Utilizing HPC Network Technologies in High Energy Physics Experiments

Jörn Schumacher
on behalf of the ATLAS Collaboration
CERN
Geneva, Switzerland
joern.schumacher@cern.ch

ATL-DAQ-PROC-2017-020
10 July 2017

***Abstract*—Because of their performance characteristics, high-performance fabrics like Infiniband or OmniPath are interesting technologies for many local area network applications, including data acquisition systems for high-energy physics experiments like the ATLAS experiment at CERN. This paper analyzes existing APIs for high-performance fabrics and evaluates their suitability for data acquisition systems in terms of performance and domain applicability.**

The study finds that existing software APIs for high-performance interconnects are focused on applications in high-performance computing with specific workloads and are not compatible with the requirements of data acquisition systems. To evaluate the use of high-performance interconnects in data acquisition systems, a custom library called NetIO has been developed and is compared against existing technologies.

NetIO has a message queue-like interface which matches the ATLAS use case better than traditional HPC APIs like MPI. The architecture of NetIO is based on an interchangeable back-end system which supports different interconnects. A libfabric-based back-end supports a wide range of fabric technologies including Infiniband. On the front-end side, NetIO supports several high-level communication patterns that are found in typical data acquisition applications like client/server and publish/subscribe. Unlike other frameworks, NetIO distinguishes between high-throughput and low-latency communication, which is essential for applications with heterogeneous traffic patterns. This feature of NetIO allows experiments like ATLAS to use a single network for different traffic types like physics data or detector control.

Benchmarks of NetIO in comparison with the message queue implementation ØMQ are presented. NetIO reaches up to 2x higher throughput on Ethernet and up to 3x higher throughput on FDR Infiniband compared to ØMQ on Ethernet. The latencies measured with NetIO are comparable to ØMQ latencies.

I. Introduction

Data-acquisition (DAQ) systems for High-Energy Physics experiments like the Large Hadron Collider experiment ATLAS ([1], Figure 1) are designed to process and filter the data generated by the experiment. The physics processes under study are typically very rare, and thus the experiments generate a large amount of data to gain statistical significance. In normal operation the detectors of the ATLAS experiment generate 60 TB/s of raw data. The data are pre-filtered in hardware. The pre-filtered data are fed into the ATLAS DAQ system at a rate in the order of hundred Gigabytes per second.

To process data at this rate in real-time, the ATLAS DAQ system is designed as a distributed system composed of thousands of interconnected compute nodes organized in a complex architecture. A fast interconnect is an essential component of the ATLAS DAQ system that has a direct impact on overall system performance.

The traffic requirements on the DAQ network are heterogeneous. On the one hand the high data rates require a network with a large bandwidth. For the network software this implies the use of adequate buffering techniques to provide a high throughput and good link utilization. On the other hand systems like detector control, monitoring, or calibration rely on low latencies for message transfers while the required bandwidth is low. This is crucial for safe operation of the detector. For example, a sudden rise of temperature in one of the experiment's detector components needs to be detected and handled with appropriate counter measures within a short window of time to protect the sensitive electronics from damage. The network technology and the network software stack need to incorporate these heterogeneous requirements.

The requirements of ATLAS on the network software stack are considerably different than typical HPC requirements. A typical paradigm in HPC that is manifested in APIs like MPI or PGAS is that a large computing problem is subdivided into smaller chunks that are processed by individual nodes in a cluster. The actual network topology is transparent to the application. Communication in HPC applications is typically

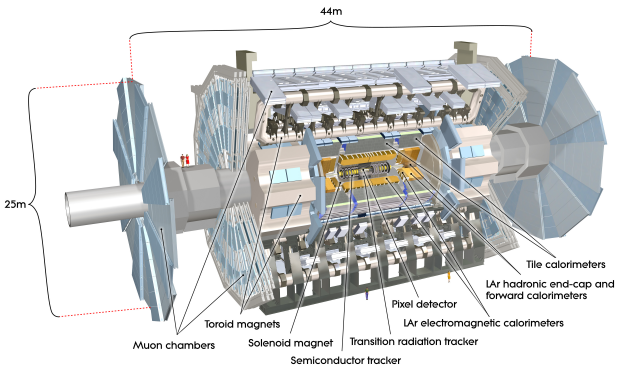

Fig. 1. The detectors of the ATLAS experiment. The experiment is one of the four experiments at the Large Hadron Collider, a circular particle collider with a 27 km circumference. The experiment itself is housed in an underground cavern circa 100 m below surface level.

very structured and nodes are organized in fixed topologies like a grid or mesh. In contrast, DAQ systems are organized in much more complex architectures. The ATLAS DAQ system is composed of several sub-systems for specific functions like data buffering, event building, event filtering, storage and others. The sub-systems are interconnected using different communication patterns such as push/pull, request/reply, or publish/subscribe, and employ techniques for dynamic load balancing.

