A Brief Introduction to the OpenFabrics Interfaces

A New Network API for Maximizing High Performance Application Efficiency

Paul Grun*, Sean Hefty†, Sayantan Sur†, David Goodell‡, Robert D. Russell§, Howard Pritchard¶, Jeffrey M. Squyres‡
*Cray, Inc. grun@cray.com
†Intel Corp. {sean.hefty, sayantan.sur}@intel.com
‡Cisco Systems, Inc. {dgoodell, jsquyres}@cisco.com
§UNH InterOperability Laboratory rdr@iol.unh.edu
¶Los Alamos National Laboratory howardp@lanl.gov

***Abstract*—OpenFabrics Interfaces (OFI) is a new family of application program interfaces that exposes communication services to middleware and applications. Libfabric is the first member of OFI and was designed under the auspices of the OpenFabrics Alliance by a broad coalition of industry, academic, and national labs partners over the past two years. Building and expanding on the goals and objectives of the verbs interface, libfabric is specifically designed to meet the performance and scalability requirements of high performance applications such as Message Passing Interface (MPI) libraries, Symmetric Hierarchical Memory Access (SHMEM) libraries, Partitioned Global Address Space (PGAS) programming models, Database Management Systems (DBMS), and enterprise applications running in a tightly coupled network environment. A key aspect of libfabric is that it is designed to be independent of the underlying network protocols as well as the implementation of the networking devices. This paper provides a brief discussion of the motivation for creating a new API, describes the novel requirements gathering process which drove its design, and summarizes the API's high-level architecture and design.**

***Keywords*-fabric; interconnect; networking; interface**

I. Introduction

OpenFabrics Interfaces, or OFI, is a framework focused on exporting communication services to applications. OFI is specifically designed to meet the performance and scalability requirements of high-performance computing (HPC) applications such as MPI, SHMEM, PGAS, DBMS, and enterprise applications running in a tightly coupled network environment. The key components of OFI are: application interfaces, provider libraries, kernel services, daemons, and test applications.

Libfabric is a library that defines and exports the user-space API of OFI, and is typically the only software that applications deal with directly. Libfabric is supported on commonly available Linux based distributions. The libfabric API is independent of the underlying networking protocols, as well as the implementation of particular networking devices over which it may be implemented.

OFI is based on the notion of application centric I/O, meaning that the libfabric library is designed to align fabric services with application needs, providing a tight semantic fit between applications and the underlying fabric hardware. This reduces overall software overhead and improves application efficiency when transmitting or receiving data over a fabric.

II. Motivations

The motivations for developing OFI evolved from experience gained by developing OpenFabrics Software (OFS), which is produced and distributed by the OpenFabrics Alliance (OFA) [1]. Starting as an implementation of the InfiniBand Trade Association (IBTA) Verbs specification [2], OFS evolved over time to include support for both the iWARP [3–5] and RoCE [6] specifications. As use of these technologies grew, new ideas emerged about how to best access the features available in the underlying hardware. New applications appeared with the potential to utilize networks in previously unanticipated ways. In addition, demand arose for greatly increased scalability and performance. More recently new paradigms, such as Non-Volatile Memory (NVM), have emerged.

All the above events combined to motivate a "Birds-of-a-Feather" session at the SC13 Conference. The results of that session provided the impetus for the OFA to form a new project known as OpenFabrics Interfaces, and a new group called the OpenFabrics Interface Working Group (OFIWG). From the beginning, this project and the associated working group sought to engage a wide spectrum of user communities who were already using OFS, who were already moving beyond OFS, or who had become interested in high-performance interconnects. These communities were asked to contribute their ideas about the existing OFS and, most importantly, they were asked to describe their requirements for interfacing with high-performance interconnects.

