Journal of Physics: Conference Series

PAPER • **OPEN ACCESS**

High-Throughput and Low-Latency Network Communication with NetIO

To cite this article: Jörn Schumacher *et al* 2017 *J. Phys.: Conf. Ser.* **898** 082003

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 082003 doi:10.1088/1742-6596/898/8/082003

High-Throughput and Low-Latency Network Communication with NetIO

Jörn Schumacher[1,2], Christian Plessl[2] and Wainer Vandelli[1]

[1] CERN, Geneva, Switzerland

[2] Paderborn University, Germany

E-mail: jorn.schumacher@cern.ch

Abstract. HPC network technologies like Infiniband, TrueScale or OmniPath provide low-latency and high-throughput communication between hosts, which makes them attractive options for data-acquisition systems in large-scale high-energy physics experiments. Like HPC networks, DAQ networks are local and include a well specified number of systems. Unfortunately traditional network communication APIs for HPC clusters like MPI or PGAS exclusively target the HPC community and are not suited well for DAQ applications. It is possible to build distributed DAQ applications using low-level system APIs like Infiniband Verbs, but it requires a non-negligible effort and expert knowledge.

At the same time, message services like ZeroMQ have gained popularity in the HEP community. They make it possible to build distributed applications with a high-level approach and provide good performance. Unfortunately, their usage usually limits developers to TCP/IP-based networks. While it is possible to operate a TCP/IP stack on top of Infiniband and OmniPath, this approach may not be very efficient compared to a direct use of native APIs.

NetIO is a simple, novel asynchronous message service that can operate on Ethernet, Infiniband and similar network fabrics. In this paper the design and implementation of NetIO is presented and described, and its use is evaluated in comparison to other approaches. NetIO supports different high-level programming models and typical workloads of HEP applications. The ATLAS FELIX project [1] successfully uses NetIO as its central communication platform.

The architecture of NetIO is described in this paper, including the user-level API and the internal data-flow design. The paper includes a performance evaluation of NetIO including throughput and latency measurements. The performance is compared against the state-of-the-art ZeroMQ message service. Performance measurements are performed in a lab environment with Ethernet and FDR Infiniband networks.

1. Introduction

High-performance networking is crucial for large data-acquisition (DAQ) systems. Such systems are composed of thousands of individual components that communicate with each other. Performance requirements for DAQ can vary for different use cases within a given system: some subsystems might require high-throughput and efficient link utilization, while others require low latencies to reduce communication delays as much as possible. Networking technologies including high performance fabrics, network topologies, software stacks and APIs are well researched topics in fields like HPC. Network infrastructures in online DAQ systems for high-energy physics (HEP) experiments, however, have fundamentally different requirements and require different methodologies and paradigms. The typical HPC use case for high-performance fabrics is large-scale computing with a single-program-multiple-data (SPMD) approach. The communication

layer is often implemented with software layers like MPI [2], PGAS [3], or similar message passing or distributed shared memory schemes.

Networking aspects such as hardware (eg. fabrics and switches), network topologies, and low-level protocols are similar in DAQ systems and HPC installations.

DAQ systems, however, are distributed systems with many different applications and thus do not match the SPMD paradigm well. Different networking software stacks compared to HPC are required for DAQ systems.

DAQ networks are subject to different requirements compared to HPC. A DAQ system, for example, has to be maintainable for decades due to the longevity of HEP experiments, which has an impact on the choice of hardware technology. Furthermore, DAQ system network infrastructures have to span relatively long distances of several hundred metres. For example, the ATLAS read-out system is located underground, in a service cavern next to experiment. The high level trigger farm is instead housed in a surface data-center. To connect systems in the two locations, distances of up to 150 m have to be bridged.

In this paper we present NetIO, a message-based communication library that provides high-level communication patterns for typical DAQ use cases. NetIO differentiates itself from other message services in two key ways:

(i) NetIO distinguishes between low-latency communication and high-throughput communication. Both are very common in DAQ systems. NetIO offers different communication sockets which are optimized for one or the other workload scenario.

