The Common Communication Interface (CCI)

Scott Atchley*, David Dillow*, Galen Shipman*,
Patrick Geoffray†, Jeffrey M. Squyres‡, George Bosilca§ and Ronald Minnich¶ *Oak Ridge National Laboratory, Oak Ridge, TN †Myricom, Inc., Arcadia, CA ‡Cisco Systems, Inc., San Jose, CA §University of Tennessee, Knoxville, TN ¶Sandia National Laboratories, Livermore, CA

***Abstract*—There are many APIs for connecting and exchanging data between network peers. Each interface varies wildly based on metrics including performance, portability, and complexity. Specifically, many interfaces make design or implementation choices emphasizing some of the more desirable metrics (e.g., performance) while sacrificing others (e.g., portability). As a direct result, software developers building large, network-based applications are forced to choose a specific network API based on a complex, multi-dimensional set of criteria. Such trade-offs inevitably result in an interface that fails to deliver some desirable features.**

In this paper, we introduce a novel interface that both supports many features that have become standard (or otherwise generally expected) in other communication interfaces, and strives to export a small, yet powerful, interface. This new interface draws upon years of experience from network-oriented software development best practices to systems-level implementations. The goal is to create a relatively simple, high-level communication interface with low barriers to adoption while still providing important features such as scalability, resiliency, and performance. The result is the Common Communications Interface (CCI): an intuitive API that is portable, efficient, scalable, and robust to meet the needs of network-intensive applications common in HPC and cloud computing.

I. Introduction

In 1981, the Internet Protocol (IP) [1] began the era of ubiquitous networking. The Transmission Control Protocol (TCP) [2] was layered on top of IP, freeing application developers from many network-specific issues such as reliable delivery and message ordering. Even though TCP/IP was implemented in a wide variety of compute and networking hardware, developers still needed a common application programming interface (API) to create portable software. The BSD Sockets API filled that role, and became the *de facto* networking API for most platforms.

But even though both the BSD Socket wire protocol and API are still widely used today, the networking hardware that they are used on has changed radically. For example, typical half round trip (HRT) TCP ping-pong latencies were measured in multiple milliseconds on early technologies such as 10BASE5 Ethernet, Token Ring, and Token Bus. Modern networking technologies such as 10 Gigabit Ethernet (10GE), Quad Data Rate (QDR) InfiniBand (IB), Cray's Gemini, IBM's Torrent, and SGI's NumaLink have HRT ping pong latencies as low as 1μs.

Current generation networking technologies also provide features such as remote direct memory access (RDMA), operating system (OS) bypass, "zero copy" hardware support, one-sided operations, and asynchronous operations. While some implementations of the BSD Sockets interface can utilize a subset of these features, its API does not directly expose any of them to applications. As a direct result, several next-generation network APIs have evolved, such as the Virtual Interface Architecture (VIA), OpenFabrics Verbs, Myrinet Express (MX), and the Portals API – just to name a few. While these interfaces adequately expose the underlying network's capabilities, none of them have garnered the support from application developers that the Sockets interface has. Two key barriers exist to widespread adoption: simplicity (from an application developer perspective) and portability across a variety of OS and networking technologies (including performance portability).

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.

Additionally, metrics that used to be the sole purview of High Performance Computing (HPC) – such as ultra-low HRT latencies and maximum bandwidth at small message sizes – are becoming important to other types of common applications, including: internet search technologies, cloud computing, financial trading, and rich media serving. Indeed, as the raw bandwidth of network technologies continues to increase, compute servers need to be able to utilize the entire network capacity. A portable network interface API is needed that removes the inefficiencies of TCP, provides convenient abstractions for applications, and enables portability across a wide variety of OS, compute server, and network platforms.

In this paper, we propose a new networking interface: the Common Communication Infrastructure (CCI). 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.

The remainder of this paper is organized as follows: Section II depicts the state-of-the-art of the common messaging APIs, followed by a section describing our designs goals. In Section IV, we detail the CCI abstraction entities (e.g.,

endpoints and *connections*) as well as the API. Finally, we present our current implementation performance over UDP, Portals, and MX interfaces, and then conclude the paper.

II. State of the Art

Over the years, many communication interfaces have come and gone. The few that have remained and seen wide-spread adoption are BSD Sockets [3], the Message Passing Interface (MPI) [4], and some vendor-specific APIs.

A. Sockets

The Sockets interface is the most widely used by far. All major operating systems provide support for Sockets; the Internet and all the services it provides relies upon it. The popularity of BSD Sockets can be attributed to:

- Simple API
- Robustness
- Implicit buffering

The API provides stream and datagram based modes, connection-oriented and connection-less modes, and client/server semantics for connection-oriented modes. Based on the transport, the API can supports multiple delivery modes (unicast, multicast, and/or broadcast). The API does not provide for collective communication nor does it provide one-sided operations.

Sockets implementations are mature and well understood. It does not assume or require special hardware features (nor can it exploit them if they exist).

Both sends and receives are buffered allowing operations to complete immediately (if buffer space is available for sends or data exists in the buffer for receives). Applications may also choose to not block if send buffers are full or receive buffers are empty if need be. The downside of buffering is more work is required by the CPU which can result in lower throughput over the network.

Sockets, when used with SOCK_STREAM, inherits the well-known TCP performance constraints [5] related to the scaling window and the bandwidth-delay product.