The response was overwhelming. OFIWG spent months interacting with enthusiastic representatives from various groups, including MPI, SHMEM, PGAS, DBMS, NVM and others. The result was a comprehensive requirements document containing 168 specific requirements. Some requests were educational – complete, on-line documentation. Some were practical – a suite of examples and tests. Some were organizational – a well-defined revision and distribution mechanism. Some were obvious but nevertheless challenging

– scalability to millions of communication peers. Some were specific to a particular user community – provide tag-matching that could be utilized by MPI. Some were an expansion of existing OFS features – provide a full set of atomic operations. Some were a request to improved existing OFS features – redesign memory registration. Some were aimed at the fundamental structure of the interface – divide the world into applications and providers, and allow users to select specific providers and features. Some were entirely new – provide remote byte-level addressing.

After examining the major requirements, including a requirement for independence from any given network technology and a requirement that a new API be more abstract than other network APIs and more closely aligned with application usage, the OFIWG concluded that a new API based solely on application requirements was the appropriate direction.

III. Architectural Overview

Figure 1 highlights the general architecture of the two main OFI components, the libfabric library and an OFI provider, shown situated between OFI enabled applications and a hypothetical NIC that supports process direct I/O.

The libfabric library defines the interfaces used by applications, and provides some generic services. However, the bulk of the OFI implementation resides in the providers. Providers hook into libfabric and supply access to fabric hardware and services. Providers are often associated with a specific hardware device or NIC. Because of the structure of libfabric, applications access the provider implementation directly for most operations in order to ensure the lowest possible software latencies.

As captured in Figure 1, libfabric can be grouped into four main services.

A. Control Services

These are used by applications to discover information about the types of communication services available in the system. For example, discovery will indicate what fabrics are reachable from the local node, and what sort of communication each fabric provides.

Discovery services allow an application to request specific features, or capabilities, from the underlying provider, for example, the desired communication model. In response, a provider can indicate what additional capabilities an application may use without negatively impacting performance or scalability, as well as requirements on how an application can best make use of the underlying fabric hardware. A provider indicates the latter by setting mode bits which encode restrictions on an application's use of the interface. Such restrictions are due to performance reasons based on the internals of a particular provider's implementation.

The result of the discovery process is that a provider uses the application's request to select a software path that is best suited for both that application's needs and the provider's restrictions.

B. Communication Services

These services are used to set up communication between nodes. They include calls to establish connections (connection management) as well as the functionality used to

