

# Event-Driven RDMA Network Communication in the ATLAS DAQ System with *NetIO*



Jörn Schumacher, CERN  
On behalf of the ATLAS TDAQ Collaboration  
[jorn.schumacher@cern.ch](mailto:jorn.schumacher@cern.ch)

# About NetIO

NetIO is a general-purpose communication library optimized for RDMA networks:  
Infiniband, OmniPath, Ethernet+RoCE, ...

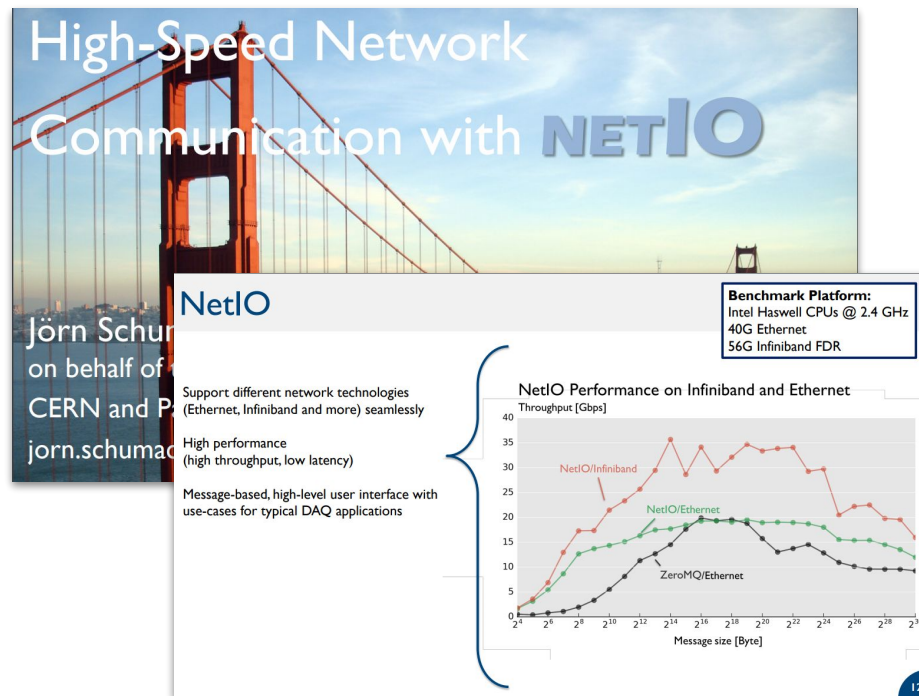
Can be used for HEP Data Acquisition Systems. Enables use of HPC technology without having to use HPC-centric APIs like MPI

**The library has been around for some time, but has recently been rewritten to a large extent following a new event-driven approach**

Apart from network communication, NetIO can be used to drive the readout of DAQ controllers, file I/O, ... using the same event-driven approach

This talk is about some lessons learned while (re-)implementing NetIO. Some of the ideas can be applied to other IO-heavy applications

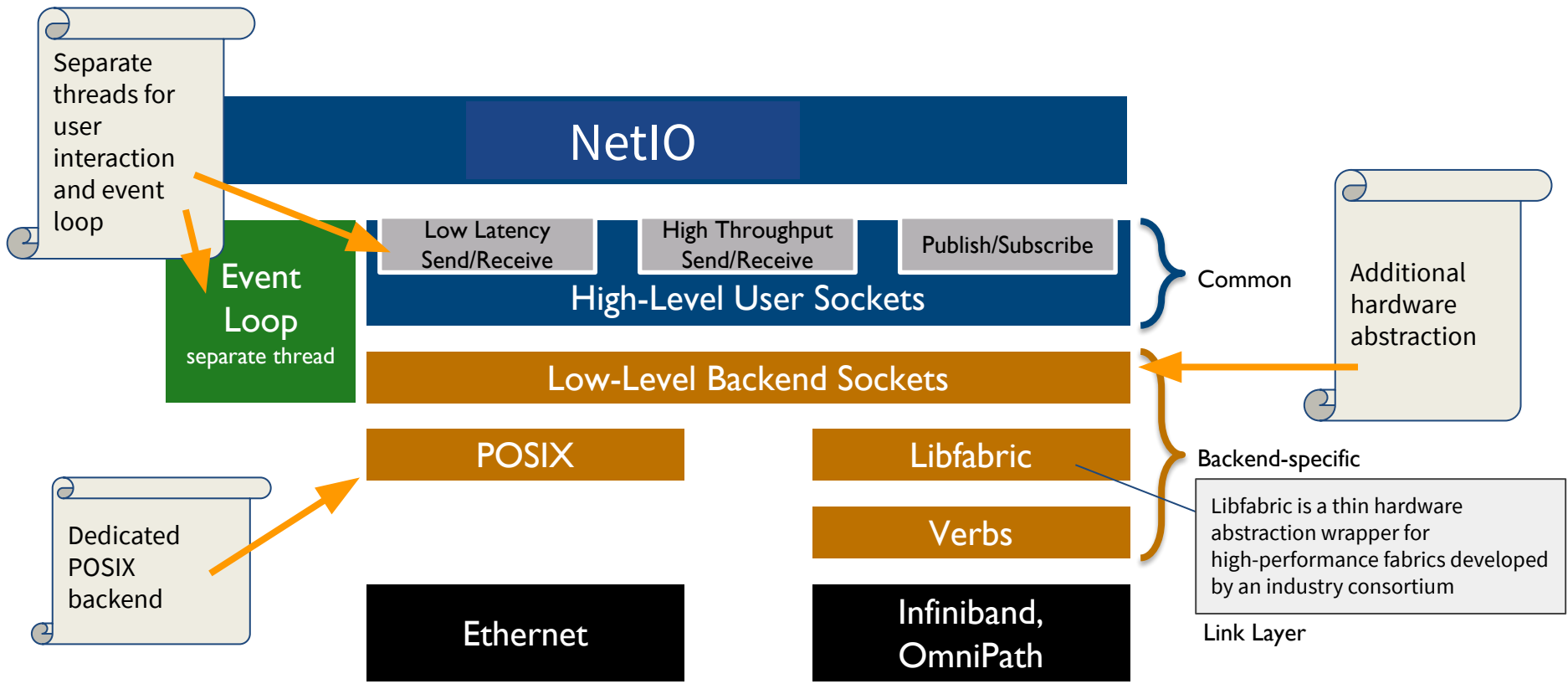
*“IO-heavy”*: data rate ~100 Gpbs, frame rate in the MHz range