(ii) The transport layer in NetIO can be provided by different back-ends, so that NetIO can communicate natively on Ethernet, Infiniband, and other network technologies. This allows users of NetIO to leverage technologies from the HPC domain for DAQ systems, which have been traditionally dominated by Ethernet.

This paper will describe the architecture and design of NetIO, including how the library uses the network stack and tunes for the different workload scenarios (low-latency and high-throughput). The implementation of the transport back-end architecture will also be presented. Finally a performance evaluation of NetIO will be presented, with demonstrations in the two workload scenarios.

2. Related Work

NetIO has been developed in the context of the upgrade of the ATLAS experiment at the LHC. The development teams at the other LHC experiments (ALICE, CMS and LHCb) are preparing upgrades for their respective DAQ systems as well. An important topic is the choice of DAQ network technologies. An overview of different 100 Gbps interconnect technologies with a focus on future DAQ applications is given in [4], which compares 100 Gigabit Ethernet, Intel OmniPath and EDR Infiniband. EDR Infiniband is a newer standard than FDR which is used in this paper. FDR supports signaling rates of 56 Gb/s on four lanes, EDR supports 100 Gb/s.

LHCb are investigating the potential use of Infiniband as network technology for their event builder network [5; 6]. On the network side, ALICE has used a mixture of Ethernet and Infiniband in the past, and is investigating future network technologies [7]. The CMS experiment currently uses a mixture of Ethernet and Infiniband networks for their DAQ system [8; 9].

ZeroMQ [10] is a message queue implementation that has gained increasing popularity in high energy physics applications. The CERN Middleware Project is building a common middleware framework based on ZeroMQ [11; 12] that replaces old CORBA-based middleware. ALICE is also considering the use of ZeroMQ as a networking software framework.

A project in the ALICE experiment is evaluating the use of nanomsg [13], a system similar to ZeroMQ, and developing an OFI (OpenFabrics Interface) transport for nanomsg that allows it to be run on Infiniband, OmniPath and other high performance interconnects [14].

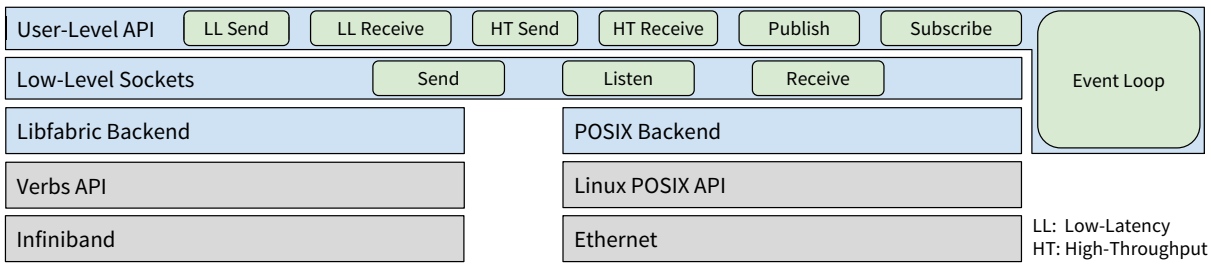


Figure 1. The NetIO architecture.

3. Architecture of the NetIO Message Service

NetIO is implemented 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.

A modular back-end system enables NetIO to support different network technologies and APIs. At the time of writing two different back-ends exist. The first back-end uses POSIX sockets to establish reliable connections to endpoints. Typically this back-end is used for TCP/IP connections in Ethernet networks. The second back-end uses libfabric [15] for communication, providing support for Infiniband and similar network technologies. Libfabric is a network API that is provided by the OpenFabrics Working Group.

The NetIO architecture is illustrated in Figure 1. There are two software layers within NetIO. The upper layer contains user-level sockets. Application-level code interacts with this layer. The lower architecture level provides a common interface to the underlying network API. Both socket layers 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 [16] in the Linux kernel.

IP address and port are used for addressing network endpoints, even for back-ends that do not natively support this form of addressing. For the Infiniband back-end the librdmacm [17] compatibility layer is used to enable addressing by IPv4 or IPv6 address and port.

3.1. User-level sockets

There are six different user-level sockets, of which four are point-to-point sockets, one publish and one subscribe socket. The point-to-point sockets consist of one send socket and one receive socket, each in a high-throughput and a low-latency version. The publish/subscribe sockets internally use the point-to-point sockets for data communication, either in the high-throughput or in the low-latency fashion.