B. MPI

In the High Performance Computing (HPC) arena, MPI is the dominant interface for inter-process communication. Designed for maximum scalability, MPI has a richer, but much more complicated API than Sockets.

MPI provides point-to-point (i.e. send-recv or two-sided semantics), collective, and one-sided operations. For point-to-point communication, MPI provides a variety of modes including blocking and non-blocking, synchronous and asynchronous, as well as *ready* mode (only send if a matching receive has been posted).

Multiple implementations exist [6]–[8] supporting multiple operating systems and/or interconnects. Rather than connections, the API uses the notion of communication groups (i.e. communicators) which include all processes (MPI_COMM_WORLD) and may be split to include subsets of processes. MPI does provide a notion of dynamic process management which includes MPI_COMM_ACCEPT and MPI_COMM_CONNECT, but this is less mature and much less used.

The MPI standard does not define an underlying network protocol and each MPI implementation has written its own network abstraction layer (NAL). These NALs typically support BSD Sockets as well as one or more vendor specific APIs.

MPI is a less mature technology – compared to Sockets – likely due to its relative complexity (over 300 functions), niche market penetration (HPC), and relative youth. While MPI semantics are fairly well defined by the MPI standard, implementations vary in their compliance to the MPI specification and may exhibit differing behavior when the standard is silent on a specific semantic. Applications that rely upon the semantic embodied in a particular implementation may not be portable across other MPI implementations. MPI provides a rigid fault model in which a process fault within a communication group imposes failure to all processes within that communication group (MPI_ERRORS_ABORT). Although work has been done on fault-tolerant MPI [9], [10], that work has yet to be adopted by the broader HPC community.

The MPI standard remains silent on a number of areas that are often performance critical. For basic send/recv operations MPI implementations often adopt two common strategies that attempt to balance buffering and communication overhead. "Eager" mode will send data immediately to the receiver regardless of the receiver having posted a matching receive. The data is buffered on the receiver if the matching receive is not posted upon receipt of the data. "Rendezvous" mode defers sending the data until a posted receive is matched on the receiver. Performance of an MPI application can therefore be largely implementation-dependent and may vary even within a single implementation depending on the MPI configuration settings used for a particular invocation. Perhaps more importantly, this ambiguity can result in receiver buffer overruns or out-of-memory (OOM) faults on the receiver.

MPI has added support for one-sided operations as currently defined in the MPI-2 standard [11]. The MPI-2 one-sided interface has received limited adoption due to API complexity and overly constrained semantics [12]. Current efforts in the MPI-3 standardization process are aimed at addressing these issues.

C. Specialized APIs

There are numerous vendor- and organization-specific APIs available today including OpenFabrics Verbs, Cray/Sandia's Portals [13], QLogic's PSM, Myricom's MX, LBL's GASNet [14], DAPL, IBM's LAPI [15], IBM's DCMF [16] among others. Overall, they provide a lot of choice but none has appeared as a widely used communication interface. Since most are targeted to specific network technologies, the APIs tend to reflect the design of the underlying hardware. In the rest of this section, we will present each of these interfaces.

Based on the earlier VIA specification [17], the InfiniBand specification does not specify an API; it only specifies which *Verbs* must be supported. After many vendors created separate Verbs APIs, they eventually coalesced into the OpenFabrics Association's (OFA) Verbs. Verbs has support for

two-sided and one-sided operations, always asynchronous. In addition, Verbs has support for reliable and unreliable modes, connection-oriented and connection-less. Buffer management is left to the application, all receives must be posted before sending. Also, all data movement operations require registering the memory in advance. Verbs use the concept of Queue Pair (QP) to represent a logical connection between two processes.

The Portals API provides one-sided semantics (i.e. Put/Get) but uses match tags to steer messages to the correct buffers. The API is connection-less and leaves it up to the NAL to maintain any necessary connection state internally. Portals is mostly used on the large Cray systems such as ORNL's Jaguar [18]. The Lustre distributed file system NAL, LNET, was originally based on Portals [19].

Both designed for efficiently implementing MPI, Myricom's MyrinetExpress (MX) and QLogic's PathScale Messages (PSM) have many similarities. Both provide a two-sided matching interface which use buffering for smaller messages and zero-copy for larger messages. Both are connection-less in that the target does not have to accept connection requests. Both provide reliable in-order matching with out-of-order completion.

LAPI [15] and DCMF [16] are both proprietary software stacks developed by IBM, using as targets the RS-series and the BlueGene P/Q machines. While some support from the scientific community outside IBM exists, it has failed to broaden and remains limited. DCMF supports an interface for one-sided and two-sided message semantics, with contiguous or discontiguous memory layout, providing transparent support for link aggregation. The DCMF communication layer supports multiple programming paradigms such as Aggregate Remote Memory Copy Interface (ARMCI) and Global Arrays (GA), in addition to MPI. It also provides a collective API, allowing processes to execute asynchronous collective communications in an optimized way.

Among all of these interfaces, the OpenFabrics Verbs API has the broadest adoption in HPC, representing 43% of the machines in the Top 500 [20]. Despite limited success in the enterprise world, Verbs aspires to widen its audience beyond HPC, especially over Ethernet through RoCEE [21]. As such, we will refer to Verbs alongside Sockets and MPI in the rest of this paper.