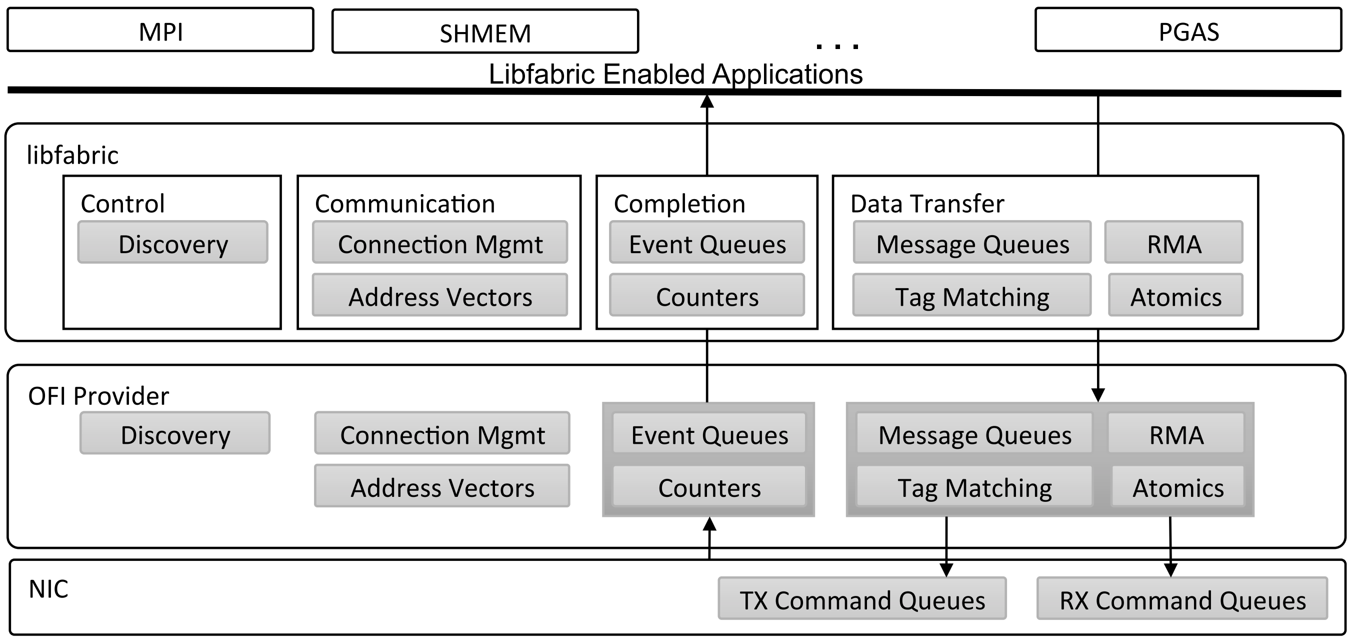


Figure. 1: Architecture of libfabric and an OFI provider layered between applications and a hypothetical NIC

address connectionless endpoints (address vectors). Communication interfaces are designed to abstract fabric and hardware specific details used to connect and configure communication endpoints.

The connection interfaces are modeled after sockets to support ease of use. However, address vectors are designed around minimizing the amount of memory needed to store addressing data for potentially millions of remote peers.

C. Completion Services

Libfabric exports asynchronous interfaces, and completion services are used by a provider to report directly to an application the results of previously initiated asynchronous operations. Completions may be reported either by using event queues or lower-impact counters. Entries in event queues provide details about the completed operation in several formats that an application can select in order to minimize the data that must be set by the provider. Counters simply report the number of completed operations.

D. Data Transfer Services

These services are sets of interfaces designed around different communication paradigms. Figure 1 shows four basic data transfer interface sets. These data transfer services give an application direct access to the provider's implementation of the corresponding service.

(i) Message queues expose the ability to send and receive data in which message boundaries are maintained. They act as FIFOs, with messages arriving from a remote sender being matched with enqueued receive requests in the order that they are received by the local provider.

(ii)Tag matching is similar to message queues in that it maintains message boundaries, but differs in that received messages are directed to enqueued receive requests based on small steering tags that are carried in the message.

(iii) RMA stands for "Remote Memory Access". RMA transfers allow an application to write data from local memory directly into a specified memory location in a target process, or to read data into local memory directly from a specified memory location in a target process.

(iv) Atomic operations are similar to RMA transfers in that they allow direct access to a specified memory location in a target process, but are different in that they allow for manipulation of the value found in that memory, such as incrementing or decrementing it.

Data transfer interfaces are designed to eliminate branches that would occur within the provider implementation and reduce the number of memory references, for example, by enabling it to preformat command buffers to further reduce the number of instructions executed in a transfer.

IV. Object Model

The libfabric architecture is based on object-oriented design concepts. At a high-level, individual fabric services are associated with a set of interfaces. For example, RMA services are accessible using a set of well-defined functions. Interface sets are associated with objects exposed by libfabric. The relationship between an object and an interface set is roughly similar to that between an object-oriented class and its member functions, although the actual implementation differs for performance and scalability reasons.

An object is configured based on the results of the discovery services. In order to enable optimized code paths between the application and fabric hardware, providers dynamically associate objects with interface sets based on the modes supported by the provider and the capabilities requested by the application.

Figure 2 shows a high-level view of the parent-child relationships between libfabric objects.

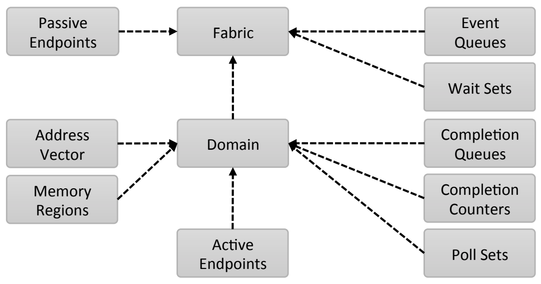


Figure. 2: Object Model of libfabric

(i) Fabric: A fabric represents a collection of hardware and software resources that access a single physical or virtual network. All network ports on a system that can communicate with each other through the fabric belong to the same fabric domain. A fabric not only includes local and remote NICs, but corresponding software, switches, routers, and any necessary fabric or subnet management components.

