CCI

2.0.0

Generated by Doxygen 1.8.3.1

Tue Jun 28 2016 13:58:50

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 Scalability

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 connections. 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 then optionally calls 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_finalize().

1.3.2 Communication Endpoints

All communication in CCI revolves around an endpoint. A single endpoint can communicate with any number of peers.

Each endpoint has some number of device-sized buffers available for sending and receiving 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().

The endpoint provides a context pointer for the application to use. The application may use the context pointer to provide access to additional state allocated by the application related to that endpoint.

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 many event types: CCI_EVENT_SEND, CCI_EVENT_RECV, CCI_EVENT_CONNECT_REQUEST, etc.

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, the connection attribute, and a context pointer.

As mentioned above, some applications may need reliable delivery while others 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). Typically, most networks can provide higher performance for unordered versus ordered connections.

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 provides a client/server semantic for connection establishment. Every open endpoint is able to initiate and receive connection requests.

To initiate a connection, the client calls cci_connect() with parameters including an endpoint, a string URI for the server, 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 polls for events which may include connection requests. When a connection request event is returned, it includes 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 will initiate the accept portion of the connection handshake. When the handshake is complete, the server will get a CCI_EVENT_ACCEPT with a status of CCI_SUCCESS and the new connection pointer or the status will indicate why the accept failed and the connection pointer will not be valid. The client gets an CCI_EVENT_CONNECT event with a status of CCI_SUCCESS, the context passed to cci_connect(), and the new connection pointer. If the server calls cci_reject(), the client gets a CCI_EVENT_CONNECT event with a status of CCI_ECONNREFUSED and the context passed to cci_connect(). On the server, the connection request event must then be returned using cci_return_event() just like every other event. If the server does not reply within the timeout set in the client's cci_connect(), the client gets an CCI_EVENT_CONNECT event with a status of CCI_ETIMEDOUT and the context passed to cci_connect(). When a process no longer needs a connection, it can call cci_disconnect().

1.3.6 Messages

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

CCI MSGs 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 MSG size limited to a single MTU vastly simplifies the requirements for message completion — either it arrives or it does not.

On receipt of a MSG, 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, a data pointer and length, an application context pointer, and flags. The pointer may be NULL. 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 a pointer to the data, its length, 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 MSGs.

1.3.7 Remote Memory Access

Clearly, MSGs 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 a pointer to the endpoint, the start of the region to be registered, the length of the region, and flags indicating if CCI should READ or WRITE or both access. The function returns a RMA handle. 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 MSGs.

The cci_rma() call takes the connection pointer, an optional MSG 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 MSG pointer and length are set, the initiator will send a completion message to the target that arrives as an MSG with the data.

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 MSG 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

Macros

- #define CCI_ABI_VERSION 2

  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_ENETDOWN = ENETDOWN, CCI_EBUSY = EBUSY, CCI_ERANGE = ERANGE,
  CCI_EAGAIN = EAGAIN, CCI_ENOBUFS = ENOBUFS, CCI_EMSGSIZE = EMSGSIZE, CCI_ENOMSG = ENOMSG,
  CCI_EADDRNOTAVAIL = EADDRNOTAVAIL, CCI_ECONNREFUSED = ECONNREFUSED }

  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 (cci_endpoint_t *endpoint, enum cci_status status)

  Returns a string corresponding to a CCI status enum.

4.1.1 Detailed Description

4.1.2 Macro Definition Documentation

4.1.2.1 #define CCI_ABI_VERSION 2

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.

4.1.3 Typedef Documentation

4.1.3.1 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>.

These status codes may be stringified with cci_strerror().

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

4.1.4 Enumeration Type Documentation

4.1.4.1 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>.

These status codes may be stringified with cci_strerror().

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.

For a connection request, this error code means that the initiator did not get anything back from the target within a timeout. It is unknown whether the target received the request and ignored it, did not receive it at all, or receive it too late.

CCI_ENOMEM No more memory.

CCI_ENODEV No device available.

CCI_ENETDOWN The requested device is down.

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.

CCI_ECONNREFUSED Connection request rejected.

4.1.5 Function Documentation

4.1.5.1 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

in	*abi_ver,:*	A constant describing the ABI version that this application requires (one of the CCI_ABI_* values).
in	*flags,:*	A constant describing behaviors that this application requires. Currently, 0 is the only valid value.
out	*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 transports or CCI_CONFIG.

CCI_ERROR Unable to parse CCI_CONFIG.
Errno if fopen() fails.
Each transport may have additional error codes.

If cci_init() completes successfully, then CCI is loaded and available to be used in this application. The application must call cci_finalize() to free all CCI resources at the end of its execution.

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.

4.1.5.2 CCI_DECLSPEC const char* cci_strerror (cci_endpoint_t * *endpoint,* enum cci_status *status*)

Returns a string corresponding to a CCI status enum.

Parameters

in	*endpoint,:*	The CCI endpoint that returned this status, NULL if none applicable.
in	*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 ∗∗devices)

  Get an array of devices.

4.2.1 Detailed Description

4.2.2 Typedef Documentation

4.2.2.1 typedef struct cci_device cci_device_t

Structure representing one CCI device.

4.2.3 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 "transport".  It indicates
# which CCI transport should be applied to this device.
transport = 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 transport.  The transport can do whatever it wants with
# these values (e.g., system admins can set values to configure the
# transport).  transport 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
# transport; 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]
transport = psm
device = ipath0

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

# storage is a device that uses the UDP transport.  Note that this transport
# 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]
transport = 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.

4.2.4 Function Documentation

4.2.4.1 CCI_DECLSPEC int cci_get_devices (cci_device_t *const ** *devices*)

Get an array of devices.

Returns a NULL-terminated array of (struct cci_device *)'s. 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

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

Returns

CCI_SUCCESS The array of devices is available.
CCI_EINVAL Devices is NULL.
Each transport 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.

If cci_get_devices() is invoked again later, it may return a larger array if some new devices appeared in the system. All previously returned devices are guaranteed to be in the new array, but their status may have changed. For instance, if the corresponding physical devices is not available anymore, the CCI device will have its "up" field unset.