III. Design Goals

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

A. Portability

Application and middleware developers do not have the resources to continuously port their code on different communication interfaces. Selecting a vendor-specific API reduces competition in the market place, thus increasing prices and adding the risk of business failure or market disruptions. It also slows down the adoption of new and improved technology. Similarly, vendors do not have the resources to properly support a large set of middleware. The whole ecosystem would clearly benefit from a truly unified communication layer. 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.

B. 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) is listed in Table I. 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.

TABLE I
Lines of Code per BTL

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

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.

C. Performance

Performance is a major driver 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.

OS-bypass allows direct interaction between the application and a virtualized instance of the network hardware, without involving the operating system. This technique is essential for

low latency as it removes the need for interrupts in the critical path. Furthermore, a process or a thread blocking in the kernel is often scheduled on a different core when awakened. Avoiding the operating system can greatly improve NUMA locality. To support OS-bypass the network adapter must be able to demultiplex incoming packets into corresponding queues in each application. Most of this functionality is commonly used in Ethernet adapters that support Receive Side Scaling (RSS).

Zero-copy reduces CPU overhead and increases bandwidth by eliminating memory copies in the critical path. The network adapter fetches or delivers data directly into the memory space of the application via Direct Memory Access (DMA) operations. To this end, the related memory pages must be pinned so that the network adapter can safely access it. An important drawback of zero-copy is its synchronous nature. Since there is no intermediate copies, the memory on the send side cannot be reused until the data has been delivered to the receiver (or at least put on the wire for unreliable connections).

Zero-copy is often confused with *one-sided operations*, which allow a communication to complete without the involvement of the application thread on the remote side. All of the required information, mainly the remote address of the data to access, is provided on the origin side. One-sided operations may or may not be associated with zero-copy, and may use the help of a progression thread on the target side. Similarly, zero-copy may be implemented with receive matching instead or remote addressing, as it is the case with MX and Portals.

Asynchronous operations are used to decouple the initiation of a communication from its progress and subsequent completion. This allows communication to be overlapped with computation. More practically, asynchronous operations enable initiation of concurrent data movements without blocking the application context.

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.

D. Scalability

Projections for leadership scale systems in HPC include hundreds of thousands of nodes and millions of cores [22]. 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 [23] and dynamic buffer management to work around the QPs memory footprint problem. Scalability was further improved with the addition of Shared Receive Queues (SRQ) [24], but distinct QPs are still required on the send side [25]. 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.

E. 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. 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.

IV. The CCI API Interface

The CCI API aims to encompass all of these design goals. It leverages *Endpoints* to transparently manage buffering, message demultiplexing and notifications. It uses *connections* to represent the state of communication channels between endpoints, with minimal footprint. Communications between endpoints use either *Active Messages* (AM) or *Remote Memory Access* (RMA) operations, depending on data size and the desired buffering semantic.

In this section, we will present the core elements of the CCI API and refer to the full documentation [26] for details.

A. Endpoints

An endpoint is a virtualized instance of a device, it is the logical source or destination of all communications in CCI. An endpoint contains both a send and receive queue and their associated buffers, it is a complete container of all resources used by an application process or thread to perform CCI operations. As such, endpoints naturally fit the NUMA architecture, they can be bound to particular cores to maximize memory locality. From the application point of view, an endpoint is an opaque object. The allocation and recycling of buffers is entirely managed by the CCI library.

Endpoints interact with the application through *events*. CCI provides the function $cci\_get\_event()$ to immediately retrieve the next event for a particular endpoint. Optionally, the application can block until an event is available, allowing the application thread to be scheduled out by the operating system. To facilitate the integration of the blocking semantic with non-CCI operations, each endpoint exposes an OS-specific handle such as a file descriptor in Linux or a Windows object.

Event-driven designs are common in reactive environments such as Graphical User Interfaces (GUIs). As such, they are well suited for communication interfaces. In CCI, events may represent receive notifications, send completions, connection transitions, etc. In addition, events may contain resources such as receive buffers. The ownership of these resources is transfered to the application when an event is retrieved, with the expectation that they would be returned to the CCI library at some point through a call to $cci\_return\_event()$, possibly out of order. Such a mechanism can be leveraged by advanced users to delay processing of a particular event. A simple example of a event processing loop is shown in code listing 1.

```
rc = cci_get_event(endpoint, &event);
if (rc == CCI_SUCCESS) {
  switch (event.type) {
  case CCI_EVENT_SEND:
    /* process send completion */
    context = event.send.context;
    break;
  case CCI_EVENT_RECV:
    /* can access message in place
       or copy it to app buffer */
    buffer = event.recv.data_ptr;
    length = event.recv.data_len;
    sender = event.recv.connection;
    /* for example, send it back */
    cci_send(sender, NULL, 0, buffer,
             length, NULL, 0);
    break;
  case CCI_EVENT_CONNECT_REQUEST:
    /* accept connection on server */
    cci_accept(event.connect.request,
               endpoint, &connection);
    break;
  case CCI_EVENT_CONNECT_SUCCESS:
    /* new connection is established */
    connection = event.connect.connection;
    break;
  default:
    printf("Unknown␣CCI␣event␣!\n");
  }
  cci_return_event(event);
}
```

Listing 1. CCI event processing loop

The concept of the endpoint is key to scalability. By multiplexing incoming messages into a shared receive queue and similarly buffering outgoing messages in a shared send queue, the overall memory footprint is independent of the number of peers communicating with the endpoint. On the time dimension, an endpoint offers a unified completion queue for events, allowing for OS-bypass implementations to provide low latency at scale.