Furthermore, DAQ applications need to be robust against failure. If a node in the network fails or a software application crashes, traffic needs to be redirected to other nodes that need to process the additional load. Data loss up to a certain degree is tolerated, as partial event data can still be meaningful for physics analysis. However, the system needs to recover from error scenarios quickly, ideally within seconds. For example, in the case of a failure of a data buffer, events that were stored in this buffer are lost. In contrast, in HPC applications data loss is unacceptable since every node contributes to the end result of a computation. However, since there are no real time requirements a job can be restarted or reverted to a checkpoint in case of a system failure.

For these reasons DAQ network applications are built using dynamic communication patterns like client/server or publish/subscribe. Robustness and error tolerance are easier to achieve than with message passing APIs. High-level communication patterns also allow for easy scalability. For instance, additional subscribers in a publish/subscribe system can be added transparently without impacting the overall architecture of the distributed system.

The ATLAS experiment traditionally only used Ethernet networks. However, alternative interconnects from the HPC community like Infiniband or OmniPath exhibit promising performance characteristics and are an interesting alternative for a local area network for the ATLAS DAQ system. The use of Infiniband is currently being evaluated as a technology candidate for the planned LHC upgrade in 2019.

The rest of this paper is organized as follows: Section II gives an overview of different network software stacks for high performance interconnects with a focus on suitability for DAQ applications. In Section III the architecture of the custom NetIO library is discussed. Performance results are presented in Section IV. Section V lists related work. Section VI gives a conclusion.

II. Overview of Network Software Stacks for HPC

In this section different network technologies are compared based on two factors: (a) the throughput performance that can be achieved for different message sizes, and (b) the suitability of the API for applications in HEP, specifically data acquisition systems. The performance measurements (Figure 2) were performed on two servers connected via 56G Infiniband FDR. An overview of the different APIs discussed in this section is also given in Table I.

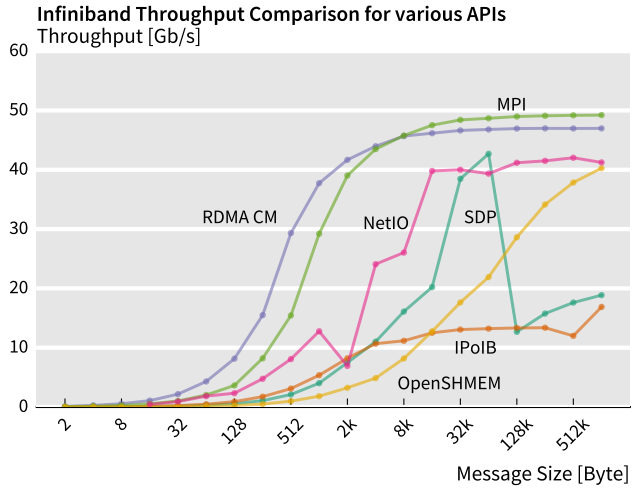

Fig. 2. Throughput for various network software stacks on 56G Infiniband FDR. The best throughput is achieved with RDMA CM (native) and MPI (message passing). OpenSHMEM, a PGAS implementation, shows poor performance for small message sizes. The measurements for the compatibility layers IPoIB and SDP were taken with the iperf3 benchmark. Performance is limited in both cases, however SDP yields much higher throughput. NetIO has a similar peak performance as SDP, but can achieve this throughput over a much bigger span of message sizes. Software releases used: RDMA CM, IPoIB and SDP from Mellanox OFED 3.4, OpenMPI 1.10, OpenSHMEM 1.2, NetIO 0.6.

A. Native APIs

Verbs or RDMA CM for Infiniband networks or PSM for OmniPath networks are native APIs for high-performance fabrics. These APIs allow users to retain full control over the underlying fabric. Their low-level nature, however, can make development cumbersome. Native APIs enable the highest possible performance, but this has to be achieved by manual tuning. Native APIs are often not portable among different fabrics. The low-level libfabric library [2] allows portability by providing a thin abstraction layer on top of native APIs.

Native APIs are of limited use to high-energy physics experiments. While the performance that can be achieved with native APIs is excellent, the development effort is high due to the lack of high-level communication patterns. The limited portability can lead to vendor lock-in, which is an issue for experiments like ATLAS with a lifespan of many decades.

