Lane15 Channel Interface Specification

January 2002

LANE 15
SOFTWARE

DISCLAIMERS

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

No license, expressed or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Lane15 Software disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. Lane15 Software does not warrant or represent that such use will not infringe such rights.

Nothing in this document constitutes a guarantee, warranty, or license, express or implied. Lane15 Software disclaims all liability for all such guaranties, warranties, and licenses, including but not limited to: fitness for a particular purpose; merchantability; non-infringement of intellectual property or other rights of any third party or of Lane15 Software; indemnity; and all others. The reader is advised that third parties may have intellectual property rights which may be relevant to this document and the technologies discussed herein, and is advised to seek the advice of competent legal counsel, without obligation to Lane15 Software.

Lane15 Software retains the right to make changes to this document at any time, without notice. Lane15 Software makes no warranty for the use of this document and assumes no responsibility for any errors, which may appear in the document, nor does it make a commitment to update the information contained herein.

Copyright © 2002 Lane15 Software. All Rights Reserved.
Protected by U.S. Patents Pending. No part may be reproduced or transmitted, in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Lane15 Software, Inc.

This document created or updated on January 22, 2002.

Lane15, Lane15 Fabric Manager, and the Lane15 logo are trademarks of Lane15 Software Inc. in the United States, other countries, or both.

InfiniBand is a service mark of the InfiniBand Trade Association.

Other company, product and service names may be trademarks or service marks of others.

Lane15 Software
8501 North Mopac Expressway
Suite 300
Austin, Texas 78759
512•233•1515
www.lane15.com

Table of Contents

1 INTRODUCTION 6

1.1 DIRECT ACCESS TO INFINIBAND HARDWARE 6
1.2 PURPOSE OF THE CI 7
1.3 THREADING MODEL 7

2 USING CHANNEL SERVICES 9

2.1 QUALITY OF SERVICE 9
2.1.1 Path selection and Path migration 9
2.2 CHANNEL RESOURCES 9
2.2.1 Direct access 9
2.2.2 Connection End Point Transitions 10
2.3 EVENT MODEL 11
2.3.1 Service connection events 12
2.3.1.1 Reliable connections 12
2.3.1.2 Unreliable Datagram services 13
2.3.2 Channel connection events 13
2.3.3 Completion events 13
2.3.4 Asynchronous errors and events 13
2.3.4.1 IChannelCa errors 14
2.3.4.2 IEeContext errors 14
2.3.4.3 IChannel errors and events 14
2.3.4.4 ICompletionQueue errors 15
2.4 CONNECTING TO A SERVICE 15
2.4.1 Peer Connections 15

3 INTERFACES 16

3.1 CHANNEL MANAGER CLASS 17
3.1.1 CChannelManager::QueryCaCount 17
3.1.2 CChannelManager::QueryCaGuids 17
3.1.3 CChannelManager::OpenChannelCa 18
3.2 CHANNEL ADAPTER INTERFACE 19
3.2.1 IChannelCa::RegisterService 19
3.2.2 IChannelCa::DeregisterService 20
3.2.3 IChannelCa::FindUdService 20
3.2.4 IChannelCa::CreateCq 21
3.2.5 IChannelCa::CreatePd 21
3.2.6 IChannelCa::CreateRdd 22
3.2.7 IChannelCa::Query 22
3.2.8 IChannelCa::QueryPort 22

3.2.9 IChannelCa::QueryPortTable 23
3.2.10 IChannelCa::Destroy 23
3.3 ASYNCHRONOUS EVENT INTERFACE 24
3.3.1 IAsyncEvent::AsyncEvent 24
3.4 UD SERVICE RESPONSE INTERFACE 24
3.4.1 IUdServiceResponse::ServiceReply 24
3.5 CHANNEL SERVICES EVENT INTERFACE 25
3.5.1 IChannelServiceEvent::ConnectRequest 25
3.5.2 IChannelServiceEvent::ConnectReject 25
3.5.3 IChannelServiceEvent::UdServiceRequest 26
3.6 CHANNEL SERVICE ACTION INTERFACE 26
3.6.1 IChannelServiceAction::Accept 26
3.6.2 IChannelServiceAction::Reject 27
3.6.3 IChannelServiceAction::SendMra 27
3.7 CHANNEL RELIABLE DATAGRAM DOMAIN INTERFACE 28
3.7.1 IChannelRdd::CreateEeContext 28
3.7.2 IChannelRdd::Deallocate 28
3.8 END-TO-END CONTEXT INTERFACE 28
3.8.1 IEeContext::Modify 28
3.8.2 IEeContext::Query 29
3.8.3 IEeContext::Destroy 29
3.9 COMPLETION QUEUE INTERFACE 30
3.9.1 ICompletionQueue::Resize 30
3.9.2 ICompletionQueue::Query 30
3.9.3 ICompletionQueue::Destroy 31
3.9.4 ICompletionQueue::PollForCompletion 31
3.9.5 ICompletionQueue::RequestCompletionNotification 31
3.10 COMPLETION EVENT INTERFACE 32
3.10.1 ICompletionEvent::CompletionEvent 32
3.11 CHANNEL PROTECTION DOMAIN INTERFACE 33
3.11.1 IChannelPd::CreateChannel 33
3.11.2 IChannelPd::RegisterMemoryRegion 33
3.11.3 IChannelPd::AllocateMemoryWindow 34
3.11.4 IChannelPd::CreateAddress 34
3.11.5 IChannelPd::Destroy 35
3.12 ADDRESS INTERFACE 35
3.12.1 IAddress::Modify 35
3.12.2 IAddress::Query 35
3.12.3 IAddress::Destroy 36
3.13 MEMORY REGION INTERFACE 36
3.13.1 IMemoryRegion::Query 36
3.13.2 IMemoryRegion::Reregister 36
3.13.3 IMemoryRegion::ReregisterPhysical 37
3.13.4 IMemoryRegion::RegisterShared 38
3.13.5 IMemoryRegion::Deregister 38
3.14 MEMORY WINDOW INTERFACE 39

3.14.1 IMemoryWindow::Query ... 39
3.14.2 IMemoryWindow::Deallocate ... 39
3.15 CHANNEL INTERFACE ... 39
3.15.1 IChannel::Connect ... 39
3.15.2 IChannel::Disconnect ... 40
3.15.3 IChannel::PostSend ... 40
3.15.4 IChannel::PostRecv ... 41
3.15.5 IChannel::Reset ... 41
3.15.6 IChannel::Query ... 41
3.15.7 IChannel::Modify ... 42
3.15.8 IChannel::Destroy ... 42
3.16 CHANNEL EVENT INTERFACE ... 43
3.16.1 IChannelEvent::AsyncEvent ... 43
3.16.2 IChannelEvent::ConnectionEvent ... 43
3.16.3 IChannelEvent::ConnectError ... 45
3.17 CHANNEL ACTION INTERFACE ... 45
3.17.1 IChannelAction::Accept ... 45
3.17.2 IChannelAction::Reject ... 46
3.17.3 IChannelAction::SendMra ... 46

4 ATTRIBUTES ... 47

4.1 CSCONNECTREQUESTSTRUCT ... 47
4.2 CSCONNECTREPLYSTRUCT ... 49
4.3 CSRTUSTRUCT ... 50
4.4 SERVICEATTRSSTRUCT ... 50
4.5 EVENTRECORDSTRUCT ... 51
4.6 CHANNELATTRSSTRUCT ... 52
4.7 CSDISCONNECTREQUESTSTRUCT ... 53
4.8 CSDISCONNECTREPLYSTRUCT ... 53
4.9 CSCONNECTREJECTSTRUCT ... 53
4.10 CSMRASTRUCT ... 56

1 Introduction

Channel Services exposes a consumer level API for use by upper level protocols. This interface exposes the full capabilities of InfiniBand (IB) and provides the highest performance possible. Privileged level operations as well as IB management and policy issues are hidden from the Channel Services Interface (CI) consumer.

This document is intended to be a guide for developers of upper level protocols such as DAT or MPI. The upper level protocol implementation is a consumer of the CI.

In general, most application developers should not code directly to the CI, but instead to one of the upper level protocols such as DAT or MPI.

1.1 Direct access to InfiniBand hardware

As shown in Figure 1, Channel Services sits between all upper level protocols and interfaces directly with IB hardware (verbs) and connection management facilities. This gives CI consumers direct access to IB hardware. In this role, CI does not introduce any performance penalties. For example, Data Transfer Operations (DTO) for send, receive, and completion notification are passed directly to the hardware level verbs interface. For these operations, CI supplies the consumer with the correct interface, then “gets out of the way”.