A high-throughput send socket does not send out messages immediately. It maintains an internal buffer where messages are copied to. As a consequence, the network interface receives large packets at a reduced rate. This approach is more efficient and yields a higher throughput, but at the cost of increasing the average transmission latency of any specific message. Once a buffer is filled the whole buffer is sent out to the receiving end. Additionally, a timer (driven by the central event loop) flushes the buffer at regular intervals to avoid starvation and infinite latencies on connections with a low message rate. A message is split if it does not fit into a single buffer. The original message is reconstructed on the receiving side.

A high-throughput receive socket receives buffers (here referred to as *pages*) that contain one or more messages or partial messages. The messages are encoded by simply prepending an 8 byte length field to the messages. The high-throughput receive socket maintains two queues: a page queue and a message queue. The page queue stores unprocessed pages that have been

received from a remote process. When a page is removed from the page queue and processed, the messages that are contained in the page are placed in the message queue. The high-throughput receive socket places received pages in the page queue. When user code calls `recv()` on a high-throughput receive socket, it will return the next message from the message queue. The `recv()` call is blocking. If the message queue is empty, the next page from the page queue is processed and the contained messages are stored in the message queue. When processing a page the contained messages are copied. The additional copy could be optimized in the future with a more sophisticated buffer management system.

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.

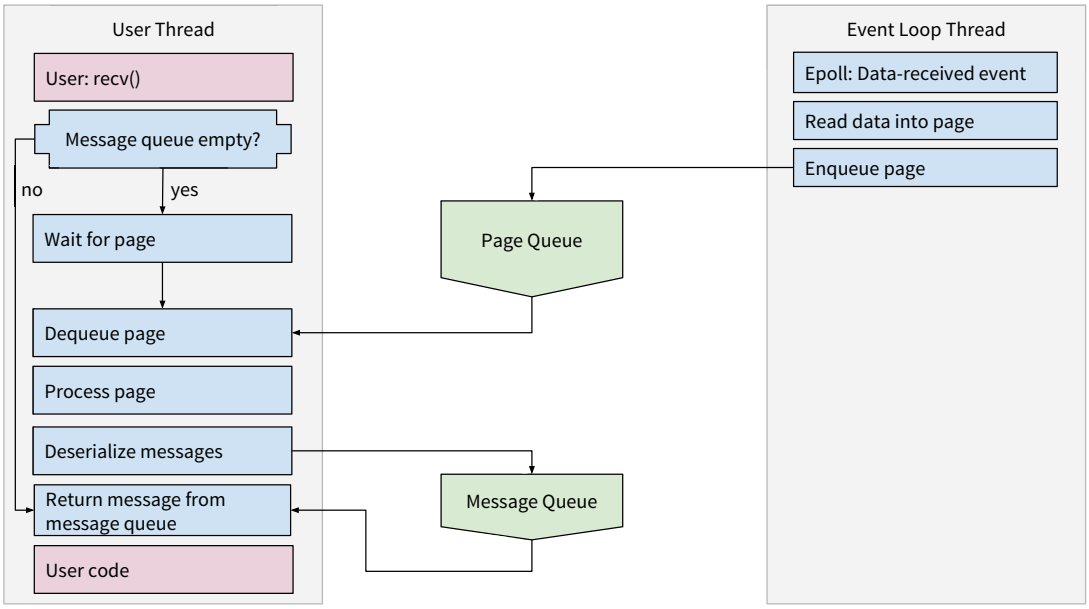
A low-latency receive socket handles incoming messages by passing them to the application code via a user-provided callback routine, instead of placing the messages in a message queue. This approach allows incoming messages to be processed immediately. A page queue is still used for compatibility reasons on the source code level, but this might be optimized in the future. In contrast to high-throughput receive sockets no data copy is taking place, the receive buffer memory is passed to the user level code. After execution of the callback routine the receive buffer will be freed and accessing the memory by user-level code is an illegal operation. If necessary, a user can decide to copy the buffer in the 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 buffer receive notification from a low-level receive socket is handled in the event loop thread. In high-throughput receive sockets the buffer is immediately pushed into the receive buffer queue, after which the event handler returns and the event loop thread is free to process further events. Parsing the buffer, extracting the messages and processing them with user code is done in the user thread (Figure 2a). 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 (Figure 2b). 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 that no latency is added by queuing of messages.

