data. In this case, an application must perform a path query, select the new alternate path, and load the new alternate path using the load AP function (see “`cm_api_init`” on page 41).

The following diagrams illustrate `cm_load_ap`:



1. For more information about path migration, refer to section 10.4 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
2. "Auto-LAP" is a term defined by the VIEO CM.

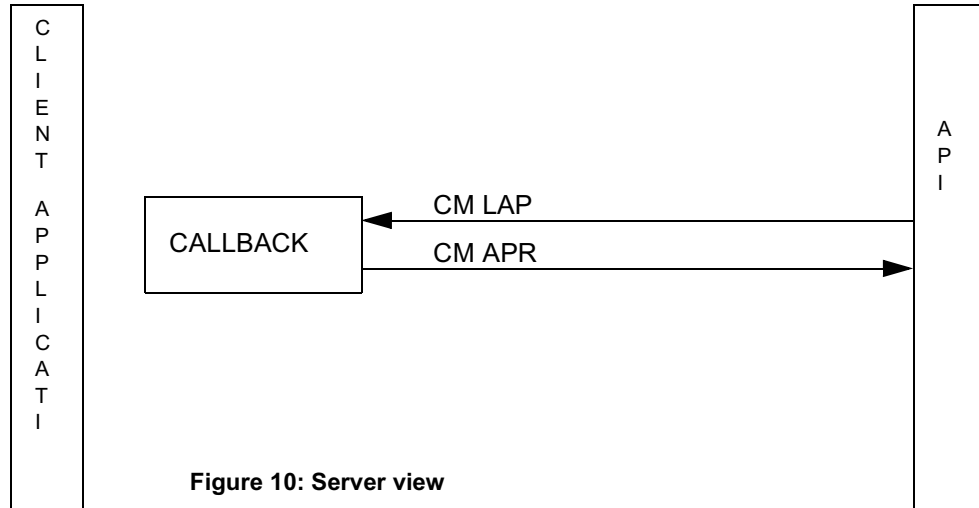


Figure 10: Server view



CM API Data Structures

Service Data Structure

Struct Member Bit Map Values (map to SA component bitmap values)

```
#define CM_SERV_serviceLease    (0x00000002ULL)
#define CM_SERV_partition      (0x00000004ULL)
#define CM_SERV_sflags         (0x00000008ULL)
#define CM_SERV_gflags         (0x00000010ULL)
#define CM_SERV_gid            (0x00000040ULL)
#define CM_SERV_serviceID      (0x00000080ULL)
#define CM_SERV_serviceName    (0x00000020ULL)
```

Definition

```
#define CM_API_VERSION          1

#define CM_API_EYE_SERVICE     "SERV"
#define CM_API_EYE_SERVICE_32  0x53455256

typedef struct _CMapiMagic_t {
    uint8_t      eye[4];          // Eye catcher above
    uint8_t      reserved[4];
} CMapiMagic_t;

#define CM_MAX_SERVICE_NAME (992 / 8) // See 15.2.5.151
typedef uint8_t CMapiServName_t[CM_MAX_SERVICE_NAME];

typedef struct _CMapiService_t {

    uint32_t      version;
    CMapiMagic_t  magic;
    void          *context;

    // The following two fields are used to specify the adapter and
    // port from which the service queries should be sent

    Cal_handle_t  caHandle;
    int16_t       port;

    // The componentMask is used to specify the search criteria, when
    // used as input, and specifies the fields that are valid when
    // provided as output

    uint64_t      componentMask;
    uint64_t      invalidMask;
```

1. References to the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001." within a code definition or sample will be noted with the section number only.



```
// Currently, only the SA service records are represented. As
// other service information sources become available, they will
// be added.

    uint32_t      serviceLease; // Service lease period
    uint16_t      partition;    // Partition of this service
    uint16_t      sflags;      // Service specific flags (12)
    uint8_t       gflags;      // Service generic flags (4)
    uint8_t       gid[16];     // GID of service
    uint64_t      serviceID;    // Service ID
    CMapiServName_t serviceName; // Service name (UTF-8 encoded)
} CMapiService_t;

#define CM_API_INIT_SERVICE_T(_SP)
{
    memset((void *)_SP, 0, sizeof(CMapiService_t));
    *((uint32_t *) (_SP)->magic.eyec) = CM_API_EYE_SERVICE_32;
    (_SP)->version = CM_API_VERSION;
}
```

