# CCI

0.1

Generated by Doxygen 1.5.6

Fri Jun 3 11:45:02 2011

# Contents

# Chapter 1

# CCI: The Common Communication Interface

## 1.1 Introduction

Over the years, many networking application programming interfaces (APIs) have be developed. The most widely used is the BSD Sockets interface due to its implementation on nearly all hardware. Designed to provide an interface for TCP, the Sockets interface does not allow applications to take advantage of newer hardware and the features that they provide. These features include remote direct memory access (RDMA), operating system (OS) bypass, "zero-copy" support, one-sided operations, and asynchronous operations.

Many different APIs evolved to expose these features such as the Virtual Interface Architecture (VIA), OpenFabrics Verbs, Myrinet Express (MX), and Portals. None have had the widespread adoption that Sockets has had. Application developers are therefore forced to make substantial tradeoffs in the selection of a user-level network interface for their network-based applications. While the use of BSD Sockets guarantees portability across nearly every type of existing network, the emulation of the Sockets API over an underlying network-native software API can substantially limit both performance and scalability. On the other hand, the use of a native networking API may satisfy performance and scalability requirements, but limit the application's portability to future platforms.

CCI balances the needs of portability and simplicity while preserving the performance capabilities of advanced networking technologies. In designing CCI, we have drawn upon prior research with a variety of low-level networking interfaces as well as our experience in working directly with application developers in the use of these APIs. Whenever possible, we adhered to our primary goal of simplicity in order to foster wide-spread adoption, yet preserving both performance and portability.

## 1.2 Design Goals

In setting out to design a new communication's interface, we had several goals in mind: portability, simplicity, performance, scalability, and robustness.

### 1.2.1 Portability

Application and middleware developers do not have the resources to continuously port their code on different communication interfaces. Selecting a vendor-specific API introduces lock-in and reduces future migration options. At the same time, vendors do not have the resources to properly support a large set of middleware. BSD Sockets and MPI both provide this high-level of portability. For any new communication interface to gain acceptance in the broader community, it needs to provide a similar breadth of implementations on currently available hardware, by supporting the semantics that are common to most vendor APIs.

### 1.2.2 Simplicity

Simplicity is paramount to the success of a programming interface. Critical mass cannot be reached by limiting the targeted audience to a few networking experts. However, ease of use involves many elements beyond just expertise. Code size is a common, albeit subjective, metric used to compare programming interfaces. The rationale is that larger codes are harder to debug and maintain. For example, an analysis of the Open MPI version 1.4.3 implementation shows substantial differences between the seven supported communication APIs (excluding self and shared memory). The total lines of code of each Byte Transfer Layer (BTL) for various APIs include:

- Elan 1,656
- MX 2,333
- Portals 2,469
- GM 2,779
- Sockets (TCP) 4,192
- UDAPL 6,208
- OpenIB (Verbs) 21,574

The Verbs BTL is the largest, five times the size of the TCP sockets BTL, third largest, and 8 to 13 times larger than the BTLs of the vendor interfaces. Another indicator of complexity is the number of functions available. Choice is good but too much choice is worse. Fortunately, software programmers are efficient at reducing overly complex interfaces to a minimum set of useful semantics. For example, MPI specifies over 300 functions but the vast majority of MPI applications only use a fraction of them.

Similarly, relative simplicity was the main drive behind the wide adoption of the BSD Socket interface. A communication interface should aspire to find the right balance between richness of semantics and ease of use.

### 1.2.3 Performance

Performance is major drive for innovation in networking, from HPC to Cloud Computing. All modern network technologies leverage common techniques developed in the last two decades: OS-bypass, zero-copy, one-sided, and asynchronous operations. To deliver the best performance, a communication interface should present semantics that can efficiently leverage all these techniques as provided by modern high-speed networks.

### 1.2.4 Scalabity

Projections for leadership scale systems in HPC include hundreds of thousands of nodes and millions of cores. In the commercial space, Cloud Computing data centers are fast approaching these levels. In this context, scalability is an important requirement. The time and space overhead of a scalable communication interface should not grow linearly with the number of communicating partners. BSD Sockets are inefficient in both dimensions, as buffers and file handles are allocated for each new socket. Through adaptive socket buffering and use of epoll(), Sockets implementations have so far managed to reasonably handle large number of connections. MPI is inherently more scalable and it has successfully been deployed on large HPC machines. However, it is not clear if MPI in its present form can efficiently scale to millions of cores. Scalability of the Verbs interface was originally quite poor due to its Queue Pair model. MPI implementations used various techniques such as connection on demand and dynamic buffer management to work around the QPs memory footprint problem. Scalability was further improved with the addition of Shared Receive Queues (SRQ), but distinct QPs are still required on the send side. To address the Cloud Computing and leadership class HPC requirements, a communication interface should aim for constant buffer and polling overhead, independently of the number of nodes in the fabric.

### 1.2.5 Robustness

Hardware and software failures occur frequently, often proportional with the size of the system. As system sizes continue to increase, ignoring such failures will no longer be an option. Most MPI implementations currently abort on failures that an application might otherwise tolerate. To address this, there have been several efforts aimed at designing fault-tolerant MPI libraries and adding fault recovery to the MPI specification.

Thus far these efforts have had limited success. The loose semantic about status completions was actually a benefit in making MPI a simpler interface, developers would send messages and trust MPI to always deliver them. Unfortunately, real-world applications eventually had to implement checkpoint/restart functionality to tolerate system faults and it is the only practical solution available today on large HPC systems today. Both Sockets and Verbs fare better than MPI on this issue. They use connections to represent the state of communication channels without reliance on a single consistent distributed process space (MPI_COMM_WORLD). Connections provide a simplified model for robustness; they contain faults and allow for their recovery by resetting the state of the affected communication channels. Unfortunately, both Sockets and Verbs associate buffers to a connection, which negatively affects scalability. A robust and scalable communication interface should provide connection-oriented semantics without per-connection resources.

Communication reliability is often seen as a way to improve overall robustness. For some applications such as Media Content Delivery (IPTV), Financial Trading (HFT) or system-health monitoring, the provided reliability may be incompatible with their timing requirements. Furthermore, the most scalable multicast implementations are unreliable. For these reasons, a large share of applications use unreliable connec- tions. A communication interface should provide different levels of connection reliability, as well as support for multicast.

## 1.3 The CCI Interface

In this section, we provide a brief overview of the CCI API to allow us to discuss how CCI can meet the goals outlined above.

### 1.3.1 Initialization

Before calling any function, the application must call cci_init(). The application may call cci_init() multiple times with different parameters. The application can then call cci_get_devices() to obtain an array of available devices. The devices are parsed from a config file and each device has a name, an array keyword/value strings, a maximum send size in bytes, and PCI information if needed. Each device's maximum send size is equivalent to the network MTU (less wire headers). When no more communication is needed, the application calls cci_free_devices().

### 1.3.2 Communication Endpoints

All communication in CCI revolves around an endpoint. Each endpoint has some number of device-sized buffers available for sending and receiving messages of small, unexpected messages. The application calls cci_create_endpoint() and cci_destroy_-endpoint(), respectively, to obtain or release an endpoint. The application may alter the number of send and/or receive buffers using cci_get_opt() and cci_set_opt().

### 1.3.3 Event Handling

CCI is inherently asynchronous and all communication functions only initiate communication. When a communication completes, it generates an event. There are three event types: CCI_EVENT_SEND, CCI_EVENT_RECV, and CCI_EVENT_-OTHER. The CCI_EVENT_OTHER event returns connection success, rejection or timeout events as well as endpoint and/or device failure events.

An application can poll for an event with cci_get_event(), which returns an event structure of which the contents vary depending on the event's type. When a process is finished with an event, it uses cci_return_event() to release it resources, if any, back to CCI.

In addition to returning an endpoint, cci_create_endpoint() also returns an operating system-specific handle that can be passed to `select()` or other OS functions to allow blocking until an event is available.

### 1.3.4 Connections

CCI defines a connection struct which includes the maximum send size negotiated by the two instances of CCI, a pointer to the owning endpoint, and the connection attribute.

As mentioned above, some applications may need reliable delivery while other may not. Among applications needing reliable delivery, some may need in-order completion (e.g. traditional SOCK_STREAM semantics) and others may accept out-of-order completion as long as communications are initiated in-order (e.g. MPI point-to-point).

In order to provide applications with the level of service appropriate for their needs, CCI provides multiple types of connection attributes:

- Reliable with Ordered completion (RO)
- Reliable with Unordered completion (RU)
- Unreliable with Unordered completion (UU)
- Unreliable with Unordered completion with multicast send (UU_MC_TX)
- Unreliable with Unordered completion with multicast receive (UU_MC_RX)

If a process needs a mix of types, it is allowed to open multiple connections to the other process.

### 1.3.5 Connection Establishment

CCI mirrors the client/server connection semantics of Sockets. A process willing to accept connections will first cci_bind() a device to a name service at a specified port with a backlog parameter. The call returns a pointer to a service. When a server no longer wishes to receive connection requests, it can cci_unbind() from the service.

To initiate a connection, the client calls cci_connect() with parameters including an endpoint, a string URI for the server, the port, optionally a pointer to a limited sized payload and its length, the connection attribute, a pointer to an optional application context, and a timeout.

The server then polls for connection requests using cci_get_conn_request() passing in the service pointer. If one is ready, it returns a conn_req struct which contains an array of compatible devices, the number of devices in the array, a pointer to the application payload and its length if the client sent it, and the requested connection attribute.

The server then calls either cci_accept() or cci_reject(). The cci_accept() call binds the connection request to an endpoint previously created from one of the compatible devices and returns a connection pointer. The client gets an CCI_EVENT_OTHER event with the type CONNECT_SUCCESS. If the server calls cci_reject(), the client get an other event with the type CONNECT_REJECTED. On the server, the connection request is stale after either call. If the server does not reply within the timeout set in the client's cci_connect(), the client gets an CCI_EVENT_OTHER event with a type of CONNECT_TIMEOUT. When a process no longer needs a connection, it can call cci_disconnect().

### 1.3.6 Active Messages

Once the connection is established, the two processes can start communicating. CCI provides two methods, active messages and remote memory access (RMA), which we discuss in the RMA section.

CCI's version of active messages does not fully mirror Active Messages (AM). Like the original AM, CCI's active messages have a maximum size that is device dependent. Ideally, the size is equal to the link MTU (less wire headers). The driving idea to limiting the message size to a single MTU is that future networks may have many paths through the network due to fabrics with high-radix switches and/or NICs with multiple ports connected to redundant switches for fault-tolerance. Limiting the active message size limited to a single MTU vastly simplifies the requirements for message completion — either it arrives or it does not.

Where CCI differs from the original Active Messages is handling of incoming messages. In Active Messages, the message contains an address of the handler that will process it, which assumes all processes have identical memory spaces. The difficulty with invoking handlers is there is no bound on how long the handler will run. While running, the communication library cannot process any more messages and could lead to dropping messages. Instead, CCI returns an event of type CCI_EVENT_RECV. The application can get the event and hold it without blocking CCI from continuing to service other communications.

The cci_send() parameters include the connection, header and data pointers and their respective lengths, an application context pointer, and flags. Either or both of the pointers may be NULL. The header is currently limited to a maximum length of 32 bytes. The context pointer is returned in the CCI_EVENT_SEND completion event and can be used to allow the application to retrieve its internal state.