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:

devices.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_create_endpoint_at (cci_device_t *device, const char *service, int flags, cci_endpoint_t **endpoint, cci_os_handle_t *fd)

  Create an endpoint bound to a specified service.
- CCI_DECLSPEC int cci_destroy_endpoint (cci_endpoint_t *endpoint)

  Destroy an endpoint.

4.3.1 Detailed Description

4.3.2 Typedef Documentation

4.3.2.1 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.

4.3.2.2 typedef struct cci_endpoint cci_endpoint_t

Endpoint.

4.3.2.3 typedef int cci_os_handle_t

OS-native handles.

4.3.3 Enumeration Type Documentation

4.3.3.1 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.

4.3.4 Function Documentation

4.3.4.1 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

in	*device,:*	A pointer to a device that was returned via cci_get_devices() or NULL.
in	*flags,:*	Flags specifying behavior of this endpoint.
out	*endpoint,:*	A handle to the endpoint that was created.
out	*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_ENODEV Device is NULL and no CCI device is available.
CCI_ENOMEM Unable to allocate enough memory.
Each transport 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), the application should bind the calling thread before calling cci_create_endpoint().

Advice to users: to set the send/receive buffer count on the endpoint, call cci_set|get_opt() after creating the endpoint with the applicable options.

Examples:

client.c, and server.c.

4.3.4.2 CCI_DECLSPEC int cci_create_endpoint_at (cci_device_t * *device,* const char * *service,* int *flags,* cci_endpoint_t ** *endpoint,* cci_os_handle_t * *fd*)

Create an endpoint bound to a specified service.

Parameters

in	*device,:*	A pointer to a device that was returned via cci_get_devices() or NULL.
in	*service,:*	Device-specific service hint.
in	*flags,:*	Flags specifying behavior of this endpoint.
out	*endpoint,:*	A handle to the endpoint that was created.
out	*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 Device, endpoint, or fd is NULL.
CCI_ENODEV Device is not "up".
CCI_ENOMEM Unable to allocate enough memory.
Each transport may have additional error codes.

This function creates a CCI endpoint bound to a specified service. 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 must be a pointer that was returned by cci_get_devices() to indicate that a specific device should be used for this endpoint.

The service argument is a string that provides a device-specific hint about how to bind the endpoint. Most transports use AF_INET sockets, which use a port for the service. For these transports, pass the requested port as a string. The shared memory (SM) transport, however, uses AF_UNIX, which uses a path. See README.ctp.sm for instructions on how to create a service string.

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

cci_create_endpoint_at() 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), the application should bind the calling thread before calling cci_create_endpoint_at().

Advice to users: to set the send/receive buffer count on the endpoint, call cci_set|get_opt() after creating the endpoint with the applicable options.

4.3.4.3 CCI_DECLSPEC int cci_destroy_endpoint (cci_endpoint_t * *endpoint*)

Destroy an endpoint.

Parameters

in	*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 transport may have additional error codes.

Successful completion of this function makes all data structures and state associated with the endpoint stale (including the OS handle, connections, events, event buffers, and RMA registrations). 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_connection

  Connection handle.

Macros

- #define CCI_CONN_REQ_LEN (1024) /∗ see above ∗/

  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_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_accept (cci_event_t ∗conn_req, const void ∗context)

  Accept a connection request.
- CCI_DECLSPEC int cci_reject (cci_event_t ∗conn_req)

  Reject a connection request.
- CCI_DECLSPEC int cci_connect (cci_endpoint_t ∗endpoint, const char ∗server_uri, const void ∗data_ptr, uint32_t data_len, cci_conn_attribute_t attribute, const void ∗context, int flags, const 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 Detailed Description

4.4.2 Macro Definition Documentation

4.4.2.1 #define CCI_CONN_REQ_LEN (1024) /∗ see above ∗/

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

4.4.3 Typedef Documentation

4.4.3.1 typedef enum cci_conn_attribute cci_conn_attribute_t

Connection request attributes.

CCI provides optional reliability and ordering to meet the varying needs of applications.

Unreliable connections are always unordered (Unreliable/Unordered or UU). UU connections may be unicast or multicast. UU connections offer no delivery guarantees; messages may arrive once, multiple times or never. UU connections have no timeout.

UU 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.

UU connections (unicast or multicast) only support messages (see Communications below).

Reliable connections may be ordered (Reliable/Ordered or RO) or unordered (Reliable/Unordered or RU). Reliable connections are unicast only. 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.

For reliable connections, RU connections allow the most aggressive optimization of the underlying network(s) to provide better performance. RO connections will reduce performance on most networks.

Reliable connections support both messages and remote memory access (see Communications below).

4.4.3.2 typedef struct cci_connection cci_connection_t

Connection handle.

4.4.4 Enumeration Type Documentation

4.4.4.1 enum cci_conn_attribute

Connection request attributes.

CCI provides optional reliability and ordering to meet the varying needs of applications.

Unreliable connections are always unordered (Unreliable/Unordered or UU). UU connections may be unicast or multicast. UU connections offer no delivery guarantees; messages may arrive once, multiple times or never. UU connections have no timeout.

UU 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.

UU connections (unicast or multicast) only support messages (see Communications below).

Reliable connections may be ordered (Reliable/Ordered or RO) or unordered (Reliable/Unordered or RU). Reliable connections are unicast only. 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.

For reliable connections, RU connections allow the most aggressive optimization of the underlying network(s) to provide better performance. RO connections will reduce performance on most networks.

Reliable connections support both messages and remote memory access (see Communications below).

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)

4.4.5 Function Documentation

4.4.5.1 CCI_DECLSPEC int cci_accept (cci_event_t * *conn_req*, const void * *context*)

Accept a connection request.

Parameters

in	*conn_req*	A connection request event previously returned by cci_get_event().
in	*context*	Cookie to be used to identify the connection in incoming events.

Returns

CCI_SUCCESS CCI has started completing the connection handshake.
CCI_EINVAL The event is not a connection request or it has already been accepted or rejected.
Each transport may have additional error codes.