B. Connections

CCI uses connections to represent the state of communication channels with remote peers. An endpoint can be connected to another endpoint through one or more connections. Connections have different attributes, such as reliability and order. CCI supports five different connection types:

- Reliable with Ordered completion (RO)
- Reliable with Unordered completion (RU)
- Unreliable with Unordered completion (UU)
- Multicast Send (MC_TX)
- Multicast Receive (MC_RX)

As previously stated, unreliable connections are useful for some applications. Order however is an unusual characteristic, as most network technologies such as Ethernet and InfiniBand assume order on the wire. Nevertheless, there are several situations when unordered semantics are desirable. For example, using multiple network links to aggregate bandwidth, a technique also known as *channel bonding*, inhibits global packet ordering. Similarly, switch contention avoidance techniques, such as adaptive routing, break order when different routes are selected between two endpoints. Switch contention is a major scalability concern, relaxing order at the communication interface level is believed to be essential to enable effective technological solutions.

Connections are established through a client-server process similar to BSD Sockets or RDMA-CM. However, applications such as Apache commonly use a 2-step mechanism where clients initially connect to a broker thread, which passes the requests to a different thread for processing. This allows the application to transparently handle requests with multiple processing threads. Sockets applications traditionally hand off connections by forking the broker process. However, using $fork()$ is particularly disruptive for interfaces supporting OS-bypass and zero-copy, as it changes the underlying memory mappings.

CCI provides a connection manager framework to the application. Clients initiate a connection by specifying a server Uniform Resource Identifier (URI), a string that contains information used by the connection manager to identify the requested service. In the context of HPC, it could be a batch queue system job identifier and a rank. For web services, the URI could be a standard URL. The flexibility of the URI allows for extensibility of naming and support of additional functionality, such as system-wide load-balancing and fail-over. The pseudo-code listing 2 shows a connection establishment between a client and a server, in complement of the

event processing loop described previously.

```
port = 1234;
if (server) {
  /* bind endpoint */
  cci_bind(endpoint, port);
} else {
   /* initiate connection */
   server_uri = ``foo.bar.com'';
   cci_connect(endpoint, server_uri,
               port, ...);
}

[...]
cci_disconnect(conn);
```

Listing 2. CCI connection establishment example

On the server side, the application binds a specific endpoint to a service to receive connection requests. The incoming connection request carries a payload that can be used for identification or authentication. Upon accepting the request, an endpoint is selected to complete the connection, potentially a different endpoint than the one used to receive the connection request. This indirection allows the application to choose a local endpoint at the last step of the server connection process, effectively managing multiple endpoints on multi-core systems.

C. Active Messages

Buffering is the most efficient way to provide asynchronous semantics, which is essential for scalability. BSD Sockets exclusively relies on buffering on both send and receive sides; a send returns as soon as the data has been written to the send buffer. Similarly, an incoming payload is first written to the receive buffer and then retrieved by the application.

The Verbs interface provides asynchronous semantics through its send/receive operations. However, it delegates the buffer allocation to the application. Receive buffers have to be posted on a QP prior to messages arrival, failure to do so transitions the QP to an error state. Furthermore, receive buffers are matched in order, so they all have to be large enough for the biggest possible message when using a Shared Receive Queue (SRQ). This puts a practical limit on the maximum size of messages exchanged with this mechanism.

MPI offers explicitly a buffered send but it is rarely used. Instead, the standard send may or may not buffer the message; it returns when the application buffer can be reused. In most implementations, it depends on the message size as explained in Section II-B. Small messages are expected and sometimes assumed to be buffered. Larger messages block the send as long as the message is not delivered to the receive side. This assumed threshold is not defined by the MPI specification and may result in unsafe code that can deadlock, thereby breaking portability. The matching interface in MPI is powerful yet complex. Support for wild cards require a single, coherent matching stack which cannot be offloaded to effectively handle intra-node messages. In addition, matching interfaces are stateful, greatly complicating fault recovery.

CCI uses a variation of Active Messages [27] (AM) to address these issues. Most of AM's complexity, shared by GasNet [14], is related to the use of asynchronous handlers. Beyond requiring a support thread, handlers severely restrict the type of operations they can perform, such as allocating memory or send new messages. CCI uses events instead to greatly simplify the API. Events can be processed in the application context, there is no blocking limitation. Similar to the original AM implementation, buffers are managed internally from a constant pool on both send and receive sides. On the receive side the application is given access to a CCI buffer through a receive event. It may process the data in place or copy it out. When the application returns the receive event to the CCI library the corresponding receive buffer is recycled. Receive events can be returned out of order, so the application can keep some receive buffers for some time if necessary. On the send side, messages are immediately buffered.

A single function $cci\_send()$ is used to send messages, its prototype is presented in code listing 3. Two segments are supported by default to simplify data encapsulation behind an application header. The context is returned as part of a send completion event, if requested.

```
int
cci_send(cci_connection_t *connection,
         void *header_ptr, uint32_t header_len,
         void *data_ptr, uint32_t data_len,
         void *context, int flags);
```

Listing 3. CCI send prototype

As with Verbs, the receive buffers have to be large enough for the largest message. As the buffer management is removed from the application, CCI explicitly defines a Maximum Send Size (MSS), specified by the underlying device. This well-defined limit avoids the portability and unsafe assumption of the MPI model. The MSS may be different for different devices and therefore different endpoints. When two endpoints connect the smallest MSS is used for that connection.