The optional flags parameter can accept the following:

- CCI_FLAG_BLOCKING which means that the send should not return until the send completes. The send completion status is passed in the function's return value.

- CCI_FLAG_NO_COPY is a hint to CCI that the application does not need the buffer back until the send completes and is free to use zero-copy methods if supported.

- CCI_FLAG_SILENT indicates that the process does not want a completion event for this send.

On the receiver, a call to cci_get_event() returns a CCI_EVENT_RECV event which includes pointers to the header and data, their lengths, and a pointer to the connection. The receiving process can choose to simply inspect the data in-place, modify the data in-place and send it to another process, or copy it out if it needs to keep the data long-term. When the process no longer needs the buffer, it releases it back to CCI with cci_return_event(). It should be noted that if the application does not process CCI_-EVENT_RECV events and return them to CCI fast enough, that CCI may still need to drop incoming messages.

CCI also provides cci_sendv() that takes an array of data pointers and an array of lengths instead of the just the one data pointer and length in cci_send(). Lastly, CCI does not require memory registration for sending or receiving active messages.

### 1.3.7 Remote Memory Access

Clearly, messages limited to a single MTU will not meet the needs of all applications. Applications such as file systems which need to move large, bulk messages need much more. To accommodate them, CCI also provides remote memory access (RMA). RMA transfers are only allowed on reliable connections.

Before using RMA, the process needs to explicitly register the memory. CCI provides cci_rma_register() which takes pointers to the endpoint, the connection, and the start of the region to be registered as well as the length of the region and it returns a RMA handle. If the connection pointer is set, RMA operations on that handle will be limited to that one connection. If the connection is NULL, then RMA operations on that handle will be limited to any connection on that endpoint. When a process no longer needs to RMA in to or out of the region, it passes the handle to cci_rma_deregister().

For a RMA transfer to take place, both processes must register their local memory and they need to pass the handle of the target process to the initiator process using one or more active messages.

The cci_rma() call takes the connection pointer, an optional header pointer and length, the local RMA handle and offset, the remote RMA handle and offset, the transfer length, an application context pointer, and a set of flags.

If the header pointer and length are set, the initiator will send a completion message to the target that arrives as an active message with the header set and no data payload. Like with cci_send(), the header length is limited to 32 bytes.

The flag options include:

- CCI_FLAG_BLOCKING (see cci_send())
- CCI_FLAG_SILENT (see cci_send())
- CCI_FLAG_READ allows data to move from remote to local memory.
- CCI_FLAG_WRITE allows data to move from local to remote memory.
- CCI_FLAG_FENCE ensures that all previous RMA operations to complete remotely before this operation and all following RMA operations.

CCI does not guarantee delivery order within an operation (i.e. no last-byte-written-last mandate), but order is guaranteed between data delivery and the remote receive event if the header is specified.

# Chapter 2

# Module Index

## 2.1 Modules

Here is a list of all modules:

# Chapter 3

# Data Structure Index

## 3.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 4

# Module Documentation

## 4.1 Initialization / Environment

### Defines

- #define CCI_ABI_VERSION 1

  *This constant is passed in via the cci_init() function and is used for internal consistency checks.*

### Typedefs

- typedef enum cci_status cci_status_t

  *Status codes that are returned from CCI functions.*

### Enumerations

- enum cci_status {

  CCI_SUCCESS = 0, CCI_ERROR, CCI_ERR_DISCONNECTED, CCI_ERR_RNR,

  CCI_ERR_DEVICE_DEAD, CCI_ERR_RMA_HANDLE, CCI_ERR_RMA_OP, CCI_ERR_NOT_IMPLEMENTED,

  CCI_ERR_NOT_FOUND, CCI_EINVAL = EINVAL, CCI_ETIMEDOUT = ETIMEDOUT, CCI_ENOMEM = ENOMEM,

  CCI_ENODEV = ENODEV, CCI_EBUSY = EBUSY, CCI_ERANGE = ERANGE, CCI_EAGAIN = EAGAIN,

CCI_ENOBUFS = ENOBUFS, CCI_EMSGSIZE = EMSGSIZE, CCI_-ENOMSG = ENOMSG, CCI_EADDRNOTAVAIL = EADDRNOTAVAIL }

*Status codes that are returned from CCI functions.*

## Functions

- CCI_DECLSPEC int cci_init (uint32_t abi_ver, uint32_t flags, uint32_t ∗caps)

  *This is the first CCI function that must called; no other CCI functions can be invoked before this function returns successfully.*

- CCI_DECLSPEC const char ∗ cci_strerror (enum cci_status status)

  *Returns a string corresponding to a CCI status enum.*

### 4.1.1 Define Documentation

#### 4.1.1.1 #define CCI_ABI_VERSION 1

This constant is passed in via the cci_init() function and is used for internal consistency checks.

**Examples:**

client.c, devices.c, init.c, and server.c.

### 4.1.2 Typedef Documentation

#### 4.1.2.1 typedef enum cci_status cci_status_t

Status codes that are returned from CCI functions.

Note that status code names that are derived from <errno.h> generally follow the same naming convention (e.g., EINVAL -> CCI_EINVAL). Error status codes that are unique to CCI are of the form CCI_ERR_<foo>.

IF YOU ADD TO THESE ENUM CODES, ALSO EXTEND src/api/strerror.c!!

### 4.1.3 Enumeration Type Documentation

#### 4.1.3.1 enum cci_status

Status codes that are returned from CCI functions.

Note that status code names that are derived from <errno.h> generally follow the same naming convention (e.g., EINVAL -> CCI_EINVAL). Error status codes that are unique to CCI are of the form CCI_ERR_<foo>.

IF YOU ADD TO THESE ENUM CODES, ALSO EXTEND src/api/strerror.c!!

**Enumerator:**

***CCI_SUCCESS*** Returned from most functions when they succeed.

***CCI_ERROR*** Generic error.

***CCI_ERR_DISCONNECTED*** For both reliable and unreliable sends, this error code means that cci_disconnect() has been invoked on the send side (in which case this is an application error), or the receiver replied that the receiver invoked cci_disconnect().

***CCI_ERR_RNR*** For a reliable send, this error code means that a receiver is reachable, the connection is connected but the receiver could not receive the incoming message during the timeout period.

If a receiver cannot receive an incoming message for transient reasons (most likely out of resources), it returns an Receiver-Not-Ready NACK and drops the message. The sender keeps retrying to send the message until the timeout expires,

If the timeout expires and the last control message received from the receiver was an RNR NACK, then this message is completed with the RNR status. If the connection is both reliable and ordered, then all successive sends are also completed in the order in which they were issued with the RNR status.

This error code will not be returned for unreliable sends.

***CCI_ERR_DEVICE_DEAD*** The local device is gone, not coming back.

***CCI_ERR_RMA_HANDLE*** Error returned from remote peer indicating that the address was either invalid or unable to be used for access / permissions reasons.

***CCI_ERR_RMA_OP*** Error returned from remote peer indicating that it does not support the operation that was requested.

***CCI_ERR_NOT_IMPLEMENTED*** Not yet implemented.

***CCI_ERR_NOT_FOUND*** Not found.

***CCI_EINVAL*** Invalid parameter passed to CCI function call.

***CCI_ETIMEDOUT*** For a reliable send, this error code means that the sender did not get anything back from the receiver within a timeout (no ACK, no NACK, etc.

). It is unknown whether the receiver actually received the message or not.

This error code won't occur for unreliable sends.

***CCI_ENOMEM*** No more memory.

***CCI_ENODEV*** No device available.

***CCI_EBUSY*** Resource busy (e.g.
port in use)

***CCI_ERANGE*** Value out of range (e.g.
no port available)

***CCI_EAGAIN*** Resource temporarily unavailable.

***CCI_ENOBUFS*** The output queue for a network interface is full.

***CCI_EMSGSIZE*** Message too long.

***CCI_ENOMSG*** No message of desired type.

***CCI_EADDRNOTAVAIL*** Address not available.

## 4.1.4 Function Documentation

### 4.1.4.1 CCI_DECLSPEC int cci_init (uint32_t *abi_ver*, uint32_t *flags*, uint32_t ∗ *caps*)

This is the first CCI function that must called; no other CCI functions can be invoked before this function returns successfully.

**Parameters:**

← ***abi_ver,:*** A constant describing the ABI version that this application requires (one of the CCI_ABI_∗ values).

← ***flags,:*** A constant describing behaviors that this application requires. Currently, 0 is the only valid value.

→ ***caps,:*** Capabilities of the underlying library: THREAD_SAFETY

**Returns:**

CCI_SUCCESS CCI is available for use.
CCI_EINVAL Caps is NULL or incorrect ABI version.
CCI_ENOMEM Not enough memory to complete.
CCI_ERR_NOT_FOUND No driver or CCI_CONFIG.
CCI_ERROR Unable to parse CCI_CONFIG.
Errno if fopen() fails.
Each driver may have additional error codes.

If cci_init() completes successfully, then CCI is loaded and available to be used in this application. There is no corresponding "finalize" call.

If cci_init() fails, an appropriate error code is returned.

If cci_init() is invoked again with the same parameters after it has already returned successfully, it's a no-op. If invoked again with different parameters, if the CCI implementation can change its behavior to ∗also∗ accommodate the new behaviors indicated by the new parameter values, it can return successfully. Otherwise, it can return a failure and continue as if cci_init() had not been invoked again.

**Examples:**

client.c, devices.c, init.c, and server.c.

#### 4.1.4.2 CCI_DECLSPEC const char* cci_strerror (enum cci_status *status*)

Returns a string corresponding to a CCI status enum.

**Parameters:**

← ***status,:*** A CCI status enum.

**Returns:**

A string when the status is valid.
NULL if not valid.

**Examples:**

client.c, devices.c, init.c, and server.c.

## 4.2 Devices

### Data Structures

- struct cci_device

  *Structure representing one CCI device.*

### Typedefs

- typedef struct cci_device cci_device_t

  *Structure representing one CCI device.*

### Functions

- CCI_DECLSPEC int cci_get_devices (cci_device_t const ∗∗∗const devices)

  *Get an array of devices.*

- CCI_DECLSPEC int cci_free_devices (cci_device_t const ∗∗devices)

  *Frees a NULL-terminated array of (cci_device_t∗)'s that were previously allocated via cci_get_devices().*

### 4.2.1 Typedef Documentation

#### 4.2.1.1 typedef struct cci_device cci_device_t

Structure representing one CCI device.

### 4.2.2 Devices

Device types and functions.

Before launching into detail, let's first describe the CCI system configuration file. On POSIX systems, it is likely a simple INI-style text file; on Windows systems, it may be registry entries. The key thing is to support trivial namespaces and key=value pairs.