Initial presentation at [CHEP 2016](#)

NetIO is in use in the ATLAS experiment (FELIX project, see talk by W. Panduro Vazquez, Track 1 / Monday)

# NetIO in 2016



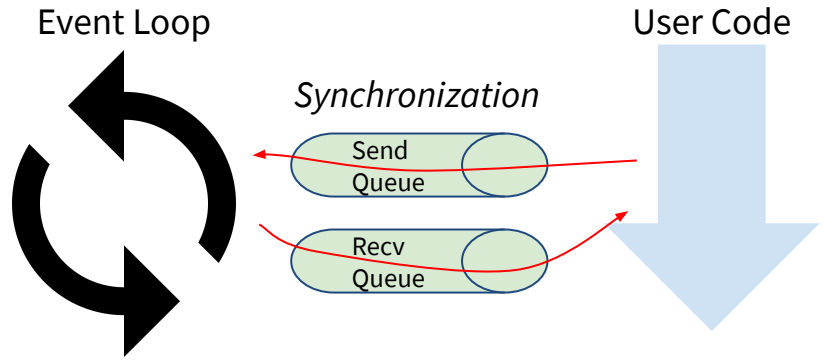
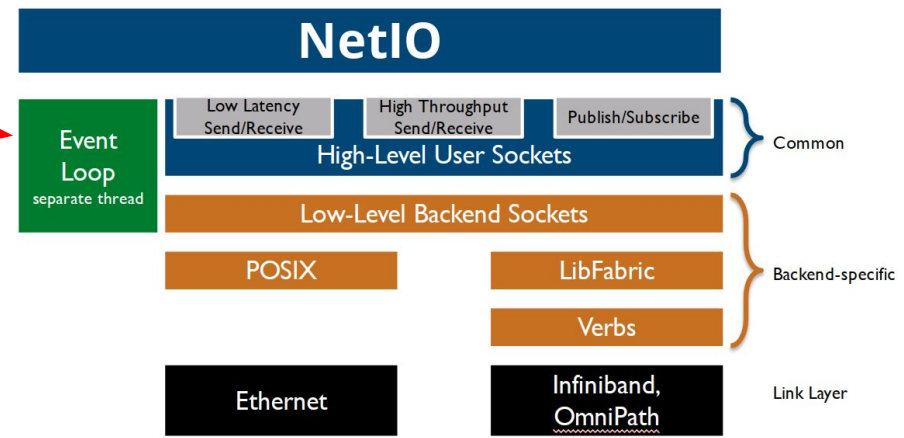
Lesson 1

# Event-Driven Architecture

# Lesson 1: Event-Driven Architecture

Early versions of NetIO used an event loop (based on Linux' epoll subsystem) running in a separate thread. The library was "internally event-driven". The event loop was not exposed to users

User code was not following the event-driven approach, so when sending or receiving messages, messages had to transition between the two paradigms



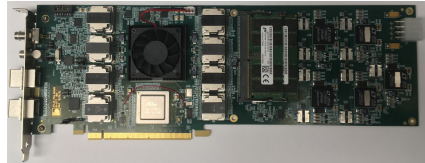
NetIO used [Intel TBBs lock-free queues](#) to buffer data at the transition point

This transition incurred significant overhead

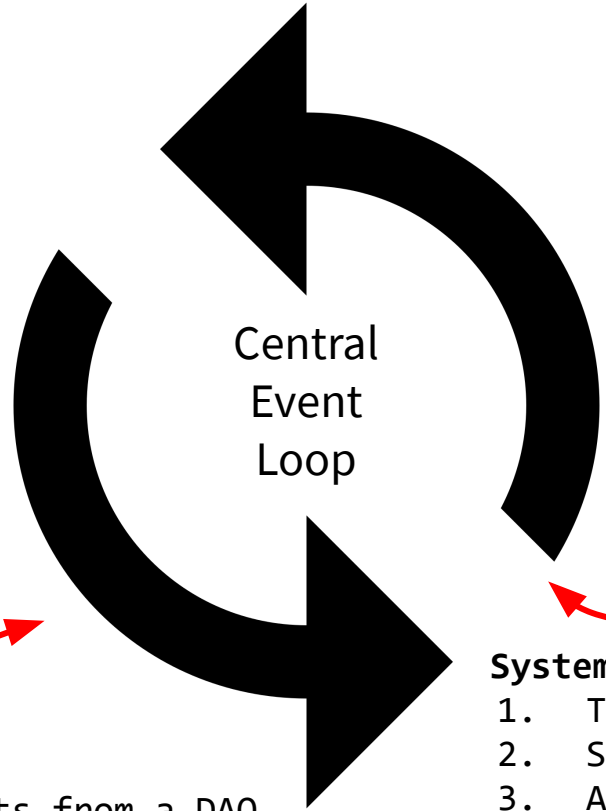
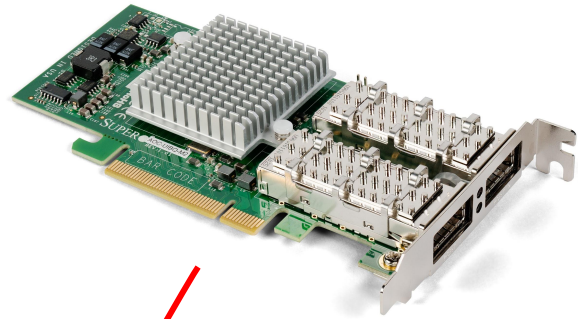
# Lesson 1: Event-Driven Architecture