This model is particularly efficient when the MSS aligns with the Maximum Transfer Unit (MTU) of the underlying device, typically 2K to 8K. In this case, no inter-packet state is needed, greatly simplifying networking hardware requirements and allowing CCI to leverage less sophisticated devices such as commodity Ethernet adapters. Some networks use a small MTU but provide in-order hardware-based segmentation and reassembly. This allows a larger MSS with minimal overhead.

If the application desires to send a message larger than the limit, it has to perform segmentation and reassembly itself. With a typical MSS in the order of a page size, the segmentation can effectively pipeline the copy into the send buffers with the packet injection itself. On the receive side, reassembly will require an extra memory copy, unless the application can manipulate the data in place over several non-contiguous receive buffers. However, the segmentation/reassembly effort is a clear incentive to restrict the messages semantic in CCI to smaller traffic such as synchronization messages.

D. Remote Memory Access

For larger data movement, CCI provides a Remote Memory Access (RMA) semantic that enables zero-copy for low CPU overhead. As RMA operations are one-sided, they are only allowed on reliable connections.

The application needs to explicitly register memory to be used for RMA transfers. The memory registration process is specific to each device implementation but it usually consists of pinning the underlying physical pages and making them suitable for DMA operations. In CCI, memory is registered for a particular connection or for all connections. This allow for simpler remote memory protection than the Protection Domains of the Verbs API. A memory area can be registered on multiple CCI connections if needed.

By requiring explicit memory registration, CCI avoids the ambiguity of MPI where registration is not part of the API but is often part of the implementation. In this case, implicit memory registration is performed in the critical path, leading to various unsafe caching mechanisms [28] (hijacking malloc and other system calls). Virtualization is driving new and improved IOMMU functionality that would make memory registration obsolete on compliant hardware. Unfortunately, it is not widely available at this time.

With portability as an important design goal, the CCI RMA semantic is designed to be efficiently implemented on top of the CCI messaging model for devices that do not provide zero-copy support in hardware. Furthermore, Open-MX [29] has demonstrated high bandwidth and low CPU overhead when using the Intel IOA/T copy engine to move data between receive buffers of an Ethernet driver in the kernel and an RMA target in user space.

Contrary to some other APIs and for performance and portability reasons, CCI does not guarantee delivery order between RMA transfers, or even for data within a single RMA operation. While OFA Verbs does not guarantee Last-Byte-Written-Last semantic as well, some early hardware provided this semantic and some applications grew to rely upon it. A common use is for an application to poll on the last byte of a RDMA write on the target side to determine when the RDMA has completed. This behavior is unsafe if the DMA operations are not guaranteed to be delivered in order as is the case on some systems.

Since RMA order is not guaranteed, a Fence flag can be used to selectively enforce order on the target side with regard to all previous RMA operations on the connection. This semantic allows for optimizations where RMA packets can use multiple links or take different routes in the fabric and the ordering cost is borne only when needed. An implementation of the SHMEM [30] API on top of CCI would use the Fence flag on the first RMA operation following a call to $shmem\_fence()$. With strong order between RDMA operations on a given QP, OFA Verbs cannot easily leverage bonding and adaptive routing to increase effective bandwidth.

Memory registration and RMA prototypes are presented in code listing 4. Memory registration produces a memory handle than can be sent to the target side of a subsequent RMA operation. The RMA call itself requires both a local and remote memory handle. In addition, it allows to optionally piggyback a header segment as an active message. This message is ordered relative to the delivery of the RMA data, the corresponding receive event can be used to notify completion of the RMA on the target. This mechanism is similar to the small *immediate* data in Verbs.

```
int
cci_rma_register(cci_endpoint_t *endpoint,
                 cci_connection_t *connection,
                 void *start, uint64_t length,
                 uint64_t *rma_handle);
int
cci_rma(cci_connection_t *connection,
        void *header_ptr,
        uint32_t header_len,
        uint64_t local_handle,
        uint64_t local_offset,
        uint64_t remote_handle,
        uint64_t remote_offset,
        uint64_t data_len,
        void *context, int flags);
```

Listing 4. CCI RMA prototypes

V. Status and Evaluation

A. Portability

We have three proof-of-concept implementations: UDP Sockets, Portals 3.3, and MX.

The Sockets prototype driver opens one SOCK_DGRAM (UDP) socket per CCI endpoint, with CCI providing optional reliability and optional ordering (UU, RO, RU). The Sockets driver multiplexes connections over a single socket. Enough functions are complete including RMA to allow simple ping pong tests to exercise the CCI API.

The Portals implementation provides UU, RO, and RU over the portals interface. Since Portals assumes a reliable interconnect, the only difference between a CCI UU connection and a RO/RU connection is that we complete a UU send when we receive the Portals' SEND_END event which indicates that the sender's buffer is no longer needed (i.e. the data is on the wire). For a reliable connection, we report a completion when we get the Portals' ACK of receipt at the peer.