Here is an example text config file:

```
# Comments are anything after the # symbols.

# Sections in this file are denoted by [section name].  Each section
```

```
# denotes a single CCI device.

[bob0]
# The only mandated field in each section is "driver".  It indicates
# which CCI driver should be applied to this device.
driver = psm

# The priority field determines the ordering of devices returned by
# cci_get_devices().  100 is the highest priority; 0 is the lowest priority.
# If not specified, the priority value is 50.
priority = 10

# The last field understood by the CCI core is the "default" field.
# Only one device is allowed to have a "true" value for default.  All
# others must be set to 0 (or unset, which is assumed to be 0).  If
# one device is marked as the default, then this device will be used
# when NULL is passed as the device when creating an endpoint.  If no
# device is marked as the default, it is undefined as to which device
# will be used when NULL is passed as the device when creating an
# endpoint.
default = 1

# All other fields are uninterpreted by the CCI core; they're just
# passed to the driver.  The driver can do whatever it wants with
# these values (e.g., system admins can set values to configure the
# driver).  Driver documentation should specify what parameters are
# available, what each parameter is/does, and what its legal values
# are.

# This example shows a bonded PSM device that uses both the ipath0 and
# ipath1 devices.  Some other parameters are also passed to the PSM
# driver; it assumedly knows how to handle them.

device = ipath0,ipath1
capabilities = bonded,failover,age_of_captain:52
qos_stuff = fast

# bob2 is another PSM device, but it only uses the ipath0 device.
[bob2]
driver = psm
device = ipath0

# bob3 is another PSM device, but it only uses the ipath1 device.
[bob3]
driver = psm
device = ipath1
sl = 3 # IB service level (if applicable)

# storage is a device that uses the UDP driver.  Note that this driver
# allows specifying which device to use by specifying its IP address
# and MAC address -- assumedly it's an error if there is no single
# device that matches both the specified IP address and MAC
# (vs. specifying a specific device name).
[storage]
driver = udp
priority = 5
ip = 172.31.194.1
```

```
mac = 01:12:23:34:45
```

The config file forms the basis for the device discussion, below.

A CCI device is a [section] from the config file, above.

## 4.2.3 Function Documentation

### 4.2.3.1 CCI_DECLSPEC int cci_get_devices (cci_device_t const ***const *devices*)

Get an array of devices.

Returns a NULL-terminated array of (struct cci_device *)'s that are "up". The pointers can be copied, but the actual cci_device instances may not. The array of devices is allocated by the CCI library; there may be hidden state that the application does not see.

**Parameters:**

→ ***devices*** Array of pointers to be filled by the function. Previous value in the pointer will be overwritten.

**Returns:**

CCI_SUCCESS The array of "up" devices is available.
CCI_EINVAL Devices is NULL.
Each driver may have additional error codes.

If cci_get_devices() succeeds, the entire returned set of data (to include the data pointed to by the individual cci_device instances) should be treated as const, and must be freed with a corresponding call to cci_free_devices().

The order of devices returned corresponds to the priority fields in the devices. If two devices share the same priority, their ordering in the return array is arbitrary.

If cci_get_devices() fails, the value returned in devices is undefined.

**Examples:**

client.c, devices.c, and server.c.

### 4.2.3.2 CCI_DECLSPEC int cci_free_devices (cci_device_t const ** *devices*)

Frees a NULL-terminated array of (cci_device_t*)'s that were previously allocated via cci_get_devices().

**Parameters:**

← ***devices,:*** array of pointers previously filled in via cci_get_devices().

**Returns:**

CCI_SUCCESS All CCI resources have been released.
CCI_EINVAL Devices is NULL.
Each driver may have additional error codes.

If cci_free_devices() succeeds, the data pointed to by the devices pointer will be stale (and should not be accessed).

If cci_free_devices() fails, the state of the data pointed to by the devices parameter is undefined.

**Examples:**

client.c, devices.c, and server.c.

# 4.3 Endpoints

## Data Structures

- struct cci_endpoint

  *Endpoint.*

## Typedefs

- typedef enum cci_endpoint_flags cci_endpoint_flags_t

  *And endpoint is a set of resources associated with a single NUMA locality.*

- typedef struct cci_endpoint cci_endpoint_t

  *Endpoint.*

- typedef int cci_os_handle_t

  *OS-native handles.*

## Enumerations

- enum cci_endpoint_flags { bogus_must_have_something_here }

  *And endpoint is a set of resources associated with a single NUMA locality.*

## Functions

- CCI_DECLSPEC int cci_create_endpoint (cci_device_t ∗device, int flags, cci_endpoint_t ∗∗endpoint, cci_os_handle_t ∗fd)

  *Create an endpoint.*

- CCI_DECLSPEC int cci_destroy_endpoint (cci_endpoint_t ∗endpoint)

  *Destroy an endpoint.*

### 4.3.1 Typedef Documentation

#### 4.3.1.1 typedef enum cci_endpoint_flags cci_endpoint_flags_t

And endpoint is a set of resources associated with a single NUMA locality.

Buffers should be pinned by the CCI implementation to the NUMA locality where the thread is located who calls create_endpoint().

Advice to users: bind a thread to a locality before calling create_endpoint().

Sidenote: if we want to someday make endpoints span multiple NUMA localities, we can add a function to say "add this locality (or thread?) to this endpoint.

Endpoints are "thread safe" by default... Meaning multiple threads can call functions on endpoints simultaneously and it's "safe". No guarantees are made about serialization or concurrency.

A set of flags that describe how the endpoint should be created.

#### 4.3.1.2 typedef struct cci_endpoint cci_endpoint_t

Endpoint.

#### 4.3.1.3 typedef int cci_os_handle_t

OS-native handles.

**Examples:**

client.c, and server.c.

### 4.3.2 Enumeration Type Documentation

#### 4.3.2.1 enum cci_endpoint_flags

And endpoint is a set of resources associated with a single NUMA locality.

Buffers should be pinned by the CCI implementation to the NUMA locality where the thread is located who calls create_endpoint().

Advice to users: bind a thread to a locality before calling create_endpoint().

Sidenote: if we want to someday make endpoints span multiple NUMA localities, we can add a function to say "add this locality (or thread?) to this endpoint.

Endpoints are "thread safe" by default... Meaning multiple threads can call functions on endpoints simultaneously and it's "safe". No guarantees are made about serialization or concurrency.

A set of flags that describe how the endpoint should be created.

**Enumerator:**

***bogus_must_have_something_here*** For future expansion.

### 4.3.3 Function Documentation

#### 4.3.3.1 CCI_DECLSPEC int cci_create_endpoint (cci_device_t ∗ *device*, int *flags*, cci_endpoint_t ∗∗ *endpoint*, cci_os_handle_t ∗ *fd*)

Create an endpoint.

**Parameters:**

← ***device,:*** A pointer to a device that was returned via cci_get_devices() or NULL.

← ***flags,:*** Flags specifying behavior of this endpoint.

→ ***endpoint,:*** A handle to the endpoint that was created.

→ ***fd,:*** Operating system handle that can be used to block for progress on this endpoint.

**Returns:**

CCI_SUCCESS The endpoint is ready for use.
CCI_EINVAL Endpoint or fd is NULL.
CCI_ENODEV Device is not "up".
CCI_ENOMEM Unable to allocate enough memory.
Each driver may have additional error codes.

This function creates a CCI endpoint. A CCI endpoint represents a collection of local resources (such as buffers and a completion queue). An endpoint is associated with a device that performs the actual communication (see the description of cci_get_devices(), above).

The device argument can be a pointer that was returned by cci_get_devices() to indicate that a specific device should be used for this endpoint, or NULL, indicating that the system default device should be used.

If successful, cci_create_endpoint() creates an endpoint and returns a pointer to it in the endpoint parameter.

cci_create_endpoint() is a local operation (i.e., it occurs on local hardware). There is no need to talk to name services, etc. To be clear, the intent is that this function can be invoked many times locally without affecting any remote resources.

If it is desirable to bind the CCI endpoint to a specific set of resources (e.g., a NUMA node), you should bind the calling thread before calling cci_create_endpoint().

Advice to users: if you want to set the send/receive buffer count on the endpoint, call cci_set|get_opt() after creating the endpoint.

**Examples:**

client.c, and server.c.

#### 4.3.3.2 CCI_DECLSPEC int cci_destroy_endpoint (cci_endpoint_t * *endpoint*)

Destroy an endpoint.

**Parameters:**

← ***endpoint,:*** Handle previously returned from a successful call to cci_create_-endpoint().

**Returns:**

CCI_SUCCESS The endpoint's resources have been released.
CCI_EINVAL Endpoint is NULL.
Each driver may have additional error codes.

Successful completion of this function makes all data structures and state associated with the endpoint stale. All open connections are closed immediately – it is exactly as if cci_disconnect() was invoked on every open connection on this endpoint.

**Examples:**

client.c, and server.c.

## 4.4 Connections

### Data Structures

- struct cci_conn_req

  *Connection request.*

- struct cci_connection

  *Connection handle.*

### Defines

- #define CCI_CONN_REQ_LEN (1024)

  *This constant is the maximum value of data_len passed to cci_connect().*

### Typedefs

- typedef enum cci_conn_attribute cci_conn_attribute_t

  *Connection request attributes.*

- typedef struct cci_conn_req cci_conn_req_t

  *Connection request.*

- typedef struct cci_connection cci_connection_t

  *Connection handle.*

### Enumerations

- enum cci_conn_attribute {

  CCI_CONN_ATTR_RO, CCI_CONN_ATTR_RU, CCI_CONN_ATTR_UU, CCI_CONN_ATTR_UU_MC_TX,

  CCI_CONN_ATTR_UU_MC_RX }

  *Connection request attributes.*

## Functions

- CCI_DECLSPEC int cci_bind (cci_device_t *device, int backlog, uint32_-t *port, cci_service_t **service, cci_os_handle_t *fd)

  *Bind a service to the connection manager using specific service port.*

- CCI_DECLSPEC int cci_unbind (cci_service_t *service, cci_device_t *device)

  *Unbind a previously-bound service.*

- CCI_DECLSPEC int cci_get_conn_req (cci_service_t *service, cci_conn_req_t **conn_req)

  *Return the next connection request, if any.*

- CCI_DECLSPEC int cci_accept (cci_conn_req_t *conn_req, cci_endpoint_-t *endpoint, cci_connection_t **connection)

  *Accept a connection request and establish a connection with a specific endpoint.*

- CCI_DECLSPEC int cci_reject (cci_conn_req_t *conn_req)

  *Reject a connection request.*

- CCI_DECLSPEC int cci_connect (cci_endpoint_t *endpoint, char *server_uri, uint32_t port, void *data_ptr, uint32_t data_len, cci_conn_attribute_t attribute, void *context, int flags, struct timeval *timeout)

  *Initiate a connection request (client side).*

- CCI_DECLSPEC int cci_disconnect (cci_connection_t *connection)

  *Tear down an existing connection.*

### 4.4.1 Define Documentation

#### 4.4.1.1 #define CCI_CONN_REQ_LEN (1024)

This constant is the maximum value of data_len passed to cci_connect().

### 4.4.2 Typedef Documentation

#### 4.4.2.1 typedef enum cci_conn_attribute cci_conn_attribute_t

Connection request attributes.

Reliable connections deliver messages once. If the packet cannot be delivered after a specific amount of time, the connection is broken; there is no guarantee regarding which messages have been received successfully before the connection was broken.

Connections can be ordered or unordered, but note that ordered unreliable connections are forbidden. Also, note that ordering of RMA operations only applies to target notification, not data delivery.

Unreliable unordered connections have no timeout.

Multicast is always unreliable unordered. Multicast connections are always unidirectional, send ∗or∗ receive. If an endpoint wants to join a multicast group to both send and receive, it needs to establish two distinct connections, one for sending and one for receiving.

#### 4.4.2.2 typedef struct cci_conn_req cci_conn_req_t

Connection request.

#### 4.4.2.3 typedef struct cci_connection cci_connection_t

Connection handle.

