A Brief Introduction to the OpenFabrics Interfaces

A New Network API for Maximizing High Performance Application Efficiency

Paul Grun
Cray, Inc.
Advanced Technology
Seattle, WA, USA
grun@cray.com

Sean Hefty
Software Solutions Group
Intel Corp.
Hillsboro, OR, USA
sean.hefty@intel.com

Sayantan Sur
Intel Corp.
sayantan.sur@intel.com

David Goodell
Cisco Systems, Inc.
dgoodell@cisco.com

Robert Russell
InterOperability Laboratory
University of New Hampshire
Durham, NH, USA
rdr@iol.unh.edu

Howard Pritchard
Los Alamos National Lab
howardp@lanl.gov

Jeffrey Squyres
Cisco Systems, Inc.
jsquyres@cisco.com

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 this family of interfaces 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 and describes the novel requirements gathering process that was used to drive its design. Next, we provide a high level overview of the API architecture and design, and finally we discuss the current state of development, release schedule and future work.**

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. Libfabric is independent of the underlying networking protocols, as well as the implementation of the networking devices.

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 by many different classes of users of the existing 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], this software evolved to deal with both the iWARP specifications [3–5] and later the IBTA RoCE specifications [6], as well as with a series of enhancements to InfiniBand and its implementations. As was inevitable during the rapid growth of these technologies, new ideas emerged about how users and middleware could best access the features available in the underlying hardware. In addition, new applications appeared with the potential to utilize network interconnects in unanticipated ways. Finally, whole new paradigms, such as Non-Volatile Memory (NVM) matured to the stage where closer integration with RDMA became both desirable and feasible.

At the 2013 SuperComputing Conference a "Birds-of-a-Feather" (BoF) meeting was held to discuss these ideas, and from that came the impetus to form what eventually came to be know as the OpenFabrics Interface Working Group (OFIWG). From the beginning, this group sought to embrace a wide spectrum of user communities who either were already using OFS, who were already moving beyond OFS, or who had become interested in high-performance interconnects. These communities were contacted about contributing their ideas, complaints, suggestions, requirements, etc. about the existing and future state of OFS, as well as about interfacing with high-performance interconnects in general.

The response was overwhelming. OFIWG spent months interacting with many enthusiastic representatives from the various groups, including but not limited to MPI, SHMEM, PGAS, DBMS, and NVM. The result was a cumulative requirements document containing 168 specific requirements. Some requests were educational – we need good user-level on-line documentation from the beginning. Some were organizational – we need a well-defined revision and distribution mechanism. Some were practical – we need a well-defined suite of examples and tests. Some were very "motherhood and apple-pie" – the future requires scalability to millions of communication peers. Some were very 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 revisit 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 appropriate providers by interrogating them to discover the features they are able to provide. 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 for an API which is more abstract than other network APIs and hence more closely aligned with application usage of the API, the OFIWG concluded that a new API based solely on application requirements was