B. MPI

MPI is one of the standard APIs in the HPC community. MPI follows a single-program-multiple-data (SPMD) paradigm [3], i.e., the same program is executed on multiple hosts and this distributed set of applications process a common dataset. MPI has only limited support for heterogeneous distributed applications with dynamic communication patterns like client/server. Instead it focuses on message passing in a cluster of homogeneous processes. It is therefore only of limited use for data acquisition systems that are composed of many individual communicating components rather than a homogeneous set of processes among which a fixed work set is

TABLE I
OVERVIEW OF HIGH-PERFORMANCE NETWORK APIS

Network API	Paradigm	Performance	Suitability
Librdma	RDMA, RC, RDM	Very good	High complexity
OpenMPI	MPI	Very good	Poor, HPC-oriented
OpenSHMEM	PGAS	Poor	Poor, HPC-oriented
IPoIB	POSIX Sockets	Poor	Average, but no high-level patterns
SDP	POSIX Sockets	Good for some message-sizes	Average, but no high-level patterns
ØMQ	Message Queue	Good for some message sizes, but no native fabric support (only SDP)	Very good
NetIO	Custom Message Queue	Good	Very good

divided. MPI also includes process management functionalities which are more suited for homogeneous clusters rather than distributed systems with complex architectures. Concerning performance many fabrics are tuned for MPI workloads, so in general a good performance can be expected.

C. Shared Address Space (PGAS)

Unlike all other presented APIs, shared address space interfaces are not based on message exchange between applications. Instead a global partitioned address space is exposed to the processes. Every process has local allocated memory regions that can be synchronized among the different nodes. PGAS implementations like OpenSHMEM [4] are well suited for SPMD applications in HPC because of the simple parallel access to distributed data. For HEP applications that rely on raw throughput the PGAS approach provides less performance than other APIs. It also more difficult to map the HEP use case to the global address space paradigm.

D. POSIX Sockets

POSIX [5] Sockets are a wide-spread API standard, mostly used for TCP communication over Ethernet. Compatibility layers are available for fabrics as well, for example IPoIB (IP-over-Infiniband, [6]) or SDP (Socket Direct Protocol, [7]). The POSIX socket API is versatile enough to implement dynamic communication patterns on top of it (see next paragraph on message queues). However, since the POSIX socket API is implemented as an abstraction layer on top of the respective native fabric API, performance might be limited compared to more low-level APIs. In Figure 2 this is the case for IPoIB, with which only a relatively low throughput can be achieved. SDP, a POSIX socket implementation based on native IB APIs promises better throughput. However, the peak throughput is only achieved for specific input parameter configurations. Compared to MPI and native APIs the performance is very sensitive to the message size being used.

E. Message Queues

Message queues like ØMQ [8] or RabbitMQ [9] have gained popularity as inter-process communication frameworks, also in the high-energy physics community [10], [11]. Message queues provide high-level communication patterns out of which complex distributed applications can be composed. Message queue APIs like ØMQ perfectly fulfill the ATLAS requirements. Unfortunately no message queue implementation supports fabrics other than Ethernet. For other fabrics a compatibility layer like IPoIB or SDP has to be used, which limits performance.

F. Custom: NetIO

Message queue APIs provide the most natural user interface for high-energy physics data acquisition applications. However, the lack of support for HPC fabrics limits their applicability. Our custom library NetIO is an implementation of a message queue that addresses this limitation by providing native support for HPC fabrics. NetIO uses libfabric as a back-end and hence supports a wide range of fabric technologies. Figure 2 shows that NetIO achieves around 75 % peak link utilization on a single connection for a large range of message sizes.

III. ARCHITECTURE OF NETIO

NetIO is designed as a generic message-based networking library that is tuned for typical use cases in DAQ systems. It supports four different communication patterns: low-latency point-to-point communication, high-throughput point-to-point communication, low-latency publish/subscribe communication, and high-throughput publish/subscribe communication.

NetIO has a back-end system to support different network technologies and APIs. Currently two different back-ends exist. The first back-end uses POSIX sockets to establish reliable connections between endpoints. Primarily this back-end is used on TCP/IP connections in Ethernet networks, but in principle a technology like SDP could also be used. The second back-end is based on libfabric [12]. It is used for communication via Infiniband and similar network technologies.

The NetIO architecture is illustrated in Figure 3. There are two software layers within NetIO. The upper level contains user-level sockets. These are the sockets that application code interacts with. The different socket types are listed in Table II.

The lower architecture level provides a common interface to the underlying network APIs. The common interface consists of three low-level socket types (a send socket, a listen socket and a receive socket), which are implemented by each back-end. The low-level sockets provide basic connection handling and simple transmission of messages between two endpoints. All higher level functionality like buffering, notification of user code via callbacks, or the publish/subscribe system are

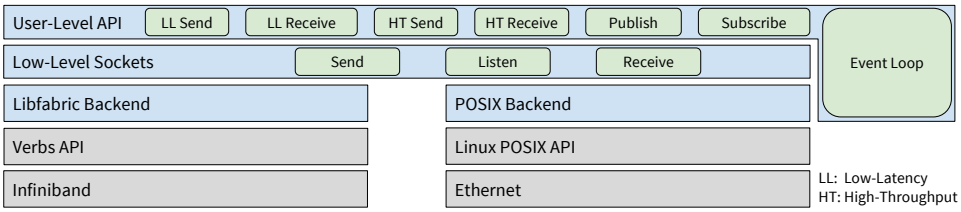

Fig. 3. The NetIO architecture.