Path Data Structure

Struct Member Bit Map Values (map to SA component mask bits)

```
#define CM_PATH_srcLID          (0x00000004ULL)
#define CM_PATH_srcGID         (0x00000008ULL)
#define CM_PATH_destLID        (0x00000010ULL)
#define CM_PATH_destGID        (0x00000020ULL)
#define CM_PATH_rawTraffic     (0x00000040ULL)

#define CM_PATH_flowLabel      (0x00000100ULL)
#define CM_PATH_hopLimit       (0x00000200ULL)
#define CM_PATH_trafficClass   (0x00000400ULL)

#define CM_PATH_numPaths       (0x00001000ULL)
#define CM_PATH_pkey           (0x00002000ULL)
#define CM_PATH_serviceLevel   (0x00004000ULL)
#define CM_PATH_MTUUseL        (0x00008000ULL)
#define CM_PATH_MTU            (0x00010000ULL)
#define CM_PATH_rateSel        (0x00020000ULL)
#define CM_PATH_rate           (0x00040000ULL)
#define CM_PATH_lifetimeSel    (0x00080000ULL)
#define CM_PATH_lifetime       (0x00100000ULL)
#define CM_PATH_injectionRate  (0x00200000ULL)

// The following are not part of the SA path record

#define CM_PATH_lclAckTimeout   (0x10000000ULL)
#define CM_PATH_subnetLocal     (0x20000000ULL)
#define CM_PATH_serviceID       (0x40000000ULL)
#define CM_PATH_pkeyIndex       (0x80000000ULL)
```



Definition

```

#define CM_API_EYE_PATH      "PATH"
#define CM_API_EYE_PATH_32  0x50415448

typedef struct _CMapiPath_t {

    uint32_t      version;
    CMapiMagic_t  magic;
    void          *context;

    // The following two fields are used to specify the adapter and
    // port from which the service queries should be sent
    Cal_handle_t  caHandle;
    int16_t       port;

    // The componentMask is used to specify the search criteria, when
    // used as input, and specifies the fields that are valid when
    // provided as output
    uint64_t      componentMask;
    uint64_t      invalidMask;
    uint8_t       numPaths;      // Number of path records to return

    // The following is set by the application and ties the path to
    // a particular service.
    uint64_t      serviceID;     // Service associated with path

    // The following members are maintained by the SA.
    // They should not be changed.
    uint16_t      pkey;          // Partition key
    uint8_t       pathMTUssel;  // See 15.2.5.17
    uint8_t       pathMTU;      // See 12.7.28

    uint8_t       rateSel;      // See 15.2.5.17
    uint8_t       rate;         // See 15.2.5.17
    uint8_t       lifetimeSel;  // See 15.2.5.17
    uint8_t       lifetime;     // See 15.2.5.17
    uint8_t       serviceLevel; // See 12,7,16

    uint8_t       injectionRate; // See 12.7.25 & 9.11.1
    uint8_t       lclAckTimeout; // See 12.7.34 & 9.7.6.1.3
    uint8_t       rawTraffic;    // see 15.2.5.17
    uint8_t       subnetLocal;  // If set, no GRH needed
                                // See 12.7.7
    uint16_t      srcLID;        // See 12.7.11
    uint8_t       srcGID[16];   // See 12.7.10

    uint16_t      destLID;      // See 12.7.21
    uint8_t       destGID[16];  // See 12.7.20

    uint8_t       trafficClass; // See 12.7.17
    uint32_t      flowLabel;    // See 12.7.18
    uint8_t       hopLimit;     // See 12.7.19

} CMapiPath_t;

#define CM_API_INIT_PATH_T(_SP)
{
    memset((void *)_SP, 0, sizeof(CMapiPath_t));
    *((uint32_t *)((_SP)->magic.eyeptr)) = CM_API_EYE_PATH_32;
    (_SP)->version = CM_API_VERSION;
}
    
```



Connect Data Structure

Struct Member Bit Map Values

```
#define CM_CON_state (0x000000000001ULL)
#define CM_CON_lclCommID (0x000000000002ULL)
#define CM_CON_rmtCommID (0x000000000004ULL)
#define CM_CON_lclAdaptGuid (0x000000000008ULL)

#define CM_CON_lclCMqkey (0x000000000010ULL)
#define CM_CON_lclStartingPSN (0x000000000020ULL)
#define CM_CON_lclQPN (0x000000000040ULL)
#define CM_CON_lclQPhandle (0x000000000080ULL)

#define CM_CON_lclQkey (0x000000000100ULL)
#define CM_CON_lclEECN (0x000000000200ULL)
#define CM_CON_lclEEChandlep (0x000000000400ULL)
#define CM_CON_lclRNRretry (0x000000000800ULL)

#define CM_CON_lclFlowControl (0x000000010000ULL)
#define CM_CON_rmtStartingPSN (0x000000020000ULL)
#define CM_CON_rmtQPN (0x000000040000ULL)
#define CM_CON_rmtQkey (0x000000080000ULL)

#define CM_CON_rmtEECN (0x000000100000ULL)
#define CM_CON_rmtRNRretry (0x000000200000ULL)
#define CM_CON_rmtFlowControl (0x000000400000ULL)
#define CM_CON_serviceType (0x000000800000ULL)

#define CM_CON_partitionKey (0x000001000000ULL)
#define CM_CON_lclRespResources (0x000002000000ULL)
#define CM_CON_lclInitDepth (0x000004000000ULL)
#define CM_CON_existRDC (0x000008000000ULL)

#define CM_CON_lclCMtimeout (0x000010000000ULL)
#define CM_CON_rmtCMtimeout (0x000020000000ULL)
#define CM_CON_maxCMretries (0x000040000000ULL)
#define CM_CON_retryCount (0x000080000000ULL)

#define CM_CON_rmtRespResources (0x000100000000ULL)
#define CM_CON_rmtInitDepth (0x000200000000ULL)
#define CM_CON_targetAckDelay (0x000400000000ULL)
#define CM_CON_failoverAccepted (0x000800000000ULL)

#define CM_CON_priPath (0x001000000000ULL)
#define CM_CON_altPath (0x002000000000ULL)
#define CM_CON_autoLAP (0x004000000000ULL)
#define CM_CON_apStatus (0x008000000000ULL)

#define CM_CON_sidrCode (0x010000000000ULL)
#define CM_CON_serviceTimeout (0x020000000000ULL)
#define CM_CON_privateData (0x040000000000ULL)
#define CM_CON_rejCode (0x080000000000ULL)

#define CM_CON_rejARI (0x100000000000ULL)
```