(ii) Domain: A domain represents a logical connection into a fabric. For example, a domain may map to a physical or virtual NIC. A domain defines the boundary within which fabric resources may be associated. Each domain belongs to a single fabric.

The properties of a domain describe how associated resources will be used. Domain attributes include information about the application's threading model, and how fabric resources may be distributed among threads. It also defines interactions that occur between endpoints, completion queues and counters, and address vectors. The intent is for an application to convey enough information that the provider can select an optimized implementation tailored to its needs.

(iii) Passive Endpoint: Passive endpoints are used by connection-oriented protocols to listen for incoming connection requests, conceptually equivalent to listening sockets.

(iv) Active Endpoint: An active endpoint (or, simply, endpoint) represents a communication portal, and is conceptually similar to a socket. All data transfer operations are initiated on endpoints.

Endpoints are usually associated with a transmit context and/or a receive context. These contexts are often implemented using hardware queues that are mapped directly into the process's address space, which enables bypassing the operating system kernel for data transfers. Data transfer requests are converted by the underlying provider into commands that are inserted into transmit and/or receive contexts.

A more advanced usage model of endpoints allows for resource sharing. Because transmit and receive contexts may be associated with limited hardware resources, libfabric defines mechanisms for sharing contexts among multiple endpoints. Shared contexts allow an application or resource manager to prioritize where resources are allocated and how shared hardware resources should be used.

In contrast with shared contexts, the final endpoint model is known as a scalable endpoint. Scalable endpoints allow a single endpoint to take advantage of multiple underlying hardware resources by having multiple transmit and/or receive contexts. Scalable contexts allow applications to separate resources to avoid thread synchronization or data ordering restrictions, without increasing the amount of memory needed for addressing.

(v) Event Queue: An event queue (EQ) is used to collect and report the completion of asynchronous operations and events. It handles control events that are not directly associated with data transfer operations, such as connection requests and asynchronous errors.

(vi) Completion Queue: A completion queue (CQ) is a high-performance queue used to report the completion of data transfer operations. Transmit and receive contexts are associated with completion queues. The format of events read from a completion queue is determined by an application. This enables compact data structures with minimal writes to memory. Additionally, the CQ interfaces are optimized around reporting operations that complete successfully, with error completions handled "out of band". This allows error events to report additional data without incurring additional overhead that would be unnecessary in the common case of a successful transfer.

(vii) Completion Counter: A completion counter is a lightweight alternative to a completion queue, in that its use simply increments a counter rather than placing an entry into a queue. Similar to CQs, an endpoint is associated with one or more counters. However, counters provide finer granularity in the types of completions that they can track.

(viii) Wait Set: A wait set provides a single underlying wait object to be signaled whenever a specified condition occurs on an event queue, completion queue, or counter belonging to the set. Wait sets enable optimized methods for suspending and signaling threads. Applications can request that a specific type of wait object be used, such as a file descriptor, or allow the provider to select an optimal object. The latter grants flexibility in current or future underlying implementations.

(ix) Poll Set: Although libfabric is architected to support providers that offload data transfers directly into hardware, it supports providers that use the host CPU to progress operations. Libfabric defines a manual progress model where the application agrees to use its threads for this purpose, avoiding the need for underlying software libraries to allocate additional threads. A poll set enables applications to group together completion queues or counters, allowing one poll call to make progress on multiple completions.

(x) Memory Region: A memory region describes an application's local memory buffers. In order for a fabric provider to access application memory during certain types of data transfer operations, such as RMA and atomic operations, the application must first grant the appropriate permissions to the fabric provider by constructing a memory region.