TABLE II
THE DIFFERENT TYPES OF USER-LEVEL SOCKETS IN NETIO

Socket Type	Description
Low-Latency	
Send	A message that is posted is immediately sent to the remote endpoint without any delay.
Receive	A message that is received is immediately passed to the user code via a callback.
High-Throughput	
Send	Messages are copied into a large connection buffer instead of being sent immediately. The buffer is transmitted when it is full or after a timeout occurs.
Receive	When a message buffer is received the contained messages are copied into separate message datastructures and enqueued in a message queue. User code can read the received messages by calling `recv()` on the receive socket.
Publish/Subscribe	
Publish	When a message is published under a given tag, the tag is matched against a subscription table and the message is sent to all subscribed remote endpoints via either low-latency or high-throughput send sockets.
Subscribe	Sends subscription requests to publish sockets via low-latency send sockets and then receives messages via either a low-latency or a high-throughput receive socket.

implemented in the user-level sockets. This maximizes code sharing among the back-ends, as only code that is specific to the underlying network technology is implemented in the low-level sockets.

Both architecture levels use a central event loop to handle I/O events like connection requests, transmission completions, error conditions, or timeouts. The event loop is executed in a separate thread. Its implementation is based on the epoll framework [13] in the Linux kernel.

Network endpoints are addressed by IP address and port, even for back-ends that do not natively support this form of addressing. For the Infiniband back-end the librdma compatibility layer is used to enable addressing by IP and port.

A. User-level sockets

There are six different user-level sockets, of which four are point-to-point sockets (one send socket and one receive socket, each in a high-throughput and a low-latency version), and two publish/subscribe sockets (one publish and one subscribe socket). The publish/subscribe sockets internally use the point-to-point sockets for data communication. Subscribe sockets can subscribe to multiple data tags from multiple publish sockets. Each subscription can either be created in high-throughput or low-latency mode.

A high-throughput send socket does not send out messages immediately but maintains buffers in which messages are copied. The buffers are referred to as pages. Due to the buffering fewer, larger packets are sent on the network link. This approach is more efficient and yields a higher throughput. However, the average transmission latency of any specific message is increased due to the buffering. A typical page size is 1 MB. Once a page is filled the whole page is sent to the receiving end. Additionally, a timer (driven by the central event loop) flushes the page at regular intervals to avoid starvation and infinite latencies on connections with a low message rate. A typical timeout interval is 2 s. A message is split if it does not fit into a single page. The original message is reconstructed on the receiving side.

A high-throughput receive socket receives pages that contain one or more messages or partial messages. The messages are encoded by simply prepending an 8 byte length field to the message contents. The high-throughput receive socket maintains two queues: a page queue, which contains unprocessed pages that have been received from a remote, and a message queue, in which messages are stored that are extracted from pages when they are processed. The high-throughput receive socket enqueues received pages in the receive page queue. When user code calls `recv()` on a high-throughput receive socket, it will return the next message from the message queue. If the message queue is empty, the next page from the receive page queue is processed and the contained messages are stored in the message queue. When processing a received page the contained messages are copied. A new zero-copy mode is currently under development in which it is also possible to

avoid this additional copy.

A low-latency send socket does not buffer messages. Messages are immediately sent to the remote process. Unlike for high-throughput send sockets, there is also no additional copy: the message buffer is directly passed to the underlying low-level socket. These design decisions minimize the added latency of a message send operation.

A low-latency receive socket handles incoming messages by passing them to the application code via a user-provided callback routine, instead of enqueuing the messages in a message queue. This approach enables incoming messages to be processed immediately. The message buffer is only valid during execution of the callback. After execution of the callback routine the receive buffer will be freed. If necessary, a user can decide to copy the buffer in the callback routine.

High-throughput and low-latency receive sockets also differ in the way threading is involved in processing incoming messages. In both cases a page receipt notification from a low-level receive socket is handled in the event loop thread. In high-throughput receive sockets the page is immediately pushed into the receive page queue, after which the event handler returns and the event loop thread is free to process further events. Parsing the page, extracting the messages and processing them with user code is done in the user thread. In low-latency sockets the event handler routine executed by the event loop thread will call the user-provided callback. Thus, all user code is executed by the event loop thread. The event handler will only return after the user callback is processed. This might block the event loop from processing further events for any amount of time. Users have to take care to implement sensible callback routines that do not block the event loop too long, or otherwise performance might degrade. The benefit of executing user code in the event loop thread is, however, that no latency is added by queuing of messages. Message processing in high-throughput and low-latency sockets is illustrated in Figures 4 and 5.