In the optimized NetIO, everything is event-driven - including user code

DAQ Device



RDMA NIC



Central Event Loop

### RDMA Events:

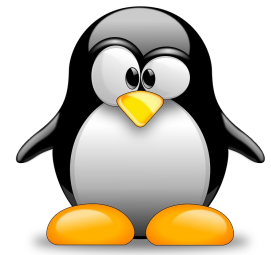
- 1. Send completed
- 2. Data received
- 3. Buffer available for sending

### User Events:

For example, interrupts from a DAQ card

### System Events:

- 1. Timer events (timerfd)
- 2. Signals (eventfd)
- 3. Any file descriptor event

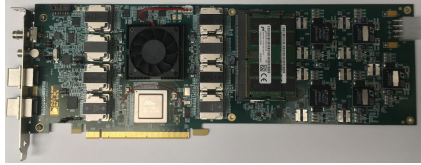


LINUX

# Lesson 1: Event-Driven Architecture

In the optimized NetIO,  
everything is event-driven -  
including user code

DAQ Device



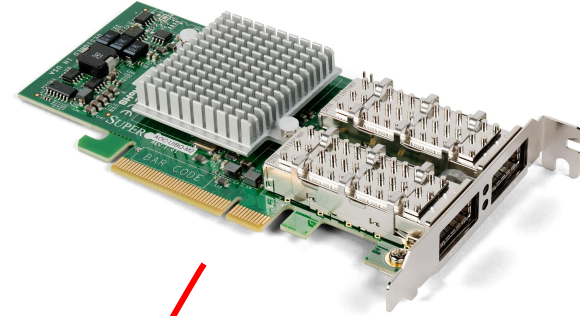
Event-driven user code that  
reads out a DAQ device (file  
descriptor API)

## User Events:

For example, interrupts from a DAQ  
card

Central  
Event  
Loop

RDMA NIC

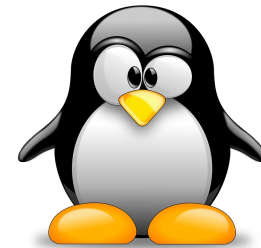


## RDMA Events:

1. Send completed
2. Data received
3. Buffer available for sending

## System Events:

1. Timer events (timerfd)
2. Signals (eventfd)
3. Any file descriptor event

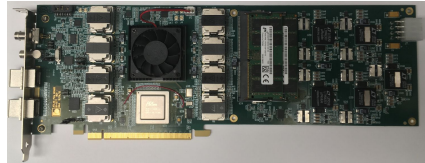


LINUX

# Lesson 1: Event-Driven Architecture

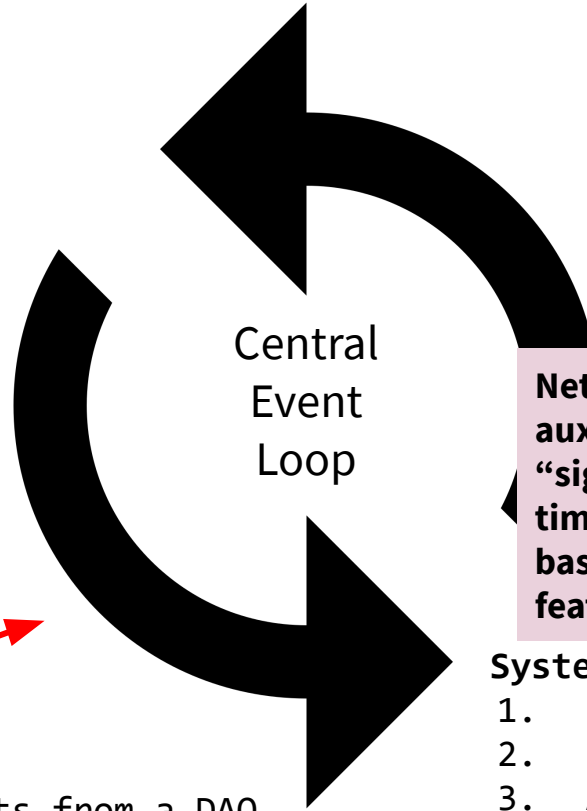
In the optimized NetIO, everything is event-driven - including user code

DAQ Device



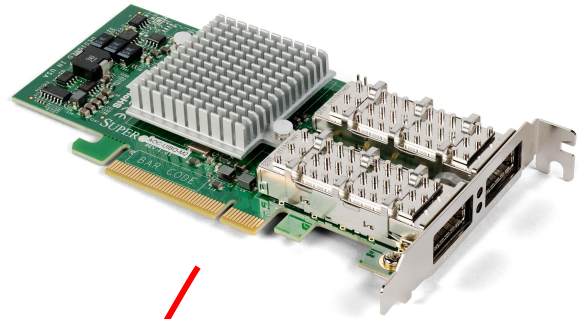
Event-driven user code that reads out a DAQ device (file descriptor API)