Definition

```

#define CM_API_EYE_CONNECT      "CONN"
#define CM_API_EYE_CONNECT_32  0x434f4e4e

typedef struct _CMapiConnect_t {

    uint32_t      version;
    CMapiMagic_t  magic;
    void          *context;

    // The following two fields are used to specify the adapter and
    // port from which the service queries should be sent

    Cal_handle_t  caHandle;
    int16_t       port;

    // The following is where the CID IPC tag is saved
    CMapiIPctag_t tagCID;

    uint64_t      componentMask; // Specifies valid fields
    uint64_t      invalidMask;

    // The following members are set by the CM
    // They are read-only to an application

    CMapiState_e  state;          // Connection state
    uint32_t      lclCommID;      // Local CID, see 12.7.1
    uint32_t      rmtCommID;      // Remote CID, see 12.7.2
    uint8_t       adapterGuid[8]; // GUID of sending CA
    uint32_t      lclCMqkey;      // Q_Key of sending CM

    // The following members are always set by the local application

    uint32_t      lclStartingPSN; // Starting PSN of local sender
    uint32_t      lclQPN;         // Applications QP number
    Cal_handle_t  qpHandle;       // Handle for QP
    uint32_t      lclQkey;        // Local Q_Key (RD only)
    uint32_t      lclEECN;        // EEC number (RD only)
    Cal_handle_t  eecHandle;      // Handle for EEC
    uint8_t       lclRNRretry;    // Number of RNR retries allowed
    uint8_t       lclFlowControl; // See 12.7.26

    // The following members are always set by the remote
    // application, and are read-only to the local application.

    uint32_t      rmtStartingPSN; // Starting PSN of local sender
    uint32_t      rmtQPN;         // Applications QP number
    uint32_t      rmtQkey;        // Local Q_Key (RD only)
    uint32_t      rmtEECN;        // EEC number (RD only)
    uint8_t       rmtRNRretry;    // Number of RNR retries allowed
    uint8_t       rmtFlowControl; // See 12.7.26

    // The following members can only be set when the local
    // application is initiating a connection (cm_connect).

    CMapiProto_e  serviceType; // Transport type being initiated
    uint32_t      partitionKey; // P_Key for connection
                                // See 12.7.24
    uint8_t       lclRespResources; // See 12.7.29
    uint8_t       lclInitDepth;    // See 12.7.30
    uint8_t       existRDC;        // If 1, RD connection already exists

```



```
uint8_t    lclCMtimeout; // See 12.7.5
uint8_t    rmtCMtimeout; // See 12.7.4
uint8_t    maxCMretries; // See 12.7.27
uint8_t    retryCount;   // See 12.7.38

// The following members can only be set when the local
// application is sinking a connection (cm_listen).
uint8_t    rmtRespResources; // Actual responder resources
uint8_t    rmtInitDepth;     // Actual initiator depth
uint8_t    targetAckDelay;   // See 12.7.33
uint8_t    failoverAccepted; // See 12.7.36

// The following members specify the primary and alternate path
// for the connection.
CMapiPath_t primPath;
CMapiPath_t altPath;
uint8_t     autoLAP;        // If set, auto-LAP enabled

// The following data areas are fields specific to particular CM
// MADS

// Alternate path status (see page 45)
CMapiAPcodes_e apStatus; // See 12.8.2.1

// SIDR code (UD only) (see page 45)
CMapiSIDcodes_e sidrCode; // See 12.11.2

// MRA service timeout
uint8_t     serviceTimeout; // See 12.7.32

// Private data buffer area (page 44)
// Note that the private data is in network byte order.
uint8_t     privateData[CM_MAX_PRIVATE_DATA_SIZE];

// Reject code (see page 44)
// If reject, this has the additional reason. Note that the
// addition reject information (ARI) is in network byte order.
CMapiRejCodes_e rejCode; // See 12.6.7.2
uint8_t     rejARI[CM_MAX_REJ_ARI];

} CMapiConnect_t;

#define CM_API_INIT_PATH_T(_SP)
{
    memset(_SP, 0, sizeof(CMapiPath_t));
    *((uint32_t *)((_SP)->magic.ey)) = CM_API_EYE_CONNECT_32;
    (_SP)->version = CM_API_VERSION;
}
```