Publish and subscribe sockets internally use point-to-point user sockets for communication. On top of that, publish sockets maintain a dynamic subscription table to manage subscriptions from subscribe sockets.

B. Low-Level Sockets

The interface to the NetIO back-ends is provided to the user-level sockets by three types of low-level sockets: back-end send sockets, back-end listen sockets, and back-end receive sockets. Back-end listen and receive sockets are used on the receiving side of a connection. Back-end listen sockets open a port and listen for incoming connections by back-end send sockets. A back-end receive socket is created when a connection request arrives at a back-end listen socket. A back-end receive socket represents a single connection. A back-end send socket is used on the sending side of a connection. A back-end send socket can connect to a port opened by a back-end listen socket and send messages when the connection is established.

The back-end sockets provide callback entry points for user-level sockets that are called when a connection has been successfully established, a remote has disconnected, or data has arrived.

The low-level API also provides an interface for back-end buffers. These are buffers that are used by the back-end sockets and can be transmitted over the network. Back-ends might have special requirements on buffers. Libfabric for example requires that all buffers are previously registered in a central registry.

C. The POSIX Back-end

The POSIX back-end is straight-forward and uses the socket API that is defined in the POSIX standard [5]. The back-end uses sockets of the SOCK_STREAM type, i.e. TCP/IP connections. The socket option TCP_NODELAY is set, which disables Nagle's algorithm [14]. Nagle's algorithm can temporarily delay packet sends to reduce the number of TCP packets on the wire. The buffering capabilities of the user-level sockets however allow a more fine-grained control over packet delay, so Nagle's algorithm can be deactivated.

The POSIX socket API uses file descriptors to represent the sockets. These file descriptors are registered in the central event loop. Thus, when a connection request or a new message arrives, the corresponding sockets are informed and handler routines are executed. The POSIX sockets are configured to asynchronous, non-blocking mode, i.e., the O_NONBLOCK flag is set.

D. The FI/Verbs Back-end

Libfabric provides several communication modes to the user, for example reliable datagram (RDM) communication, reliable connection (RC) communication (which works like RDM but additionally provides message ordering), or RDMA. Libfabric be can used on top of several network stacks. For Infiniband, the library utilizes librdma and libibverbs. For Intel OmniPath, the native PSM2 interface can be used.

The NetIO FI/Verbs back-end uses the reliable connection communication model from libfabric. Libfabric also provides reliable datagram communication, but the reliable connection model preserves message order. Preservation of message order is important since a message can span multiple NetIO buffers (see Section III-A on high-throughput sockets).

Libfabric provides so-called active and passive endpoints to manage connections. Passive endpoints listen to incoming connections, while active endpoints are the equivalents of sockets and are used to send and receive messages. The libfabric API is fully asynchronous, and connection management notifications are presented to the user as events that need to be handled. Each endpoint has an event queue in which connection events are stored. Libfabric allows a file descriptor to be registered with an event queue. When a new event arrives, the file descriptor becomes readable. The NetIO FI/Verbs back-end uses these event queue file descriptors and registers them in the central event loop.

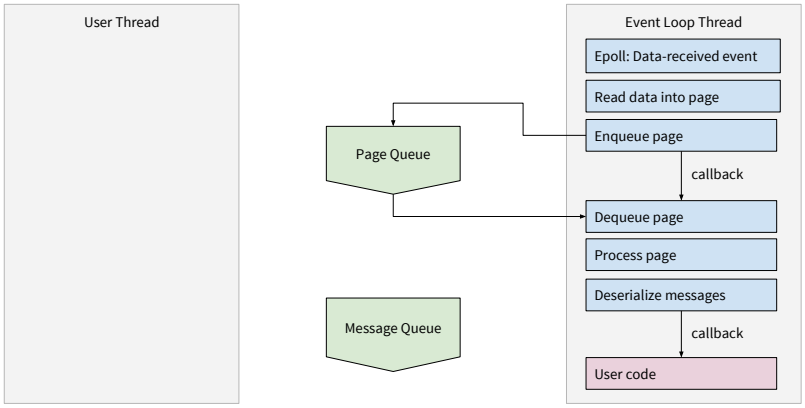

Fig. 4. Processing of incoming messages in NetIO low-latency sockets. Note that all processing, including the execution of user code, is performed in the event loop thread.