User Events:  
For example, interrupts from a DAQ card



Central Event Loop

RDMA NIC



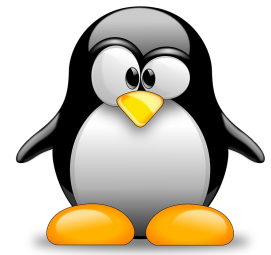
RDMA Events:

NetIO supports different auxiliary event source like "signals" and "periodic timers" (implementation is based on Linux kernel features)

... completed  
... received  
... per available  
... sending

System Events:

1. Timer events (timerfd)
2. Signals (eventfd)
3. Any file descriptor event



LINUX



# Example: simple timer

```

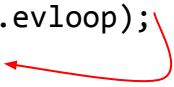
#include <stdio.h>
#include "netio.h"

struct netio_context ctx;
struct netio_timer timer;

void on_timer(void* ptr) {
    int* ctr = (int*)ptr;
    printf("%d\n", (*ctr)--);
    if(*ctr == 0) {
        netio_terminate(&ctx.evloop);
    }
}

```

This is a callback,  
in this case a timer  
that is periodically  
executed



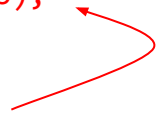
```

int main(int argc, char** argv) {
    int counter = 10;
    netio_init(&ctx);
    netio_timer_init(&ctx.evloop,
                    &timer);

    timer.cb = on_timer;
    timer.data = &counter;
    netio_timer_start_s(&timer, 1);

    // run event loop
    netio_run(&ctx.evloop);
    return 0;
}

```



This executes the event loop which runs until  
terminated

Lesson 2

# Reduce Thread Synchronization

# Lesson 2: Reduce Thread Synchronization

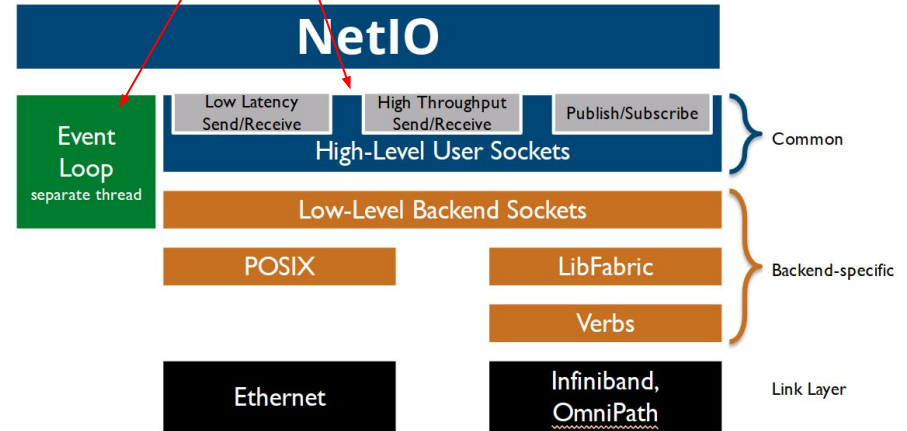
Having multiple threads is a nice way to increase performance of an application by parallelising the load for many applications

However, some overhead is incurred due to thread synchronization

In I/O-heavy applications, synchronization overhead may outweigh the advantages of parallelism. **Most work is done by the DMA controller of the NIC, not by the CPU.**

NetIO switched to a single thread approach. **A single thread is enough to saturate a modern 100 Gbps RDMA network link**

Multiple Threads that need to be synchronized  
(mutexes, concurrent queues, semaphores, spinlocks, ...)



Multiple threads may still be used at a higher level (user code) and may be beneficial if the application is also doing significant processing on top of the I/O work

# Lesson 2: Reduce Thread Synchronization

Having only a single thread executing an event loop has implications on user code:

User code may not block, as that would stall the event loop

No further events can be processed, performance can degrade

```
daq_device.callback_data_available = on_data_available;
socket.callback_buffers_available = on_data_available;
```

```
void on_data_available(...) {
    for(i = last_item_processed; i<available; i++) {
        int res = netio_send(socket, &big_buffer[i]);
        if(res == AGAIN) {
            last_item_processed = i;
            return;
        }
    }
}
```

**Function is called by the event loop as a result of an event (e.g. an interrupt)**

**Or when the output socket is ready to send data again**

- 1. No call is ever blocking**
- 2. Caller needs to handle cases where a call fails because it would have to block**
- 3. Save state and continue**

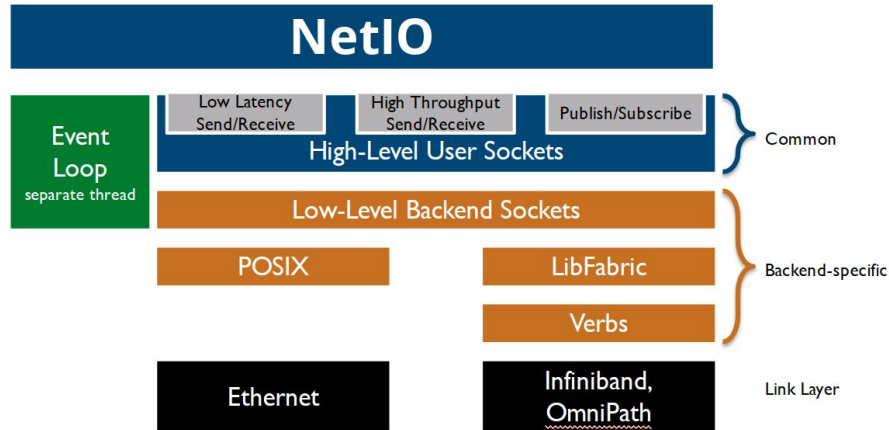
**User code paradigm:** do not wait for conditions. Instead, let the event loop notify you about condition changes

*Note: syntax simplified for illustration*

Lesson 3

# **Avoid unnecessary complexity**

# Lesson 3: Avoid unnecessary complexity



NetIO included an additional hardware abstraction layer. This made it possible to implement a separate POSIX backend for non-RDMA network technologies.