Libfabric defines multiple modes for creating memory regions. It supports a method that aligns well with existing InfiniBand^{TM} and iWARP^{TM} hardware, but, in order to scale to millions of peers, also allows for addressing using offsets and user-specified memory keys.

(xi) Address Vector: An address vector is used by connectionless endpoints to map higher-level addresses which may be more natural for an application to use, such as IP addresses, into fabric-specific addresses. This allows providers to reduce the amount of memory required to maintain large address look-up tables, and to eliminate expensive address resolution and look-up methods during data transfer operations.

Libfabric borrowed and expanded on concepts found in other APIs, then brought them together in an extensible framework. Additional objects can easily be introduced, or new interfaces to an existing object can be added. However, object definitions and interfaces are designed specifically to promote software scaling and low-latency, where needed. Effort went into ensuring that objects provided the correct level of abstraction in order to avoid inefficiencies in either the application or the provider.

V. Current State

An initial (1.0) release of libfabric is now available [7, 8] with complete user-level documentation ("man pages") [9]. New releases are planned quarterly. This release provides enough support for HPC applications to adapt to using its interfaces. Areas where improvements can be made should be reported back to the OFI working group, either by posting concerns to the ofiwg mailing list, bringing it to the work group's attention during one of the weekly conference calls, or by opening an issue in the libfabric GitHub^{TM} database. Although the API defined by the 1.0 release is intended to enable optimized code paths, provider optimizations that take advantage of those features will be phased in over the next several releases.

The 1.0 release supports several providers. A sockets provider is included for developmental purposes, and runs on both Linux and Mac OS X systems. It implements the full set of features exposed by libfabric. A general verbs provider allows libfabric to run over hardware that supports the libibverbs interface. A "usnic" provider supports the `usNIC` (user-space NIC) feature of Cisco's Virtualized Interface Card (VIC) hardware. Finally, the PSM provider supports Intel's Performance Scaled Messaging (PSM) interface.

In addition to the current providers, support for additional hardware is actively under development. A Cray Aries network provider will be available in a post 1.0 release. An MXM provider will enhance libfabric support for Mellanox hardware. And future support will also include Intel's new Omni Path Architecture. Optimizations are also under development for select hardware and vendors. The details of this work will become available as it moves closer to completion, and may be tracked through the GitHub repositories.

VI. Analysis of the Interface

The libfabric interfaces aim to achieve multiple objectives. Among them are hardware implementation independence, improved software scalability, and decreased software overhead. In order to analyze whether the proposed interfaces meet these objectives, we provide a comparison of using libfabric alongside a widely used interface for HPC middleware, libibverbs. In order to ensure as fair a comparison as reasonable, we restrict the comparison to both using InfiniBand based hardware, the architecture that libibverbs is based upon. Additionally, we focus on the impact that the API itself has on application performance, not differences that may arise as a result of the underlying implementation.

A. *Scalability*

The address vector interfaces of libfabric are specifically designed to improve software scalability. For the purposes of this comparison, we analyze the memory footprint needed to access peer processes when using an unconnected endpoint. A summary for a 64-bit platform is shown in Figure 3.

libibverbs with InfiniBand			libfabric with InfiniBand		
Type	**Data**	**Size**	**Type**	**Data**	**Size**
struct *	ibv_ah	8	uint64	fi_addr_t	8
uint32	QPN	4			
uint32	QKey	4 [0]			
ibv_ah					
struct *	ibv_context	8			
struct *	ibv_pd	8			
uint32	handle	4			
[uint32]	[padding]	0 [4]			
Total		**36**			**8**

Figure. 3: libibverbs compared to libfabric when accessing an unconnected endpoint

An application that uses the libibverbs interface requires a total of 36 bytes of related addressing metadata for every remote peer. Applications submitting a transfer request must provide 3 input parameters: a pointer to an address handle structure (ibv_ah), a destination queue pair number (QPN), and a queue key (qkey). For this analysis, it is assumed that an application uses a single qkey, which may be ignored. An address handle is required to send to each peer process, and is allocated by the libibverbs library. The minimal size of the address handle is given in Figure 3, and consists of two pointers, plus a 32-bit kernel identifier or handle. To account for data alignment, we include 4 additional bytes of padding which we will make use of below.