Default values for Connect parameters

```
#define CM_DEFAULT_RNR_RERTRY    7 // lclRNRretry
#define CM_DEFAULT_PKEY         0xffff // partitionKey
#define CM_DEFAULT_RESOURCES    1 // lclRespResources
#define CM_DEFAULT_IDEPH       1 // lclInitDepth
#define CM_DEFAULT_CM_TIMEOUT   18 // lclCMtimeout
#define CM_DEFAULT_CM_RETRIES   7 // maxCMretries
#define CM_DEFAULT_RETRIES     7 // retryCount
#define CM_DEFAULT_ACK_DELAY    8 // targetAckDelay
```



CM API Function Definitions

cm_service_query

DESCRIPTION Find a service. This function must be called until the iterator becomes NULL, or called with the command set to CMapiResCmdEnd. Failure to do so will result in incorrect (stale) service data.

SYNTAX Status_t cm_service_query (CMapiResTable_t *resp, CMapiResCmd_e cmd, uint64_t timeout, void **iterator, CMapiParamRes_t servp);

PARAMETERS **Input:**

*resp: Pointer to a service resolution table.
 cmd: Command.
 timeout: Number of microseconds to wait.
 **iterator: Initially passed in as NULL. For subsequent calls, iterator is the value returned by the prior call.
 servp: Pointer to service template.

Output:

servp: Pointer to a current service.
 **iterator: Updated to reflect next item to access; if NULL, no more services are available.

RETURNS VSTATUS_OK: Status good.
 VSTATUS_ILLPARM: Invalid parameter.
 VSTATUS_TIMEOUT: Requested timeout.

EXAMPLE

```
Status_t          status;
CMapiParamRes_t   param;
CMapiService_t    serv;
void              *iterator = NULL;

serv.serviceID = 0x01ffffffffffffffffULL;
param.servp = &serv;
status = cm_service_query (NULL,
                           CMapiResCmdInitial,
                           1000000ULL /* 1 sec */,
                           &iterator,
                           param
                           );
if (status != VSTATUS_OK) {
    /* failure */
}

while ( /* not found */ ) {
    if (iterator != NULL) {
        status = cm_service_query (NULL,
                                   CMapiResCmdNext,
                                   1000000ULL, /* 1 sec */,
                                   &iterator,
                                   param);
    }

    if (status != VSTATUS_OK) {
        /* failure*/
    }
}
```



Communication Manager (CM) API Reference Manual CM API Function Definitions

```
if (( status == VSTATUS_OK) && (iterator != NULL)) {  
    (void)cm_service_query (NULL,  
        CMapiResCmdEnd,  
        1000000ULL, /* 1 sec */  
        &iterator,  
        param);  
}
```

SEE ALSO “Service Query” on page 15; “Service Data Structure” on page 25.



cm_path_query

DESCRIPTION Find a path. This function must be called until the iterator becomes NULL, or called with the command set to CMapiResCmdEnd. Failure to do so will result in incorrect (stale) service data.

SYNTAX `Status_t cm_path_query (CMapiResTable_t *resp, CMapiResCmd_e cmd, uint64_t timeout, void **iterator, CMapiParamRes_t pathp);`

PARAMETERS **Input:**

`*resp`: Pointer to a service resolution table.
`cmd`: Command.
`timeout`: Number of microseconds to wait.
`**iterator`: Initially passed in as NULL. For subsequent calls, `iterator` is the value returned by the prior call.
`pathp`: Pointer to a path template.

Output:

`**iterator`: Updated to reflect next item to access; if NULL, no more services are available.
`pathp`: Pointer to a current path.

RETURNS `VSTATUS_OK`: Status good.
`VSTATUS_ILLPARM`: Invalid parameter.
`VSTATUS_TIMEOUT`: Requested timeout.

EXAMPLE

```
Status_t          status;
CMapiParamRes_t   param;
CMapiPath_t       path;
void              *iterator = NULL;

memcpy (path.sgid, myGID, 16);
memcpy (path.dgid, serviceGID, 16);
path.numPaths = 1;
param.pathp = &path;
status = cm_path_query (NULL,
                        CMapiResCmdInitial,
                        1000000ULL /* 1 sec */,
                        &iterator,
                        param
                        );

if (status != VSTATUS_OK) {
    /* failure */
}

while ( /* not found */ ) {
    if (iterator != NULL) {
        status = cm_path_query (NULL,
                                CMapiResCmdNext,
                                1000000ULL, /* 1 sec */,
                                &iterator,
                                param);
    }

    if (status != VSTATUS_OK) {
        /* failure*/
    }
}
```



```
if (( status == VSTATUS_OK) && (iterator != NULL)) {  
    (void)cm_path_query (NULL,  
                        CMapiResCmdEnd,  
                        1000000ULL, /* 1 sec */  
                        &iterator,  
                        param);  
}
```

SEE ALSO “Path Query” on page 16; “Path Data Structure” on page 26.



cm_service_reg

DESCRIPTION Register a service on a port on an adapter on this node.