Upon success, CCI will attempt to complete the connection handshake. Once completed, CCI will return a CCI_EVENT_ACCEPT event. If successful, the event will contain a pointer to the new connection. It will always contain the context passed in here.

The connection request event must still be returned to CCI via cci_return_event().

Examples:

server.c.

4.4.5.2 CCI_DECLSPEC int cci_reject (cci_event_t * *conn_req*)

Reject a connection request.

Parameters

in	*conn_req*	Connection request event to reject.

Returns

CCI_SUCCESS Connection request has been rejected.
CCI_EINVAL The event is not a connection request or it has already been accepted or rejected.
Each transport may have additional error codes.

Rejects an incoming connection request. The connection request event must still be returned to CCI via cci_return_event().

Examples:

server.c.

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

Initiate a connection request (client side).

Request a connection from a specific endpoint. The server endpoint's 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

in	*endpoint*	Local endpoint to use for requested connection.
in	*server_uri*	Uniform Resource Identifier of the server and is generated by the server's endpoint when it is created.
in	*data_ptr*	Pointer to connection data to be sent in the connection request (for authentication, etc).
in	*data_len*	Length of connection data. Implementations must support data_len values <= 1,024 bytes.
in	*attribute*	Attributes of the requested connection (reliability, ordering, multicast, etc).
in	*context*	Cookie to be used to identify the completion through a connect accepted, rejected, or failed event, and used to identify the connection in incoming events.
in	*flags*	Currently unused.
in	*timeout*	NULL means forever.

Returns

CCI_SUCCESS The request is buffered and ready to be sent or has been sent.
CCI_EINVAL data_len is strictly larger than CCI_CONN_REQ_LEN.
Each transport may have additional error codes.

Examples:

client.c.

4.4.5.4 CCI_DECLSPEC int cci_disconnect (cci_connection_t * *connection*)

Tear down an existing connection.

Operation is local, remote side is not notified. Any future attempt to use the connection will result in undefined behavior.

Parameters

in	*connection*	Connection to server.

Returns

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

Examples:

client.c.

4.5 Events

Data Structures

- struct cci_event_send

  Send event.
- struct cci_event_recv

  Receive event.
- struct cci_event_connect

  Connect request completion event.
- struct cci_event_connect_request

  Connection request event.
- struct cci_event_accept

  Accept completion event.
- struct cci_event_keepalive_timedout

  Keepalive timeout event.
- struct cci_event_endpoint_device_failed

  Endpoint device failed event.
- union cci_event

  Generic event.

Typedefs

- typedef enum cci_event_type cci_event_type_t

  Event types.
- 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_connect cci_event_connect_t

  Connect request completion event.
- typedef struct cci_event_connect_request cci_event_connect_request_t

  Connection request event.
- typedef struct cci_event_accept cci_event_accept_t

  Accept completion event.
- typedef struct cci_event_keepalive_timedout cci_event_keepalive_timedout_t

  Keepalive timeout event.
- typedef struct cci_event_endpoint_device_failed cci_event_endpoint_device_failed_t

  Endpoint device failed event.

Enumerations

- enum cci_event_type {
  CCI_EVENT_NONE, CCI_EVENT_SEND, CCI_EVENT_RECV, CCI_EVENT_CONNECT,
  CCI_EVENT_CONNECT_REQUEST, CCI_EVENT_ACCEPT, CCI_EVENT_KEEPALIVE_TIMEDOUT, CCI_EVENT_ENDPOINT_DEVICE_FAILED }

  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 **event)

  Get the next available CCI event.
- CCI_DECLSPEC int cci_return_event (cci_event_t *event)

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

4.5.1 Detailed Description

4.5.2 Typedef Documentation

4.5.2.1 typedef enum cci_event_type cci_event_type_t

Event types.

Each event has a unique type and the first element is always the event type. A detailed description of each event is provided with the event structure.

The CCI_EVENT_NONE event type is never passed to the application and is for internal CCI use only.

4.5.2.2 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 (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.

4.5.2.3 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.

4.5.2.4 typedef struct cci_event_connect cci_event_connect_t

Connect request completion event.

The status field may contain the following values:

CCI_SUCCESS The connection was accepted by the server and successfully established. The corresponding connection structure is available in the event connection field.

CCI_ECONNREFUSED The server rejected the connection request.

CCI_ETIMEDOUT The connection could not be established before the timeout expired.

Some transports may also return specific return codes.

The connection field is only valid for use if status is CCI_SUCCESS. It is set to NULL in any other case.

The context field is always set to what was passed to cci_connect(). On success, the connection structure context field is also set accordingly.

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.

4.5.2.5 typedef struct cci_event_connect_request cci_event_connect_request_t

Connection request event.

An incoming connection request from a client. It includes the requested connection attributes (reliability and ordering) and an optional payload.

The number of fields in this struct is intentionally limited in order to reduce costs associated with state storage, caching, updating, copying.

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

This event should be passed to either cci_accept() or cci_reject() before being returned with cci_return_event().

4.5.2.6 typedef struct cci_event_accept cci_event_accept_t

Accept completion event.

The status field may contain the following values:

CCI_SUCCESS The accepted connection was successfully established. The corresponding connection structure is available in the event connection field.

Some transports may also return specific return codes.

The connection field is only valid for use if status is CCI_SUCCESS. It is set to NULL in any other case.

The context field is always set to what was passed to cci_accept(). On success, the connection structure context field is

also set accordingly.

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.

4.5.2.7 typedef struct cci_event_keepalive_timedout cci_event_keepalive_timedout_t

Keepalive timeout event.

We were unable to send a periodic message to the peer. The application can attempt communication or disconnect. The connection will continue to consume resources until the application calls cci_disconnect().

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.

4.5.2.8 typedef struct cci_event_endpoint_device_failed cci_event_endpoint_device_failed_t

Endpoint device failed event.

The endpoint's device has failed.

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.

4.5.3 Enumeration Type Documentation

4.5.3.1 enum cci_event_type

Event types.

Each event has a unique type and the first element is always the event type. A detailed description of each event is provided with the event structure.