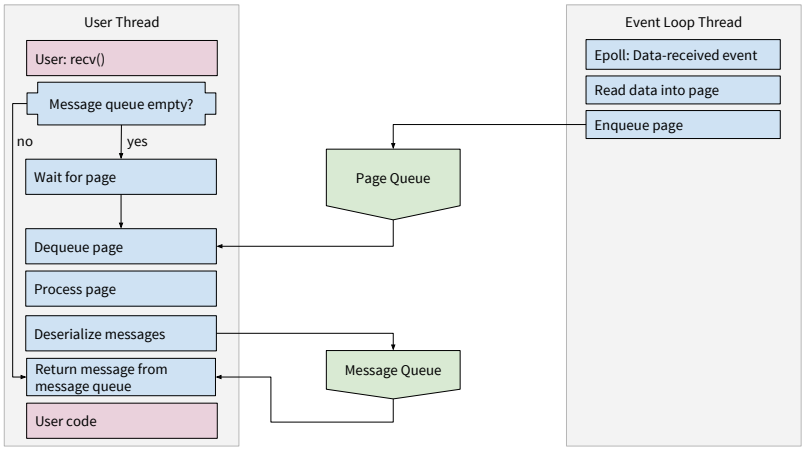

Fig. 5. Processing of incoming messages in NetIO high-throughput sockets. After the data is received in the event loop thread, all processing is done in the user level thread. The event loop thread is freed up to process further incoming data.

POSIX sockets have internal buffers. When a message is sent on a POSIX socket, the data are copied into the internal buffer, from which the data are then sent to the remote process. The user-supplied buffer is usable again immediately after the send call. Similarly, a receiving POSIX socket receives data in an internal buffer, and a receive call will copy the data out of the internal buffer. The user does not need to supply a buffer in which data from the network can be received.

The FI/Verbs back-end is asynchronous and makes it possible to send and receive messages without data copies. Libfabric endpoints do not have internal buffers. When a message is sent, the user-supplied message buffer is used and no data are copied. A user needs to provide receive buffers to receive messages. Each active endpoint has a queue for completion events to manage the send and receive buffers. Completions notify the user-space application of the result of the send or receive operation. After a send completion arrives, the corresponding send buffer can be reused for new send operations. After a receive completion arrives, the corresponding receive buffer is filled with a message from a remote host and can be processed. Like the connection management events the completion events can trigger a file descriptor. NetIO uses such completion file descriptors and registers them in the central event loop.

Libfabric requires send and receive buffers to be registered with the call `fi_mr_req`. The FI/Verbs back-end provides a data buffer interface that performs this registration step.

E. The Intel OmniPath Back-end

Intel OmniPath [15] is a recent fabric technology that is based on the TrueScale technology [16] formerly by the QLogic company. On the software side, OmniPath has a Verbs interface and is thus directly supported by NetIO via libfabric. OmniPath additionally provides a native API called PSM, which is also supported by libfabric. However, the libfabric PSM provider does not currently support the reliable connection mode, which is needed for NetIO. NetIO on OmniPath therefore currently only works using the Verbs interface and is considered experimental.

TABLE III
SYSTEMS USED FOR NETIO BENCHMARKS

	System 1	System 2
CPU Type	Intel Xeon E5-2630 v3	Intel Xeon E5-2660 v3
CPU Clock Speed	2.40 GHz	2.60 GHz
Nr of cores		
real	8 per CPU	10 per CPU
threads	16 per CPU	20 per CPU
Nr of CPUs	2	2
Memory	64 GB	64 GB

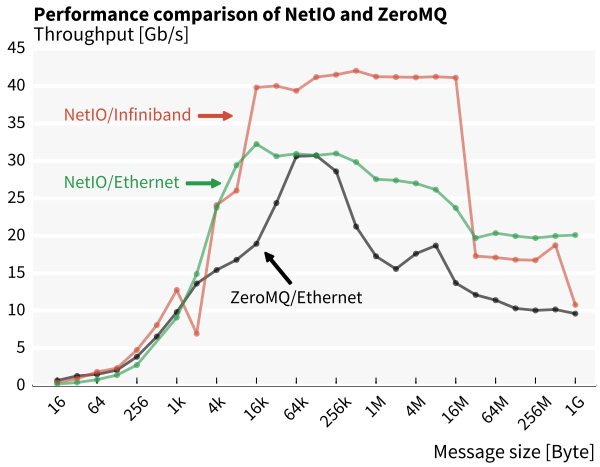

Fig. 6. Throughput measured with NetIO and ØMQ on 40G Ethernet and NetIO on 56G Infiniband FDR for various message sizes. Note that only a single connection is used between the two systems, hence the link is not fully utilized.