### 4.4.3 Enumeration Type Documentation

#### 4.4.3.1 enum cci_conn_attribute

Connection request attributes.

Reliable connections deliver messages once. If the packet cannot be delivered after a specific amount of time, the connection is broken; there is no guarantee regarding which messages have been received successfully before the connection was broken.

Connections can be ordered or unordered, but note that ordered unreliable connections are forbidden. Also, note that ordering of RMA operations only applies to target notification, not data delivery.

Unreliable unordered connections have no timeout.

Multicast is always unreliable unordered. Multicast connections are always unidirectional, send ∗or∗ receive. If an endpoint wants to join a multicast group to both send and receive, it needs to establish two distinct connections, one for sending and one for receiving.

**Enumerator:**

***CCI_CONN_ATTR_RO*** Reliable ordered.

Means that both completions and delivery are in the same order that they were issued.

***CCI_CONN_ATTR_RU*** Reliable unordered.

Means that delivery is guaranteed, but both delivery and completion may be in a different order than they were issued.

***CCI_CONN_ATTR_UU*** Unreliable unordered (RMA forbidden).

Delivery is not guaranteed, and both delivery and completions may be in a different order than they were issued.

***CCI_CONN_ATTR_UU_MC_TX*** Multicast send (RMA forbidden).

***CCI_CONN_ATTR_UU_MC_RX*** Multicast recv (RMA forbidden).

## 4.4.4 Function Documentation

### 4.4.4.1 CCI_DECLSPEC int cci_bind (cci_device_t * *device*, int *backlog*, uint32_t * *port*, cci_service_t ** *service*, cci_os_handle_t * *fd*)

Bind a service to the connection manager using specific service port.

It returns a service handle and an OS-specific handle that can be used for blocking (e.g., via POSIX poll(), select(), or Windows' WaitOnMultipleObjects(), or other OS-specific methods).

If a specific service port is not required, passing "0" will allocate an unused one. If the requested service port is already used by another application, an error is returned. The lowest 4096 (?) ports are reserved for privileged processes.

**Parameters:**

← ***device*** Device to bind to, can be NULL.

← ***backlog*** Incoming connection requests queue depth.

↔ ***port*** Port number used by client to identify the service accepting connection requests.

→ ***service*** Handle representing the service accepting connection requests through the connection manager.

→ ***fd*** OS-specific file descriptor/handle to block on incoming connection requests.

**Returns:**

CCI_SUCCESS Service successfully bound on that device.
CCI_EINVAL Port, service, or fd is NULL.
CCI_EINVAL Backlog is zero.
CCI_ENODEV Device is NULL and no default device found.
CCI_ENODEV Device is not "up".
CCI_ENOMEM Unable to allocate enough memory.

CCI_EBUSY The service port is already bound on that device.
Each driver may have additional error codes.

If you use the same service port, you get the same service, even for different devices. The connection request will contain all the devices that are compatible for the connection.

**Examples:**

server.c.

#### 4.4.4.2 CCI_DECLSPEC int cci_unbind (cci_service_t ∗ *service*, cci_device_t ∗ *device*)

Unbind a previously-bound service.

**Parameters:**

← ***service*** Service that was previously returned from cci_bind().

← ***device*** Specific device to unbind from the service. If 0, unbinds all devices bound to that service.

**Returns:**

CCI_SUCCESS Device has been unbound from the service.
CCI_EINVAL Service or device is NULL.
CCI_ENODEV Device is not bound on the service.
Each driver may have additional error codes.

The service could become stale if there is no more device bound to that service. This does not affect established connections.

**Examples:**

server.c.

#### 4.4.4.3 CCI_DECLSPEC int cci_get_conn_req (cci_service_t ∗ *service*, cci_conn_req_t ∗∗ *conn_req*)

Return the next connection request, if any.

**Parameters:**

← ***service*** Service to check for incoming requests.

→ ***conn_req*** New connection request.

**Returns:**

CCI_SUCCESS A new connection request is available.
CCI_EINVAL Service or conn_req is NULL.
CCI_EAGAIN No connection request was ready.
Each driver may have additional error codes.

This function always returns immediately, even if nothing is available. The application can block on the OS-specific handle returned by cci_bind(), if desired.

The connection request structure contains the connection information, including pointer to the connection request data.

**Examples:**

server.c.

#### 4.4.4.4 CCI_DECLSPEC int cci_accept (cci_conn_req_t ∗ *conn_req*, cci_endpoint_t ∗ *endpoint*, cci_connection_t ∗∗ *connection*)

Accept a connection request and establish a connection with a specific endpoint.

**Parameters:**

← ***conn_req*** A connection request previously returned by cci_get_conn_req().

← ***endpoint*** The local endpoint to use for this connection. It must be bound to one of the devices specified in the connection request.

↔ ***connection*** Pointer to a connection request structure.

**Returns:**

CCI_SUCCESS The connection has been established.
CCI_EINVAL Conn_req, endpoint, or connection is NULL.
CCI_EINVAL The endpoint is not bound to one of the devices in the connection request.
CCI_ETIMEDOUT The incoming connection request timed out on the client.
Each driver may have additional error codes.

Upon success, the incoming connection request is bound to the desired endpoint and a connection handle is filled in. The connection request handle then becomes stale.

**Examples:**

server.c.

#### 4.4.4.5 CCI_DECLSPEC int cci_reject (cci_conn_req_t * *conn_req*)

Reject a connection request.

**Parameters:**

← ***conn_req*** Connection request to reject.

**Returns:**

CCI_SUCCESS Connection request has been rejected.
CCI_ETIMEDOUT The incoming connection request timed out on the client.
Each driver may have additional error codes.

Rejects an incoming connection request. The connection request becomes stale after this function returns successfully; no further interaction with this connection is possible after rejecting it.

**Examples:**

server.c.

#### 4.4.4.6 CCI_DECLSPEC int cci_connect (cci_endpoint_t * *endpoint*, char * *server_uri*, uint32_t *port*, void * *data_ptr*, uint32_t *data_len*, cci_conn_attribute_t *attribute*, void * *context*, int *flags*, struct timeval * *timeout*)

Initiate a connection request (client side).

Request a connection through a connection manager on a given machine for a given CCI service port. The connection manager address is described by a Uniform Resource Identifier. The use of an URI allows for flexible description (IP address, hostname, etc).

The connection request can carry limited amount of data to be passed to the server for application-specific usage (identification, authentication, etc).

The connect call is always non-blocking, reliable and requires a decision by the server (accept or reject), even for an unreliable connection, except for multicast.

Multicast connections don't necessarily involve a discrete connection server, they may be handled by IGMP or other distributed framework.

Upon completion, an ...

**Parameters:**

← ***endpoint*** Local endpoint to use for requested connection.

← ***server_uri*** Uniform Resource Identifier of the server. The URI is flexible and can encode different values. Coma-separated arguments can be added after a colon.

- IP address: "ip://172.31.194.2"
- Resolvable name: "ip://foo.bar.com"
- IB LID or GID: "ib://TBD"
- Blah id: "blah://crap0123"
- With arguments: "ip://foo.bar.com:eth1,eth3"

← ***port*** The CCI port number use to identify the service on the server.

← ***data_ptr*** Pointer to connection data to be sent in the connection request (for authentication, etc).

← ***data_len*** Length of connection data. Implementations must support a data_len values <= 1,024 bytes.

← ***attribute*** Attributes of the requested connection (reliability, ordering, multicast, etc).

← ***context*** Cookie to be used to identify the completion through an Other event.

← ***flags*** Currently unused.

← ***timeout*** NULL means forever.

**Returns:**

CCI_SUCCESS The request is buffered and ready to be sent or has been sent.
CCI_EINVAL Endpoint or server_uri is NULL.
CCI_EINVAL Data_ptr is NULL but data_len > 0.
Each driver may have additional error codes.

The server_uri is used to identify/reach a specific machine (it does not necessarily imply a specific destination endpoint). The URIs are strings so that we can easily accommodate special needs. The URIs are typically passed by the environment, as a hostname, an IP address, or whatever makes sense to identify a remote machine. The main part of the URI is device independent, it's only the identification of the remote machine. The arguments are device-specific. On the client side, the device to use is dictated by the local endpoint. On the server side, multiple devices can be used for the connection, depending on connectivity and arguments from the client.

**Examples:**

client.c.

#### 4.4.4.7 CCI_DECLSPEC int cci_disconnect (cci_connection_t ∗ *connection*)

Tear down an existing connection.

Operation is local, remote side is not notified. From that point, both local and remote side will get a DISCONNECTED communication error if sends are initiated on this connection.

**Parameters:**

← ***connection*** Connection to sever.

**Returns:**

CCI_SUCCESS The connection's resources have been released.
CCI_EINVAL Connection is NULL.
Each driver may have additional error codes.

**Examples:**

client.c.

## 4.5 Endpoint / Connection Options

### Data Structures

- union cci_opt_handle

    *Handle defining the scope of an option.*

### Typedefs

- typedef union cci_opt_handle cci_opt_handle_t

    *Handle defining the scope of an option.*

- typedef enum cci_opt_level cci_opt_level_t

    *Level defining the scope of an option.*

- typedef enum cci_opt_name cci_opt_name_t

    *Name of options.*

### Enumerations

- enum cci_opt_level { CCI_OPT_LEVEL_ENDPOINT, CCI_OPT_LEVEL_-CONNECTION }

    *Level defining the scope of an option.*

- enum cci_opt_name {

    CCI_OPT_ENDPT_MAX_HEADER_SIZE, CCI_OPT_ENDPT_SEND_-TIMEOUT, CCI_OPT_ENDPT_RECV_BUF_COUNT, CCI_OPT_ENDPT_-SEND_BUF_COUNT,

    CCI_OPT_ENDPT_KEEPALIVE_TIMEOUT, CCI_OPT_CONN_SEND_-TIMEOUT }

    *Name of options.*

### Functions

- CCI_DECLSPEC int cci_set_opt (cci_opt_handle_t ∗handle, cci_opt_level_-t level, cci_opt_name_t name, const void ∗val, int len)

    *Set an endpoint or connection option value.*

- CCI_DECLSPEC int cci_get_opt (cci_opt_handle_t *handle, cci_opt_level_t level, cci_opt_name_t name, void **val, int *len)

  *Get an endpoint or connection option value.*

### 4.5.1 Typedef Documentation

#### 4.5.1.1 typedef union cci_opt_handle cci_opt_handle_t

Handle defining the scope of an option.

#### 4.5.1.2 typedef enum cci_opt_level cci_opt_level_t

Level defining the scope of an option.

#### 4.5.1.3 typedef enum cci_opt_name cci_opt_name_t

Name of options.

### 4.5.2 Enumeration Type Documentation

#### 4.5.2.1 enum cci_opt_level

Level defining the scope of an option.

**Enumerator:**

***CCI_OPT_LEVEL_ENDPOINT*** Flag indicating that the union is an endpoint.

***CCI_OPT_LEVEL_CONNECTION*** Flag indicating that the union is a connection.

#### 4.5.2.2 enum cci_opt_name

Name of options.

**Enumerator:**

***CCI_OPT_ENDPT_MAX_HEADER_SIZE*** Max header size (in bytes) on the endpoint, for both sends and RMA operations.

cci_get_opt() only.

***CCI_OPT_ENDPT_SEND_TIMEOUT*** Default send timeout for all new connections.

cci_get_opt() and cci_set_opt().