The CCI_EVENT_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 A message has been received.

CCI_EVENT_CONNECT An outgoing connection request has completed.

CCI_EVENT_CONNECT_REQUEST An incoming connection request from a client.

CCI_EVENT_ACCEPT An incoming connection accept has completed.

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

CCI_EVENT_ENDPOINT_DEVICE_FAILED A device on this endpoint has failed.

4.5.4 Function Documentation

4.5.4.1 CCI_DECLSPEC int cci_arm_os_handle (cci_endpoint_t * *endpoint*, int *flags*)

4.5.4.2 CCI_DECLSPEC int cci_get_event (cci_endpoint_t * *endpoint*, cci_event_t ** *event*)

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, connection request, etc.). If the application wants to block, it should pass the OS handle to the 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 behavior of the OS handle when used with the OS blocking mechanism is to return the equivalent of a POLLIN which indicates that the application should call cci_get_event(). It does not, however, guarantee that cci_get_event() will return something other than CCI_EAGAIN. For example, the application has two threads: one blocking on the OS handle and another calling cci_get_event(). By the time the blocking thread wakes and calls cci_get_event(), the other thread may have already reaped all the queued events. Note, the application must never directly read from or write to the OS handle. The results are undefined.

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

Parameters

in	*endpoint*	Endpoint to poll for a new event.
in	*event*	New event, if any.

Returns

CCI_SUCCESS An event was retrieved.
CCI_EAGAIN No event is available.
CCI_ENOBUFS No event is available and there are no available receive buffers. The application must return events before any more messages can be received.
Each transport may have additional error codes.

To discuss:

- 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.

4.5.4.3 CCI_DECLSPEC int cci_return_event (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

in	*event*	Event to return.

Returns

CCI_SUCCESS The event was returned to CCI.
CCI_EINVAL The event is a connection request and it has not been passed to cci_accept() or cci_reject().
Each transport may have additional error codes.

Examples:

client.c, and server.c.

4.6 Endpoint / Connection Options

Typedefs

- typedef enum cci_opt_name cci_opt_name_t

  Name of options.

Enumerations

- enum cci_opt_name {
  CCI_OPT_ENDPT_SEND_TIMEOUT, CCI_OPT_ENDPT_RECV_BUF_COUNT, CCI_OPT_ENDPT_SEND_BUF_COUNT, CCI_OPT_ENDPT_KEEPALIVE_TIMEOUT,
  CCI_OPT_ENDPT_URI, CCI_OPT_ENDPT_RMA_ALIGN, CCI_OPT_CONN_SEND_TIMEOUT, CCI_OPT_CONN_KEEPALIVE_TIMEOUT }

  Name of options.

Functions

- CCI_DECLSPEC int cci_set_opt (cci_opt_handle_t *handle, cci_opt_name_t name, const void *val)

  Set an endpoint or connection option value.
- CCI_DECLSPEC int cci_get_opt (cci_opt_handle_t *handle, cci_opt_name_t name, void *val)

  Get an endpoint or connection option value.

4.6.1 Detailed Description

4.6.2 Typedef Documentation

4.6.2.1 typedef enum cci_opt_name cci_opt_name_t

Name of options.

4.6.3 Enumeration Type Documentation

4.6.3.1 enum cci_opt_name

Name of options.

Enumerator

CCI_OPT_ENDPT_SEND_TIMEOUT Default send timeout for all new connections. cci_get_opt() and cci_set_opt().

The parameter must point to a uint32_t.

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().

The parameter must point to a uint32_t.

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().

The parameter must point to a uint32_t.

CCI_OPT_ENDPT_KEEPALIVE_TIMEOUT Send a periodic message over each reliable connection on the endpoint. Sending keepalive messages can determine if a peer has silently disconnected. The CCI transport will periodically send a message over each connection. If the transport determines that the message was successfully received, it will repeat at the next period. If the transport determines that the message was not successfully delivered, it will raise the CCI_EVENT_KEEPALIVE_TIMEDOUT event on the connection and the keepalive timeout for that connection is set to 0 (i.e. disabled).

If a keepalive event is raised, the connection is *not* disconnected. Recovery decisions are up to the application; it may choose to disconnect the connection, re-arm the keepalive timeout, send a MSG or RMA, etc. The application may "re-arm" the keepalive timeout for the connection individually using CCI_OPT_CONN_KEEPALIVE_TIMEOUT or re-arm all connections with this option.

The keepalive timeout is expressed in microseconds. The default is 0 (i.e. disabled). Using this option enables the same timeout on all connections, currently opened and those opened in the future.

The messages are sent internally within CCI and are never visible to the application either locally or at the peer. Using keepalives may cause spurious wake ups when using the OS handle for blocking.

cci_get_opt() and cci_set_opt().

The parameter must point to a uint32_t.

CCI_OPT_ENDPT_URI Retrieve the endpoint's URI used for listening for connection requests. The application should never need to parse this URI.

```
cci_get_opt() only.

The parameter must point to a char *.
The application is responsible for freeing the pointer that is
stored in this char *.
```

CCI_OPT_ENDPT_RMA_ALIGN RMA registration alignment requirements, if any, for this endpoint. This option needs the address of a cci_alignment_t pointer passed in. The CTP will allocate and fill in the struct with the minimal alignment needed for each member of the struct. A value of 0 indicates that there are no alignment requirements for that member. A value of 4, for example, indicates that that member must be 4-byte aligned.

If the CTP requires RMA alignment and the application passes in an un-aligned parameter, the CTP may need to allocate a temporary buffer, register it, and use it instead. This will also require a copy of the data to the correct location. This will decrease performance for these cases.

cci_get_opt() only.

The parameter must point to a cci_alignment_t.

CCI_OPT_CONN_SEND_TIMEOUT Reliable send timeout in microseconds. cci_get_opt() and cci_set_opt().

The parameter must point to a uint32_t.