IV. BENCHMARKS

We performed several experiments to evaluate the performance of NetIO. As a reference point we use the ØMQ [17] library to compare NetIO against. ØMQ is a library that gained popularity in the HEP community and is used in several projects in the LHC experiments. ØMQ provides point-to-point communication as well as a publish/subscribe system. Benchmarks are performed between two nodes connected via a single switch. The benchmark system configuration is described in Table III. The systems are equipped with Mellanox ConnectX-3 VPI network interface cards, which can be operated in 40G Ethernet mode or 56G Infiniband FDR mode.

The first benchmark scenario consists of point-to-point communication between the two systems using NetIO high-throughput sockets and a single connection. The sending side uses the NetIO test tool `netio_throughput` to send messages to the receiving node, which uses the program `netio_recv` to receive the messages. The throughput achieved for various message sizes is shown in Figure 6.

NetIO on Ethernet and ØMQ on Ethernet have a very similar peak performance of around 30 Gb/s. NetIO, however, reaches higher throughput values for small and large message sizes. For message sizes less than 64 kB an up to two-fold better

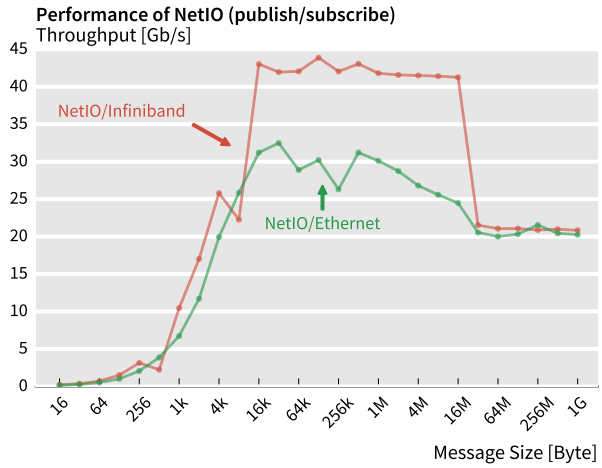

Fig. 7. Throughput performance of NetIO publish/subscribe sockets on 40G Ethernet and 56G Infiniband FDR. The peak performance of NetIO with the Infiniband back-end is more than 30% faster than NetIO with the Ethernet back-end.

throughput is measured with NetIO than with ØMQ. NetIO on Infiniband outperforms both NetIO and ØMQ on Ethernet for message sizes smaller than 32 MB. For larger message sizes the CPU cache limits performance. The peak performance is around 40 Gb/s.

An experiment with NetIO high-throughput publish/subscribe sockets is shown in Figure 7. Similar to the previous benchmark NetIO on Infiniband outperforms NetIO on Ethernet. The achieved peak performance in each case is comparable to the point-to-point benchmarks.

A third benchmark analyzes the performance of NetIO low-latency sockets. We measure the round-trip time (RTT) between two systems in Table III. In both cases, Ethernet and Infiniband, there is one switch in the middle. The results of the measurements are shown in Figure 8.

NetIO on Ethernet, NetIO on Infiniband, and ØMQ show all very similar RTT values. The average RTT is in the case of Ethernet around $40\,\mu s$, for Infiniband it is just slightly higher. The difference for Infiniband indicates that the NetIO Infiniband back-end is slightly less efficient and still needs to be optimized.

V. RELATED WORK

ØMQ has limited support for Infiniband via SDP, the Sockets Direct Protocol. SDP is an implementation of a TCP-like stream protocol on top of RDMA network hardware. SDP can show performance improvements over IPoIB [18].

One has to resort to native APIs like Verbs to utilize the full capability of HPC fabrics like Infiniband. The RDMA Connection Manager (RDMA CM) from the OpenFabrics Alliance simplifies the addressing part of RDMA transfers by using IP (using IPoIB) for addressing and connection management. Data transfers are done using the native Verbs calls. This approach is described in [19]. The advantage of this approach is full flexibility and the availability of all features of

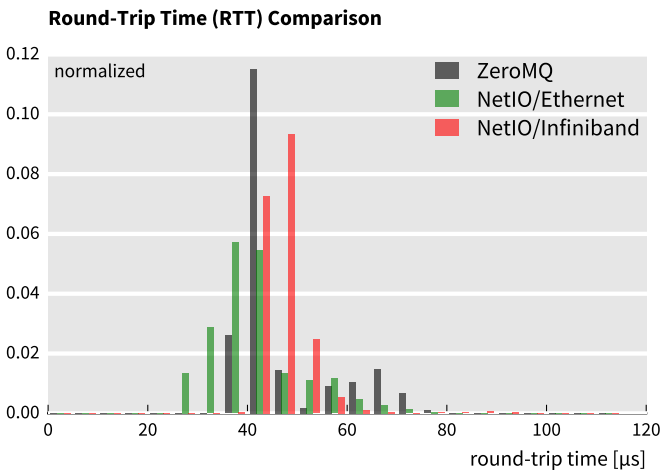

Fig. 8. Round-trip time comparison between NetIO on Ethernet, NetIO on Infiniband, and ØMQ. All three implementations have a similar average value. NetIO on Infiniband has a slightly higher round-trip time than NetIO on Ethernet which can likely be explained by the different NetIO back-end implementation.