SYNTAX `Status_t cm_service_reg(CMapiResTable_t *resp, CMapiParamRes_t servp, uint64_t timeout);`

PARAMETERS **Input:**

`*resp`: Pointer to a resolution table.
`servp`: Pointer to a service.
`timeout`: Number of microseconds to wait.

Output:

None

RETURNS `VSTATUS_OK`: Status good.
`VSTATUS_ILLPARAM`: Invalid parameter.
`VSTATUS_TIMEOUT`: Requested timeout.

EXAMPLE `#define SERVICE_NAME "MY_SERVICE_NAME"`

```
Status_t          status;
CMapiParamRes_t   param;
CMapiService_t    serv;
CMapiIPCTag_t     tagSID;

serv.serviceID = 0x01ffffffffffffffffFULL;
memcpy (serv.serviceName, SERVICE_NAME, strlen(SERVICE_NAME));
param.servp = &serv;
status = cm_service_reg ( NULL, param,1000000);
if (status != VSTATUS_OK) {
    /* failure */
}
tagSID = serv.tagSID;
```

SEE ALSO "Service Registration" on page 17; "Service Data Structure" on page 25.



cm_service_del

DESCRIPTION Unregister a service on a port on this adapter on this node.

SYNTAX `Status_t cm_service_del(CMapiResTable_t *resp, CMapiParamRes_t servp, uint64_t timeout);`

PARAMETERS `*resp`: Pointer to a resolution table.
`servp`: Pointer to a service.
`timeout`: Number of microseconds to wait.

RETURNS `VSTATUS_OK`: Status good.
`VSTATUS_ILLPARM`: Invalid parameter.
`VSTATUS_TIMEOUT`: Requested timeout.

EXAMPLE

```
Status_t          status;
CMapiParamRes_t  param;
CMapiService_t   serv;
CMapiIPCTag_t    tagSID;

serv.serviceID = 0x01ffffffffffffffffULL;
serv.tagSID    = tagSID;
param.servp = &serv;
status = cm_service_del ( NULL, param, 1000000);
if (status != VSTATUS_OK) {
    /* failure */
}
```

SEE ALSO “Service Delete” on page 18; “Service Data Structure” on page 25.



cm_connect

DESCRIPTION Initiate a connection.

SYNTAX `Status_t cm_connect(CMapiCallBack_t *callBackp, CMapiConnect_t *conp);`

PARAMETERS **Input:**

*callBackp: Callback function.
 *conp: Parameters to establish connection.

Output:

*conp: Parameters of established connection.

RETURNS VSTATUS_OK: Status good.
 VSTATUS_ILLPARM: Invalid parameter.
 VSTATUS_TIMEOUT: Requested timeout.
 VSTATUS_INVALID_HANDL: Invalid handle.
 VSTATUS_UNINIT: Library not initialized.
 VSTATUS_BUSY: Busy.
 VSTATUS_INVALID_PROTO: Invalid protocol (not RC,RD,UC,UD,RAW).

EXAMPLE

```
CMapiConnect_t con;
CMapiPath_t path;

static CMapiCallBackMask_t
callBack (CMapiCallBackMask_t mask, CMapiConnect_t **conp)
{
    CMapiConnect_t *lisConp;
    CMapiCallBackMask_t rmask=CMapiCBmaskNext;

    return (rmask);
}

/* get my GID and service GID */
/* get path between my GID and service GID */

con.primPath = path;
con.autoLAPpath.numPaths = 0;

status = cm_connect (&callBack, &con);
if (status != VSTATUS_OK) {
    /* failure */
}
```

SEE ALSO "Connect" on page 19; "Connect Data Structure" on page 28.



cm_listen

DESCRIPTION Wait for a connection request.

SYNTAX `Status_t cm_listen(CMapiIPCTag_t tagSID, CMapiCallBack_t *callBackp, CMapiConnect_t *conp, CMapiConnect_t **newConp);`

PARAMETERS **Input:**

tagSID: SID IPC tag used to access CM.
*callBackp: Callback function.
*conp: Parameters to establish connection.

Output:

**newConp: Parameters of established connection.

RETURNS VSTATUS_OK: Status good.
VSTATUS_ILLPARM: Invalid parameter.
VSTATUS_TIMEOUT: Requested timeout.
VSTATUS_INVALID_HANDL: Invalid handle.
VSTATUS_UNINIT: Library not initialized.
VSTATUS_BUSY: Busy.

```
EXAMPLE #define SERVICE_NAME          "MY_SERVICE_NAME"

Status_t          status;
CMapiConnect_t   con;
CMapiConnect_t   *newConp;
CMapiParamRes_t  param;
CMapiIPCTag_t    tagSID;

static CMapiCallBackMask_t
callBack (CMapiCallBackMask_t mask, CMapiConnect_t **conp)
{
    CMapiConnect_t      *lisConp;
    CMapiCallBackMask_t  rmask=CMapiCBmaskNext;

    switch (mask) {
        case CMapiCBmaskREQ: /* Allocate memory for lisCon */
            *conp = lisConp;
            break;
    }
    return (rmask);
}

/* register service */

status = cm_listen (tagSID, &callBack, &con, &newConp);
if (status != VSTATUS_OK) {
    /* failure */
}
```

SEE ALSO "Listen" on page 22.



cm_disconnect

DESCRIPTION Terminate a connection.