***CCI_OPT_ENDPT_RECV_BUF_COUNT*** How many receiver buffers on the endpoint.

It is the max number of messages the CCI layer can receive without dropping.

cci_get_opt() and cci_set_opt().

***CCI_OPT_ENDPT_SEND_BUF_COUNT*** How many send buffers on the endpoint.

It is the max number of pending messages the CCI layer can buffer before failing or blocking (depending on reliability mode).

cci_get_opt() and cci_set_opt().

***CCI_OPT_ENDPT_KEEPALIVE_TIMEOUT*** The "keepalive" timeout is to prevent a client from connecting to a server and then the client disappears without the server noticing.

If the server never sends anything on the connection, it'll never realize that the client is gone, but the connection is still consuming resources. But note that keepalive timers apply to both clients and servers.

The keepalive timeout is expressed in microseconds. If the keepalive timeout value is set:

- If no traffic at all is received on a connection within the keepalive timeout, the CCI_EVENT_KEEPALIVE_TIMEOUT event is raised on that connection.
- The CCI implementation will automatically send control hearbeats across an inactive (but still alive) connection to reset the peer's keepalive timer before it times out.

If a keepalive event is raised, the keepalive timeout is set to 0 (i.e., it must be "re-armed" before it will timeout again), but the connection is ∗not∗ disconnected. Recovery decisions are up to the application; it may choose to disconnect the connection, re-arm the keepalive timeout, etc.

cci_get_opt() and cci_set_opt().

***CCI_OPT_CONN_SEND_TIMEOUT*** Reliable send timeout in microseconds.

cci_get_opt() and cci_set_opt().

## 4.5.3 Function Documentation

### 4.5.3.1 CCI_DECLSPEC int cci_set_opt (cci_opt_handle_t ∗ *handle*, cci_opt_level_t *level*, cci_opt_name_t *name*, const void ∗ *val*, int *len*)

Set an endpoint or connection option value.

**Parameters:**

← ***handle*** Endpoint or connection handle.

← ***level*** Indicates type of handle.

← ***name*** Which option to set the value of.

← ***val*** Pointer to the value.

← ***len*** Length of value to be set.

**Returns:**

CCI_SUCCESS Value successfully set.
CCI_EINVAL Handle or val is NULL or len is 0.
CCI_EINVAL Level/name mismatch.
CCI_ERR_NOT_IMPLEMENTED Not supported by this driver.
Each driver may have additional error codes.

Note that the set may fail if the CCI implementation cannot actually set the value.

**Examples:**

client.c.

#### 4.5.3.2 CCI_DECLSPEC int cci_get_opt (cci_opt_handle_t ∗ *handle*, cci_opt_level_t *level*, cci_opt_name_t *name*, void ∗∗ *val*, int ∗ *len*)

Get an endpoint or connection option value.

**Parameters:**

← ***handle*** Endpoint or connection handle.

← ***level*** Indicates type of handle.

← ***name*** Which option to set the value of.

← ***val*** Address of the pointer to the value.

← ***len*** Address of the length of value.

**Returns:**

CCI_SUCCESS Value successfully retrieved.
CCI_EINVAL Handle or val is NULL or len is 0.
CCI_EINVAL Level/name mismatch.
CCI_ERR_NOT_IMPLEMENTED Not supported by this driver.
Each driver may have additional error codes.

## 4.6 Events

### Data Structures

- struct cci_event_send

  *Send event.*

- struct cci_event_recv

  *Receive event.*

- struct cci_event_other

  *Other event.*

- struct cci_event

  *Generic event.*

### Typedefs

- typedef struct cci_event_send cci_event_send_t

  *Send event.*

- typedef struct cci_event_recv cci_event_recv_t

  *Receive event.*

- typedef struct cci_event_other cci_event_other_t

  *Other event.*

- typedef enum cci_event_type cci_event_type_t

  *Event types.*

- typedef struct cci_event cci_event_t

  *Generic event.*

### Enumerations

- enum cci_event_type {

  CCI_EVENT_NONE, CCI_EVENT_SEND, CCI_EVENT_RECV, CCI_-EVENT_CONNECT_SUCCESS,

CCI_EVENT_CONNECT_TIMEOUT, CCI_EVENT_CONNECT_-REJECTED, CCI_EVENT_KEEPALIVE_TIMEOUT, CCI_EVENT_-ENDPOINT_DEVICE_FAIL }

*Event types.*

## Functions

- CCI_DECLSPEC int cci_arm_os_handle (cci_endpoint_t ∗endpoint, int flags)
- CCI_DECLSPEC int cci_get_event (cci_endpoint_t ∗endpoint, cci_event_-t ∗∗const event, uint32_t flags)

  *Get the next available CCI event.*

- CCI_DECLSPEC int cci_return_event (cci_endpoint_t ∗endpoint, cci_event_t ∗event)

  *This function returns the buffer associated with an event that was previously obtained via cci_get_event().*

### 4.6.1 Typedef Documentation

#### 4.6.1.1 typedef struct cci_event_send cci_event_send_t

Send event.

A completion struct instance is returned for each cci_send() that requested a completion notification.

On a reliable connection, a sender will generally complete a send when the receiver replies for that message. Additionally, an error status may be returned (UNREACH-ABLE, DISCONNECTED, RNR).

On an unreliable connection, a sender will return CCI_SUCCESS upon local completion (i.e., the message has been queued up to some lower layer – there is no guarantee that it is "on the wire", etc.). Other send statuses will only be returned for local errors.

The number of fields in this struct is intentionally limited in order to reduce costs associated with state storage, caching, updating, copying. For example, there is no field pointing to the endpoint used for the send because it can be obtained from the cci_connection, or through the endpoint passed to the cci_get_event() call.

If it is desirable to match send completions with specific sends (it usually is), it is the responsibility of the caller to pass a meaningful context value to cci_send().

The ordering of fields in this struct is intended to reduce memory holes between fields.

#### 4.6.1.2 typedef struct cci_event_recv cci_event_recv_t

Receive event.

A completion struct instance is returned for each message received.

The number of fields in this struct is intentionally limited in order to reduce costs associated with state storage, caching, updating, copying. For example, there is no field pointing to the endpoint because it can be obtained from the cci_connection or through the endpoint passed to the cci_get_event() call.

The ordering of fields in this struct is intended to reduce memory holes between fields.

#### 4.6.1.3 typedef struct cci_event_other cci_event_other_t

Other event.

Other event.

A completion struct to handle non-send and non-receive events.

It contains a context pointer for application-specific data such as the state of a connection request waiting for a connection accept or reject message (i.e., passed to cci_-connect()).

The event also contains a union depending on the type of other event. If it is CONNECT_SUCCESS (i.e. the server accepted the connection request), the new connection is returned in the union. For all other events, the union has no meaning.

**Note:**

We will need to add a union member for keepalive timeouts that will have a pointer to the connection that timed out.

#### 4.6.1.4 typedef enum cci_event_type cci_event_type_t

Event types.

There are three board categories of events: send, receive, and other. The other class includes connect success, rejected, and timeout as well as a generic endpoint device failure.

The NONE event type is never passed to the application and is for internal CCI use only.

#### 4.6.1.5 typedef struct cci_event cci_event_t

Generic event.

This is the union of Send, Recv and Other events.

## 4.6.2 Enumeration Type Documentation

### 4.6.2.1 enum cci_event_type

Event types.

There are three board categories of events: send, receive, and other. The other class includes connect success, rejected, and timeout as well as a generic endpoint device failure.

The NONE event type is never passed to the application and is for internal CCI use only.

**Enumerator:**

***CCI_EVENT_NONE*** Never use - for internal CCI use only.

***CCI_EVENT_SEND*** A send or RMA has completed.

***CCI_EVENT_RECV*** An active message has been received.

***CCI_EVENT_CONNECT_SUCCESS*** A new outgoing connection was successfully accepted at the peer; a connection is now available for data transfer.

***CCI_EVENT_CONNECT_TIMEOUT*** A new outgoing connection did not complete the accept/connect handshake with the peer in a finite time.

CCI has therefore given up attempting to continue to create this connection.

***CCI_EVENT_CONNECT_REJECTED*** A new outgoing connection was rejected by the server.

***CCI_EVENT_KEEPALIVE_TIMEOUT*** This event occurs when the keepalive timeout has expired (see CCI_OPT_ENDPT_KEEPALIVE_TIMEOUT for more details).

***CCI_EVENT_ENDPOINT_DEVICE_FAIL*** A device on this endpoint has failed.

## 4.6.3 Function Documentation

### 4.6.3.1 CCI_DECLSPEC int cci_arm_os_handle (cci_endpoint_t * *endpoint*, int *flags*)

### 4.6.3.2 CCI_DECLSPEC int cci_get_event (cci_endpoint_t * *endpoint*, cci_event_t **const *event*, uint32_t *flags*)

Get the next available CCI event.

This function never blocks; it polls instantly to see if there is any pending event of any type (send completion, receive, or other events – errors, incoming connection requests, etc.). If you want to block, use the OS handle to use your OS's native blocking mechanism (e.g., select/poll on the POSIX fd). This also allows the app to busy poll for a while and then OS block if nothing interesting is happening. The default OS handle returned when creating the endpoint will return the equivalent of a POLL_IN when any event is available.

This function borrows the buffer associated with the event; it must be explicitly returned later via cci_return_event().

**Parameters:**

← ***endpoint*** Endpoint to poll for a new event.

← ***event*** New event, if any.

← ***flags***
- CCI_PE_SEND_EVENT
- CCI_PE_RECV_EVENT
- CCI_PE_OTHER_EVENT
- CCI_PE_I_SET_THE_DATA_BUFFER_PLEASE_COPY Flag value of 0 means (CCI_PE_SEND_EVENT | CCI_PE_RECV_EVENT | CCI_PE_OTHER_EVENT).

**Returns:**

CCI_SUCCESS An event was retrieved.
CCI_EINVAL Endpoint or event is NULL.
CCI_EAGAIN No event is available.
Each driver may have additional error codes.

To discuss:

- How do we know if the event was filled? Via the function return value?
- it may be convenient to optionally get multiple OS handles; one each for send completions, receives, and "other" (errors, incoming connection requests, etc.). Should that be part of endpoint creation? If we allow this concept, do we need a way to pass in a different CQ here to get just those types of events?
- How do we have CCI-implementation private space in the event – bound by size? I.e., how/who determines the max inline data size?

**Examples:**

client.c, and server.c.

#### 4.6.3.3 CCI_DECLSPEC int cci_return_event (cci_endpoint_t ∗ *endpoint*, cci_event_t ∗ *event*)

This function returns the buffer associated with an event that was previously obtained via cci_get_event().

The data buffer associated with the event will immediately become stale to the application.

Events may be returned in any order; they do not need to be returned in the same order that cci_poll_event() issued them. All events must be returned, even send completions and "other" events – not just receive events. However, it is possible (likely) that returning send completion and "other" events will be no-ops.

**Parameters:**

← ***endpoint*** Endpoint that provided the event.

← ***event*** Event to return.

**Returns:**

CCI_SUCCESS The event was returned to CCI.
CCI_EINVAL Endpoint is NULL.
CCI_EINVAL Event did not come from endpoint.
Each driver may have additional error codes.

**Examples:**

client.c, and server.c.