CCI_OPT_CONN_KEEPALIVE_TIMEOUT Send a periodic message over this reliable connection. This option is similar to CCI_OPT_ENDPT_KEEPALIVE_TIMEOUT except that it modifies the keepalive timeout on a single connection only. The application may use it to re-arm a connection that has raised a CCI_EVENT_KEEPALIVE_TIMEDOUT, to selectively arm only some connections, or to set a timeout different from the endpoint's keepalive timeout period.

cci_get_opt() and cci_set_opt().

The parameter must point to a uint32_t.

4.6.4 Function Documentation

4.6.4.1 CCI_DECLSPEC int cci_set_opt (cci_opt_handle_t * *handle*, cci_opt_name_t *name*, const void * *val*)

Set an endpoint or connection option value.

Parameters

in	*handle*	Endpoint or connection handle.
in	*name*	Which option to set the value of.
in	*val*	Pointer to the input value. The type of the value must match the option name.

Depending on the value of name, handle must be a cci_endpoint_t* or a cci_connection_t*.

Returns

CCI_SUCCESS Value successfully set.
CCI_EINVAL Handle or val is NULL.
CCI_EINVAL Trying to set a get-only option.
CCI_ERR_NOT_IMPLEMENTED Not supported by this transport.
Each transport may have additional error codes.

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

Examples:

client.c.

4.6.4.2 CCI_DECLSPEC int cci_get_opt (cci_opt_handle_t * *handle*, cci_opt_name_t *name*, void * *val*)

Get an endpoint or connection option value.

Parameters

in	*handle*	Endpoint or connection handle.
in	*name*	Which option to get the value of.
in	*val*	Pointer to the output value. The type of the value must match the option name.

Depending on the value of name, handle must be a cci_endpoint_t* or a cci_connection_t*.

Returns

CCI_SUCCESS Value successfully retrieved.
CCI_EINVAL Handle or val is NULL.
CCI_ERR_NOT_IMPLEMENTED Not supported by this transport.
Each transport may have additional error codes.

Examples:

client.c, and server.c.

4.7 Communications

Functions

- CCI_DECLSPEC int cci_send (cci_connection_t *connection, const void *msg_ptr, uint32_t msg_len, const void *context, int flags)

  Send a short message.
- CCI_DECLSPEC int cci_sendv (cci_connection_t *connection, const struct iovec *data, uint32_t iovcnt, const void *context, int flags)

  Send a short vectored (gather) message.
- CCI_DECLSPEC int cci_rma_register (cci_endpoint_t *endpoint, void *start, uint64_t length, int flags, cci_rma_handle_t **rma_handle)

  Register memory for RMA operations.
- CCI_DECLSPEC int cci_rma_deregister (cci_endpoint_t *endpoint, cci_rma_handle_t *rma_handle)

  Deregister memory.
- CCI_DECLSPEC int cci_rma (cci_connection_t *connection, const void *msg_ptr, uint32_t msg_len, cci_rma_handle_t *local_handle, uint64_t local_offset, cci_rma_handle_t *remote_handle, uint64_t remote_offset, uint64_t data_len, const void *context, int flags)

  Perform a RMA operation between local and remote memory on a valid connection.

4.7.1 Detailed Description

4.7.2 Function Documentation

4.7.2.1 CCI_DECLSPEC int cci_send (cci_connection_t * *connection*, const void * *msg_ptr*, uint32_t *msg_len*, const void * *context*, int *flags*)

Send a short message.

A short message limited to the size of cci_connection::max_send_size, which may be lower than the cci_device::max_send_size.

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

in	*connection*	Connection (destination/reliability/ordering).
in	*msg_ptr*	Pointer to local segment.
in	*msg_len*	Length of local segment (limited to max send size).
in	*context*	Cookie to identify the completion through a Send event when non-blocking.
in	*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.
Each transport 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 to close the connection.

Note, using both CCI_FLAG_NO_COPY and CCI_FLAG_SILENT is only allowed on RO connections.

Examples:

client.c, and server.c.

4.7.2.2 CCI_DECLSPEC int cci_sendv (cci_connection_t * *connection,* const struct iovec * *data,* uint32_t *iovcnt,* const 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

in	*connection*	Connection (destination/reliability).
in	*data*	Array of local data buffers.
in	*iovcnt*	Count of local data array.
in	*context*	Cookie to identify the completion through a Send event when non-blocking.
in	*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.
Each transport may have additional error codes.

4.7.2.3 CCI_DECLSPEC int cci_rma_register (cci_endpoint_t * *endpoint,* void * *start,* uint64_t *length,* int *flags,* cci_rma_handle_t ** *rma_handle*)

Register memory for RMA operations.

Prior to accessing memory using RMA, the application must register the memory with an endpoint. Memory registered with one endpoint may not be accessed via another endpoint, unless also registered with that endpoint (i.e. an endpoint serves as a protection domain).

Registration may take awhile depending on the underlying device and should not be in the critical path.

It is allowable to have overlapping registrations.

Parameters

in	*endpoint*	Local endpoint to use for RMA.
in	*start*	Pointer to local memory.
in	*length*	Length of local memory.
in	*flags*	Optional flags: • CCI_FLAG_READ: Local memory may be read from other endpoints. • CCI_FLAG_WRITE: Local memory may be written by other endpoints.
out	*rma_handle*	Handle for use with cci_rma().

flags may be 0 if this handle will never be accessed by any other endpoint.

Returns

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

4.7.2.4 CCI_DECLSPEC int cci_rma_deregister (cci_endpoint_t * *endpoint*, cci_rma_handle_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

in	*endpoint*	Local endpoint to use for RMA.
in	*rma_handle*	Handle for use with cci_rma().

Returns

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

4.7.2.5 CCI_DECLSPEC int cci_rma (cci_connection_t * *connection*, const void * *msg_ptr*, uint32_t *msg_len*, cci_rma_handle_t * *local_handle*, uint64_t *local_offset*, cci_rma_handle_t * *remote_handle*, uint64_t *remote_offset*, uint64_t *data_len*, const void * *context*, int *flags*)

Perform a RMA operation between local and remote memory on a valid connection.

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 on the same connection 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 msg_ptr and msg_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.