SYNTAX `Status_t cm_disconnect(CMapiCallBack_t *callBackp, CMapiConnect_t *conp);`

PARAMETERS **Input:**

`*callBackp`: Callback function.

`*conp`: Specifies the parameters to terminate connect.

Output:

`*conp`: Results of disconnect.

RETURNS `VSTATUS_OK`: Status good.

`VSTATUS_ILLPARM`: Invalid parameter.

`VSTATUS_TIMEOUT`: Requested timeout.

`VSTATUS_INVALID_HANDL`: Invalid handle.

`VSTATUS_UNINIT`: Library not initialized.

`VSTATUS_CONNECT_GONE`: Connection has been torn down.

EXAMPLE `CMapiConnect_t con;
CMapiPath_t path;`

```
static CMapiCallBackMask_t  
callBack (CMapiCallBackMask_t mask, CMapiConnect_t **conp)  
{  
    CMapiCallBackMask_t rmask=CMapiCBmaskNext;  
  
    return (rmask);  
}  
  
status = cm_disconnect (&callBack, &con);  
if (status != VSTATUS_OK) {  
    /* failure */  
}
```

SEE ALSO "Disconnect" on page 22.



cm_load_ap

DESCRIPTION Load an alternate path.

SYNTAX `Status_t cm_load_ap(CMapiCallBack_t *callBackp, CMapiConnect_t *conp);`

PARAMETERS **Input:**

`*callBackp`: Callback function.
`*conp`: Specifies the parameters to load.

Output:

`*conp`: Results.

RETURNS `VSTATUS_OK`: Status good.
`VSTATUS_ILLPARM`: Invalid parameter.
`VSTATUS_INVALID_HANDL`: Invalid handle.
`VSTATUS_UNINIT`: Library not initialized.
`VSTATUS_CONNECT_GONE`: Connection has been torn down.

EXAMPLE

```
CMapiConnect_t con;
CMapiPath_t     newPath;

static CMapiCallBackMask_t
callBack (CMapiCallBackMask_t mask, CMapiConnect_t **conp)
{
    CMapiCallBackMask_t rmask=CMapiCBmaskNext;

    return (rmask);
}

/* get new path between my GID and service GID */

con.altPath = newPath;

status = cm_load_ap (&callBack, &con);
if (status != VSTATUS_OK) {
    /* failure */
}
```



cm_api_init

DESCRIPTION Initialize CM API library.

SYNTAX `Status_t cm_api_init(uint32_t poolSize);`

PARAMETERS **Input:**

poolSize: Number of bytes to allocate for API memory pool.

Output:

None

RETURNS VSTATUS_OK: Status good.

VSTATUS_ILLPARM: Invalid parameter.

EXAMPLE

```
status = cm_api_init (1024*1024);
if (status != VSTATUS_OK) {
    /* failure */
}
```



[This page intentionally left blank.]



Appendix: Additional CM API Structures and Definitions

General Definitions

Transport Protocol Values¹

```
#define CMversion 1

typedef void *CMapiIPCTag_t;

typedef enum {

    CMapiAttrProtoRC=0, // Reliable connection
    CMapiAttrProtoUC=1, // Unreliable connection
    CMapiAttrProtoRD=2, // Reliable datagram
    CMapiAttrProtoUD=3, // Unreliable datagram
    CMapiAttrProtoLast // Always the last item in enum

} CMapiProto_e;
```

Connection State Values²

```
typedef enum {

// Active states
    CMapiStateIdle           = 1,
    CMapiStateREQsent       = 2,
    CMapiStatePeerCompare   = 3,
    CMapiStateREPwait       = 4,
    CMapiStateREPrvd        = 5,
    CMapiStateMRAsentAct    = 6,
    CMapiStateEstablishedAct= 7,
    CMapiStateDREQsentAct   = 8,
    CMapiStateDREQrcvdAct   = 9,
    CMapiStateTimeWaitAct   = 10,
    CMapiStateTimeout       = 11,
    CMapiStateDREPToAct     = 12,

// Passive states
    CMapiStateListen        = 13,
    CMapiStateREQrcvd       = 14,
    CMapiStateMRAsentPas    = 15,
    CMapiStateREPsent       = 16,
    CMapiStateMRArcvd       = 17,
    CMapiStateEstablishedPas= 18,
    CMapiStateDREQrcvdPas   = 19,
    CMapiStateDREQsentPas   = 20,
    CMapiStateRTUto         = 21,
    CMapiStateTimeWaitPas   = 22,
    CMapiStateDREPToPas     = 23,

    CMapiStatelast

} CMapiState_e;
```

-
1. Assigned transport protocol values are as defined in section 12.7.6 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
 2. Connection state values are arbitrary; for more information, see section 12.9.95 - 12.9.7 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."