## 4.7 Communications

### Data Structures

- struct **cci_sg**

  *This data structure should map to the native scatter/gather list that is used down in the kernel.*

### Typedefs

- typedef struct cci_sg cci_sg_t

  *This data structure should map to the native scatter/gather list that is used down in the kernel.*

### Functions

- CCI_DECLSPEC int cci_send (cci_connection_t ∗connection, void ∗header_ptr, uint32_t header_len, void ∗data_ptr, uint32_t data_len, void ∗context, int flags)

  *Send a short message.*

- CCI_DECLSPEC int cci_sendv (cci_connection_t ∗connection, void ∗header_ptr, uint32_t header_len, struct iovec ∗data, uint8_t iovcnt, void ∗context, int flags)

  *Send a short vectored (gather) message.*

- CCI_DECLSPEC int cci_rma_register (cci_endpoint_t ∗endpoint, cci_connection_t ∗connection, void ∗start, uint64_t length, uint64_t ∗rma_handle)

  *Register memory for RMA operations.*

- CCI_DECLSPEC int cci_rma_deregister (uint64_t rma_handle)

  *Deregister memory.*

- CCI_DECLSPEC int cci_rma (cci_connection_t ∗connection, void ∗header_ptr, uint32_t header_len, uint64_t local_handle, uint64_t local_offset, uint64_t remote_handle, uint64_t remote_offset, uint64_t data_len, void ∗context, int flags)

  *Perform a RMA operation between local and remote memory.*

## 4.7.1 Typedef Documentation

### 4.7.1.1 typedef struct cci_sg cci_sg_t

This data structure should map to the native scatter/gather list that is used down in the kernel.

## 4.7.2 Function Documentation

### 4.7.2.1 CCI_DECLSPEC int cci_send (cci_connection_t ∗ *connection*, void ∗ *header_ptr*, uint32_t *header_len*, void ∗ *data_ptr*, uint32_t *data_len*, void ∗ *context*, int *flags*)

Send a short message.

am_max_size maximum, no order guaranteed, completion is local.

Two segments for Header and Data. When CCI_FLAG_ASYNC is used and the call returns, data has been buffered.

A short message may have two segments, header and data. The header has a limited size which is retrievable using cci_get_opt() with the CCI_OPT_ENDPT_MAX_HEADER_SIZE flag. The data length is limited to the cci_connection::max_send_size, which may be lower than the cci_device::max_send_size. The application may specify both the header and data, only one, or neither (although nothing will be delivered, the peer will still ack the message on a reliable connection).

If the application needs to send a message larger than cci_connection::max_send_size, the application is responsible for segmenting and reassembly or it should use cci_rma().

When cci_send() returns, the application buffer is reusable. By default, CCI will buffer the data internally.

**Parameters:**

← ***connection*** Connection (destination/reliability).

← ***header_ptr*** Pointer to local header segment.

← ***header_len*** Length of local header segment (limited to 32 bytes).

← ***data_ptr*** Pointer to local data segment.

← ***data_len*** Length of local data segment (limited to max send size).

← ***context*** Cookie to identify the completion through a Send event when non-blocking.

← ***flags*** Optional flags: CCI_FLAG_BLOCKING, CCI_FLAG_NO_COPY, CCI_FLAG_SILENT. These flags are explained below.

**Returns:**

CCI_SUCCESS The message has been queued to send.
CCI_EINVAL Connection is NULL.
CCI_EINVAL header_ptr is NULL and header_len is > 0.
CCI_EINVAL data_ptr is NULL and data_len is > 0.
Each driver may have additional error codes.

The send will complete differently in reliable and unreliable connections:

- Reliable: only when remote side ACKs complete delivery – but not necessary consumption (i.e., remote completion).
- Unreliable: when the buffer is re-usable (i.e., local completion).

When cci_send() returns, the buffer is re-usable by the application.

If the CCI_FLAG_BLOCKING flag is specified, cci_send() will *also* block until the send completion has occurred. In this case, there is no event returned for this send via cci_get_event(); the send completion status is returned via cci_send().

If the CCI_FLAG_NO_COPY is specified, the application is indicating that it does not need the buffer back until the send completion occurs (which is most useful when CCI_FLAG_BLOCKING is *not* specified). The CCI implementation is therefore free to use "zero copy" types of transmission with the buffer – if it wants to.

CCI_FLAG_SILENT means that no completion will be generated for non-CCI_-FLAG_BLOCKING sends. For reliable ordered connections, since completions are issued in order, the completion of any non-SILENT send directly implies the completion of any previous SILENT sends. For unordered connections, completion ordering is not guaranteed – it is **not** safe to assume that application protocol semantics imply specific unordered SILENT send completions. The only ways to know when unordered SILENT sends have completed (and that the local send buffer is "owned" by the application again) is either to close the connection or issue a non-SILENT send. The completion of a non-SILENT send guarantees the completion of all previous SILENT sends.

**Examples:**

client.c, and server.c.

#### 4.7.2.2 CCI_DECLSPEC int cci_sendv (cci_connection_t ∗ *connection*, void ∗ *header_ptr*, uint32_t *header_len*, struct iovec ∗ *data*, uint8_t *iovcnt*, void ∗ *context*, int *flags*)

Send a short vectored (gather) message.

Like cci_send(), cci_sendv() sends a short message bound by cci_connection::max_-send_size. Instead of a single data buffer, cci_sendv() allows the application to gather an array of iovcnt buffers pointed to by struct iovec ∗data.

**Parameters:**

← ***connection*** Connection (destination/reliability).

← ***header_ptr*** Pointer to local header segment.

← ***header_len*** Length of local header segment (limited to 32 bytes).

← ***data*** Array of local data buffers.

← ***iovcnt*** Count of local data array.

← ***context*** Cookie to identify the completion through a Send event when non-blocking.

← ***flags*** Optional flags: CCI_FLAG_BLOCKING, CCI_FLAG_NO_COPY, CCI_FLAG_SILENT. See cci_send().

**Returns:**

CCI_SUCCESS The message has been queued to send.
CCI_EINVAL Connection is NULL.
CCI_EINVAL header_ptr is NULL and header_len is > 0.
CCI_EINVAL data is NULL and iovcnt is > 0.
Each driver may have additional error codes.

#### 4.7.2.3 CCI_DECLSPEC int cci_rma_register (cci_endpoint_t ∗ *endpoint*, cci_connection_t ∗ *connection*, void ∗ *start*, uint64_t *length*, uint64_t ∗ *rma_handle*)

Register memory for RMA operations.

The intent is that this function is invoked frequently – "just register everything" before invoking RMA operations.

In the best case, the implementation is cheap/fast enough that the invocation time doesn't noticeably affect performance (e.g., MX and PSM). If the implementation is slow (e.g., IB/iWARP), this function should probably have a registration cache so that at least repeated registrations are fast.

If the connection is provided, the memory is only exposed to that connection. If it is NULL, then any reliable connection on that endpoint can access that memory.

It is allowable to have overlapping registerations.

**Parameters:**

← ***endpoint*** Local endpoint to use for RMA.

← ***connection*** Restrict RMA to this connection.

← ***start*** Pointer to local memory.

← ***length*** Length of local memory.

→ ***rma_handle*** Handle for use with cci_rma().

**Returns:**

CCI_SUCCESS The memory is ready for RMA.
CCI_EINVAL endpoint, start, or rma_handle is NULL.
CCI_EINVAL connection is unreliable.
CCI_EINVAL length is 0.
Each driver may have additional error codes.

#### 4.7.2.4 CCI_DECLSPEC int cci_rma_deregister (uint64_t *rma_handle*)

Deregister memory.

If an RMA is in progress that uses this handle, the RMA may abort or the deregistration may fail.

Once deregistered, the handle is stale.

**Parameters:**

← ***rma_handle*** Handle for use with cci_rma().

**Returns:**

CCI_SUCCESS The memory is deregistered.
Each driver may have additional error codes.

#### 4.7.2.5 CCI_DECLSPEC int cci_rma (cci_connection_t ∗ *connection*, void ∗ *header_ptr*, uint32_t *header_len*, uint64_t *local_handle*, uint64_t *local_offset*, uint64_t *remote_handle*, uint64_t *remote_offset*, uint64_t *data_len*, void ∗ *context*, int *flags*)

Perform a RMA operation between local and remote memory.

Initiate a remote memory WRITE access (move local memory to remote memory) or READ (move remote memory to local memory). Adding the FENCE flag ensures all previous operations are guaranteed to complete remotely prior to this operation and all subsequent operations. Remote completion does not imply a remote completion event, merely a successful RMA operation.

Optionally, send a remote completion event to the target. If header_ptr and header_len are provided, send a completion event to the target after the RMA has completed. It is guaranteed to arrive after the RMA operation has finished.

CCI makes no guarantees about the data delivery within the RMA operation (e.g., no last-byte-written-last).

Only a local completion will be generated.

**Parameters:**

← ***connection*** Connection (destination).

← ***header_ptr*** Pointer to local header segment.

← ***header_len*** Length of local header segment (limited to 32 bytes)

← ***local_handle*** Handle of the local RMA area.

← ***local_offset*** Offset in the local RMA area.

← ***remote_handle*** Handle of the remote RMA area.

← ***remote_offset*** Offset in the remote RMA area.

← ***data_len*** Length of data segment.

← ***context*** Cookie to identify the completion through a Send event when non-blocking.

← ***flags*** Optional flags:

- CCI_FLAG_BLOCKING: Blocking call (see cci_send() for details).
- CCI_FLAG_READ: Move data from remote to local memory.
- CCI_FLAG_WRITE: Move data from local to remote memory
- CCI_FLAG_FENCE: All previous operations are guaranteed to complete remotely prior to this operation and all subsequent operations.
- CCI_FLAG_SILENT: Generates no local completion event (see cci_send() for details).

**Returns:**

CCI_SUCCESS The RMA operation has been initiated.
CCI_EINVAL connection is NULL.
CCI_EINVAL connection is unreliable.
CCI_EINVAL header_ptr is NULL and header_len > 0.
CCI_EINVAL data_len is 0.
CCI_EINVAL Both READ and WRITE flags are set.
CCI_EINVAL Neither the READ or WRITE flag is set.
Each driver may have additional error codes.

**Note:**

CCI_FLAG_FENCE only applies to RMA operations for this connection. It does not apply to sends on this connection.
READ may not be performance efficient.

# Chapter 5

# Data Structure Documentation

## 5.1 cci_conn_req Struct Reference

Connection request.

`#include <cci.h>`

### Data Fields

- cci_device_t const **const devices

  *Array of compatible devices.*

- uint32_t devices_cnt

  *Number of compatible devices.*

- const void * data_ptr

  *Pointer to connection data received with the connection request.*

- uint32_t data_len

  *Length of connection data.*

- cci_conn_attribute_t attribute

  *Attribute of requested connection.*

### 5.1.1 Detailed Description

Connection request.

**Examples:**

server.c.

### 5.1.2 Field Documentation

#### 5.1.2.1 cci_device_t const** const cci_conn_req::devices

Array of compatible devices.

#### 5.1.2.2 uint32_t cci_conn_req::devices_cnt

Number of compatible devices.

#### 5.1.2.3 const void* cci_conn_req::data_ptr

Pointer to connection data received with the connection request.

#### 5.1.2.4 uint32_t cci_conn_req::data_len

Length of connection data.

#### 5.1.2.5 cci_conn_attribute_t cci_conn_req::attribute

Attribute of requested connection.