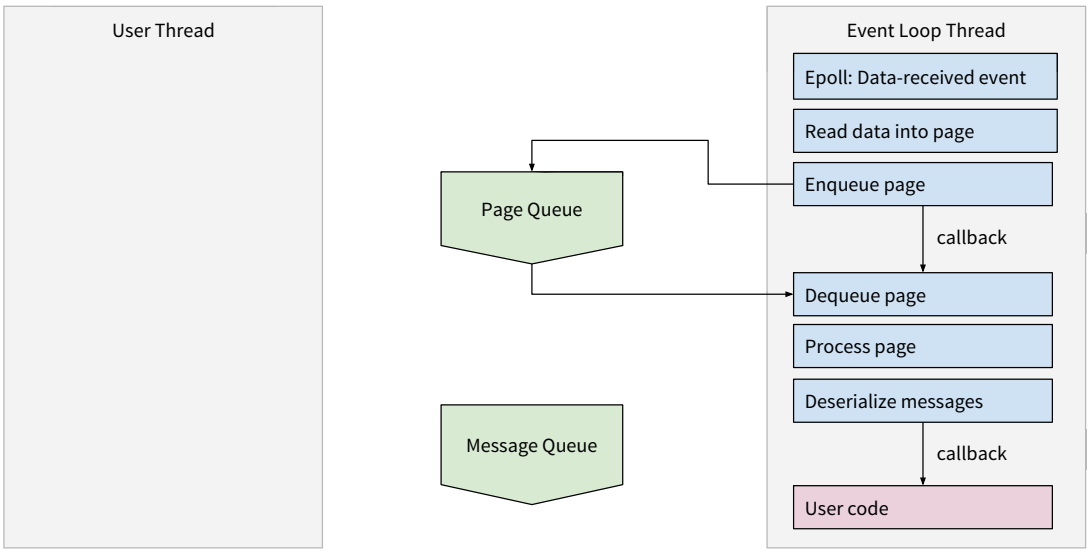
Publish/subscribe sockets use point-to-point sockets internally. The publish socket contains a low-latency receive socket on which subscription requests are received. Once a subscription request arrives, a new send socket is created and connected to the remote subscribe socket. The subscriber can either request a high-throughput connection or a low-latency connection in the subscription request, and the send socket is created of the requested type. Subscribers can subscribe to a specific type of data by specifying a *subscription tag*, which is a 64 bit integer. Subscriptions are registered in a hashmap that is maintained in the publish socket. The hashmap maps subscription tags to send sockets. When a message is published on the publish socket, this hashmap is used to look up the send sockets on which the message has to be sent. Subscribe sockets contain a low-latency send socket to send subscription requests, as well as a low-latency receive socket and a high-throughput receive socket to receive messages for both types of subscription.

3.2. 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



(a) High-throughput sockets.



(b) Low-latency sockets.

Figure 2. Processing of messages in NetIO high-throughput and low-latency sockets. Threads have different responsibilities in the two cases.

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 socket has disconnected, or data have arrived. The low-level API also provides an interface for back-end buffers. These are back-end-

specific memory areas used to transmit the data to the network stack. Different back-ends have different buffering requirements (Section 3.4).

3.3. The POSIX Back-End

The POSIX back-end is based on the API defined by the POSIX standard [18]. 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 [19]. 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 allow a more fine-grained control over packet delay. Nagle's algorithm can be deactivated since buffering is already implemented at a higher level.

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 is set.

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.

3.4. The FI/Verbs Back-End

Libfabric provides several communication modes to the user, for example reliable datagram (RDM) communication, reliable connection 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 [20], for Intel OmniPath the native PSM2 [21] interface can be used.