Private Data Maximum Size Values¹

```
#define CM_MAX_PRIVATE_DATA_SIZE      256
#define CM_MAX_DREP_PRIVATE_SIZE      (1792 >> 3)
#define CM_MAX_DREQ_PRIVATE_SIZE      (1760 >> 3)
#define CM_MAX_MRA_PRIVATE_SIZE       (1776 >> 3)
#define CM_MAX_REJ_PRIVATE_SIZE       (1184 >> 3)
#define CM_MAX_REP_PRIVATE_SIZE       (1632 >> 3)
#define CM_MAX_REQ_PRIVATE_SIZE       ( 736 >> 3)
#define CM_MAX_RTU_PRIVATE_SIZE       (1792 >> 3)
#define CM_MAX_LAP_PRIVATE_SIZE       (1344 >> 3)
#define CM_MAX_APR_PRIVATE_SIZE       (1208 >> 3)
#define CM_MAX_REJ_ARI                 ( 576 >> 3)
```

Rejection Codes²

```
typedef enum {
    rejNoQPavail          = 1,
    rejNoEECavai         = 2,
    rejNoResources        = 3,
    rejTimeout            = 4,
    rejUnsupported        = 5,
    rejCommIdInvalid      = 6,
    rejCommInstanceInvalid = 7,
    rejServiceIDInvalid   = 8,
    rejTransServTypeInvalid = 9,
    rejStaleConnection    = 10,
    rejNoSuchRDC          = 11,
    rejPrimRmtGIDrej      = 12,
    rejPrimRmtLIDrej      = 13,
    rejPrimSlInvalid      = 14,
    rejPrimTclassInvalid  = 15,
    rejPrimHopLimitInvalid = 16,
    rejPrimPktRateInvalid = 17,
    rejAltRmtGIDrej       = 18,
    rejAltRmtLIDrej       = 19,
    rejAltSlInvalid       = 20,
    rejAltTclassInvalid   = 21,
    rejAltHopLimitInvalid = 22,
    rejAltPktRateInvalid  = 23,
    rejCMredirect         = 24,
    rejPortRedirect       = 25,
    rejInvalidMTU         = 26,
    rejShortage           = 27,
    rejConsumer           = 28,
    rejRNRretryCount      = 29,
    rejLast
} CMapiRejCodes_e;
```

-
1. Private data maximum size values depends on CM MAD attributes; for more information, see sections 12.6.5 - 12.6.8 and 12.8.1 - 12.8.2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
 2. For more information, see section 12.6.7.2 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."



Alternate Path Status Codes¹

```
typedef enum {  
  
    apInfoLoaded           = 0,  
    apInvalidCommID       = 1,  
    apNotSupported         = 2,  
    apOther                = 3,  
    apRejectedRedirect     = 4,  
    apSameAsPrimary       = 5,  
    aplast  
  
} CMapiAPcodes_e;
```

Service ID Resolution (SIDR) Codes²

```
typedef enum {  
  
    sidrOK                 = 0,  
    sidrNoServiceID        = 1,  
    sidrServReject         = 2,  
    sidrNoQP               = 3,  
    sidrRedir              = 4,  
    sidrLast  
  
} CMapiSIDcodes_e;
```

-
1. For more information, see section 12.8.2.1 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."
 2. For more information, see section 12.11.2.1 of the "InfiniBand Architecture Specification Volume 1, Release 1.0.a; June 19, 2001."



Callback Definitions

The primary purpose of this callback function is to allow the client or server code to look at connection parameters, private data, change QP state, and add receive buffers prior to connection completion.

The callback function is invoked for each protocol message. For example, when `cm_listen` is called, the callback function will be invoked when the REQ MAD is received. It will be called with the mask set to `CMapiCBmaskREQ` and any private data will be accessible in `comp->privateData`. This is the normal processing path for a server. The server can elect to respond with MRA or reject if the connection request is erroneous, or if it cannot allocate the resources that it needs. In order for the CM finite state machine (FSM) to take the appropriate action, the callback function needs to inform the FSM of the appropriate action to take. This is the purpose of the value returned by the callback function. If the `callBack` function returns zero, the FSM proceeds to the next normal state.