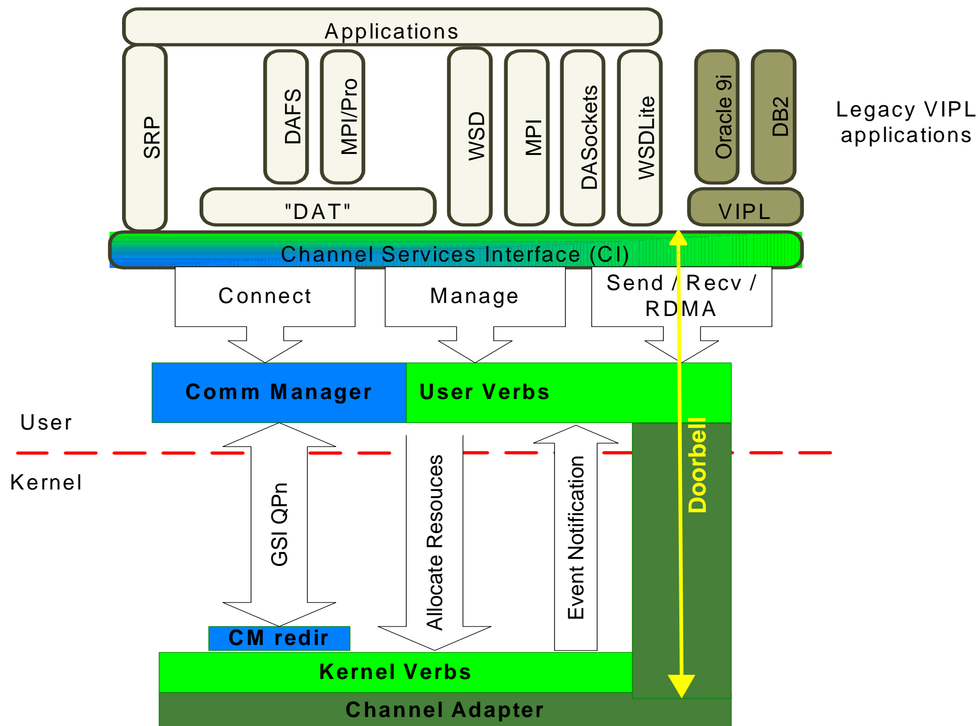


Figure 1 Channel Services Architecture Overview

1.2 Purpose of the CI

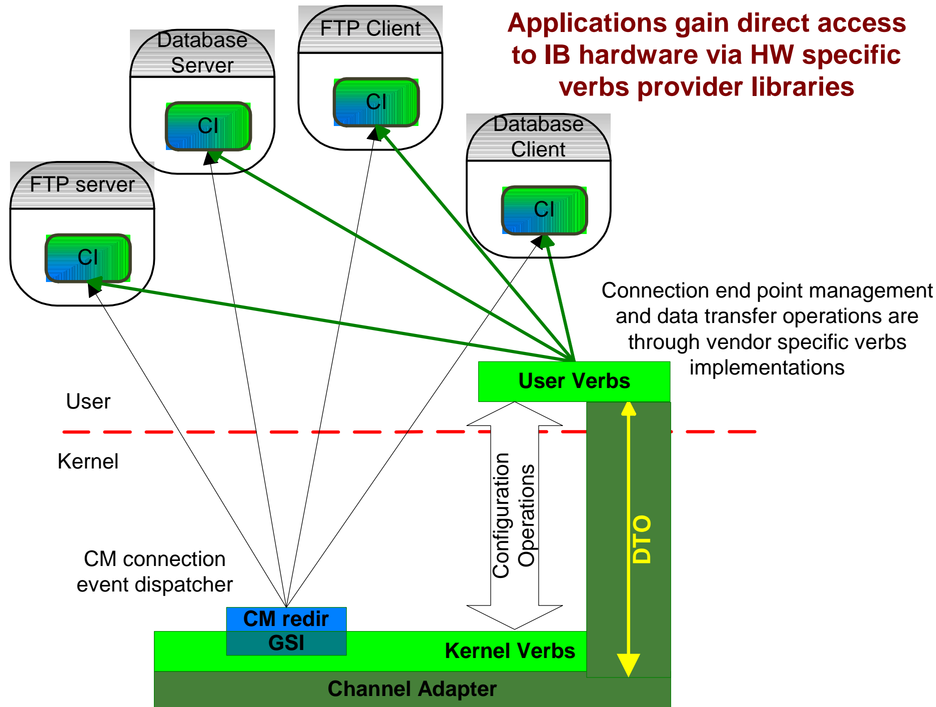
The channel interface exists for the following reasons:

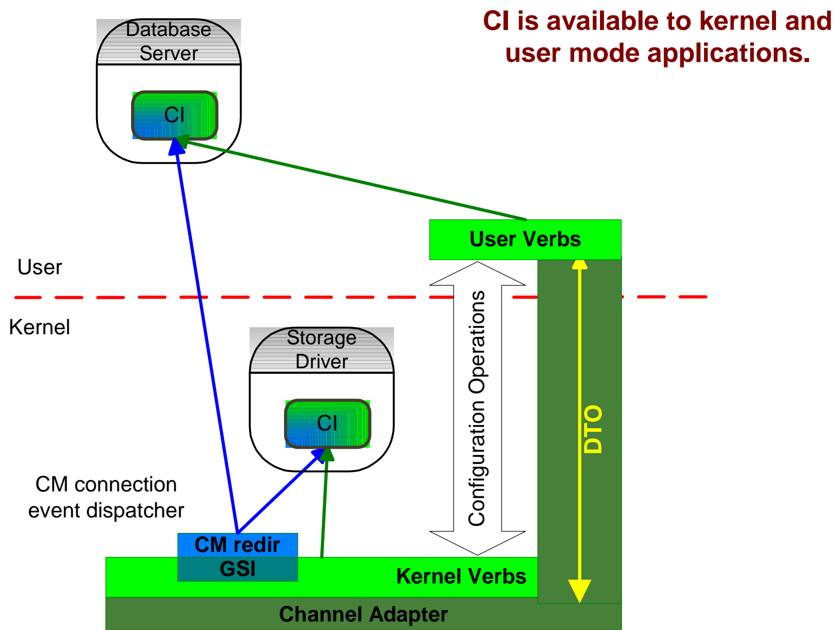
- Hide privileged-level operations from non-privileged consumers.
- Provide a meaningful intersection of Verbs, Service Registration, and Communication Management (CM) functions.
- Provide a single interface for all higher-level protocol channel drivers.
- Provide a single interface to expose from kernel to user space.
- Provide a single point to address threading model and kernel boundary issues.
- Provide a single point for Quality of Service (QoS), monitoring, and control.
- Hide the complexities of interactions between CM, Connection End Point (CEP), and verbs from the consumer. This keeps developers from ‘rediscovering’ these complexities.
- Provide automatic path migration across channel adapters, even if the channel adapters are from different vendors.

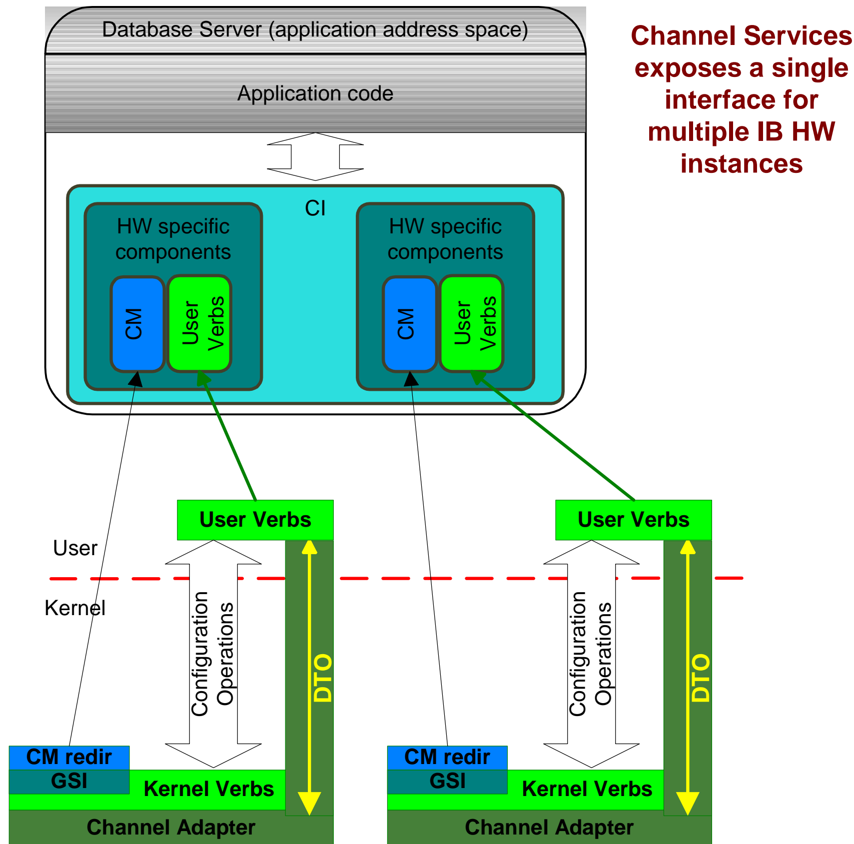
1.3 Threading model

The CI inherits its threading model from the CI consumer, the Verbs implementation, and the CM implementation. Typically, CM inherits its threading model from the GSI (and subsequently the verbs) implementation. The CI itself does not implement a thread pool.

Data transfer operations (DTO) bypass the channel services component. The threading model for DTO completions is inherited directly from the IB verbs implementation. For kernel mode this is typically a Deferred Procedure Call (DPC). In user mode, this is typically a thread executing in the address space of the application. The user mode verbs provider assigns the priority of that thread.


Applications gain direct access to IB hardware via HW specific verbs provider libraries
Database Server
CI
FTP Client
CI
Database Client
CI
FTP server
CI
Connection end point management and data transfer operations are through vendor specific verbs implementations
User Verbs
User
Kernel
Configuration Operations
DTO
CM connection event dispatcher
CM redir
GSI
Kernel Verbs
Channel Adapter


CI is available to kernel and user mode applications.
Database Server
CI
User Verbs
User
Kernel
Storage Driver
CI
Configuration Operations
DTO
CM connection event dispatcher
CM redir
GSI
Kernel Verbs
Channel Adapter



For additional details, see the section on the channel services Event Model.

2 Using Channel Services

2.1 Quality of Service

2.1.1 Path selection and Path migration

Path selection and path migration are handled transparently to the Channel Services consumer.

2.2 Channel resources

2.2.1 Direct access

In certain cases, channel resources allocated from an underlying component are passed directly back to the CI consumer. This eliminates performance penalties that might arise by calling through the CI to the underlying component. For example, IB completion operations are directed from the IB verbs implementation directly to the consumer without any intervention on the part of CI.

2.2.2 Connection End Point Transitions

Channel services responsibilities include management of IB queue pairs and EE contexts (QP/EE) resources. This management is transparent to the CI consumer, and does not require any direct interactions by the consumer. However, details on the process may be of interest to CI consumers that want to understand what effect associated method calls and parameter settings have on QP management.

To accomplish QP/EE management, the channel class maintains a Connection End Point, (CEP) object. During connection establishment, the CEP undergoes a sequence of state transitions. These transitions are implemented as modify QP operations. The data for these operations comes from a variety of sources. The following tables provide details on the origin of this data.

Table 1- Create CEP

Field	Source	Notes
QpType	App	See Lane15 Verbs Reference Guide.
SignalingType	App	See Lane15 Verbs Reference Guide.
MaxSndFragments	App	See Lane15 Verbs Reference Guide.
MaxRcvFragments	App	See Lane15 Verbs Reference Guide.
MaxSndRequests	App	See Lane15 Verbs Reference Guide.
MaxRcvRequests	App	See Lane15 Verbs Reference Guide.
SendCq	App	See Lane15 Verbs Reference Guide.
RecvCq	App	See Lane15 Verbs Reference Guide.
Rdd	App	See Lane15 Verbs Reference Guide.

Table 2- Reset to Init

Field	Source	Notes
PrimaryPkeyIndex	CI	
Qkey	CI	
PhysicalPort	App	See Lane15 Verbs Reference Guide.
AccessControl	App	See Lane15 Verbs Reference Guide.

Table 3- Init to Ready-To-Receive

Field	Source	Notes
RemoteNodeAddr	CI	
DestQPn	CI	
RcvQpPsn	CI	CI tracks packet sequence numbers to ensure that new CEP communications are not interrupted by expired CEP residual packet traffic (packets left floating around the subnet).
ResponderResources	CM	The CM pulls this from the ConnectRequest or ConnectReply packet it receives from the remote node.