This comes at the cost of additional complexity.

It is also redundant: libfabric has built-in support for TCP- and UDP-based networks (the support for this has significantly improved in the last versions)

The POSIX backend is not necessary anymore

Old NetIO: **19213** lines of C++

New NetIO: **8076** lines of C

# Recap: Event-Driven NetIO Architecture

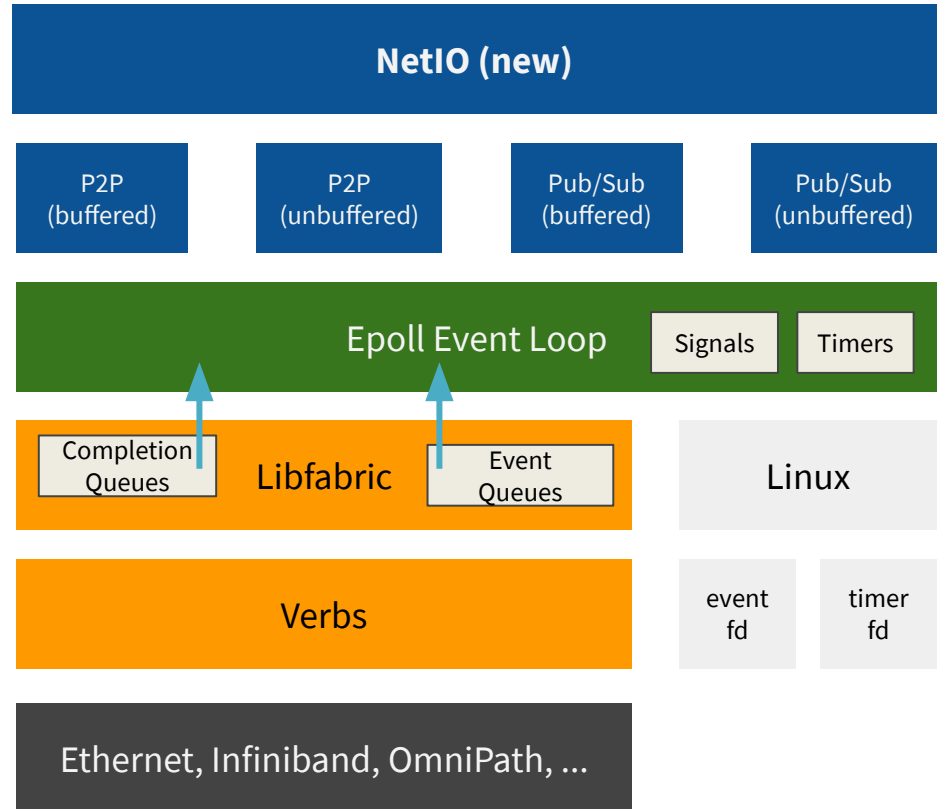
## User code and event loop in a single thread

- Callback-driven code
- Non-blocking code to avoid stalling the event loop

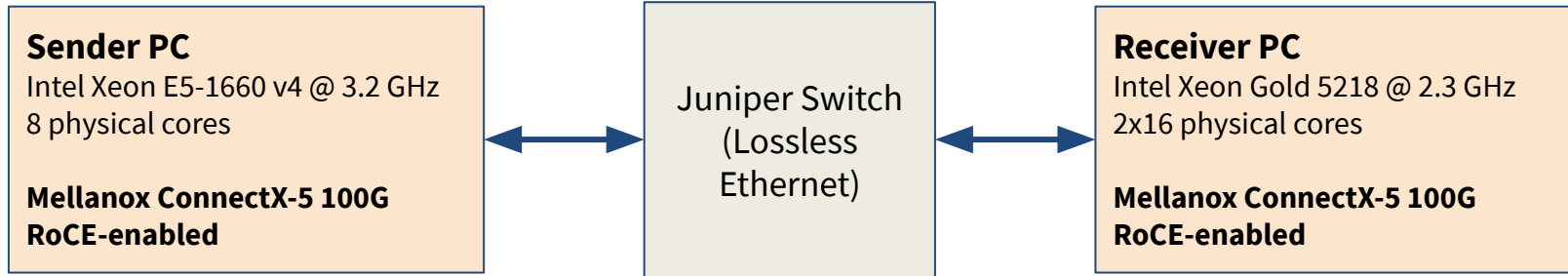
This avoids a lot of synchronization (queues etc.) that were a bottleneck in the original implementation