Callback Interface Definitions

`CMapiCallBackMask_t` is a bit map of packet types. When the callback function is executed, the mask indicates the kind of MAD that generated the callback

```
typedef CMapiCallBackMask_t (CMapiCallBack_t)
    (CMapiCallBackMask_t mask, CMapiConnect_t **comp);

typedef enum {

// The following are inputs and outputs

    CMapiCBmaskREQ      = 0x00001,
    CMapiCBmaskREP      = 0x00002,
    CMapiCBmaskRTU      = 0x00004,
    CMapiCBmaskMRA      = 0x00008,
    CMapiCBmaskREJ      = 0x00010,
    CMapiCBmaskDREQ     = 0x00020,
    CMapiCBmaskDREP     = 0x00040,
    CMapiCBmaskLAP      = 0x00080,
    CMapiCBmaskAPR      = 0x00100,
    CMapiCBmaskSIDREQ   = 0x00200,
    CMapiCBmaskSIDREP   = 0x00400,
    CMapiCBmaskMRAsent  = 0x00800,
    CMapiCBmaskIPCgone  = 0x01000,

// The following are outputs only

    CMapiCBmaskNext     = 0

} CMapiCallBackMask_t;

#define CMapiCBmaskAny (~0)
```



Resolution Definitions

```
typedef enum {  
  
    CMapiResClassCM      = 0,      // Communication Manager  
    CMapiResClassSA      = 1,      // Subnet Administrator  
    CMapiResClassLDAP    = 2,      // Lightweight Directory  
    CMapiResClassMAPI    = 3,      // Active Directory  
    CMapiResClassLast    = 0x0000, // enum terminator  
  
} CMapiResClass_e;
```

Resolution Operation Class Bit Maps

```
typedef enum {  
  
    CMapiResOpsNone      = 0,  
    CMapiResOpsReg        = 1,      // Registration  
    CMapiResOpsQuery      = 2,      // Query  
    CMapiResOpsService    = 4,      // Service entity  
    CMapiResOpsName       = 8,      // Name entity  
    CMapiResOpsPath       = 16,     // Path entity  
  
} CMapiResOp_e;
```

Command Definition

```
typedef enum {  
  
    CMapiResCmdInitial    = 0,      // Start search  
    CMapiResCmdNext       = 1,      // Get next record  
    CMapiResCmdEnd        = 2,      // Terminate search  
    CMapiResCmdLast  
  
} CMapiResCmd_e;
```



Resolution Function

DESCRIPTION The resolution function performs the actual lookup.

```
SYNTAX typedef union _CMapiParamRes_t
{

    CMapiPath_t      *pathp;
    CMapiService_t  *servp;

} CMapiParamRes_t;

typedef Status_t (CMapiResFunc_t) (CMapiResOp_e op, CMapiResCmd_e
cmd, CMapiPath_t *pathp, void **iterator, CMapiParamRes_t
entryp);
```

PARAMETERS **Input:**

op: Operation(s) to perform.

cmd: Command.

*pathp: Path to resolver.

**iterator: Initially passed in as NULL; subsequent calls, the value returned by prior call.

Output:

**iterator: If equal to NULL, nothing more.

entryp: Service/path entry found.

RETURNS STATUS_OK
STATUS_ILLPARM
STATUS_TIMEOUT

CM and SA Resolver Function Prototypes

```
Status_t cm_resolve_cm (CMapiResOp_e,
                        CMapiResCmd_e,
                        CMapiPath_t *,
                        void **,
                        CMapiParamRes_t,
                        uint64_t);
```

```
Status_t cm_resolve_sa (CMapiResOp_e,
                        CMapiResCmd_e,
                        CMapiPath_t *,
                        void **,
                        CMapiParamRes_t,
                        uint64_t);
```



Resolution Data Structures

The resolution table allows an application to set a list of functions that will evaluate service registration/unregistration and service/path queries. When the register/unregister/query API function is called, it is passed a pointer to a resolution table. Each entry in the table specifies the information needed to allow the specified function to evaluate the request.

The table is evaluated starting at the first entry and ending at the last entry, or when the scope indicates complete. The evaluation may be fairly simple (for example, register with the local CM) or it may be complex (for example, a LAPD request to a remote LAPD server).

Resolution Table Entry

```
typedef struct _CMapiResEntry_t {
    CMapiResFunc_t *funcp;    // Evaluator function
    CMapiPath_t pathp;       // Path to resolution server
    CMapiResClass_e resClass; // Class of resolution server
} CMapiResEntry_t;
```

Resolution Table

```
typedef struct _CMapiResTable_t {
    uint16_t numEntries;    // Number of entries in table
    CMapiResEntry_t *table; // Array of n CMapiResEntry_t
                           // entries; where n = numEntries
} CMapiResTable_t;
```



[This page intentionally left blank.]



INDEX

C

cm_api_init, 41
cm_connect, 37
cm_disconnect, 39
cm_listen, 38
cm_path_query, 33
cm_service_del, 36
cm_service_query, 31
cm_service_reg, 35
CMapiAPcodes_e, 45
CMapiCallBackMask_t, 46
CMapiConnect_t, 30
CMapiMagic_t, 25
CMapiParamRes_t, 48
CMapiPath_t, 27
CMapiProto_e, 43
CMapiRejCodes_e, 44
CMapiResClass_e, 47
CMapiResCmd_e, 47
CMapiResEntry_t, 49
CMapiResFunc_t, 48
CMapiResOp_e, 47
CMapiResTable_t, 49
CMapiService_t, 26
CMapiSIDcodes_e, 45
CMapiState_e, 43



[This page intentionally left blank.]