In an ordered connection, RMA completion events are ordered according to the ordering of the cci_send() or cci_rma() calls in the local peer. For instance a cci_rma() with completion message posted between two cci_send() will generate a completion event on the target between the receive events of these sends.

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

A local completion will be generated. If a completion message is provided, then a remote completion will be generated as well.

remote_handle must have been created with protection flags that match the flags passed in cci_rma() here. local_handles does not need any protection flag since it is only accessed locally here.

RMA requires a valid connection (i.e. open on both sides). If the remote peer has called disconnect(), any attempt to RMA to that peer using the half closed connection should fail.

Parameters

in	*connection*	Connection (destination).
in	*msg_ptr*	Pointer to data for the remote completion.
in	*msg_len*	Length of data for the remote completion.
in	*local_handle*	Handle of the local RMA area.
in	*local_offset*	Offset in the local RMA area.
in	*remote_handle*	Handle of the remote RMA area.
in	*remote_offset*	Offset in the remote RMA area.
in	*data_len*	Length of data segment.
in	*context*	Cookie to identify the completion through a Send event when non-blocking.
in	*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 on the same connection 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 data_len is 0.
CCI_EINVAL Both READ and WRITE flags are set.
CCI_EINVAL Neither the READ or WRITE flag is set.
CCI_ERR_DISCONNECTED The remote peer has closed the connection.
Each transport 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_alignment Struct Reference

`#include <cci.h>`

Data Fields

- uint32_t rma_write_local_addr
  WRITE local_handle->start + offset.
- uint32_t rma_write_remote_addr
  WRITE remote_handle->start + offset.
- uint32_t rma_write_length
  WRITE length.
- uint32_t rma_read_local_addr
  READ local_handle->start + offset.
- uint32_t rma_read_remote_addr
  READ remote_handle->start + offset.
- uint32_t rma_read_length
  READ length.

5.1.1 Field Documentation

5.1.1.1 uint32_t cci_alignment::rma_write_local_addr

WRITE local_handle->start + offset.

5.1.1.2 uint32_t cci_alignment::rma_write_remote_addr

WRITE remote_handle->start + offset.

5.1.1.3 uint32_t cci_alignment::rma_write_length

WRITE length.

5.1.1.4 uint32_t cci_alignment::rma_read_local_addr

READ local_handle->start + offset.

5.1.1.5 uint32_t cci_alignment::rma_read_remote_addr

READ remote_handle->start + offset.

5.1.1.6 uint32_t cci_alignment::rma_read_length

READ length.

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.
- void * context

  Application-provided, private context.

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.

5.2.2.4 void* cci_connection::context

Application-provided, private context.

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 * transport

  Name of the device driver, e.g., "sock" or "verbs".
- unsigned up

  Is this device actually up and running?
- 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 *const * 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

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 "transport".  It indicates
# which CCI transport should be applied to this device.
transport = 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 transport.  The transport can do whatever it wants with
# these values (e.g., system admins can set values to configure the
# transport).  transport 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
# transport; 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]
transport = psm
device = ipath0

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

# storage is a device that uses the UDP transport.  Note that this transport
# 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]
```

```
transport = 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:

devices.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::transport

Name of the device driver, e.g., "sock" or "verbs".

5.3.3.3 unsigned cci_device::up

Is this device actually up and running?

5.3.3.4 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.

5.3.3.5 const char* const* cci_device::conf_argv

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

5.3.3.6 uint32_t cci_device::max_send_size

Maximum send size supported by the device.

5.3.3.7 uint64_t cci_device::rate

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

0 if unknown.

5.3.3.8 uint32_t cci_device::domain

5.3.3.9 uint32_t cci_device::bus

5.3.3.10 uint32_t cci_device::dev

5.3.3.11 uint32_t cci_device::func

5.3.3.12 struct { ... } cci_device::pci

5.4 cci_endpoint Struct Reference

Endpoint.

`#include <cci.h>`

Data Fields

- cci_device_t * device

  Device that runs this endpoint.

5.4.1 Detailed Description

Endpoint.

Examples:

client.c, and server.c.

5.4.2 Field Documentation

5.4.2.1 cci_device_t* cci_endpoint::device

Device that runs this endpoint.

5.5 cci_event Union Reference

Generic event.

`#include <cci.h>`

Data Fields

- cci_event_type_t type
- cci_event_send_t send
- cci_event_recv_t recv
- cci_event_connect_t connect
- cci_event_connect_request_t request
- cci_event_accept_t accept

- cci_event_keepalive_timedout_t keepalive
- cci_event_endpoint_device_failed_t dev_failed

5.5.1 Detailed Description

Generic event.

This is union of all events and the event type. Each event must start with the type as well. The application can simply look at the event as a type to determine how to handle it.

Examples:

client.c, and server.c.

5.5.2 Field Documentation

5.5.2.1 cci_event_type_t cci_event::type

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_connect_t cci_event::connect

Examples:

client.c.

5.5.2.5 cci_event_connect_request_t cci_event::request

5.5.2.6 cci_event_accept_t cci_event::accept

5.5.2.7 cci_event_keepalive_timedout_t cci_event::keepalive

5.5.2.8 cci_event_endpoint_device_failed_t cci_event::dev_failed

5.6 cci_event_accept Struct Reference

Accept completion event.

`#include <cci.h>`

Data Fields

- cci_event_type_t type

  Type of event - should equal CCI_EVENT_ACCEPT.
- cci_status_t status

  Result of the accept.
- void * context

  The context that was passed to cci_accept()
- cci_connection_t * connection

  The new connection, if the request successfully completed.

5.6.1 Detailed Description

Accept completion event.

The status field may contain the following values:

CCI_SUCCESS The accepted connection was successfully established. The corresponding connection structure is available in the event connection field.

Some transports may also return specific return codes.

The connection field is only valid for use if status is CCI_SUCCESS. It is set to NULL in any other case.

The context field is always set to what was passed to cci_accept(). On success, the connection structure context field is also set accordingly.

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.

5.6.2 Field Documentation

5.6.2.1 cci_event_type_t cci_event_accept::type

Type of event - should equal CCI_EVENT_ACCEPT.