Table 4- Ready-To-Receive to Ready-To-Send

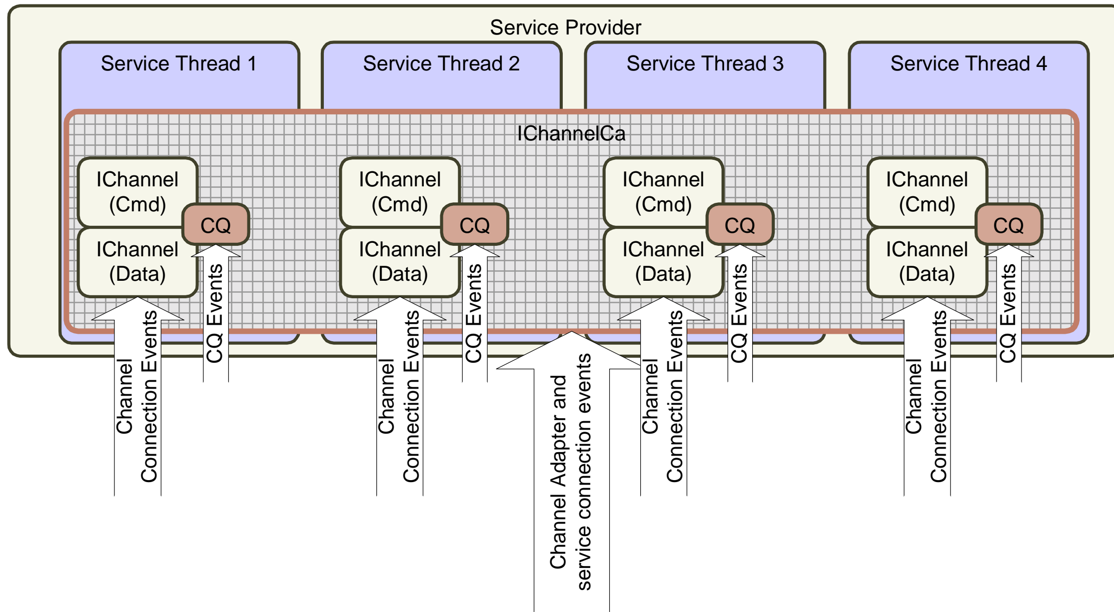
Field	Source	Notes
RetryCount	QoS	
Timeout	QoS	
RnrRetryCount	QoS	
SendQpPsn	CM	
OpLimit	App	See Lane15 Verbs Reference Guide.

2.3 Event Model

CI serves as a focal point for IB resource management, service registration, communications management and data transport. Events can originate from any of these components, and are managed and dispatched through channel services. Applications must implement event handlers to deal with these events. In order to simplify the process of implementing these event handlers and to reduce confusion over the responsibilities of each event handler, channel services provides separate interfaces for four categories of events:

- Service connection events– Connection events related to service registration.
- Channel connection events – Connection events related to specific channels.
- Completion events – events related to data transport functions.
- Asynchronous errors and events – Events related to IB resources such as channel adapters, queue pairs, completion queues, and protection domains.

Applications that take full advantage of the CI event notification infrastructure can be written in either C or C++. Regardless of the language chosen, applications should be written using a component-oriented design. The following diagram shows one example of how software components can be organized to handle various events.



Note: Multiple events can be dispatched concurrently.

2.3.1 Service connection events

Service providers must supply an `IChannelServiceEvent` interface when they register a service via the following:

`IChannelCa::RegisterService`

2.3.1.1 Reliable connections

Service connection events are used to notify a service provider, such as an FTP server, that a client is requesting a connection to the service. For reliable connections, the service provider must implement the following methods:

`IChannelServiceEvent::ConnectRequest`
`IChannelServiceEvent::ConnectReject`

Note that this interface is only used by service providers, and only to process initial connection requests. Once the connection request is accepted, all further connection events are dispatched through the `IChannelEvent` interface described in the next section.

Service providers must respond to `ConnectRequest()` events by calling a service action method. These methods are part of the `IChannelServiceAction` interface:

`IChannelServiceAction::Accept`
`IChannelServiceAction::Reject`
`IChannelServiceAction::SendMra`

LANE 15 SOFTWARE

2.3.1.2 Unreliable Datagram services

Unreliable Datagram (UD) service ID resolution requests are forwarded to consumers via the `UdServiceRequest` callback:

`IChannelServiceEvent::UdServiceRequest`

A client application initiates a query to acquire the remote channel address for the specified UD by calling `IChannelCa::FindUdService`. The results of the query are returned through the UD service response callback:

`IUdServiceResponse::ServiceReply`

2.3.2 Channel connection events

Channel connection events are used to notify `IChannel` consumers (clients, servers, or peers) of connection establishment events such as replies, rejects, or message receipt acknowledgements. This allows the consumer to monitor the connection establishment process, or to exchange private data with the remote connection endpoint.

`IChannelEvent::ConnectionEvent`

Applications must respond to `ConnectionEvent()` with one of the following `IChannelAction` calls:

`IChannelAction::Accept`
`IChannelAction::Reject`
`IChannelAction::SendMra`

2.3.3 Completion events

Completion events are used to notify `IChannel` consumers of data transport events such as send or receive completions.

Refer to the section on the Completion Event Interface for additional details.

2.3.4 Asynchronous errors and events

Asynchronous events are used to notify consumers of errors or events associated with IB channel adapter resources.

Refer to the section on the Asynchronous Event Interface for additional details.

2.3.4.1 IChannelCa errors

All unaffiliated asynchronous errors are sent to the event handler registered with the `CChannelManager::OpenChannelCa` call. All `IChannelCa` instances receive notification of these events.

Catastrophic Error: Generated when an error occurs that cannot be associated with a specific resource and the channel adapter's behavior is unpredictable.

Port Error: Generated when a channel adapter port becomes unavailable.

2.3.4.2 IEeContext errors

Asynchronous errors affiliated with EE contexts are propagated to the event handler registered when the EE context was created (see `IChannelRdd::CreateEeContext`).

Errors:

Catastrophic Error: Generated when an error occurs while accessing or processing the EE Context that prevents reporting of completions.

Path Migration Request Error: Generated when an incoming path migration request to this EE was not accepted.

2.3.4.3 IChannel errors and events

Asynchronous errors affiliated with queue pairs (channels) are propagated to the event handler registered when the channel was created (see `IChannelPd::CreateChannel`).

Events:

Path Migrated: Generated when the connection has migrated to the alternate path.

Communications Established: Generated when the first packet arrives for the Receive Work Queue where the QP/EE is still in the Ready-To-Receive state.

Send Queue Drained: Generated when the Send Queue of the specified Queue Pair has completed the outstanding messages in progress when the state change was requested and, if applicable, has received all acknowledgements for those messages.

Errors:

Path Migration Error: Generated when an incoming path migration request was not accepted.

2.3.4.4 ICompletionQueue errors

Asynchronous errors affiliated with completion queues are propagated to the event handler registered when the CQ was created (see `IChannelCa::CreateCq`).

Errors:

Write Error: Generated when an error occurs when writing an entry to the Completion Queue.

Work Queue Catastrophic Error: Generated when an error occurs while accessing or processing a Work Queue that prevents the reporting of completions.

2.4 Connecting to a service

2.4.1 Peer Connections

Peer connections are initiated when two processes on different nodes actively initiate connection establishment to each other. To do this, both processes create an `IChannel` instance, and initiate connection establishment by calling the `IChannel::Connect` method with the peer flag set to true.

For peer connections, one of the consumers assumes the active role; the other consumer assumes the passive role. The CM performs the role negotiation without intervention by either consumer. The consumers are indirectly notified of their respective roles when they receive connection event notification through their `IChannelEvent` interface. The consumer assuming the active role receives `IChannelEvent::ConnectionEvent` `ConnectRequest` notification, while the consumer assuming the passive role receives `IChannelEvent::ConnectionEvent` `ConnectReply` notification.

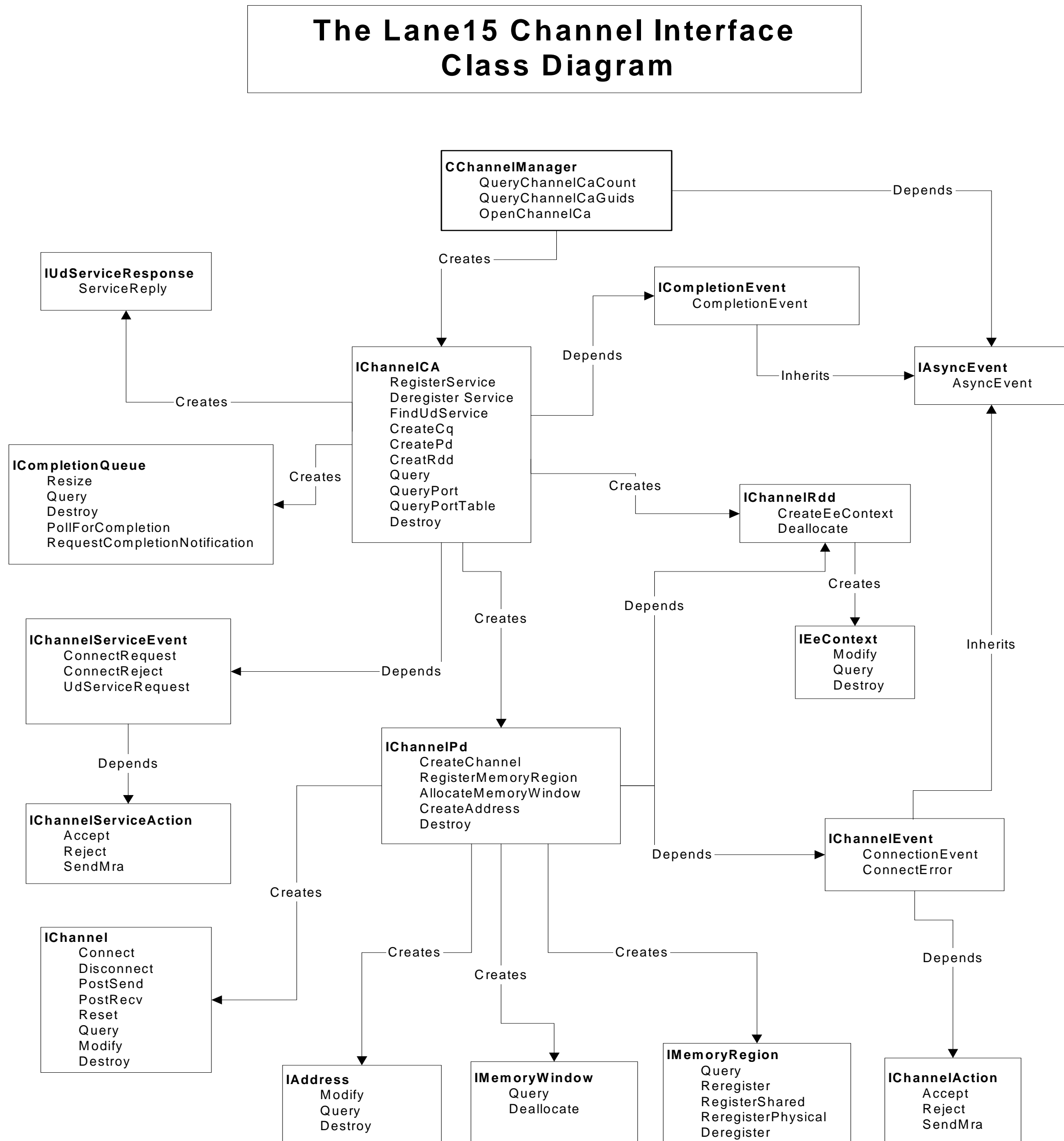
The combination of Service ID, local port GID and remote port GID are used by the CM to associate pairs of peer connection end points.