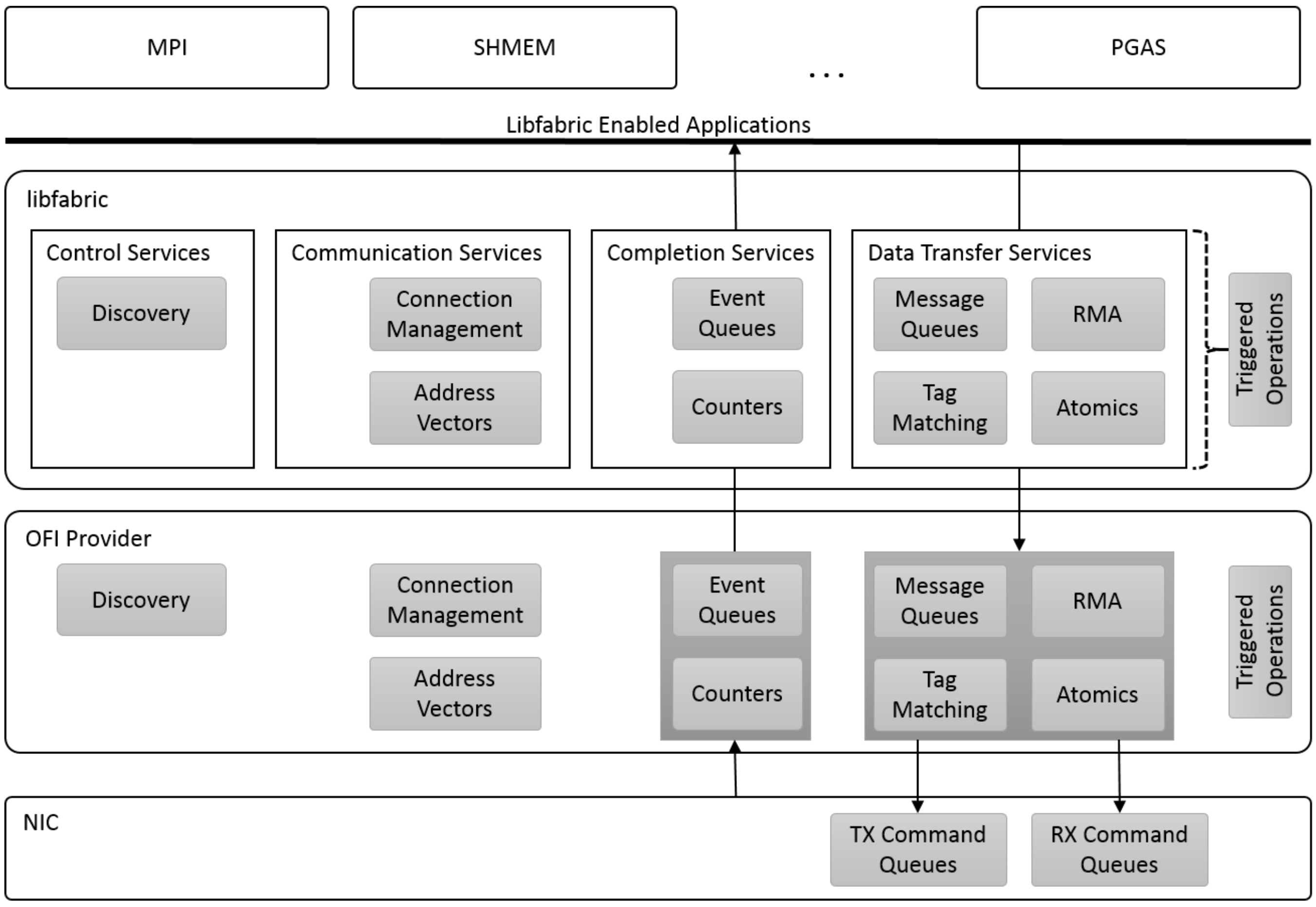

Figure. 1: Architecture of libfabric and an OFI provider layered between applications and a hypothetical NIC

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, as they are 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 plug 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.

The discovery services are defined such that an application can request specific features, referred to as capabilities, from the underlying provider, for example, the desired communication model. In responding, a provider can indicate what additional capabilities an application may use without negatively impacting performance or scalability. In order to ensure the tightest semantic fit between the application software and the underlying fabric hardware, a provider can use the discovery interface to inform applications on how it may best be used. It does this by setting mode bits which specify limitations on how a provider should be used in order to provide the best performance. Mode bits are restrictions that often have a lower impact on performance if implemented by higher level software than at the provider level.

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 that application's needs.

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 address connectionless endpoints (address vectors).

Connection interfaces are integrated directly into libfabric, which allows hiding fabric and hardware specific details used to connect and configure communication endpoints. Although libfabric does not define a connection protocol, it supports a 3-way handshake that is commonly used. Libfabric also allows for applications to exchange application specific data as part of their connection setup.

Address vectors are designed around minimizing the amount of memory needed to store addressing data for potentially millions of remote peers. A remote peer can be referenced by simply assigning it an index into an address vector, which maps well to applications such as MPI and SHMEM. Alternatively, a remote peer can be referenced using a provider specified address, an option that allows data transfer calls to pass encoded addressing data directly to the hardware, avoiding additional memory reads during the data transmission.

C. Completion Services

Libfabric exports asynchronous interfaces, and completion services are used to report the results of previously initiated asynchronous operations. Completions may be reported either by using event queues, which provide details about the operation that completed, or by lower-impact counters, which simply return the number of operations that have completed.

Applications select the type of completion structure. Different completion structures provide varying amounts of information about a completed request. This allows for creating compact arrays of completion structures, and ensures that data fields that would otherwise be ignored by an application are not filled in by a provider.

D. Data Transfer Services

These services are sets of interfaces designed around different communication paradigms. Figure 1 shows four basic data transfer interface sets, plus triggered operations that are strongly related to the data transfer operations.