With libfabric, data transfers require that applications provide an input value of data type `fi_addr_t`. This is defined as a uint64. Address vectors in libfabric can be one of two different types: `FI_AV_TABLE` or `FI_AV_MAP`. In the case of `FI_AV_TABLE`, applications reference peers using a simple index. For applications, such as MPI or SHMEM, the index can be mapped either to rank or PE number, eliminating any application level storage needed for addressing. With `FI_AV_MAP`, however, applications must store an opaque `fi_addr_t` address, which requires 8 bytes of storage per peer.

It should be noted that the memory footprint required by an application to use an interface is meaningless if the metadata is simply moved underneath the interface. Although the amount of metadata ultimately required is fabric specific, for InfiniBand hardware, the metadata required to send a transfer to an unconnected queue pair is shown in Figure 4.

IB Data:	**DLID**	**SL**	**QPN**
Size:	2	1	3

Figure. 4: InfiniBand metadata for an unconnected transfer

An InfiniBand path within a subnet is defined as a tuple: `<SLID, DLID, SL>`. Packets carry this metadata when transferring between peer endpoints. As mentioned, unconnected transfers also require a qkey (which may be constant for a given job) and a qpn (which is chosen randomly for each peer queue pair). With the `SLID` determined by the local endpoint, to reach a given peer requires storing the tuple: `<DLID, SL, QPN>`. As shown in Figure 4, this requires 6 bytes of metadata. (Note, that the `SL` is only 4-bits, but expanded to use a full byte.) The `QPN` is already accounted for in the libibverbs API Figure 3. However, a libibverbs provider must maintain the `DLID` and `SL` for each destination. We propose that a provider may store this metadata in the address handle (ibv_ah) in the space where a compiler would normally provide structure padding. This optimally keeps the memory footprint to 36 bytes per peer.

Using similar metadata, Figure 4 shows that the libfabric address vector of type `FI_AV_TABLE` reduces the memory footprint to only 6 bytes, which requires a table lookup on any transfer call, a cost similar to dereferencing a pointer to struct ibv_ah. If `FI_AV_MAP` is used, the memory footprint remains at 8 bytes per peer, because the `fi_addr_t` encodes the `<DLID, SL, QPN>` directly, with the extra bits unused.

With both address vector types, the memory footprint is reduced approximately 80%. Application selection of address vector type then becomes a matter of further optimizing for memory footprint or execution time on transfers.

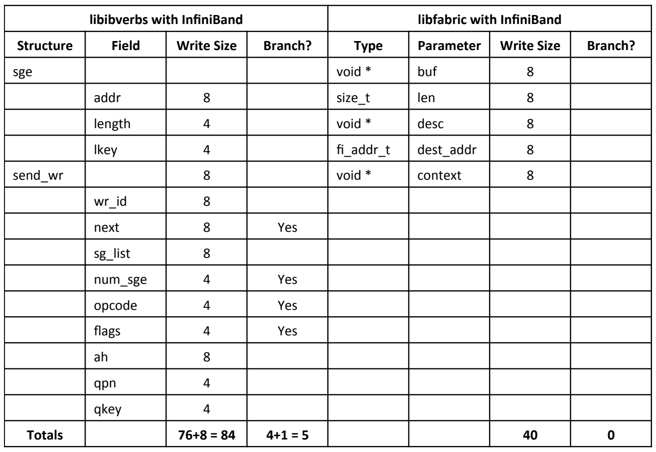
B. Performance Optimizations

To analyze performance optimizations, we examine the transmit code path that results from the interface definition itself. For this analysis, we measure the count and number of bytes of memory writes that an application must invoke in order to use the interface. Additionally, we determine where the API results in the underlying provider needing to take conditional branches, including structured loops, in order to ensure correct operation. Figure 5 summarizes this analysis.