The Service ID used by the consumer implicitly defines whether the service is client/server or peer-to-peer. If an application attempts to establish a peer-to-peer connection using a Service ID that has already been registered through `RegisterService` the connect call will fail with an error status.

3 Interfaces

This section contains details on channel interfaces, methods, and associated parameters.

The following diagram illustrates the class hierarchy of the various interfaces classes comprising the Channel Services:



LANE 15 SOFTWARE

`ICompletionEvent`, `IAsyncEvent`, `IAddress`, `IMemoryWindow`, `IMemoryRegion`, and `IEeContext` are exposed directly from the verbs interface. `IChannelCa`, `IChannelPd`, `IChannelRdd`, and `IChannel` expose methods which are a direct pass through to the verbs interface. See the Lane15 Verbs documentation and verbs interface definitions for more details.

In the following sections the interfaces are organized in a functional hierarchy (as opposed to a UML class dependency hierarchy). This helps to keep common functional elements grouped together.

3.1 Channel Manager Class

The channel manager is the class factory that applications use to acquire interfaces to channel adapters on host or target systems. The channel adapter's GUID is used to identify the channel adapter the application wants to gain access to.

The channel manager also provides a mechanism to query the channel adapter GUIDs for all channel adapters on a platform.

3.1.1 CChannelManager::QueryCaCount

Used to find the number of available channel adapters. Consumers should use this to determine the size of the `guidArray` to pass to `QueryCaGuids`.

Declaration:

```
CReturnStatus CChannelManager::QueryCaCount(
      UINT16 *                guidCount
);
```

Parameters:

```
guidCount:   [out] The number of channel adapter GUIDs that would
             be returned by calling QueryCaGuids.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.1.2 CChannelManager::QueryCaGuids

Returns a list of GUIDs for all available channel adapters. This list could change between calls, for example, if a new PnP device driver is installed. Application should supply an array of sufficient length to store all channel adapter GUIDs. The function will copy up to the smaller of `guidArraySize` or `guidCount` GUIDs into the provided array. Consumers can use `QueryCaCount` to determine the size of the array to allocate. Since new PnP adapters may appear between the call to `QueryCaCOunt` and `QueryCaGuids`,

consumers are encouraged to allocate an array larger than that suggested by `QueryCaCount`.

Declaration:

```
CReturnStatus CChannelManager::QueryCaGuids(
      UINT16                  guidArraySize,
      GuidType *              guidArray,
      UINT16 *                guidCount
);
```

Parameters:

```
guidArraySize:  [in] Size of the guidArray (in number of GUIDS).
                If this is larger than guidArray, then memory
                beyond quidArray will be overwritten.
guidArray:      [out] Array of length guidCount containing the
                GUIDs of all avalable channel adaptors.
                Application must supply an array of sufficient
                length to store all the channel adapter GUIDs. If
                the array is not large enough (determined by
                guidCount)  to store all of the available GUIDs,
                then up to guidCount entries will be copied.
guidCount:      [out] number of GUIDs copied into the guidArray.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.1.3 CChannelManager::OpenChannelCa

Opens an interface to a channel adapter, and associates an event handler and event contexts with the given instance of the channel adapter.

Declaration:

```
CReturnStatus CChannelManager::OpenChannelCa(
      GuidType                 caGuid,
      IAsyncEvent            * asyncEventHandler,
      IchannelCa             * channelInterface
);
```

Parameters:

```
caGuid:             [in] GUID of the channel adapter to open. The
                    caGuid can be retrieved by calling
                    CChannelManager::QueryCaGuids.
asyncEventHandler:  [in] Consumer implemented event callback
                    interface. See description of Asynchronous
                    Event Interface.

                    Note that the event handler registered with
                    this call will only receive unaffiliated
                    events. The unaffilited errors directed to
```


LANE 15 SOFTWARE

```
                          the CA event handler are:

                          L15_UER_CA_CATASTROPHIC_ERROR
                          L15_UER_CA_PORT_ERROR

                          Affiliated events are directed to the event
                          handler registered with the specific
                          resource. See IAsyncEvent::AsyncEvent.
channelInterface:         [out] Pointer to a channel interface for the
                          specified CA GUID.  If the call fails, then
                          the interface pointer is unchanged.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.2 Channel Adapter Interface

3.2.1 IChannelCa::RegisterService

Registers a service with the associated channel interface.

- This function can be used to register on a single port on the adapter. If the consumer needs to register on all the ports or a subset of the ports on an adapter, then this function will need to be called once for each port.
- If multiple entities register with the same service ID, which service entity gets called is undefined.
- An application can register multiple services. However, each service should have a unique service ID.

Declaration:

```
CReturnStatus IChannelCa::RegisterService(
      ServiceAttrsStruct *    serviceAttributes,
      IChannelServiceEvent *  channelServiceEvent,
      void **                 serviceRegistrationHandle
);
```

Parameters:

```
serviceAttributes:          [in] Attributes for the service. See
                            ServiceAttrsStruct.
channelServiceEvent:        [in] Implementation of the
                            IChannelServiceEvent interface
                            supplied by the consumer.
                            See Channel Services Event Interface
                            for details.
serviceRegistrationHandle:  [out] Used to deregister the service.
                            See IChannelCa::DeregisterService
```

3.2.2 IChannelCa::DeregisterService

Removes registration from CM service lookup table.

Declaration:

```
CReturnStatus IChannelCa::DeregisterService(
      Void *                  serviceRegistrationHandle
);
```

Parameters:

serviceRegistrationHandle:	[in] Service registration handle returned by a previous call to IChannelCa::RegisterService.

3.2.3 IChannelCa::FindUdService

Initiates a query to acquire the remote channel address for the specified UD service on a specified device. The results of the query are returned to the CI consumer through a consumer supplied `IUdServiceResponse` interface (see IUdServiceResponse::ServiceReply).

This is used only for unreliable datagram services.

Declaration:

```
CReturnStatus IChannelCa::FindUdService(
      PathStruct *            path,
      UINT64                  serviceId,
      UINT8                   privateDataLength,
      void *                  privateData,
      IUdServiceResponse *    serviceResponse
);
```

Parameters:

path:	[in] Path to the remote service. This path can be acquired using the same mechanisms as those used to acquire paths for reliable services. However, for an unreliable service this path is incomplete. FindUdService, is used to acquire the rest of the path.
serviceId:	[in] IB compliant service ID. For IANA defined services, this may be the 16 bit IANA port number (not to be confused with an IB port number).
privateDataLength:	[in] Length in bytes of the privateData. For UD service ID resolution, this must not exceed 216 bytes.
privateData:	[in] Private data passed to the service provider. This data is application specific.
serviceResponse:	[in] Event interface used to process service ID resolution responses. See

IUdServiceResponse::ServiceReply.

Notes:

Use IChannel::Connect to connect to reliable services.

3.2.4 IChannelCa::CreateCq

Declaration:

```
CReturnStatus IChannelCa::CreateCq(
      UINT32                     minEntries,
      ICompletionEvent        *  completionEvent,
      ICompletionQueue        ** completionQueue,
      UINT32                  *  actualEntries
);
```

Parameters:

minEntries:	[in] The minimum number of entries the caller will accept for the completion queue.
completionEvent:	[in] Implementation of an ICompletionEvent interface. Must not be null. It is up to the implementor of this object to associate the object with the newly created completion queue. For example, the completion queue can be a member of the implementation class. See ICompletionEvent::CompletionEvent.
completionQueue:	[out] Interface to newly created completion queue.
actualEntries:	[in] Actual number of entries allowed for this completion queue.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.2.5 IChannelCa::CreatePd

Create a protection domain.

Declaration:

```
CReturnStatus IChannelCa::CreatePd(
      IChannelPd              **  channelPd
);
```

Parameters:

channelPd:	[out] Newly created protection domain.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.2.6 IChannelCa::CreateRdd

Create a reliable datagram domain.

Declaration:

```
CReturnStatus IChannelCa::CreateRdd(
      IChannelRdd **          channelRdd
);
```

Parameters:

channelRdd: [out] Newly created reliable data domain.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.2.7 IChannelCa::Query

Returns the attributes for the specified Channel Adapter.

Declaration:

```
CReturnStatus IChannelCa::Query(
      QueryCaStruct*          attributes
);
```

Parameters:

attributes: [out] Channel Adapter attributes for specified adapter.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.2.8 IChannelCa::QueryPort

Returns the attributes for the specified port of the Channel Adapter.

Declaration:

```
CReturnStatus IChannelCa::QueryPort(
      UINT8                   portNumber,
      QueryCaPortStruct*      portAttributes
);
```

Parameters:

portNumber:	[in] Port number on the Channel Adapter to query.
portAttributes:	[out] Attributes for the specified port on the adapter.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.2.9 IChannelCa::QueryPortTable

Returns the requested table for the specified port of the Channel Adapter. The length specified must match the length of the table requested.

Declaration:

```
CReturnStatus IChannelCa::QueryPortTable(
        UINT8                   portNumber,
        UINT8                   type,
        UINT32                  length,
        UINT64                  table
);
```

Parameters:

portNumber:	[in] Port number on the Channel Adapter.
type:	[in] Type of table (Partition or SourceGid).
length:	[in] Length of the table (this value should be acquired via QueryPort from the appropriate member of QueryCaPortStruct: PartitionTableLength or SourceGidTableLength).
table:	[in] Address of a buffer for the requested table for the specified port on the adapter.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.2.10 IChannelCa::Destroy

Destroys the `IChannel` object.

Declarations:

```
CReturnStatus IChannel::Destroy();
```

Parameters:

None.

3.3 Asynchronous Event Interface

`IAsyncEvent` is the callback interface used to notify CI consumers of errors or events associated with a specific channel adapter. The CI consumer binds this interface to a particular `IChannelCa` instance when calling `CChannelManager::OpenChannelCa`.

These errors or events may be associated with a resource such as a channel, completion queue, or EE context. Refer to section 2.3 for an overview of Channel Services event mechanisms, and section 2.3.4 for details on asynchronous errors and events.

3.3.1 IAsyncEvent::AsyncEvent

Declaration:

```
CReturnStatus IAsyncEvent::AsyncEvent(
      EventRecordStruct       const * eventRecord
);
```

Parameters:

`IAsyncEvent (this)`: Context originally supplied by the CI consumer when the channel adapter instance was created. See the asyncEventHandler parameter to `CChannelManager::OpenChannelCa`.

`eventRecord`: An EventRecordStruct. See `EventRecordStruct`.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.4 UD Service Response Interface

3.4.1 IUdServiceResponse::ServiceReply

Declaration:

```
CReturnStatus IUdServiceResponse::ServiceReply(
      SidrRespStruct          const * serviceResponse
);
```

Parameters:

`serviceResponse`: Address of UD service.

Notes:

See: `IChannelCa::FindUdService`
`IChannelServiceEvent::UdServiceRequest`

3.5 Channel Services Event Interface

3.5.1 IChannelServiceEvent::ConnectRequest

Used to notify a service provider that a client is requesting a connection to the service. This is only called when a client first requests a connection from a service provider. Once the service provider associates an `IChannel` with the request (by calling `IChannelServiceAction::Accept`), `IChannelEvent::ConnectionEvent` is used for all subsequent connection events.

When in the listen state, no channel has been associated with the connect request. Once the service decides to accept a connect request, it (the service) allocates a channel and associates it with the connect request by calling Accept and passing in the channel instance. All further events associated with that channel are sent through the channel's associated `IChannelEvent` object.

Declaration:

```
CReturnStatus IChannelServiceEvent::ConnectRequest(
      const CsConnectRequestStruct  * connectRequest,
      IchannelServiceAction         * connectRequestAction
);
```

Parameters:

connectRequest:	[in] ConnectRequest sent by the client. See IChannel::Connect for details on the connect request structure.
connectRequestAction:	[in] The consumer uses this interface to accept or reject the connection. If the consumer needs time to process the connect request before accepting or rejecting the connection, then this interface can be used to send an MRA to the client. The interface can be preserved until such time that the accept or reject is sent.

Notes:

See `IChannelServiceAction::Accept`, `IChannelServiceAction::Reject`, & `IChannelServiceAction::SendMra.`

3.5.2 IChannelServiceEvent::ConnectReject

A client can send a connect reject at any time, even before the service has had the opportunity to accept or reject the request. This method is used to forward these early rejects to the service provider. If a connect reject is received after the request has been accepted, then the reject will be sent to the `IChannelEvent::ConnectionEvent` method.

Consumers cannot respond to rejects, so no context or action interface is supplied.

Declaration:

```
CReturnStatus IChannelServiceEvent::ConnectReject(
      const CsConnectRejectStruct   * connectReject
);
```

Parameters:

connectReject: [in] CsConnectReject sent by the client. See IChannel::Connect for details on the connect request structure.

3.5.3 IChannelServiceEvent::UdServiceRequest

UD service ID resolution requests are forwarded to consumers via the `UdServiceRequest` callback.

Declaration:

```
CReturnStatus IChannelServiceEvent::UdServiceRequest(
      const CsSidrReqStruct * sidrRequest,
      IChannel**              channel
);
```

Parameters:

sidrRequest: [in] Used by CM to notify the application running as a UD service that a remote node is requesting that service.

channel: [out] IChannel interface returned to the consumer.

3.6 Channel Service Action Interface

Service providers use this interface to respond to client connect requests. If the service provider wants to accept the connection request, then the service provider must create a channel object and pass it in to the Accept method before returning from the `ConnectionEvent()` callback.

3.6.1 IChannelServiceAction::Accept

Used by a service provider to respond to a client's request to establish a channel. If the service provider wants to establish the connection, then the service provider must create a channel object and pass it in to the Accept method before returning from the `ConnectionEvent()` callback.

Declaration:

```
CReturnStatus IChannelServiceAction::Accept(
      IChannel *                channel,
      CsConnectReplyStruct *    acceptInfo
);
```

Parameters:

channel: [in] Channel previously created using IChannelPd::CreateChannel.
acceptInfo: [in] See CsConnectReplyStruct.

Notes:

The channel passed into this method must be in the disconnected state. If the channel is in the disconnected state due to a previous error, then the channel will be automatically reset before use.

3.6.2 IChannelServiceAction::Reject

Reject a connect request.

Declaration:

```
CReturnStatus IChannelServiceAction::Reject(
      CsConnectRejectStruct * rejectInfo
);
```

Parameters:

rejectInfo: [in]See CsConnectRejectStruct.

3.6.3 IChannelServiceAction::SendMra

Delay processing of a connect request.

Declaration:

```
CReturnStatus IChannelServiceAction::SendMra(
      CsMraStruct *            mraInfo
);
```

Parameters:

mraInfo: [in]See CsMraStruct.

3.7 Channel Reliable Datagram Domain Interface

3.7.1 IChannelRdd::CreateEeContext

Creates an EE Context for the specified CA. On success, a handle to the newly created EE Context is returned. The values for Remote Node Address, Send Sequence Number, Receive Sequence Number are all zero. The EE Context is created in the Reset state.

Declaration:

```
CReturnStatus IChannelRdd::CreateEeContext(
        void*                   asyncEventContext,
        IEeContext**            eeContext
);
```

Parameters:

```
AsyncEventContext:   [in]  An opaque object used by the verbs
                     consumer to maintain context between the newly
                     created EE Context object and subsequent async
                     event notifications. This context is opaque to
                     the verbs and will be passed to the verbs
                     consumer during event notifications. May be
                     NULL.
EeContext:           [out] EE Context object returned to caller.
```

3.7.2 IChannelRdd::Deallocate

Returns a previously allocated reliable datagram domain object for reuse by the Allocate Reliable Datagram Domain Verb. The reliable datagram domain object cannot be deallocated if it is still associated with a Queue Pair or an EE Context.

Declaration:

```
CReturnStatus IChannelRdd::Deallocate();
```

Parameters:

None.

3.8 End-to-End Context Interface

3.8.1 IEeContext::Modify

If the CI supports RD Service, upon invocation of this Verb, the CI shall modify the attributes for the specified EE Context and then shall cause the EE Context to transition to the specified EE Context state.

Declaration:

```
CReturnStatus IEeContext::Modify(
        ModifyEecReqStruct* reqEecAttrs,
        UINT16              optAttrsFlags,
        ModifyEecOptStruct* optEecAttrs
);
```

Parameters:

reqEecAttrs:	[in] The required attribute values of the EEC to modify based on the state transition.
optAttrsFlags:	[in] Flag bit-mask indicating which (if any) of the optional attributes have been provided by the caller.
optEecAttrs:	[in] The optional attribute values of the EEC to modify based on the state transition. If optAttrsFlag is 0, this parameter should be NULL.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.8.2 IEeContext::Query

Returns the attribute list and current values for the specified EE Context.

Declaration:

```
CReturnStatus IEeContext::Query(
        EeContextAttrsStruct* eeContextAttrs
);
```

Parameters:

eeContextAttrs:	[out] The values of the EE Context attributes.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.8.3 IEeContext::Destroy

Destroys the specified EE Context. Any resources allocated by the Channel Interface for use by the EE Context are freed from use. If the CI supports RD Service, after this Verb is invoked, any outstanding or subsequently submitted Work Requests that depend on the EE Context shall complete with an Invalid EE Context Number error.

Declaration:

```
CReturnStatus IEeContext::Destroy();
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.9 Completion Queue Interface

3.9.1 ICompletionQueue::Resize

Resizes the CQ. A CQ must be able to be resized with outstanding Work Completions on the CQ and Work Requests on queues associated with the specified CQ.

The resize operation is allowed to adversely affect the performance while the CQ is being resized. The act of resizing is not allowed to directly generate completion or asynchronous errors. Completions must not be lost due to a CQ resize.

Declaration:

```
CReturnStatus ICompletionQueue::Resize(
        UINT32                minNbrOfEntries,
        UINT32*               actualNbrOfEntries
);
```

Parameters:

```
minNbrOfEntries:      [in]  The minimum number of entries the
                      caller will accept for the size of the
                      completion queue.
actualNbrOfEntries:   [out] The actual number of entries in the
                      completion queue after the resize operation.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.9.2 ICompletionQueue::Query

Returns the number of entries in the specified CQ.

Declaration:

```
CReturnStatus ICompletionQueue::Query(
      UINT32*                 numberOfEntries
);
```

Parameters:

```
numberOfEntries:  [out]  The current size of the completion queue
                  in terms of number of entries.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.9.3 ICompletionQueue::Destroy

Destroys the specified CQ. Resources allocated by the Channel Interface to implement the CQ must be deallocated during the destroy operation.

The CQ cannot be destroyed if any Work Queue is still associated with the CQ.

Any completions that have not been retrieved from the CQ prior to being destroyed are discarded.

Declaration:
```
CReturnStatus ICompletionQueue::Destroy();
```

Parameters:
```
None.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.9.4 ICompletionQueue::PollForCompletion

Polls the specified CQ for a Work Completion. A Work Completion indicates that a Work Request for a Work Queue associated with the CQ is done.

If an entry is present, the Work Completion at the head of the CQ is returned to the Consumer.

Declaration:
```
CReturnStatus ICompletionQueue::PollForCompletion(
      CmpReqStruct*           workCompletion
);
```

Parameters:
```
workCompletion:   [out]  The work completion request containing
                  information relating to a completed work
                  request if and only if an entry is present on
                  the completion queue.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.9.5 ICompletionQueue::RequestCompletionNotification

Requests the CQ event handler be called when the next completion entry of the specified type is added to the specified CQ. The handler is called at most once per Request Completion Notification call for a particular CQ. Any CQ entries that existed before the

notify is enabled will not result in a call to the handler. See the IB specification for more details.