## No more dedicated Ethernet backend, this is supported via libfabric

This reduces overhead by abstraction



# Benchmarks



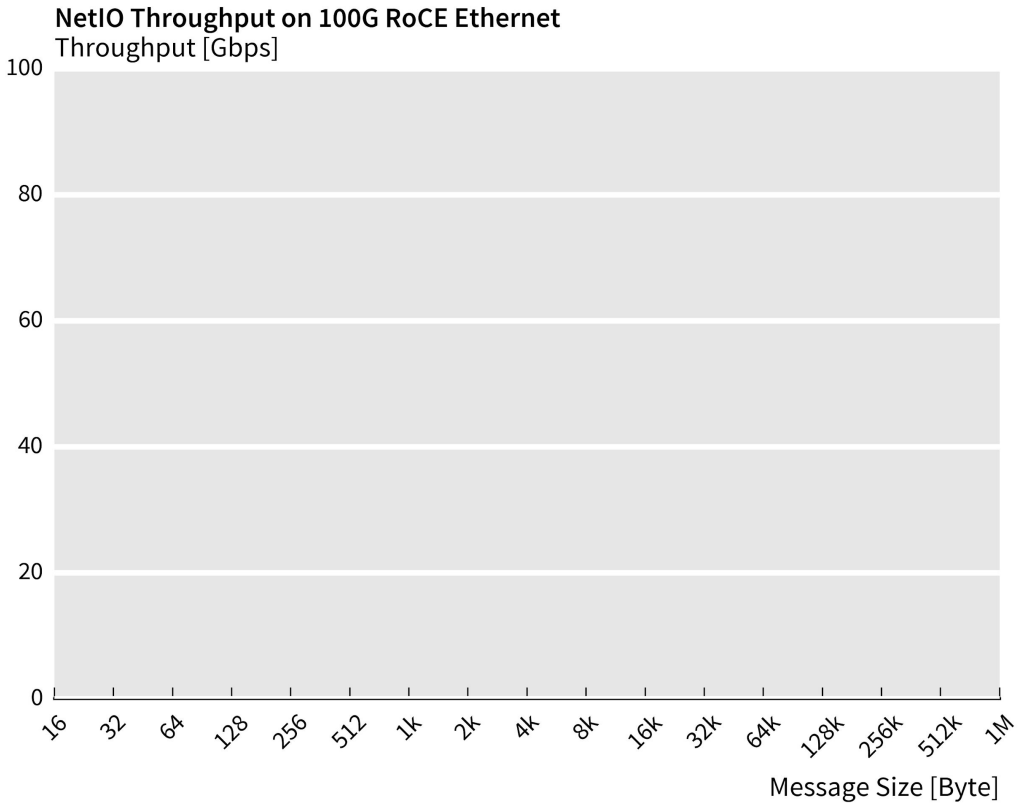
In the tests we use Ethernet, but with the *RDMA-over-Converged-Ethernet (RoCE)* extension

The same API (Verbs, RDMA) as for Infiniband is used, but on top of lossless Ethernet hardware

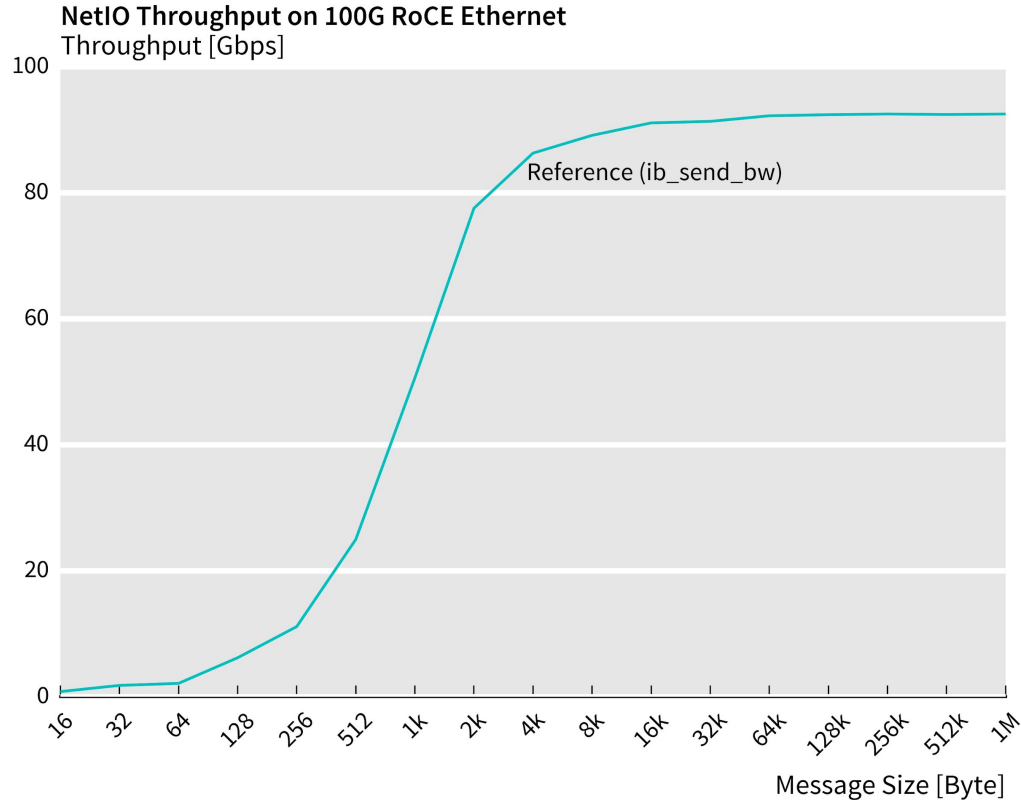
Maximum bandwidth measured using *ib\_send\_bw*: ~**92 Gbps**