The MX driver currently only implements UU active messages. It takes advantage of MX's unexpected handler to handle all incoming active messages and connection requests. MX limits the unexpected handler to 1 KB messages which we adopt as the max send size for this driver. MX send semantics closely mirror those of MPI, so the completion is returned once the buffer is modifiable (typically once the contents are copied out to an intermediate buffer). We will implement RU active messages by using MX's synchronous (rendezvous) send.

B. Performance

We wrote ping pong applications for Portals, MX, and CCI. We tested CCI in lock-less and locking versions, *CCI* and *CCI Thread-Safe* respectively, for active message ping pong tests. All binaries were compiled with `-O3` and `-fno-builtin`. For each message size we conducted an average of 500,000 iterations. Typically, we saw $\pm$10ns variation for messages up to 1 KB.

We evaluated two different native Portals strategies for sending. The first strategy binds a buffer, puts the buffer, and then unlinks the buffer; we refer to it as *Portals Bind*. This is the most intuitive strategy and probably the more commonly used method by developers. The second strategy allocates a intermediate buffer and binds it once. The application then

copies into the bound buffer before calling Put. We refer to this as *Portals Copy*. Portals Copy performs better than Portals bind for messages less than 8 KB.

We ran the Portals tests on a Cray XT6 and locked the processes to the same cores for both tests. The CCI HRT ping pong is about 200-300ns more than the Portals Copy performance up to 1 KB. For an eight byte message, for example, CCI's latency is 5.80µs versus 5.56µs for native Portals. Figures 1 and 2 show latency and bandwidth for active messages up to 8 KB on the Cray XT6. Figure 3 shows latency and overhead for RMA messages up to 4 MB.

For MX tests, we used a pair of nodes with dual Intel Nehalems and Myri-10G NICs. Figure 4 shows CCI overhead of around 60ns without locks and around 120ns with locks while figure 5 shows bandwidth for MX, CCI, and MPI.

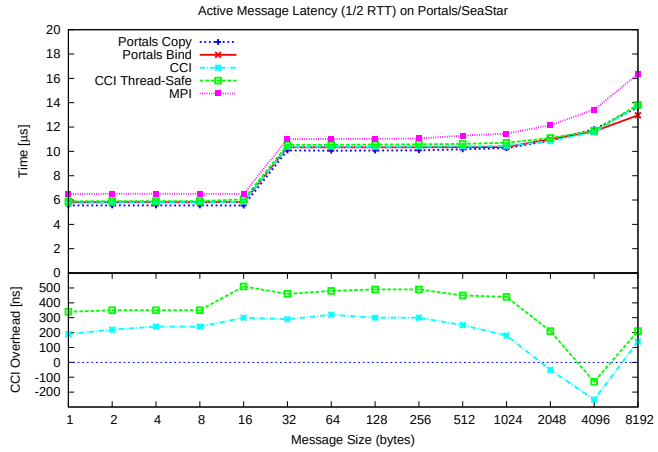

Fig. 1. Ping pong Latency when using Portals on SeaStar

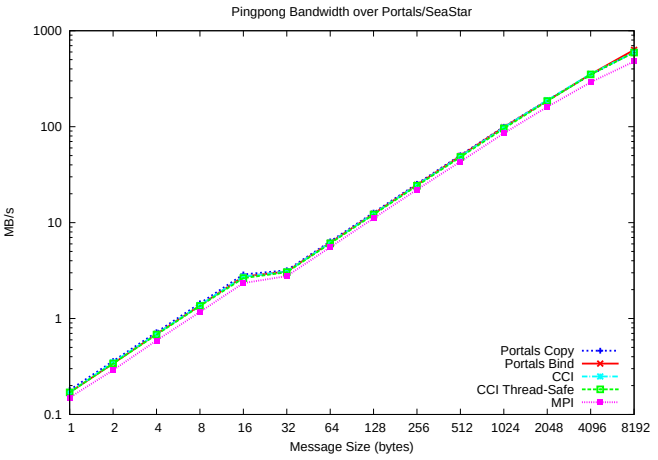

Fig. 2. Ping pong Bandwidth when using Portals on SeaStar

C. Scalability

CCI uses connection-oriented semantics with minimal per-connection resources. For the Portals driver, each connection requires 104 bytes on 64-bit machines (20 bytes for the public CCI connection struct, 20 bytes for the private CCI struct, and