(i) Message queues expose the ability to send and receive data in which message boundaries are maintained. They act as FIFOs, with sent messages matched with receive requests in the order that messages are received.

(ii)Tag matching is similar to message queues in that it maintains message boundaries, but differs in that received messages are directed to receive requests based on small steering tags that are carried in the sent 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 the differ in that they allow for manipulation of the value found in that memory, such as incrementing or decrementing it.

Different data transfer interfaces are defined in order to eliminate branches that would occur in the provider if only a single, more general function call were defined. This eliminates checks inside the provider, 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. These interface sets are then 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. Providers dynamically associate objects with interface sets based on the modes supported by the provider and the capabilities requested by the application. This enables optimized code paths between the application and fabric hardware that is based on the expected usage model, with the usage conveyed during initialization.

Figure 2 shows a high-level view of the parent-child relationships between libfabric objects, with details described below.

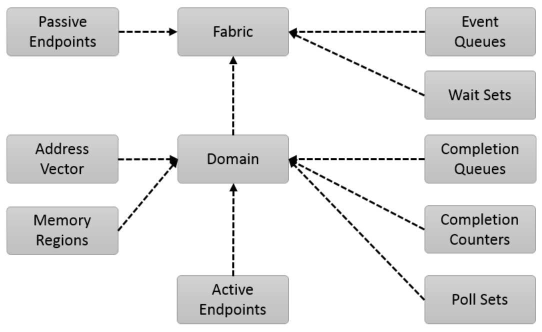

Figure. 2: Object Model of libfabric

(i) Fabric: A fabric represents a collection of hardware and software resources that access a single physical or virtual network. For example, a fabric may be a single network subnet. All network ports on a system that can communicate with each other through the fabric belong to the same fabric domain. A fabric can share network addresses with multiple providers.

A fabric not only includes local and remote NICs, but corresponding software, switches, routers, and any necessary fabric or subnet management components. Fabrics are identified by a named string and encapsulate network based events and topology data.

(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 data 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, they are equivalent to listening sockets. Passive endpoints often map to software constructs, and may span multiple domains.

(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. Transmit and receive 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, though OFI does not require this implementation. 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. An application can direct data transfers to use a specific context, or the provider can select which context to use. Each context may be associated with its own completion queue. 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.

Applications directly or indirectly control the types of events that are received on an event queue. An event queue supports an interface that is flexible enough to handle almost any type of event in an efficient manner.

(vi) Completion Queue: A completion queue (CQ) is a high-performance queue used to report the completion of data transfer operations. An endpoint is associated with one or more completion queues. An endpoint may direct completed transmit and receive operations to separate completion queues, or the same queue. 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 completions for operations that completed 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. Completion counters and queues may be used together.

(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 applications to use in place of more generic native operating system constructs. 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 data transfer operations. In the latter case, a thread running on the host CPU may be needed to execute software needed to complete a data transfer. Libfabric defines a manual progress model where the application agrees to use its threads for this purpose, which avoids the need for additional threads being allocated by the underlying software libraries. A poll set is defined to optimize for this situation. A poll set allows an application to group together multiple objects such that progress can be driven across all associated data transfers.

(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$^{\text{TM}}$ and iWARP$^{\text{TM}}$ hardware, but also allows for mechanisms needed to scale to millions of parallel peers.

(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.

The libfabric object model that has been defined is extensible. 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]. 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$^{\text{TM}}$ 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. The sockets provider runs on both Linux and Mac OS X systems and implements the full set of features exposed by libfabric. A general verbs provider allows libfabric to run over hardware that supports the libibverbs interface. The verbs provider is limited to supporting only connection-oriented endpoints and select data transfer services (message queue and RMA), but future versions will add support for unconnected endpoints and all data transfer services. A `usNIC` (user-space NIC) provider supports Cisco’s usNIC Ethernet hardware. The `usNIC` provider enables direct hardware access for UDP based applications. It supports libfabric’s connectionless endpoints and message queue service. Future versions of the `usNIC` provider will increase the types of endpoints and data transfer services that are supported. The last provider supports Intel’s Performance Scaled Messaging (PSM) interface. The `PSM` provider allows direct application access to the InfiniBand link layer. It supports all data transfer services over reliable, unconnected endpoints. Future support will include Intel’s new Omni Path Architecture, and the Cray Aries Network.

In addition to the current OFI providers, support for additional hardware is actively under development. Optimizations are also under development for select hardware and vendors. The details of this work will become available as it moves closer to completion.

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."