# Performance - Throughput



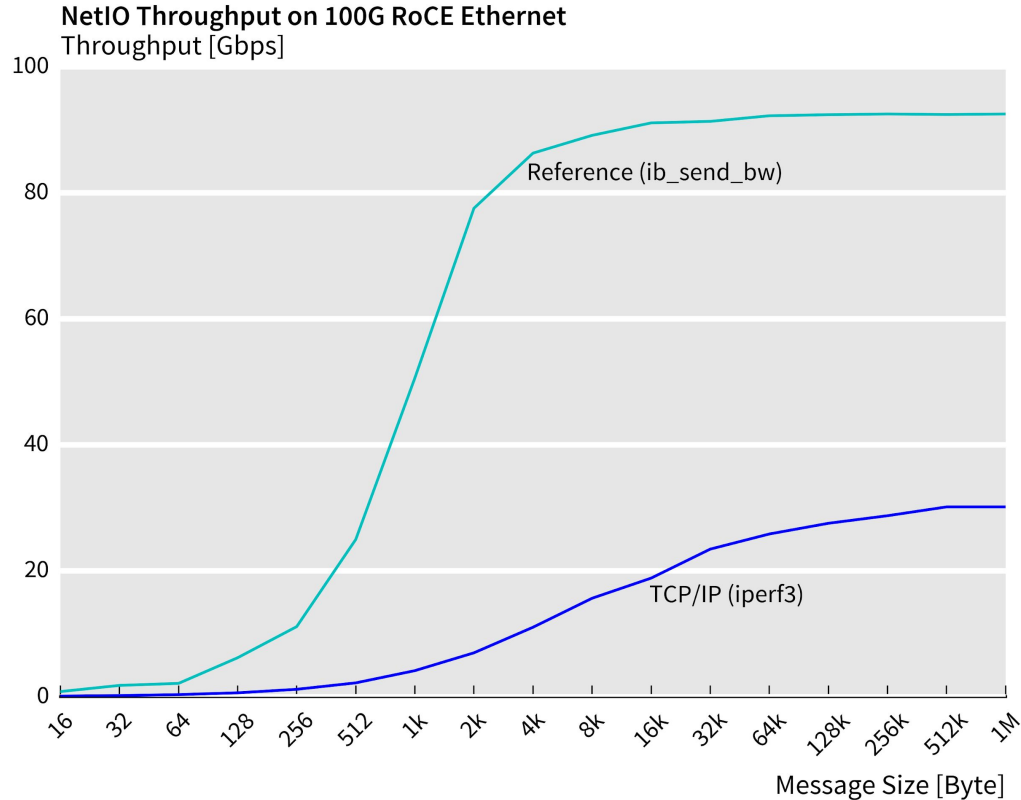
# Performance - Throughput



The baseline is acquired using a native verbs benchmark tool, *ib\_send\_bw*

With a single thread one can achieve close to link speed

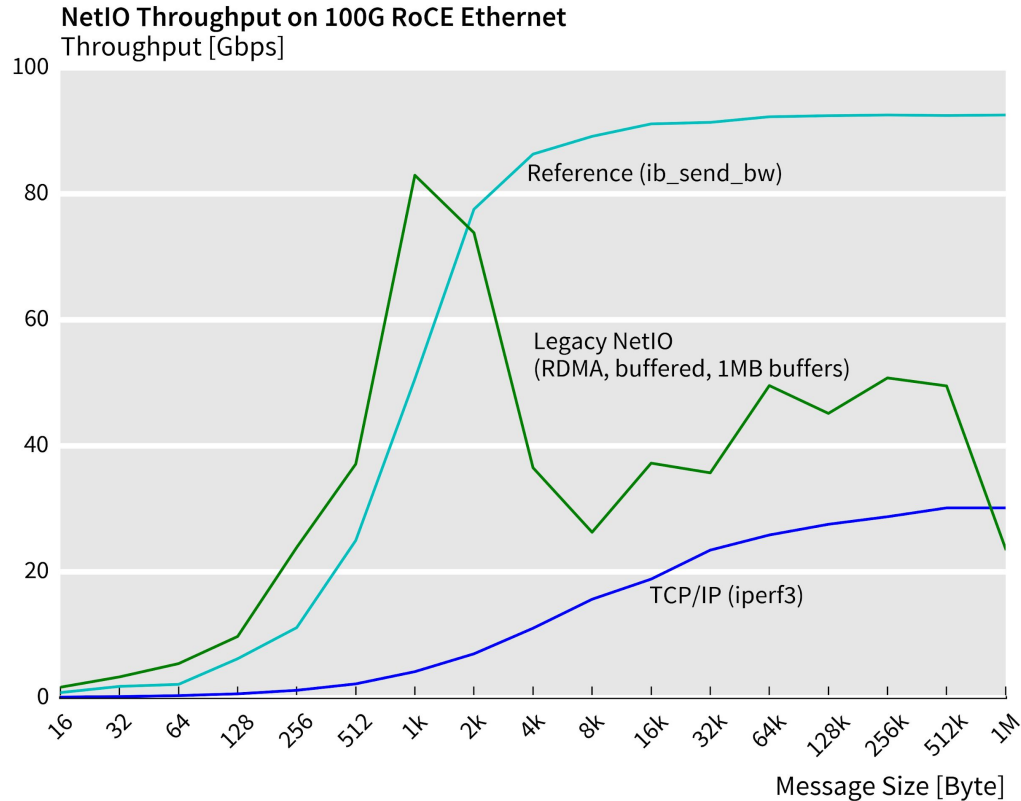
# Performance - Throughput



TCP performs poorly

In order to scale up close to the 100 Gbps that the NIC is capable of, one would have to run up to 8 parallel instances of the benchmark application (iperf3). Essentially, this uses up the entire CPU