In contrast to POSIX sockets, FI/Verbs sockets are buffer-less. The back-end therefore makes it possible to send and receive messages without data copies. When a message is sent, the user-supplied message buffer is used and no data are copied. To receive messages, a user needs to provide receive buffers. The libfabric API is asynchronous. Thus, a user needs to actively manage the access to the buffers. For this purpose, each active endpoint has a queue for completion events. Completions notify the user-space application of the result of the send or receive operations. 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. Completion events can trigger a file descriptor in the same way as connection management events. NetIO uses such completion file descriptors and registers them in the central event loop.

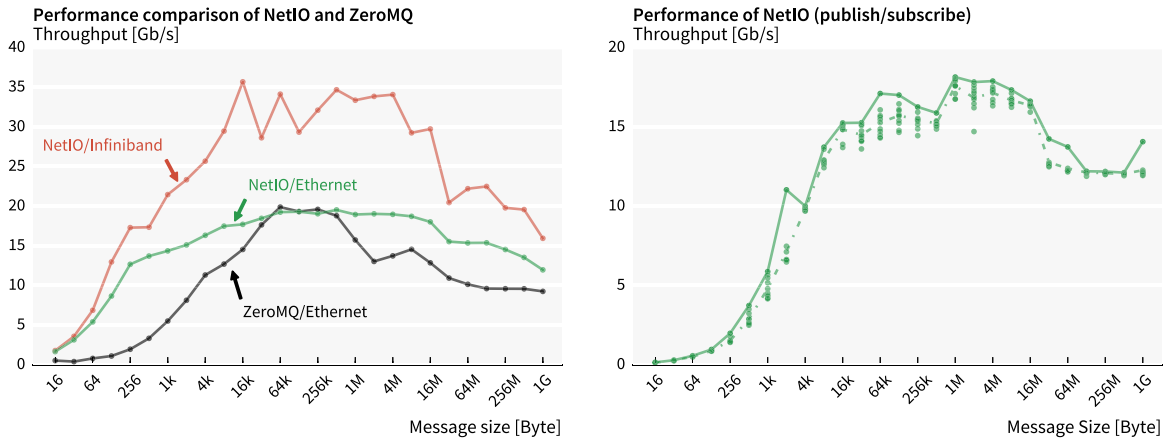
Libfabric requires send and receive buffers to be registered with the `fi_mr_req` call. The NetIO FI/Verbs back-end provides a data buffer interface that performs this registration step.

4. Benchmarks and Tests with NetIO

To evaluate the performance of NetIO several experiments have been performed. As a reference point the ZeroMQ library is used for comparison with NetIO. ZeroMQ is a library that gained popularity in the HEP community and is used in several projects in the LHC and the LHC experiments. ZeroMQ 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 1. The systems are equipped with Mellanox ConnectX-3 VPI network interface cards, which can be operated in either 40 Gigabit Ethernet mode or 56 Gigabit Infiniband FDR mode.

	System 1	System 2
CPU Type	Intel Xeon E5-2630 v3 [22]	Intel Xeon E5-2660 v3 [23]
CPU Clock Speed	2.40 GHz	2.60 GHz
Nr of cores per CPU	8 (real) / 16 (logical)	10 (real) / 20 (logical)
Nr of CPUs	2	2
Memory	64 GB	64 GB

Table 1. Systems used for NetIO benchmarks.



(a) Point-to-Point (b) Publish/Subscribe

Figure 3. Throughput performance of NetIO sockets.

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 program `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 3a.

NetIO on Ethernet and ZeroMQ on Ethernet have a very similar peak performance of around 22 Gb/s. NetIO reaches higher throughput values for small and large message sizes. NetIO reaches an up to five-fold better throughput compared to ZeroMQ for messages sizes less than 1 kB. NetIO on Infiniband outperforms both NetIO and ZeroMQ on Ethernet. The peak performance is around 36 Gb/s. An experiment with NetIO high-throughput publish/subscribe sockets is shown in Figure 3b.

A third benchmark analyzes the performance of NetIO low-latency sockets. The round-trip time (RTT) is measured between two systems in Table 1. The hardware setup is the same as in the previous measurements. NetIO on Ethernet, NetIO on Infiniband, and ZeroMQ all show very similar RTT values. The average RTT is in each case around 40μs.