Declaration:

```
CReturnStatus ICompletionQueue::RequestCompletionNotification(
        UINT8                 completionNotificationType
);
```

Parameters:

```
completionNotificationType:   [in]  Type of completion
                              notification requested.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.10 Completion Event Interface

3.10.1 ICompletionEvent::CompletionEvent

Completion events are used to notify `IChannel` consumers of data transport events such as send or receive completions. To register for CQ events, the CI consumer must supply an `ICompletionEvent` instance when creating the completion queue (see `IChannelCa::CreateCq`).

Declaration:

```
CReturnStatus ICompletionEvent::CompletionEvent();
```

Parameters:

None.

Notes:

The `CompletionEvent()` could be running in a shared thread, a dedicated thread, a software DPC interrupt, or a hardware interrupt. The choice is up to the channel adapter (or IB verbs) vendor. Host applications running in user space (Win2k or Linux) can assume that the completion notification will execute in single thread dedicated to the application. RTOS target application developers should consult with the channel adapter or IB verbs vendor for implementation details.

See the Lane15 Verbs Reference Guide for more information.

3.11 Channel Protection Domain Interface

3.11.1 IChannelPd::CreateChannel

Creates a channel with an associated Connection End Point (i.e., QP).

Declaration:

```
CReturnStatus IChannelPd::CreateChannel(
      ChannelAttrsStruct *    attributes,
      IChannelEvent *         channelEventHandler,
      IChannel **             channel,
      QpPropertiesStruct *    actualProperties
);
```

Parameters:

```
attributes:            [in] See ChannelAttrsStruct.
channelEventHandler:   [in] Event handler to receive connection
                       events and affiliated asynchronous events.
                       See IAsyncEvent::AsyncEvent.
channel:               [out] IChannel interface returned to the
                       consumer.
actualProperties:      [out] QP properties.
```

3.11.2 IChannelPd::RegisterMemoryRegion

Prepares a virtually addressed memory region for use by an HCA.

Declaration:

```
CReturnStatus IChannelPd::RegisterMemoryRegion(
        UINT64                    virtualAddress,
        UINT32                    length,
        UINT8                     accessControl,
        IMemoryRegion**           memHandle,
        UINT32*                   localKey,
        UINT32*                   remoteKey
);
```

Parameters:

```
virtualAddress  [in] The address of the first byte of the region
                to be registered. The maximum size of a virtual
                address is 64 bits.
length:         [in] Length of region to be registered in bytes.
accessControl:  [in] Memory region access control mask.
memHandle:      [out] Memory region interface object returned for
                the registered memory region.
localKey:       [out] L_Key (local key) for this memory region.
```

remoteKey: [out] R_Key (remote key) for this memory region if and only if remote access was requested.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.11.3 IChannelPd::AllocateMemoryWindow

Allocates a memory window that is associated with a protection domain. It is not inherently associated with any memory region when allocated.

Declaration:

```
CReturnStatus IChannelPd::AllocateMemoryWindow(
        IMemoryWindow**       windowHandle,
        UINT32*               remoteKey
);
```

Parameters:

windowHandle: [out] Window Handle used to identify this specific Memory Window to other Memory Management Verbs.

remoteKey: [out] An unbound R_Key for use in specifying the Window with the Bind Memory Window Verb.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.11.4 IChannelPd::CreateAddress

Creates an address handle for the address vector passed in through the verbs.

Declaration:

```
CReturnStatus IChannelPd::CreateAddress(
        UdAddrDescStruct*     addressDescription,
        IAddress**            address
);
```

Parameters:

addressDescription: [in] Address vector description.

address: [out] Address interface object.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.11.5 IChannelPd::Destroy

Returns a previously allocated Channel Protection Domain object for reuse by the Create Protection Domain method. See `IChannelCa::CreatePd`. The Channel Protection Domain object cannot be deallocated if it is still associated with any Queue Pair, Memory Region, or Memory Window.

Declaration:

```
CReturnStatus IChannelPd::Destroy();
```

Parameters:

None.

3.12 Address Interface

3.12.1 IAddress::Modify

Changes an address vector associated with the address handle passed in by the Consumer.

Declaration:

```
CReturnStatus IAddress::Modify(
        UdAddrDescStruct*     addressDescription
);
```

Parameters:

addressDescription: [in] Address vector description.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.12.2 IAddress::Query

Obtains the address vector associated with the address handle passed in by the Consumer.

Declaration:

```
CReturnStatus IAddress::Query(
        UdAddrDescStruct*     addressDescription
);
```

Parameters:

addressDescription: [in] Address vector description.

Notes:
See the Lane15 Verbs Reference Guide for more information.

3.12.3 IAddress::Destroy

Removes an address vector and its associated address handle from the CA. After the address handle is removed, it can no longer be used to reference the destination.

Declaration:
```
CReturnStatus IAddress::Destroy();
```

Parameters:
None.

Notes:
See the Lane15 Verbs Reference Guide for more information.

3.13 Memory Region Interface

3.13.1 IMemoryRegion::Query

Query a previously registered memory region.

Declaration:
```
CReturnStatus IMemoryRegion::Query(
        QueryMrStruct*        memRegionAttrs
);
```

Parameters:

memRegionAttrs: [out] Attributes for the Memory Region.

Notes:
See the Lane15 Verbs Reference Guide for more information.

3.13.2 IMemoryRegion::Reregister

Modifies the attributes of an existing Memory Region.

Declaration:
```
CReturnStatus IMemoryRegion::Reregister(
        ModifyMrStruct*       changeRequest,
        IMemoryRegion**       memHandle,
        UINT32*               localKey,
```

```
        UINT32*                remoteKey
);
```

Parameters:

```
changeRequest:  [in] Change Request.
memHandle:      [out] Memory Region handle. Must be used for
                future references to this Memory Region. Might or
                might not be the same as the previous Memory
                Region handle.
localKey:       [out] L_Key used for local access.
remoteKey:      [out] R_Key used for remote access. Only returned
                whenremote access is requested.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.13.3 IMemoryRegion::ReregisterPhysical

Modifies the attributes of an existing Physical Memory Region.

Declaration:

```
CReturnStatus IMemoryRegion::ReregisterPhysical(
        ModifyPhysMrStruct*   changeRequest,
        IMemoryRegion**       memHandle,
        UINT64*               iovaActual,
        UINT32*               localKey,
        UINT32*               remoteKey
);
```

Parameters:

```
changeRequest:  [in] Change Request.
memHandle:      [out] Physical Memory Region handle. Must be used
                for future references to this Memory Region. Might
                or might not be the same as the previous Memory
                Region handle.
iovaActual:     [out] I/O Virtual Address (IOVA) actually assigned
                by the Channel Interface for the first byte of the
                region.
localKey:       [out] L_Key used for local access.
remoteKey:      [out] R_Key used for remote access. Only returned
                when remote access is requested.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.13.4 IMemoryRegion::RegisterShared

Register Shared Memory Region. Given an existing Memory Region, a new independent Memory Region associated with the same physical memory locations is created, with the intention that the new Memory Region share CA mapping resources to the extent possible.

Declaration:

```
CReturnStatus IMemoryRegion:: RegisterShared (
        UINT64                virtualAddrRequest,
        UINT8                 accessControl,
        IProtectionDomain*    protectionDomain,
        IMemoryRegion**       memHandle,
        UINT64*               virtualAddrActual,
        UINT32*               localKey,
        UINT32*               remoteKey
);
```

Parameters:

virtualAddrRequest:	[in] Virtual Address requested by the Consumer for the first page of the buffer.
accessControl:	[in] Access Control selections.
protectionDomain:	[in] Protection Domain for the shared memory region (if different than current one).
memHandle:	[out] Memory Region Handle of the new Memory Region.
virtualAddrActual:	[out] Virtual Address actually assigned by the Channel Interface for the first page.
localKey:	[out] L_Key used for local access.
remoteKey:	[out] R_Key used for remote access. Only returned when remote access is requested.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.13.5 IMemoryRegion::Deregister

Deregister a previously registered memory region.

Declaration:

```
CReturnStatus IMemoryRegion::Deregister();
```

Parameters:

None.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.14 Memory Window Interface

3.14.1 IMemoryWindow::Query

This verb returns the attributes associated with the specified Memory Window.

Declaration:

```
CReturnStatus IMemoryWindow::Query(
        UINT32*              remoteKey,
        IProtectionDomain**  protectionDomain
);
```

Parameters:

```
remoteKey:          [out] The current R_Key associated with the
                    Memory Window.
protectionDomain:   [out] Protection Domain associated with the
                    Memory Window.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.14.2 IMemoryWindow::Deallocate

Deallocates the Memory Window.

Declaration:

```
CReturnStatus IMemoryWindow::Deallocate();
```

Parameters:

None.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.15 Channel Interface

3.15.1 IChannel::Connect

Declaration:

```
CReturnStatus IChannel::Connect(
      CsConnectRequestStruct  const * connectRequest,
      BOOL                    connectPeer
```

```
);
```

Parameters:

```
connectRequest:   [in] See CsConnectRequestStruct.
connectPeer:      [in] Indicates if this is a peer connection. See
                  the description of Peer Connections details.
```

Notes:

The channel must be in the disconnected state when calling this method. If the channel is in the disconnected state due to a previous error, then the channel will be automatically reset before use.

3.15.2 IChannel::Disconnect

Closes an active connection, or aborts a connection in process, or resets a channel after an error event.

Declaration:

```
CReturnStatus IChannel::Disconnect (
      CsDisconnectRequestStruct *   disconnectInfo
);
```

Parameters:

```
disconnectInfo:   [in] See CsDisconnectRequestStruct.
```

3.15.3 IChannel::PostSend

Posts a send request.

Declaration:

```
CReturnStatus IChannel::PostSend (
      PostSendRequestStruct * sendRequest
);
```

Parameters:

```
sendRequest:      [in] The work request containing the information
                  required to perform the send request.
```

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.15.4 IChannel::PostRecv

Posts a receive request.

Declaration:

```
CReturnStatus IChannel::PostRevc (
      RcvReqStruct *          recvRequest
);
```

Parameters:

recvRequest: [in] The work request containing the information required to perform the receive request.

Notes:

See the Lane15 Verbs Reference Guide for more information.

3.15.5 IChannel::Reset

Returns a channel to a usable disconnected state.

Declaration:

```
CReturnStatus IChannel::Reset();
```

Parameters:

None.