5.6.2.2 cci_status_t cci_event_accept::status

Result of the accept.

5.6.2.3 void* cci_event_accept::context

The context that was passed to cci_accept()

5.6.2.4 cci_connection_t* cci_event_accept::connection

The new connection, if the request successfully completed.

5.7 cci_event_connect Struct Reference

Connect request completion event.

`#include <cci.h>`

Data Fields

- cci_event_type_t type

  Type of event - should equal CCI_EVENT_CONNECT.
- cci_status_t status

  Result of the connect request.
- void * context

  Context value that was passed to cci_connect()
- cci_connection_t * connection

  The new connection, if the request successfully completed.

5.7.1 Detailed Description

Connect request completion event.

The status field may contain the following values:

CCI_SUCCESS The connection was accepted by the server and successfully established. The corresponding connection structure is available in the event connection field.

CCI_ECONNREFUSED The server rejected the connection request.

CCI_ETIMEDOUT The connection could not be established before the timeout expired.

Some transports may also return specific return codes.

The connection field is only valid for use if status is CCI_SUCCESS. It is set to NULL in any other case.

The context field is always set to what was passed to cci_connect(). On success, the connection structure context field is also set accordingly.

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 cci_event_type_t cci_event_connect::type

Type of event - should equal CCI_EVENT_CONNECT.

5.7.2.2 cci_status_t cci_event_connect::status

Result of the connect request.

Examples:

client.c.

5.7.2.3 void* cci_event_connect::context

Context value that was passed to cci_connect()

5.7.2.4 cci_connection_t* cci_event_connect::connection

The new connection, if the request successfully completed.

5.8 cci_event_connect_request Struct Reference

Connection request event.

`#include <cci.h>`

Data Fields

- cci_event_type_t type

  Type of event - should equal CCI_EVENT_CONNECT_REQUEST.
- uint32_t data_len

  Length of connection data.
- const void * data_ptr

  Pointer to connection data received with the connection request.
- cci_conn_attribute_t attribute

  Attribute of requested connection.

5.8.1 Detailed Description

Connection request event.

An incoming connection request from a client. It includes the requested connection attributes (reliability and ordering) and an optional payload.

The number of fields in this struct is intentionally limited in order to reduce costs associated with state storage, caching, updating, copying.

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

This event should be passed to either cci_accept() or cci_reject() before being returned with cci_return_event().

5.8.2 Field Documentation

5.8.2.1 cci_event_type_t cci_event_connect_request::type

Type of event - should equal CCI_EVENT_CONNECT_REQUEST.

5.8.2.2 uint32_t cci_event_connect_request::data_len

Length of connection data.

5.8.2.3 const void* cci_event_connect_request::data_ptr

Pointer to connection data received with the connection request.

5.8.2.4 cci_conn_attribute_t cci_event_connect_request::attribute

Attribute of requested connection.

5.9 cci_event_endpoint_device_failed Struct Reference

Endpoint device failed event.

`#include <cci.h>`

Data Fields

- cci_event_type_t type
  Type of event - should equal CCI_EVENT_ENDPOINT_DEVICE_FAILED.
- cci_endpoint_t * endpoint
  The endpoint on the device that failed.

5.9.1 Detailed Description

Endpoint device failed event.

The endpoint's device has failed.

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.

5.9.2 Field Documentation

5.9.2.1 cci_event_type_t cci_event_endpoint_device_failed::type

Type of event - should equal CCI_EVENT_ENDPOINT_DEVICE_FAILED.

5.9.2.2 cci_endpoint_t* cci_event_endpoint_device_failed::endpoint

The endpoint on the device that failed.

5.10 cci_event_keepalive_timedout Struct Reference

Keepalive timeout event.

`#include <cci.h>`

Data Fields

- cci_event_type_t type

  Type of event - should equal CCI_EVENT_KEEPALIVE_TIMEDOUT.
- cci_connection_t * connection

  The connection that timed out.

5.10.1 Detailed Description

Keepalive timeout event.

We were unable to send a periodic message to the peer. The application can attempt communication or disconnect. The connection will continue to consume resources until the application calls cci_disconnect().

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.

5.10.2 Field Documentation

5.10.2.1 cci_event_type_t cci_event_keepalive_timedout::type

Type of event - should equal CCI_EVENT_KEEPALIVE_TIMEDOUT.

5.10.2.2 cci_connection_t* cci_event_keepalive_timedout::connection

The connection that timed out.

5.11 cci_event_recv Struct Reference

Receive event.

`#include <cci.h>`

Data Fields

- cci_event_type_t type
  Type of event - should equal CCI_EVENT_RECV.
- uint32_t len
  The length of the data (in bytes).
- const void * ptr
  Pointer to the data.
- cci_connection_t * connection
  Connection that this message was received on.

5.11.1 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.

5.11.2 Field Documentation

5.11.2.1 cci_event_type_t cci_event_recv::type

Type of event - should equal CCI_EVENT_RECV.

5.11.2.2 uint32_t cci_event_recv::len

The length of the data (in bytes).

This value may be 0.

Examples:

client.c, and server.c.

5.11.2.3 const void* cci_event_recv::ptr

Pointer to the data.

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

Examples:

client.c, and server.c.

5.11.2.4 cci_connection_t* cci_event_recv::connection

Connection that this message was received on.

5.12 cci_event_send Struct Reference

Send event.

`#include <cci.h>`

Data Fields

- cci_event_type_t type
  Type of event - should equal CCI_EVENT_SEND.
- cci_status_t status
  Result of the send.
- cci_connection_t * connection
  Connection that the send was initiated on.
- void * context
  Context value that was passed to cci_send()

5.12.1 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.

5.12.2 Field Documentation

5.12.2.1 cci_event_type_t cci_event_send::type

Type of event - should equal CCI_EVENT_SEND.

5.12.2.2 cci_status_t cci_event_send::status

Result of the send.

Examples:

client.c.

5.12.2.3 cci_connection_t* cci_event_send::connection

Connection that the send was initiated on.