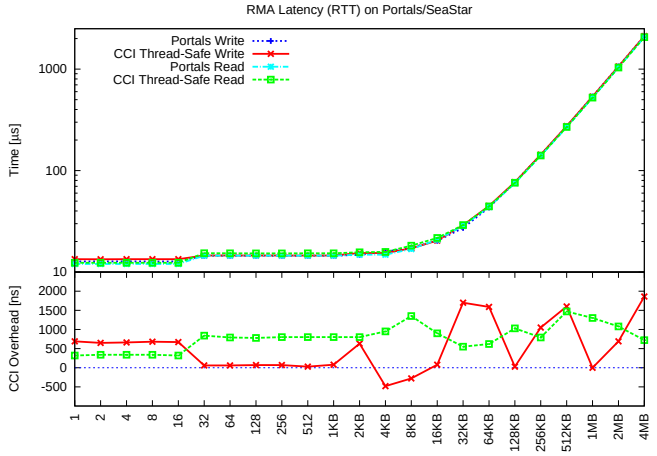

Fig. 3. RMA Latency when using Portals on SeaStar

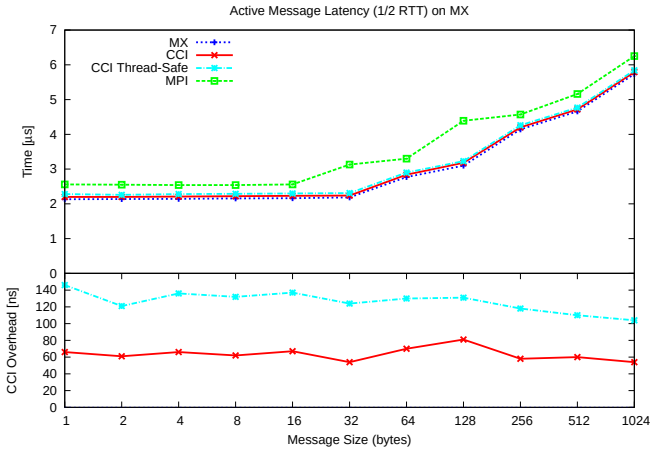

Fig. 4. Ping pong Latency when using MX

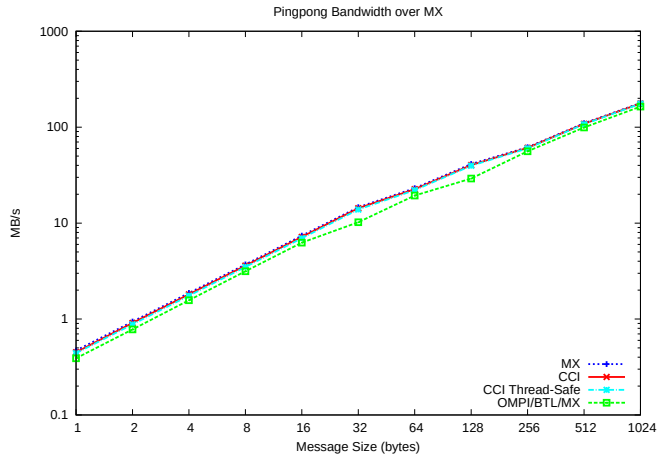

Fig. 5. Ping pong Bandwidth when using MX

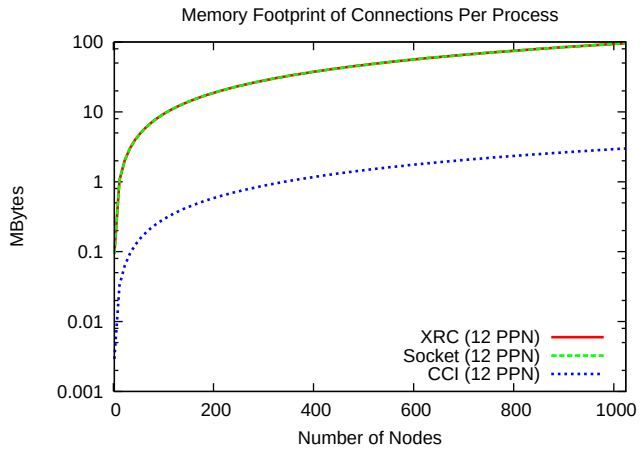

Fig. 6. Memory footprint for connections per process

64 bytes for the Portals driver connection struct). The sock driver requires 140 bytes for each connection.

Figure 6 illustrates the connection state for Verbs, BSD Sockets, and CCI. Verbs memory usage scales linearly [25] with the number of connected peers when Reliable Connected (RC) mode. This is the best case scenario for Verbs. The Sockets usage is derived from the minimum 4 KB page send and receive buffers and internal state tied to the connection.

The CCI usage conservatively assumes 256 bytes (assuming alingment padding, etc.) as used in the UDP driver. Obviously, if CCI is implemented over Verbs, for example, then this amount is on top of the underlying implementation. In order to provide support for Verbs hardware (InfiniBand, iWARP, etc.), we intend to use Verb's Unreliable Datagram (UD) mode to avoid the QP memory usage by Verb's Reliable Connected (RC) or eXtended Reliable Connected (XRC) modes.

VI. Conclusion

The need for high-performance, low-latency communication, once reserved primarily for HPC oriented applications now extends to a wide spectrum of markets such as cloud computing and web services. While these applications share similar performance requirements as their HPC counterparts, they differ in their need for an adaptive, elastic, distributed computing model. These applications have historically used the ubiquitous Sockets interface for portability, sacrificing performance and support for advanced networking features. As networking technologies have continued to advance, the gap between achievable performance (as demonstrated by HPC communication models such as MPI) and realized performance using the Sockets API has widened.

We have proposed CCI, a new networking interface to address this gap. Our design goals of portability, simplicity, performance, scalability, and robustness have been driven by the needs of a broad community of distributed computing application developers. CCI achieves portability through the use of a component architecture with a clean separation of API and underlying low-level network driver support. Similar to Sockets, simplicity has been achieved through a narrow API with well-defined semantics. High-performance is achieved through a low-overhead active message style semantic for small/control messages and RMA support for bulk data movement and zero-copy semantics. Our prototype implementation over MX achieves performance that is within 60ns of the native implementation. The use of shared resources and minimized per-peer state affords CCI substantially improved scalability characteristics over alternative API implementations. CCI maintains as little as 120 bytes per connection and shared resources on the order of 8 Megabytes per endpoint. Robustness is achieved through well-defined semantics for a variety of failure scenarios, allowing the application to respond appropriately to catastrophic network and remote end-point failure scenarios.

Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

References

[1] RFC791, "Internet protocol," September 1981, dARPA Internet Program Protocol Specification. [Online]. Available: http://www.ietf.org/rfc/rfc791.txt

[2] RFC793, "Transmission control protocol," September 1981, dARPA Internet Program Protocol Specification. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[3] S. Sechrest, "Tutorial examples of interprocess communication in Berkeley UNIX 4.2 BSD," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-84-191, Jun 1984. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1984/5970.html

[4] Message Passing Interface Forum, "MPI: A Message Passing Interface," in *Proc. of Supercomputing '93*. IEEE Computer Society Press, November 1993, pp. 878–883.

[5] A. P. Foong, T. R. Huff, H. H. Hum, J. P. Patwardhan, and G. J. Regnier, "Tcp performance re-visited," in *In IEEE International Symposium on Performance of Systems and Software*, 2003, pp. 70–79.

[6] E. Garbriel, G. Fagg, G. Bosilica, T. Angskun, J. J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[8] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2003, pp. 295–304.

[9] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovski, and J. J. Dongarra, "Fault tolerant communication library and applications for high perofrmance," in *Los Alamos Computer Science Institute Symposium*, Santa Fee, NM, October 27-29 2003.

[10] R. Batchu and et al, "MPI/FT TM: Architecture and taxonomies for fault-tolerant,message-passing middleware for performance-portable parallel computing," in *1st IEEE International Symposium of Cluster Computing and the Grid, 2001, pp. 26-33*, 2001.

[11] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: Extending the Message-Passing Interface," in *Euro-Par '96 Parallel Processing*. Springer Verlag, 1996, pp. 128–135.

[12] D. Bonachea, "The inadequacy of the MPI 2.0 one-sided communication API for implementing parallel global address-space languages."

[13] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe, "Portals 3.0: Protocol building blocks for low overhead communication," in *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, 2002.

[14] D. Bonachea, "GASNet specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

[15] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. Govindaraju, K. Gildea, P. DiNicola, and C. Bender, "Performance and experience with LAPI-a new high-performance communication library for the IBM RS/6000 SP," in *Proceedings of the First Merged International Symposium on Parallel and Distributed Processing*, 1998, pp. 260–266.

[16] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, B. Michael, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer," in *Proceedings of the 22nd annual international conference on Supercomputing*, ser. ICS '08. New York, NY, USA: ACM, 2008, pp. 94–103.

[17] T. von Eicken and W. Vogels, "Evolution of the virtual interface architecture," in *IEEE Computer, Vol. 31, pp. 61-68*, 1998.

[18] A. Bland, R. Kendall, D. Kothe, J. Rogers, and G. Shipman, "Jaguar: The world's most powerful computer," in *Proceedings of the Cray User Group Conference*, 2010.

[19] P. Braam, P. Schwan, and R. Brightwell, "Portals and networking for the lustre file system," in *IEEE International Conference on Cluster Computing*, 2002.

[20] J. Dongarra, H. Meuer, and E. Strohmaier, "Top500 supercomputing sites," http://www.top500.org, 2009.

[21] D. Cohen, T. Talpey, A. Kanevsky, U. Cummings, M. Krause, R. Recio, D. Crupnicoff, L. Dickman, and P. Grun, "Remote direct memory access over the converged enhanced ethernet fabric: Evaluating the options," in *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 123–130.

[22] J. Dongarra, "Impact of architecture and technology for extreme scale on software and algorithm design," in *The Department of Energy Workshop on Cross-cutting Technologies for Computing at the Exascale*, 2010.

[23] G. M. Shipman, T. S. Woodall, R. Graham, A. B. Maccabe, and P. G. Bridges, "InfiniBand scalability in Open MPI," in *In Proceedings, 20th IEEE International Parallel & Distributed Processing Symposium, Processing Letters*, 2006.

[24] G. M. Shipman, R. Brightwell, B. Barrett, J. M. Squyres, and G. Bloch, "Investigations on InfiniBand: Efficient network buffer utilization at scale," in *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[25] G. M. Shipman, S. Poole, P. Shamis, and I. Rabinovitz, "X-SRQ - improving scalability and performance of multi-core InfiniBand clusters," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 33–42. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87475-1_11

[26] "CCI: The Common Communication Interface, v0.1," Tech. Rep., 2011.

[27] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, Gold Coast, Australia, May 1992, pp. 430–440. [Online]. Available: http://www.cs.cmu.edu/~seth/papers/voneicken-isca92.pdf

[28] P. W. Ohio, P. Wyckoff, and J. Wu, "Memory registration caching correctness," in *In Proceedings of CCGrid05*. IEEE Computer Society, 2005.

[29] B. Goglin, "High-Performance Message Passing over generic Ethernet Hardware with Open-MX," *Elsevier Journal of Parallel Computing (PARCO)*, vol. 37, no. 2, pp. 85–100, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00533058

[30] S. W. Poole, A. Curtis, O. Hernandez, K. Feind, J. A. Kuehn, and G. M. Shipman, "OpenSHMEM: Towards a unified RMA model," Oak Ridge National Laboratory, Tech. Rep., 2011.