3.15.6 IChannel::Query

Declaration:

```
CReturnStatus IChannel::Query (
      QueryChannelStruct *    channelAttrs,
      UINT8 *                 channelState
);
```

Parameters:

channelAttrs: [out] Channel attributes.

channelState: [out] Current channel state. One of the following:
CHANNEL_STATE_IDLE
CHANNEL_STATE_CONNECTED
CHANNEL_STATE_PENDING_CONNECTION
CHANNEL_STATE_ERROR

3.15.7 IChannel::Modify

Declaration:

```
CReturnStatus IChannel::Modify (
      ModifyQpReqStruct *     reqChannelAttrs
      UINT16                  optAttrsFlag,
      ModifyQpOptStruct *     optChannelAttrs,
      UINT16 *                actualSndReq,
      UINT16 *                actualRcvReq
);
```

Parameters:

reqChannelAttrs: [in] The required attribute values of the channel to modify based on the state transition.
optAttrsFlag: [in] Flag bit-mask indicating which (if any) of the optional attributes have been provided by the caller.
optChannelAttrs: [in] The optional attribute values of the channel to modify based on the state transition. If optAttrsFlag is 0, this parameter should be NULL.
actualSndReq: [out] The actual number of Work Requests supported on the Send Queue of the channel.
actualRcvReq: [out] The actual number of Work Requests supported on the Receive Queue of the channel.

Notes:
See the Lane15 Verbs Reference Guide for more information.

3.15.8 IChannel::Destroy

Destroys the channel instance. Further calls using the destroyed channel object result in error.

Declaration:

```
CReturnStatus IChannel::Destroy ();
```

Parameters:
None.

3.16 Channel Event Interface

3.16.1 IChannelEvent::AsyncEvent

Includes:

```
verbsDef.h
```

Declaration:

```
CReturnStatus IChannelEvent::AsyncEvent(
      EventRecordStruct *     eventRecord
);
```

Parameters:

eventRecord: [in] See EventRecordStruct. Resource type will be set to L15_RESOURCE_TYPE_QP. EventOrError will be one of the following:

```
/* Affiliated Events (AEV) */
L15_AEV_QP_PATH_MIGRATED
L15_AEV_EE_PATH_MIGRATED
L15_AEV_QP_COMM_ESTABLISHED
L15_AEV_EE_COMM_ESTABLISHED
L15_AEV_QP_SEND_QUEUE_DRAINED

/* Affiliated Errors (AER) */
L15_AER_QP_PATH_MIGRATION_ERROR
L15_AER_EE_PATH_MIGRATION_ERROR

L15_AER_CQ_WRITE_ERROR
L15_AER_CQ_OVERRUN_ERROR
L15_AER_QP_CATASTROPHIC_ERROR
L15_AER_EE_CATASTROPHIC_ERROR
```

Notes:

The channel must be in the disconnected state when calling this method. If the channel is in the disconnected state due to a previous error, then the channel will be automatically reset before use.

See the Lane15 Verbs Reference Guide for more information.

3.16.2 IChannelEvent::ConnectionEvent

Channel events originate in the CM, and are propagated to the channel consumer to allow the consumer to exchange private data with the remote process.

Declaration:

```
CReturnStatus IChannelEvent::ConnectionEvent (
      UINT32                  eventType,
      void *                  attribute,
```

```
        IChannelAction *        channelAction
);
```

Parameters:

eventType: [in] One of the following:
ConnectRequest (See Peer Connections)
ConnectReply
ConnectReject
DisconnectRequest
DisconnectReply
LoadAlternatePath (can be rejected)
MraReceived (can be rejected)

attribute: [in] Depending on the event type, this could be any one of the following:
CsConnectRequestStruct
CsConnectReplyStruct
CsConnectRejectStruct
CsDisconnectRequestStruct
CsDisconnectReplyStruct
CsAltPathStruct
CsMraStruct

channelAction: [in] Pointer to an IChannelAction, or NULL. If this parameter is not null, then the consumer must respond by calling one of the methods of the channel action interface. See Channel Action Interface.

EventType	Attribute	IChannelAction::		
		Accept	Reject	Mra
Connect Request	CsConnectRequestStruct	X(1)	X	X
Connect Reply	CsConnectReplyStruct	X(2)	X	X
Connect Reject	CsConnect-RejectStruct			
Disconnect Request	CsDisconnect-RequestStruct	X(3)		
Disconnect Reply	CsDisconnectReply			
MraReceived	CsMraStruct		X	

(1) Accept with a CsConnectReplyStruct.
(2) Accept with a CsRtuStruct
(3) Accept with a CsDisconnectReplyStruct

3.16.3 IChannelEvent::ConnectError

Indicates a connection attempt failed, typically due to the connect request timing out.

Declaration:

```
CReturnStatus IChannelEvent::ConnectError (
      UINT32                  reason,
      void *                  data
);
```

Parameters:

```
reason:     Rejection reason.

data:       Private data.
```

3.17 Channel Action Interface

3.17.1 IChannelAction::Accept

Used by the active consumer to respond to a connect reply. Used by the passive consumer to accept a connect request or a disconnect request.

Declaration:

```
CReturnStatus IChannelAction::Accept (
      CsRtuStruct *           acceptInfo
);

CReturnStatus IChannelAction::Accept (
      CsConnectReplyStruct *  acceptInfo
);

CReturnStatus IChannelAction::Accept (
      CsDisconnectReplyStruct * acceptInfo
);
```

Parameters:

```
acceptInfo: See one of the following:
                  CsRtuStruct
                  CsDisconnectReplyStruct
                  CsConnectReplyStruct
```

3.17.2 IChannelAction::Reject

Reject a pending connection.

Declaration:

```
CReturnStatus IChannelAction::Reject (
      CsConnectRejectStruct * rejectInfo
);
```

Parameters:

rejectInfo: See CsConnectRejectStruct.

3.17.3 IChannelAction::SendMra

Declaration:

```
CReturnStatus IChannelAction::SendMra (
      CsMraStruct *           mraInfo
);
```

Parameters:

mraInfo: See CsMraStruct.

4 Attributes

In the following tables, the *Source* field indicates what entity is responsible for setting the value in that structure. The *Source* field will contain one of the following:

CI: Channel Services Interface
CM: InfiniBand Communications manager (below the CI)
Pol: Policy based (below the CI)
App: End user application; alternatively, this could indicate the ULP
ULP: Upper Layer Protocol
QoS: Quality of Service manager (below the CI)

4.1 CsConnectRequestStruct

Table 5- CsConnectRequestStruct field source information

Field	Source	Notes
Local Communications Id	CM	Value is assigned by the CM, and is opaque to layers above the CM.
ServiceID	ULP	
LocalCaGuid	CI	Channel services acquires the LocalCaGuid on service startup.
LocalCmQkey	CM	
LocalQkey	Pol	Policy based. Currently this is set by the app in the ChannelAttrsStruct.
LocalQpn	CI	Captured when the channel was created.
Offered-Responder-Resources	CI	Queried from the channel adapter.
LocalEECN	CI	
Offered-Initiator-Depth	CI	
RemoteEECN	CI	
RemoteCM-Response-Timeout	App	The time, expressed as (4.096 µS*$2^{\text{Remote CM Response Timeout}}$), within which the CM message recipient shall transmit a response to the sender. The recipient uses this information to determine whether it should send an MRA. For example, VIPL uses this as the count, in milliseconds, that VipConnectRequest will wait for connection to complete before returning to the caller. In this case, VIPL might ignore the MRA and send a reject once the timeout time has expired.
Transport-Service-Type	ULP	Specifies desired service type 0: RC (Reliable Connected) 1: UC (Unreliable Connected) 2: RD (Reliable Datagram) 3. Reserved
End-to-end	QoS	Capability is queried from the HCA. Whether to

Flow-Control		use this is a QoS policy decision.
Starting PSN	CI	
Local CM response Timeout	CI	This is derived from the RemoteCmResponseTimeout as follows: (note that the “turn-around” time is derived from the path record). The time, expressed as (4.096 µS*2 Local CM Response Timeout), that the remote CM shall wait for a response from the local CM to a CM message sent by the remote CM. This value is unsigned. Note that whereas Remote CM Response Timeout is the time between receipt of a message and transmission of a response, Local CM Response Timeout includes that "turn-around" time, as well as round trip packet flight time. The initiating CM is responsible for determining this value, through Subnet Management or other means.
Retry count	QoS	Retry count for CM/CI level rejects.
Partition key	CI	Note that both paths must share the same PKEY. Access: IQueuePair::Query() QueryQpStruct.PrimaryPkeyIndex QueryQpStruct.AltPkeyIndex
Path packet Payload MTU	CI	The path MTU must be the same for both the primary path and the alternate path. The CI must select the smaller of the two path MTUs. Access: IQueuePair::Query() QueryQpStruct.PrimaryAddrVector.Mtu QueryQpStruct.AltAddrVector.Mtu
RDC Exists	CI	
RNR retry count	QoS	Quality of service, potentially policy based.
Max CM retries	ULP	Combined with the remote CM response timeout, this determines the total timeout time for connection establishment attempts.
Primary Local port LID	CI	
Primary Remote port LID	CI	
Primary local port GID	CI	
Primary remote port GID	App	Address of the service. For example, this is resolved by ServiceRecord queries.
Primary Flow label	CI	
Primary packet rate	CI	
Primary traffic class	CI	
Primary hop limit	CI	

Primary SL	CI	
Primary subnet local	CI	
Primary local ACK timeout	CI	
Alternate Local port LID	CI	
Alternate Remote port LID	CI	
Alternate local port GID	CI	
Alternate remote port GID	CI	
Alternate Flow label	CI	
Alternate packet rate	CI	
Alternate traffic class	CI	
Alternate hop limit	CI	
Alternate SL	CI	
Alternate subnet local	CI	
Alternate local ACK timeout	CI	
Private Data	ULP	Optional

4.2 *CsConnectReplyStruct*

Table 6- CsConnectReplyStruct field source information

Field	Source	Notes
Local Communications ID	CM	
Remote Communications ID	CM	
Local Q_Key	Pol	Policy based.
Local QPN	CI	Queried from CA queuepair
Local EECN	CI	Queried from IqueuePair
Starting PSN	CI	Old PSN is queried from CA, new PSN is calculated by CI.
Responder Resources	CI	Queried from CA
Initiator Depth	CI	Queried from CA

Target ACK Delay	CI	Queried from the channel adapter (HCA)
Failover Accepted	CI	CI queries the channel adapter to determine if APM is supported. If not, and the requester specified an alternate path, then this field should be set to 1.
End to End Flow Control	CI/QoS	Capability is queried from the HCA. Whether to use this is a QoS policy decision.
RNR Retry	QoS	QoS supervisor function.
Private Data	ULP	Data that is opaque to the communication management protocol, passed from the sender to the recipient. The recipient may choose to accept or reject the request based on the private data. The format and meaning of the PrivateData field is specific to the ServiceID and message type, and is not specified within Communication Management.