5.12.2.4 void* cci_event_send::context

Context value that was passed to cci_send()

Examples:

client.c.

5.13 cci_rma_handle Struct Reference

Opaque RMA handle for use with cci_rma().

`#include <cci.h>`

Data Fields

- uint64_t stuff [4]

5.13.1 Detailed Description

Opaque RMA handle for use with cci_rma().

The RMA handle contains all information that a transport will need to initiate a remote RMA operation. The contents should not be inspected or modified by the application. The contents are serialized and ready for sending to peers.

5.13.2 Field Documentation

5.13.2.1 uint64_t cci_rma_handle::stuff[4]

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;

static 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://1.2.3.4:5678\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);
        if (ret == CCI_SUCCESS) {
                switch (event->type) {
                case CCI_EVENT_SEND:
                        printf("send %d completed with %s\n",
                               (int)((uintptr_t) event->send.context),
                               cci_strerror(endpoint, event->send.
      status));
                        break;
                case CCI_EVENT_RECV:
                        memcpy(buffer, event->recv.ptr, event->recv.
      len);
                        buffer[event->recv.len] = '\0';
                        fprintf(stderr, "received \"%s\"\n", buffer);
                        *done = 1;
                        break;
```

```
                case CCI_EVENT_CONNECT:
                        *done = 1;
                        if (event->connect.status == CCI_SUCCESS)
                                *connection = event->connect.connection;
                        else
                                *connection = NULL;
                        break;
                default:
                        fprintf(stderr, "ignoring event type %d\n",
                                event->type);
                }
                cci_return_event(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 */
        char *uri = NULL;
        cci_os_handle_t fd;
        cci_endpoint_t *endpoint = NULL;
        cci_connection_t *connection = NULL;
        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(NULL, 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(NULL, ret));
                exit(EXIT_FAILURE);
        }

        ret = cci_get_opt(endpoint,
                          CCI_OPT_ENDPT_URI, &uri);
        if (ret) {
                fprintf(stderr, "cci_get_opt() failed with %s\n", cci_strerror(endpoint, ret));
                exit(EXIT_FAILURE);
        }
        printf("Opened %s\n", uri);

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

        /* initiate connect */
        ret = cci_connect(endpoint, server_uri, "Hello World!", 12,
                          CCI_CONN_ATTR_UU, NULL, 0, NULL);
        if (ret) {
                fprintf(stderr, "cci_connect() returned %s\n",
                        cci_strerror(endpoint, 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 data[128];

                memset(data, 0, sizeof(data));
                sprintf(data, "Hello World!");
                ret = cci_send(connection, data, (uint32_t) strlen(data),
                               (void *)(uintptr_t) i, 0);
                if (ret)
                        fprintf(stderr, "send %d returned %s\n", i,
                                cci_strerror(endpoint, 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(endpoint, ret));
                exit(EXIT_FAILURE);
        }
        ret = cci_destroy_endpoint(endpoint);
        if (ret) {
                fprintf(stderr, "cci_destroy_endpoint() returned %s\n",
                        cci_strerror(NULL, ret));
                exit(EXIT_FAILURE);
        }
        /* add cci_finalize() here */

        return 0;
}
```

6.2 devices.c

This is an example of using get_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 * devices, * const * d;

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

        ret = cci_get_devices(&devices);
        if (ret != CCI_SUCCESS) {
                fprintf(stderr, "cci_get_devices() returned %s\n",
                        cci_strerror(NULL, 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);
        }

        /* Add cci_finalize() here */

        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(NULL, ret));

        return 0;
}
```

6.4 server.c

This application demonstrates opening an endpoint, 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;
        char *uri = NULL;
        cci_endpoint_t *endpoint = NULL;
        cci_os_handle_t ep_fd;
        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(NULL, 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(NULL, ret));
                exit(EXIT_FAILURE);
        }

        ret = cci_get_opt(endpoint,
                          CCI_OPT_ENDPT_URI, &uri);
        if (ret) {
                fprintf(stderr, "cci_get_opt() failed with %s\n", cci_strerror(endpoint, ret));
                exit(EXIT_FAILURE);
        }
        printf("Opened %s\n", uri);

        while (1) {
                char *buffer;
                cci_event_t *event;

                ret = cci_get_event(endpoint, &event);
                if (ret != CCI_SUCCESS) {
                        if (ret != CCI_EAGAIN) {
                                fprintf(stderr, "cci_get_event() returned %s",
                                        cci_strerror(endpoint, ret));
                        }
                        continue;
                }
                switch (event->type) {
                case CCI_EVENT_RECV:
                        {
                                memcpy(buffer,
                                       event->recv.ptr, event->recv.
        len);
                                buffer[event->recv.len] = 0;
                                printf("recv'd \"%s\"\n", buffer);

                                /* echo the message to the client */
                                ret = cci_send(connection,
                                               event->recv.ptr,
                                               event->recv.len, NULL, 0);
                                if (ret != CCI_SUCCESS)
                                        fprintf(stderr,
                                                "send returned %s\n",
                                                cci_strerror(endpoint, ret));
                                break;
                        }
                case CCI_EVENT_SEND:
                        printf("completed send\n");
                        break;
                case CCI_EVENT_CONNECT_REQUEST:
                        {
                                int accept = 1;

                                if (accept) {
                                        ret = cci_accept(event, NULL);
                                        if (ret != CCI_SUCCESS) {
                                                fprintf(stderr,
                                                        "cci_accept() returned %s",
                                                        cci_strerror(endpoint, ret));
                                        }

                                } else {
                                        cci_reject(event);
                                }
                        }
                        break;
                case CCI_EVENT_ACCEPT:
                        connection = event->accept.connection;
                        if (!buffer) {
                                buffer = calloc(1, connection->max_send_size + 1);
                                /* check for buffer ... */
                        }
                        break;
                default:
                        fprintf(stderr, "unexpected event %d", event->type);
                        break;
                }
                cci_return_event(event);
        }
```

```
        /* clean up */
        cci_destroy_endpoint(endpoint);
        /* add cci_finalize() here */

        return 0;
}
```

Index