## 5.2 cci_connection Struct Reference

Connection handle.

`#include <cci.h>`

### Data Fields

- uint32_t max_send_size

  *Maximum send size for the connection.*

- cci_endpoint_t * endpoint

  *Local endpoint associated to the connection.*

- cci_conn_attribute_t attribute

  *Attributes of the connection.*

### 5.2.1 Detailed Description

Connection handle.

**Examples:**

client.c, and server.c.

### 5.2.2 Field Documentation

#### 5.2.2.1 uint32_t cci_connection::max_send_size

Maximum send size for the connection.

**Examples:**

server.c.

#### 5.2.2.2 cci_endpoint_t* cci_connection::endpoint

Local endpoint associated to the connection.

#### 5.2.2.3 cci_conn_attribute_t cci_connection::attribute

Attributes of the connection.

## 5.3 cci_device Struct Reference

Structure representing one CCI device.

`#include <cci.h>`

### Data Fields

- const char * name

  *Name of the device from the config file, e.g., "bob0".*

- const char * info

  *Human readable description string (to include newlines); should contain debugging info, probably the network address of the device at a bare minimum.*

- const char ** conf_argv

  *Array of "key=value" strings from the config file for this device; the last pointer in the array is NULL.*

- uint32_t max_send_size

  *Maximum send size supported by the device.*

- uint64_t rate

  *Data rate per specification: data bits per second (not the signaling rate).*

- struct {
  uint32_t domain
  uint32_t bus
  uint32_t dev
  uint32_t func
  } pci

  *The PCI ID of this device as reported by the OS/hardware.*

### 5.3.1 Detailed Description

Structure representing one CCI device.

### 5.3.2 Devices

Device types and functions.

Before launching into detail, let's first describe the CCI system configuration file. On POSIX systems, it is likely a simple INI-style text file; on Windows systems, it may be registry entries. The key thing is to support trivial namespaces and key=value pairs.

Here is an example text config file:

```
# Comments are anything after the # symbols.

# Sections in this file are denoted by [section name].  Each section
# denotes a single CCI device.

[bob0]
# The only mandated field in each section is "driver".  It indicates
# which CCI driver should be applied to this device.
driver = psm

# The priority field determines the ordering of devices returned by
# cci_get_devices().  100 is the highest priority; 0 is the lowest priority.
# If not specified, the priority value is 50.
priority = 10

# The last field understood by the CCI core is the "default" field.
# Only one device is allowed to have a "true" value for default.  All
# others must be set to 0 (or unset, which is assumed to be 0).  If
# one device is marked as the default, then this device will be used
# when NULL is passed as the device when creating an endpoint.  If no
# device is marked as the default, it is undefined as to which device
# will be used when NULL is passed as the device when creating an
# endpoint.
default = 1

# All other fields are uninterpreted by the CCI core; they're just
# passed to the driver.  The driver can do whatever it wants with
# these values (e.g., system admins can set values to configure the
# driver).  Driver documentation should specify what parameters are
# available, what each parameter is/does, and what its legal values
# are.

# This example shows a bonded PSM device that uses both the ipath0 and
# ipath1 devices.  Some other parameters are also passed to the PSM
# driver; it assumedly knows how to handle them.

device = ipath0,ipath1
capabilities = bonded,failover,age_of_captain:52
qos_stuff = fast

# bob2 is another PSM device, but it only uses the ipath0 device.
[bob2]
driver = psm
device = ipath0

# bob3 is another PSM device, but it only uses the ipath1 device.
[bob3]
driver = psm
device = ipath1
sl = 3 # IB service level (if applicable)
```

```
# storage is a device that uses the UDP driver.  Note that this driver
# allows specifying which device to use by specifying its IP address
# and MAC address -- assumedly it's an error if there is no single
# device that matches both the specified IP address and MAC
# (vs. specifying a specific device name).
[storage]
driver = udp
priority = 5
ip = 172.31.194.1
mac = 01:12:23:34:45
```

The config file forms the basis for the device discussion, below.

A CCI device is a [section] from the config file, above.

**Examples:**

client.c, devices.c, and server.c.

## 5.3.3 Field Documentation

### 5.3.3.1 const char* cci_device::name

Name of the device from the config file, e.g., "bob0".

### 5.3.3.2 const char* cci_device::info

Human readable description string (to include newlines); should contain debugging info, probably the network address of the device at a bare minimum.

### 5.3.3.3 const char** cci_device::conf_argv

Array of "key=value" strings from the config file for this device; the last pointer in the array is NULL.

### 5.3.3.4 uint32_t cci_device::max_send_size

Maximum send size supported by the device.

### 5.3.3.5 uint64_t cci_device::rate

Data rate per specification: data bits per second (not the signaling rate).

#### 5.3.3.6 uint32_t cci_device::domain

#### 5.3.3.7 uint32_t cci_device::bus

#### 5.3.3.8 uint32_t cci_device::dev

#### 5.3.3.9 uint32_t cci_device::func

#### 5.3.3.10 struct { ... } cci_device::pci

The PCI ID of this device as reported by the OS/hardware.

All values will be ((uint32_t) -1) for non-PCI devices (e.g., shared memory)

## 5.4 cci_endpoint Struct Reference

Endpoint.

```
#include <cci.h>
```

### Data Fields

- uint32_t max_recv_buffer_count

  *Maximum number of receive buffers on this endpoint that can be loaned to the application.*

### 5.4.1 Detailed Description

Endpoint.

**Examples:**

client.c, and server.c.

### 5.4.2 Field Documentation

#### 5.4.2.1 uint32_t cci_endpoint::max_recv_buffer_count

Maximum number of receive buffers on this endpoint that can be loaned to the application.

When this number of buffers have been loaned to the application, incoming messages may be dropped.

# 5.5 cci_event Struct Reference

Generic event.

`#include <cci.h>`

## Data Fields

- cci_event_type_t type

  *Type of the event.*

- union {
  cci_event_send_t send
  cci_event_recv_t recv
  cci_event_other_t other
  } info

  *union of event types*

### 5.5.1 Detailed Description

Generic event.

This is the union of Send, Recv and Other events.

**Examples:**

client.c, and server.c.

### 5.5.2 Field Documentation

#### 5.5.2.1 cci_event_type_t cci_event::type

Type of the event.

**Examples:**

client.c, and server.c.

#### 5.5.2.2 cci_event_send_t cci_event::send

**Examples:**

client.c.

#### 5.5.2.3 cci_event_recv_t cci_event::recv

**Examples:**

client.c, and server.c.

#### 5.5.2.4 cci_event_other_t cci_event::other

#### 5.5.2.5 union { ... } cci_event::info

union of event types

**Examples:**

client.c, and server.c.

## 5.6 cci_event_other Struct Reference

Other event.

`#include <cci.h>`

### Data Fields

- void * context

  *Context value.*

- union {
  struct **cci_event_connect** {
  cci_connection_t * connection
  } connect
  *new connection if peer accepted our connection request*
  } u

  *union of possible other items*

### 5.6.1 Detailed Description

Other event.

Other event.

A completion struct to handle non-send and non-receive events.

It contains a context pointer for application-specific data such as the state of a connection request waiting for a connection accept or reject message (i.e., passed to cci_-connect()).

The event also contains a union depending on the type of other event. If it is CONNECT_SUCCESS (i.e. the server accepted the connection request), the new connection is returned in the union. For all other events, the union has no meaning.

**Note:**

We will need to add a union member for keepalive timeouts that will have a pointer to the connection that timed out.

### 5.6.2 Field Documentation

#### 5.6.2.1 void* cci_event_other::context

Context value.

#### 5.6.2.2 cci_connection_t* cci_event_other::connection

#### 5.6.2.3 struct { ... } ::cci_event_connect cci_event_other::connect

new connection if peer accepted our connection request

#### 5.6.2.4 union { ... } cci_event_other::u

union of possible other items

## 5.7 cci_event_recv Struct Reference

Receive event.

```
#include <cci.h>
```

### Data Fields

- const uint32_t header_len

  *The length of the header part of the message (in bytes).*

- const uint32_t data_len

  *The length of the data part of the message (in bytes).*

- void *const header_ptr

  *Pointer to the header part of the received message.*

- void *const data_ptr

  *Pointer to the data part of the received message.*

- cci_connection_t * connection

  *Connection that this message was received on.*

### 5.7.1 Detailed Description

Receive event.

A completion struct instance is returned for each message received.

The number of fields in this struct is intentionally limited in order to reduce costs associated with state storage, caching, updating, copying. For example, there is no field pointing to the endpoint because it can be obtained from the cci_connection or through the endpoint passed to the cci_get_event() call.

The ordering of fields in this struct is intended to reduce memory holes between fields.

### 5.7.2 Field Documentation

#### 5.7.2.1 const uint32_t cci_event_recv::header_len

The length of the header part of the message (in bytes).

This value may be 0.

**Examples:**

client.c, and server.c.

#### 5.7.2.2 const uint32_t cci_event_recv::data_len

The length of the data part of the message (in bytes).

This value may be 0.

**Examples:**

client.c, and server.c.

#### 5.7.2.3 void∗ const cci_event_recv::header_ptr

Pointer to the header part of the received message.

The pointer always points to an address that is 8-byte aligned, unless (header_len == 0), in which case the value is undefined.

**Examples:**

client.c, and server.c.

#### 5.7.2.4 void∗ const cci_event_recv::data_ptr

Pointer to the data part of the received message.

The pointer always points to an address that is 8-byte aligned, unless (header_len == 0), in which case the value is undefined.

**Examples:**

client.c, and server.c.

#### 5.7.2.5 cci_connection_t∗ cci_event_recv::connection

Connection that this message was received on.

## 5.8 cci_event_send Struct Reference

Send event.

`#include <cci.h>`

### Data Fields

- cci_connection_t * connection

  *Connection that the send was initiated on.*

- void * context

  *Context value that was passed to cci_send().*

- cci_status_t status

  *Result of the send.*

### 5.8.1 Detailed Description

Send event.

A completion struct instance is returned for each cci_send() that requested a completion notification.

On a reliable connection, a sender will generally complete a send when the receiver replies for that message. Additionally, an error status may be returned (UNREACHABLE, DISCONNECTED, RNR).

On an unreliable connection, a sender will return CCI_SUCCESS upon local completion (i.e., the message has been queued up to some lower layer – there is no guarantee that it is "on the wire", etc.). Other send statuses will only be returned for local errors.

The number of fields in this struct is intentionally limited in order to reduce costs associated with state storage, caching, updating, copying. For example, there is no field pointing to the endpoint used for the send because it can be obtained from the cci_connection, or through the endpoint passed to the cci_get_event() call.

If it is desirable to match send completions with specific sends (it usually is), it is the responsibility of the caller to pass a meaningful context value to cci_send().

The ordering of fields in this struct is intended to reduce memory holes between fields.

## 5.8.2 Field Documentation

### 5.8.2.1 cci_connection_t∗ cci_event_send::connection

Connection that the send was initiated on.

### 5.8.2.2 void∗ cci_event_send::context

Context value that was passed to cci_send().

**Examples:**

client.c.

### 5.8.2.3 cci_status_t cci_event_send::status

Result of the send.

**Examples:**

client.c.

## 5.9 cci_opt_handle Union Reference

Handle defining the scope of an option.