4.3 CsRtuStruct

Sent by the active consumer to the passive consumer. Used to notify the passive consumer that the active connection is ready to use.

Table 7- CsRtuStruct field source information

Field	Source	Notes
PrivateData	App	Application specific private data. See CsConnectRequestStruct for details.

4.4 ServiceAttrsStruct

Table 8 – ServiceAttrsStruct field source information

Field	Source	Notes
ServiceId	ULP	IB defined service ID. For IANA defined services, this may be the 16 bit IANA port number (not to be confused with an IB port number). A service provider may register the same service ID with multiple service types.
PortNumber	ULP	IB port the service is to be registered on. Use port 0, or IC_ALL_PORTS to register the service on all ports on the channel adapter.

TransportTypes	ULP	Specifies the transport types that the service registration supports. XPORT_TYPE_RC 1 XPORT_TYPE_UC 2 XPORT_TYPE_RD 4 XPORT_TYPE_UD 8
PeerToPeer	ULP	Set TRUE if the service is a peer to peer service. Otherwise, the service is registered as a client/server service.
RegisterSa	ULP	Set to TRUE if the service should be registered with the SA.
ServiceType	ULP	Character string identifying the service. For example "tftp", or "CFM.IBTA". This is a 124-byte long, UTF-8 encoded, null-terminated string.

4.5 EventRecordStruct

The event record struct is passed as a parameter to IChannelEvent::AsyncEvent.

The event record struct is defined in the context of the verbs interface. The header file can be found in the management verbs include directory in the file EventRecord.h

Table 9- EventRecordStruct field source information

Field	Type	Notes
ResourceType	UINT32	L15_RESOURCE_TYPE_QP L15_RESOURCE_TYPE_CQ L15_RESOURCE_TYPE_EE L15_RESOURCE_TYPE_CA
EventOrError	UINT32	/* Affiliated Events (AEV) */ L15_AEV_QP_PATH_MIGRATED L15_AEV_QP_COMM_ESTABLISHED L15_AEV_QP_SEND_QUEUE_DRAINED L15_AEV_EE_PATH_MIGRATED L15_AEV_EE_COMM_ESTABLISHED /* Affiliated Errors (AER) */ L15_AER_QP_PATH_MIGRATION_ERROR L15_AER_QP_CATASTROPHIC_ERROR L15_AER_CQ_WRITE_ERROR L15_AER_CQ_OVERRUN_ERROR L15_AER_EE_PATH_MIGRATION_ERROR L15_AER_EE_CATASTROPHIC_ERROR /* Unaffiliated Errors (UER) */ L15_UER_CA_CATASTROPHIC_ERROR L15_UER_CA_PORT_ERROR

4.6 ChannelAttrsStruct

Table 10- ChannelAttrsStruct field source information

Field	Source	Notes
SendCq	ULP	The CQ to be associated with the Send Queue.
RecvCq	ULP	The CQ to be associated with the Receive Queue.
Rdd	ULP	Reliable Datagram Domain to be associated with this QP. (Only applicable for RD QPs)
Qkey	ULP	Q_Key to associate with the QP.
MaxSendRequests	ULP	Used to specify the maximum number of outstanding Work Requests the Consumer expects to submit to the Send Queue.
MaxRecvRequests	ULP	Used to specify the maximum number of outstanding Work Requests the Consumer expects to submit to the Receive Queue.
TransportType	ULP	QueuePair/Transport Type. For Special Queue Pairs, the valid QueuePair types are: - SMI QP (QP0) - GSI QP (QP1) - Raw IPv6 - Raw Ethertype For all other Queue Pairs, the valid Transport types are: - Reliable Connection (RC) - Reliable Datagram (RD) - Unreliable Connection (UC) - Unreliable Datagram (UD)
SignalingType	ULP	The Signaling Type must be specified for the Send Queue on this QP. The valid types are: o All Work Requests submitted to the Send Queue always generate a completion entry. o Consumer must specify on each Work Requests submitted to the Send Queue whether to generate a completion entry for successful completions.
MaxSendFragments	ULP	Used to specify the maximum number of scatter/gather elements the Consumer will specify in Work Requests submitted to the Send Queue.
MaxRecvFragments	ULP	Used to specify the maximum number of scatter/gather elements the Consumer will specify in Work Requests submitted to the Receive Queue.
AccessControl	ULP	Enable/Disable o RDMA read o RDMA write o Atomic operations
Qos	ULP	Array of attribute value pairs. Initial values for QoS requirements. Over time, the channel QoS requirements may be reset by the underlying management layers.

4.7 CsDisconnectRequestStruct

Table 11- CsDisconnectRequestStruct field source information

Field	Source	Notes
PrivateData	ULP	Application specific private data. See CsConnectRequestStruct for details.

4.8 CsDisconnectReplyStruct

Table 12- CsDisconnectReplyStruct field source information

Field	Source	Notes
PrivateData	ULP	Application specific private data. See CsConnectRequestStruct for details.

4.9 CsConnectRejectStruct

Table 13- CsConnectRejectStruct field source information

Field	Source	Notes
RejectInfoLength	ULP	Length in bytes of the RejectInfo component.
Reserved		
Reason	ULP	See CS_REJECT_REASON (Table 13)
RejectInfo	ULP	See CS_REJECT_REASON – RejectInfo (Table 13)
PrivateData	ULP	Application specific private data. See CsConnectRequestStruct for details.

Table 14- CS_REJECT_REASON (connection rejection reasons)

	Reason	Description	RejectInfo
1	No QP available	The **REQ** message required the recipient to allocate a QP, and none were available	
2	No EEC available	The **REQ** message required the recipient to allocate an EE context, and none were available	
3	No resources available	The **REQ** message required the recipient to allocate resources other than QPs or EE contexts, and none were available	
4	Timeout	The CM protocol timed out waiting for a message	
5	Unsupported request	Receiving CM does not support this request.	
6	Invalid Communication ID	The recipient received a CM message in which the Local Communication ID, Remote Communication ID, or both, were invalid.	

7	Invalid Communication Instance	The Local Communication ID, Remote Communication ID, QPN/EECN tuple does not refer to any valid communication instance.	
8	Invalid Service ID	The recipient of the **REQ** message does not recognize or does not support the service associated with the specified ServiceID	
9	Invalid Transport Service Type	The recipient of the **REQ** message did not recognize the requested Transport Service Type	
10	Stale connection	The recipient of the **REQ** determined that it already had a connection with the "Local QPN" or "Local EECN" specified in the **RE**Q. Upon receiving a **REJ** with this reason, the **REJ** recipient shall cause the QP or EE context to be placed into the TimeWait state as described in section 12.9.8.4 .	
11	RDC does not exist	The Reliable Datagram Channel described in the **REQ** (Local EECN/Remote EECN) does not exist.	
12	Primary Remote Port GID rejected	The recipient of the **REQ** message could not (or would not) accept the Primary Remote Port GID	GID of acceptable port.
13	Primary Remote Port LID rejected	The recipient of the **REQ** message could not (or would not) accept the Primary Remote Port LID	LID of acceptable port.
14	Invalid Primary SL	The recipient of the **REQ** message does not support the requested Primary SL	Acceptable SL.
15	Invalid Primary Traffic Class	The recipient of the **REQ** message does not support the requested Primary Traffic Class	Acceptable Traffic Class
16	Invalid Primary Hop Limit	The recipient of the **REQ** message could not (or would not) accept the Primary Hop Limit	Acceptable Hop Limit
17	Invalid Primary Packet Rate	The recipient of the **REQ** message could not adjust its trans-mitter to send as slowly as would be required to comply with the requested Primary Packet Rate	Minimum acceptable Packet Rate
18	Alternate Remote Port GID rejected	The recipient of the **REQ** message could not (or would not) accept the Alternate Remote Port GID	GID of acceptable port.
19	Alternate Remote Port LID rejected	The recipient of the **REQ** message could not (or would not) accept the Alternate Remote Port LID	LID of acceptable port.
20	Invalid Alternate SL	The recipient of the **REQ** message does not support the requested Alternate SL	Acceptable SL.
21	Invalid Alternate Traffic Class	The recipient of the **REQ** message does not support the requested Alternate Traffic Class	Acceptable Traffic Class

22	Invalid Alternate Hop Limit	The recipient of the **REQ** message could not (or would not) accept the Altermate Hop Limit	Acceptable Hop Limit
23	Invalid Alternate Packet Rate	The recipient of the **REQ** message could not adjust its trans-mitter to send as slowly as would be required to comply with the requested Alternate Packet Rate	Minimum acceptable Packet Rate
24	Port and CM Redirection	The recipient of the **REQ** message supports the requested Ser-vice ID, but at the endpoint specified by the ARI. Further CM messages should be sent to that endpoint as well.	A ClassPortInfo data struc-ture as documented in Sec-tion 13.4.8.1 describing where to send subsequent CM messages, and also describing the GID of the port to propose in the new **RE**Q.
25	Port Redirection	The recipient of the **REQ** message supports the requested Ser-vice ID, but at the port specified by the ARI. Further CM mes-sages shall be sent to the port to which the original **REQ** was sent.	GID of port to propose in new **RE**Q.
26	Invalid Path MTU	The recipient of the **REQ** message cannot support the maximum packet payload size specified	Maximum acceptable maximum packet payload size
27	Insufficient Responder Resources	The value of Responder Resources (for RDMA Read/Atomics) in the **REP** message was insufficient.	
28	Consumer Reject	The consumer decided to reject the communication or EE context setup establishment attempt for reasons other than those listed above. (Typically this happens based upon information being conveyed in the PrivateData field of a message.)	Defined by the consumer
29	RNR Retry Count Reject	The recipient of the message rejects the RNR NAK Retry count value.	

4.10 CsMraStruct

Table 15- CsMraStruct field source information

Field	Source	Notes
ServiceTimeout	ULP	The time, expressed as (4.096 µS*2 Service Timeout) that the recipient shall wait for a REP, RTU, APR, or REJ (as appropriate) from the sender after receiving this message before timing out. (See section 12.9.8.5) This quantity should be treated as an unsigned number; negative exponents in the calculation above are not allowed.
Reserved	ULP	
PrivateData	ULP	Application specific private data. See CsConnectRequestStruct for details.