5. Conclusion

In this paper the network communication library NetIO was introduced. The NetIO library was designed with the goal to make HPC network technologies like Infiniband accessible for DAQ systems in HEP experiments. Two use cases were identified: high-throughput communication and low-latency communication. In experiments with 40G Ethernet it was shown that NetIO

high-throughput sockets outperform the state-of-the-art message queue implementation ZeroMQ in many use cases. Additionally it was shown that higher throughput can be achieved when switching to Infiniband FDR (56 Gb/s). Regarding low-latency communication it was demonstrated that NetIO low-latency sockets are on par with the ZeroMQ library.

In the future further evaluation of NetIO will be performed. The setup that was used in the measurements is a small lab setup. It will be desirable to explore the performance of NetIO in more realistic DAQ contexts. In some areas it is possible to further improve NetIO, e.g., by avoiding queuing of pages in low-latency sockets. Another area of development is the addition of more back-ends. An Intel OmniPath back-end is in development, further back-ends can be imagined.

References

[1] Jorn Schumacher et al. FELIX: a High-Throughput Network Approach for Interfacing to Front End Electronics for ATLAS Upgrades. *Journal of Physics: Conference Series*, 664(8):082050, 2015.

[2] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[3] Tim Stitt. *An introduction to the Partitioned Global Address Space (PGAS) programming model*. Connexions, Rice University, 2009.

[4] Adam Otto et al. A first look at 100 Gbps LAN technologies, with an emphasis on future DAQ applications. *Journal of Physics: Conference Series*, 664(5):052030, 2015. doi: 10.1088/1742-6596/664/5/052030.

[5] Enrico Bonaccorsi et al. Infiniband event-builder architecture test-beds for full rate data acquisition in lhcb. *Journal of Physics: Conference Series*, 331(2):022008, 2011.

[6] D.H. Campora Perez et al. The 40MHz trigger-less DAQ for the LHCb Upgrade. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 824:280–283, 2016. doi: 10.1016/j.nima.2015.10.047.

[7] F Carena et al. Preparing the ALICE DAQ upgrade. *Journal of Physics: Conference Series*, 396(1):012050, 2012.

[8] G. Bauer et al. The new CMS DAQ system for LHC operation after 2014 (DAQ2). *Journal of Physics: Conference Series*, 513(TRACK 1), 2014.

[9] Tomasz Bawej et al. Boosting Event Building Performance Using Infiniband FDR for CMS Upgrade. In *Proceedings of Technology and Instrumentation in Particle Physics 2014 (TIPP2014)*, Amsterdam, 2014.

[10] Pieter Hintjens, Martin Sústrik, and Others. ZeroMQ. URL `http://zeromq.org/`.

[11] A Dworak, M Sobczak, F Ehm, W Sliwinski, and P Charrue. Middleware trends and market leaders 2011. In *Proceedings of ICALEPCS2011*, volume 111010, Grenoble, 2011.

[12] A Dworak, F Ehm, P Charrue, and W Sliwinski. The new CERN Controls Middleware. *Journal of Physics: Conference Series*, 396(1):012017, 2012.

[13] Martin Sústrik and Others. nanomsg. URL `http://nanomsg.org/`.

[14] Ioannis Charalampidis. A libfabric-based Transport for nanomsg, 2016. URL `https://github.com/wavesoft/nanomsg-transport-ofi`.

[15] OFI Working Group. Libfabric, . URL `https://ofiwg.github.io/libfabric/`.

[16] Linux User's Manual. *epoll(7)*, April 2012. URL `https://linux.die.net/man/7/epoll`.

[17] OFI Working Group. librdmacm, . URL `https://github.com/ofiwg/librdmacm`.

[18] The Open Group. IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, 2008.

[19] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

[20] Paul Grun. Introduction to Infiniband for end users. *White paper, InfiniBand Trade Association*, 2010.

[21] Intel. *Intel Performance Scaled Messaging 2 (PSM2) Programmer's Guide*, November 2015.

[22] Intel. Xeon Processor E5-2630 v3, . URL `http://ark.intel.com/products/83356/`.

[23] Intel. Xeon Processor E5-2660 v3, . URL `https://ark.intel.com/products/81706/`.