The libibverbs interface uses a single interface, `ibv_post_send`, into the provider to transmit data, with the following prototype:

```
int ibv_post_send(struct ibv_qp *qp,
              struct ibv_send_wr *wr,
              struct ibv_send_wr **bad_wr)
```

In order to send a transfer to an unconnected peer, the fields shown in Figure 5 must be filled out. Additionally, `wr`, a reference to the send work request structure, plus the `bad_wr` parameter, must be written to the stack. The net result is that 14 memory writes for 84 total bytes of metadata must be written by the application as part of the transmit operation. (The `qp` parameter and its corresponding libfabric `ep` parameter are ignored for this study.)



libibverbs with InfiniBand				libfabric with InfiniBand			
Structure	Field	Write Size	Branch?	Type	Parameter	Write Size	Branch?
sge				void *	buf	8	
	addr	8		size_t	len	8	
	length	4		void *	desc	8	
	lkey	4		fi_addr_t	dest_addr	8	
send_wr		8		void *	context	8	
	wr_id	8					
	next	8	Yes				
	sg_list	8					
	num_sge	4	Yes				
	opcode	4	Yes				
	flags	4	Yes				
	ah	8					
	qpn	4					
	qkey	4					
Totals		76+8 = 84	4+1 = 5			40	0

Figure. 5: Transmit code path comparison

Furthermore, if we examine the work request fields, we can identify those which will result in branches in the underlying provider. Send work requests may be chained together, and the number of SGEs is variable. Both of these fields result in for-loops in the provider code. The type of operation is specified through the opcode field. Since the opcode and flags fields determine which of the other fields of the work request are valid, providers must check these fields and act accordingly. Lastly, libibverbs defines a single entry point into the provider. Transfer operations on all queue pairs, regardless of their type, branch off from the single `ibv_post_send` entry point. In total, a transmit call will take at least 5 branches before the request can be written to hardware. (A code review of one provider showed that at least 19 branches would be taken, though we suspect that number could be reduced through code restructuring.)

Libfabric, on the other hand, associates entry points per endpoint, and provides multiple calls, similar to the socket transfer calls `send, write, writev,` and `sendmsg`. Data transfer flags are specified as part of endpoint initialization, which enables them to be removed from the transmit path. For a transmit call that sends a single message, the libfabric API requires that applications write 5 values onto the stack, for 40 total bytes of metadata. The API itself does not result in provider branches.

VII. Summary

The libfabric interfaces were co-designed between application developers and hardware providers. The result is an API not just tailored to meet application needs, but also designed to allow for efficient implementation. Libfabric was designed to map well to MPI, SHMEM, PGAS, DBMS, and socket applications. In addition, strong consideration was given to how those interfaces could be implemented over a variety of hardware and fabrics.

The initial focus of the OFIWG was quickly enabling libfabric over a variety of hardware, so that application developers could begin porting their applications and exploring its use. The next phase of libfabric development is focused on the creation of native, optimized providers.

References

[1] OpenFabrics Alliance, "http://www.openfabrics.org."

[2] Infiniband Trade Association, "Infiniband Architecture Specification Volume 1, Release 1.2.1," Nov. 2007.

[3] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A Remote Direct Memory Access Protocol Specification," RFC 5040, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5040.txt

[4] H. Shah, J. Pinkerton, R. Recio, and P. Culley, "Direct Data Placement over Reliable Transports," RFC 5041, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5041.txt

[5] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier, "Marker PDU Aligned Framing for TCP Specification," RFC 5044, Oct. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5044.txt

[6] Infiniband Trade Association, "Supplement to Infiniband Architecture Specification Volume 1, Release 1.2.1: Annex A16: RDMA over Converged Ethernet (RoCE)," Apr. 2010.

[7] OpenFabrics Interfaces, "https://github.com/ofiwg/libfabric."

[8] Libfabric Programmer's Manual, "http://ofiwg.github.io/libfabric."

[9] Libfabric man pages v1.0.0 release, "http://ofiwg.github.io/libfabric/v1.0.0/man."