`#include <cci.h>`

### Data Fields

- cci_endpoint_t * endpoint

  *Endpoint.*

- cci_connection_t * connection

  *Connection.*

### 5.9.1 Detailed Description

Handle defining the scope of an option.

**Examples:**

client.c.

### 5.9.2 Field Documentation

#### 5.9.2.1 cci_endpoint_t* cci_opt_handle::endpoint

Endpoint.

**Examples:**

client.c.

#### 5.9.2.2 cci_connection_t* cci_opt_handle::connection

Connection.

## 5.10 cci_service Struct Reference

Service handle.

```
#include <cci.h>
```

### Data Fields

- int bogus

  *unused*

### 5.10.1 Detailed Description

Service handle.

**Examples:**

server.c.

### 5.10.2 Field Documentation

#### 5.10.2.1 int cci_service::bogus

unused

# Chapter 6

# Example Documentation

## 6.1 client.c

This application demonstrates opening an endpoint, connecting to a server, sending messages, and polling for events.

```
/*
 * Copyright (c) 2011 UT-Battelle, LLC.  All rights reserved.
 * Copyright (c) 2011 Oak Ridge National Labs.  All rights reserved.
 *
 * See COPYING in top-level directory
 *
 * $COPYRIGHT$
 *
 */

#include "cci.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

char *proc_name = NULL;

void
usage(void)
{
    fprintf(stderr, "usage: %s -h <server_uri>\n", proc_name);
    fprintf(stderr, "where server_uri is a valid CCI uri\n");
    fprintf(stderr, "such as ip://1.2.3.4\n");
    exit(EXIT_FAILURE);
}

static void
poll_events(cci_endpoint_t *endpoint, cci_connection_t **connection, int *done)
{
```

```
    int ret;
    char buffer[8192];
    cci_event_t *event;

    ret = cci_get_event(endpoint, &event, 0);
    if (ret == CCI_SUCCESS) {
        switch (event->type) {
        case CCI_EVENT_SEND:
            printf("send %d completed with %s\n",
                   (int)((uintptr_t) event->info.send.context),
                   cci_strerror(event->info.send.status));
            break;
        case CCI_EVENT_RECV:
            memcpy(buffer, event->info.recv.header_ptr, event->info.recv.header_len);
            buffer[event->info.recv.header_len] = '\0';
            fprintf(stderr, "received header\"%s\"\n", buffer);
            memcpy(buffer, event->info.recv.data_ptr, event->info.recv.data_len);
            buffer[event->info.recv.data_len] = '\0';
            fprintf(stderr, "received data\"%s\"\n", buffer);
            *done = 1;
            break;
        case CCI_EVENT_CONNECT_SUCCESS:
            *done = 1;
            *connection = event->info.other.u.connect.connection;
            break;
        case CCI_EVENT_CONNECT_TIMEOUT:
        case CCI_EVENT_CONNECT_REJECTED:
            *done = 1;
            *connection = NULL;
            break;
        default:
            fprintf(stderr, "ignoring event type %d\n", event->type);
        }
        cci_return_event(endpoint, event);
    }
}

int main(int argc, char *argv[])
{
        int done = 0, ret, i = 0, c;
        uint32_t caps = 0;
    char *server_uri = NULL; /* ip://1.2.3.4 */
        cci_os_handle_t fd;
        cci_device_t **devices = NULL;
        cci_endpoint_t *endpoint = NULL;
        cci_connection_t *connection = NULL;
    cci_opt_handle_t handle;
    uint32_t timeout_us = 30 * 1000000; /* microseconds */

    proc_name = argv[0];

    while ((c = getopt(argc, argv, "h:")) != -1) {
        switch (c) {
            case 'h':
                server_uri = strdup(optarg);
                break;
            default:
```

```
            usage();
        }
    }

    /* init */
        ret = cci_init(CCI_ABI_VERSION, 0, &caps);
    if (ret) {
        fprintf(stderr, "cci_init() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    /* get devices */
        ret = cci_get_devices((const cci_device_t *** const)&devices);
    if (ret) {
        fprintf(stderr, "cci_get_devices() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

        /* create an endpoint */
        ret = cci_create_endpoint(NULL, 0, &endpoint, &fd);
    if (ret) {
        fprintf(stderr, "cci_create_endpoint() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    /* set endpoint tx timeout */
    handle.endpoint = endpoint;
    cci_set_opt(&handle, CCI_OPT_LEVEL_ENDPOINT, CCI_OPT_ENDPT_SEND_TIMEOUT,
                (void *) &timeout_us, (int) sizeof(timeout_us));
    if (ret) {
        fprintf(stderr, "cci_set_opt() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

        /* initiate connect */
        ret = cci_connect(endpoint, server_uri, 54321, server_uri,
                      strlen(server_uri), CCI_CONN_ATTR_UU, NULL, 0, NULL);
    if (ret) {
        fprintf(stderr, "cci_connect() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

        /* poll for connect completion */
        while (!done)
        poll_events(endpoint, &connection, &done);

    if (!connection) {
        fprintf(stderr, "no connection\n");
        exit(EXIT_FAILURE);
    }

        /* begin communication with server */
    for (i = 0; i < 10; i++) {
        char hdr[32];
        char data[128];

        memset(hdr, 0, sizeof(hdr));
```

```
        memset(data, 0, sizeof(data));
        sprintf(hdr, "%4d", i);
        sprintf(data, "Hello World!");
        ret = cci_send(connection, hdr, (uint32_t) strlen(hdr),
                       data, (uint32_t) strlen(data), (void *)(uintptr_t) i, 0);
        if (ret)
            fprintf(stderr, "send %d returned %s\n", i, cci_strerror(ret));

        done = 0;
        while (!done)
            poll_events(endpoint, &connection, &done);
    }

        /* clean up */
        ret = cci_disconnect(connection);
    if (ret) {
        fprintf(stderr, "cci_disconnect() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }
        ret = cci_destroy_endpoint(endpoint);
    if (ret) {
        fprintf(stderr, "cci_destroy_endpoint() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }
        ret = cci_free_devices((const cci_device_t ** const)devices);
    if (ret) {
        fprintf(stderr, "cci_free_devices() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

        return 0;
}
```

## 6.2 devices.c

This is an example of using get_devices and free_devices. It also iterates over the conf_argv array.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#include "cci.h"

int main(int argc, char *argv[])
{
    int ret, i = 0;
    uint32_t caps;
    cci_device_t const ** const devices, **d;

    ret = cci_init(CCI_ABI_VERSION, 0, &caps);
    if (ret != CCI_SUCCESS) {
        fprintf(stderr, "cci_init() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    ret = cci_get_devices((cci_device_t const *** const) &devices);
    if (ret != CCI_SUCCESS) {
        fprintf(stderr, "cci_get_devices() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    for (d = devices; *d != NULL; d++) {
        char **keyval;

        printf("device %d is %s\n", i, (*d)->name);
        i++;
        for (keyval = (char **) (*d)->conf_argv; *keyval != NULL; keyval++)
            printf("\t%s\n", *keyval);
    }

    ret = cci_free_devices(devices);
    if (ret != CCI_SUCCESS) {
        fprintf(stderr, "cci_free_devices() returned %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

## 6.3 init.c

This is an example of using init and strerror.

```
#include <stdio.h>
#include <stdint.h>

#include "cci.h"

int main(int argc, char *argv[])
{
    int ret;
    uint32_t caps;

    ret = cci_init(CCI_ABI_VERSION, 0, &caps);
    if (ret != CCI_SUCCESS)
        fprintf(stderr, "cci_init() returned %s\n", cci_strerror(ret));

    return 0;
}
```

## 6.4 server.c

This application demonstrates opening an endpoint, binding to a service, getting connection requests, accepting connections, polling for events, and echoing received messages back to the client.

```
/*
 * Copyright (c) 2011 UT-Battelle, LLC.  All rights reserved.
 * Copyright (c) 2011 Oak Ridge National Labs.  All rights reserved.
 *
 * See COPYING in top-level directory
 *
 * $COPYRIGHT$
 *
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#include "cci.h"

int main(int argc, char *argv[])
{
    int ret;
    uint32_t caps = 0, port = 54321;
    cci_device_t **devices = NULL;
    cci_endpoint_t *endpoint = NULL;
    cci_os_handle_t ep_fd, bind_fd;
    cci_service_t *service = NULL;
    cci_connection_t *connection = NULL;

    /* init */
    ret = cci_init(CCI_ABI_VERSION, 0, &caps);
    if (ret) {
        fprintf(stderr, "cci_init() failed with %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    /* get devices */
    ret = cci_get_devices((cci_device_t const *** const) &devices);
    if (ret) {
        fprintf(stderr, "cci_get_devices() failed with %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    /* create an endpoint */
    ret = cci_create_endpoint(NULL, 0, &endpoint, &ep_fd);
    if (ret) {
        fprintf(stderr, "cci_create_endpoint() failed with %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    /* bind first device to the service at port 54321 */
    ret = cci_bind(devices[0], 10, &port,&service, &bind_fd);
```

```
    if (ret) {
        fprintf(stderr, "cci_bind() failed with %s\n", cci_strerror(ret));
        exit(EXIT_FAILURE);
    }

    while (1) {
        int accept = 1;
        char *buffer;
        cci_conn_req_t *conn_req;
        cci_event_t *event;

        ret = cci_get_conn_req(service, &conn_req);
        if (ret == CCI_SUCCESS) {
            /* inspect conn_req_t and decide to accept or reject */

            if (accept) {
                /* associate this connect request with this endpoint */
                ret = cci_accept(conn_req, endpoint, &connection);
                if (ret != CCI_SUCCESS) {
                    fprintf(stderr, "cci_accept() returned %s",
                                    cci_strerror(ret));
                } else if (!buffer) {
                    buffer = calloc(1, connection->max_send_size + 1);
                    /* check for buffer ... */
                }

            } else {
                cci_reject(conn_req);
            }
        }

        /* check for next event...
         * handle communication over existing connections */

again:
        ret = cci_get_event(endpoint, &event, 0);
        if (ret == CCI_SUCCESS) {
            switch (event->type) {
                case CCI_EVENT_RECV:
                {
                    memcpy(buffer, event->info.recv.header_ptr, event->info.recv.header_len);
                    buffer[event->info.recv.header_len] = 0;
                    printf("recv'd:\n");
                    printf("\theader: \"%s\"\n", buffer);
                    memcpy(buffer, event->info.recv.data_ptr, event->info.recv.data_len);
                    buffer[event->info.recv.data_len] = 0;
                    printf("\tdata: \"%s\"\n", buffer);

                    /* echo the message to the client */
                    ret = cci_send(connection,
                                   event->info.recv.header_ptr,
                                   event->info.recv.header_len,
                                   event->info.recv.data_ptr,
                                   event->info.recv.data_len,
                                   NULL, 0);
                    if (ret != CCI_SUCCESS)
                        fprintf(stderr, "send returned %s\n", cci_strerror(ret));
```

```
                    break;
                }
                case CCI_EVENT_SEND:
                    printf("completed send\n");
                    break;
                default:
                    fprintf(stderr, "unexpected event %d", event->type);
                    break;
            }
            cci_return_event(endpoint, event);
            goto again;
        }
    }

    /* clean up */
    cci_unbind(service, NULL);
    cci_destroy_endpoint(endpoint);
    cci_free_devices((cci_device_t const **) devices);

    return 0;
}
```

# Index