Infiniband like RDMA. However, even with the RDMA CM library setting up and maintaining connections is still complex compared to high-level libraries. This approach can also not natively run on Ethernet or other networks that do not support Verbs and librdma.

VI. Conclusions

In this paper we highlight the differences between data-acquisition system networks for high energy physics experiments like ATLAS and networks in HPC. We came to the conclusion that the typical requirements on the network software stack are fundamentally different in the two domains. Nevertheless, HPC interconnect technologies have promising performance features, both in terms of throughput and latency.

We could show that in order to leverage HPC interconnects like Infiniband or OmniPath, the HEP community is mostly missing appropriate software APIs that match the requirements and characteristics of typical DAQ applications. This involves a significant paradigm shift from a simple problem division approach as found in MPI, PGAS or other parallel computing frameworks, to high-level communication patterns like client/server or publish/subscribe.

Experiments with an Infiniband FDR setup showed that existing socket-based APIs that provide a DAQ-compatible interface have low performance compared to native Infiniband benchmarks when using compatibility layers like IPoIB or SDP. With NetIO we presented a library that implements high-level communication patterns with a libfabric-based back-end. NetIO was built to bridge the gap between HPC interconnect hardware and HEP requirements.

The throughput performance of NetIO is on a par with the peak performance achieved in benchmarks with SDP. However, NetIO maintains the high throughput over a large span of parameters, while the SDP benchmark only achieves high performance in a few scenarios.

There is still room for improvement for NetIO. MPI and RDMA benchmarks exhibit an up to 25 % higher throughput. Future work will focus on closing this gap. One approach is the reduction of memory copies in the NetIO data flow. Another area of work is the support for more network technologies such as Intel OmniPath. A NetIO OpenSource release is planned for later this year.

While the work described here mostly concerns applications in high-energy physics, we can envisage that the results could be applied to other data center applications, for example applications in the Big Data domain.

References

[1] The ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," *Journal of Instrumentation*, vol. 3, no. 08, 2008.

[2] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug 2015, pp. 34–39.

[3] F. Darema, "The spmd model: Past, present and future," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2001.

[4] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010, p. 2.

[5] *System Application Program Interface (API)*, ser. Information technology—Portable Operating System Interface (POSIX), 1990.

[6] J. Chu and V. Kashyap, "Transmission of IP over InfiniBand (IPoIB)," RFC 4391 (Proposed Standard), Internet Engineering Task Force, Apr. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4391.txt

[7] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin, "Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis," in *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*. IEEE, 2005, pp. 128–137.

[8] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[9] A. Richardson *et al.*, "Introduction to RabbitMQ," *Google UK, available at http://www. rabbitmq. com/resources/google-tech-talk-final/alexis-google-rabbitmq-talk. pdf, retrieved on Mar*, vol. 30, p. 33, 2012.

[10] A. Dworak, M. Sobczak, F. Ehm, W. Sliwinski, and P. Charrue, "Middleware trends and market leaders 2011," in *13th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2011.

[11] A. Dworak, F. Ehm, P. Charrue, and W. Sliwinski, "The new CERN Controls Middleware," *Journal of Physics: Conference Series*, vol. 396, no. 1, p. 012017, 2012.

[12] OpenFabrics Working Group. Libfabric. [Online]. Available: https://ofiwg.github.io/libfabric/

[13] Linux User's Manual, *epoll(7)*, April 2012.

[14] J. Nagle, "Congestion Control in IP/TCP Internetworks," RFC 896, Internet Engineering Task Force, Jan. 1984. [Online]. Available: http://www.ietf.org/rfc/rfc896.txt

[15] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel Omni-path Architecture: Enabling Scalable, High Performance Fabrics," *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 1–9, 2015.

[16] Intel, "Intel True Scale Fabric Architecture: Enhanced HPC Architecture and Performance," vol. 1, November 2012.

[17] P. Hintjens, M. Sústrik, and Others. ZeroMQ. [Online]. Available: http://zeromq.org/

[18] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, "Sockets Direct Protocol over InfiniBand in clusters: is it beneficial?" in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, 2004, pp. 28–35.

[19] T. Bedeir, "Building an RDMA-capable application with IB Verbs," *Technical report, HPC Advisory Council*, 2010. [Online]. Available: http://hpcadvisorycouncil.com/