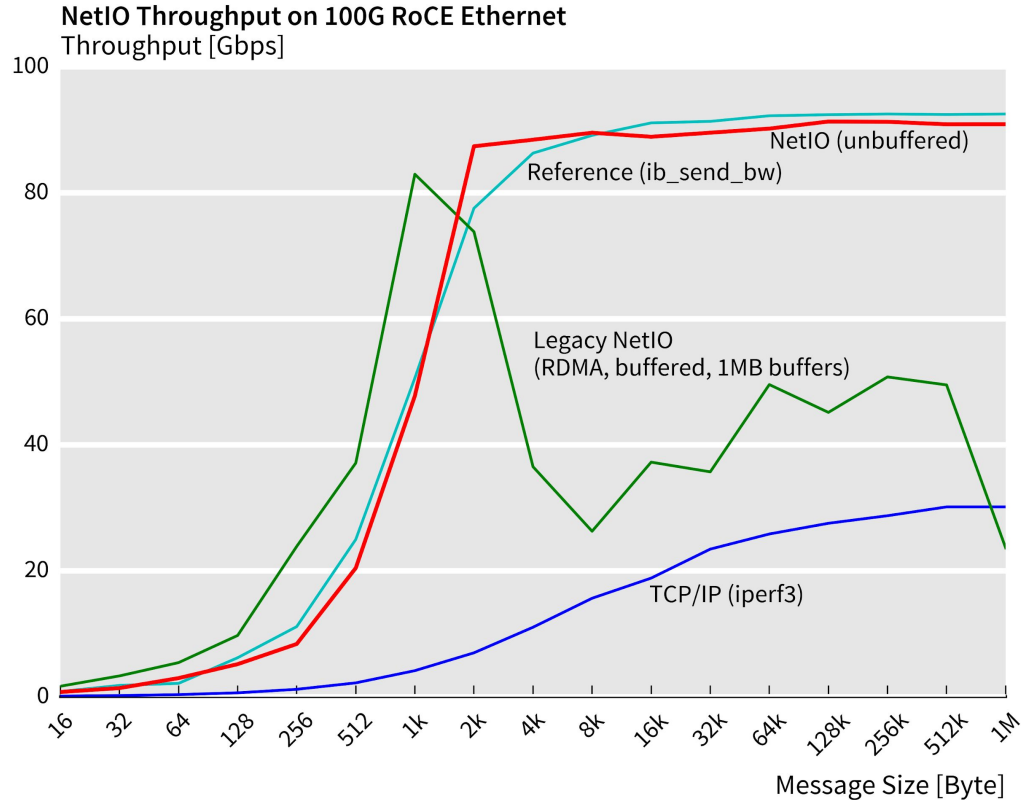
# Performance - Throughput



Legacy NetIO is an improvement compared to TCP/IP, but still it does not perform nearly as good as the native benchmark

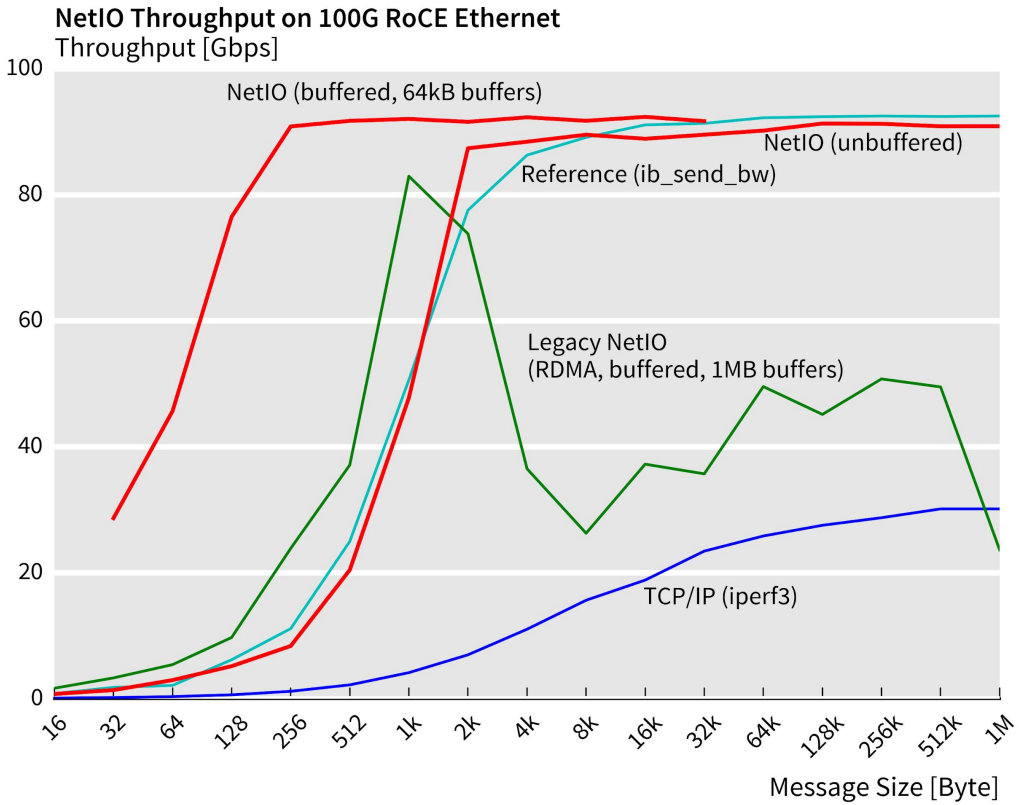
Performance advantage for lower message sizes is due to buffering in Legacy NetIO - the reference benchmark is unbuffered

# Performance - Throughput



The optimized NetIO (unbuffered) is operating at peak performance, about the same speed as the native benchmark

# Performance - Throughput

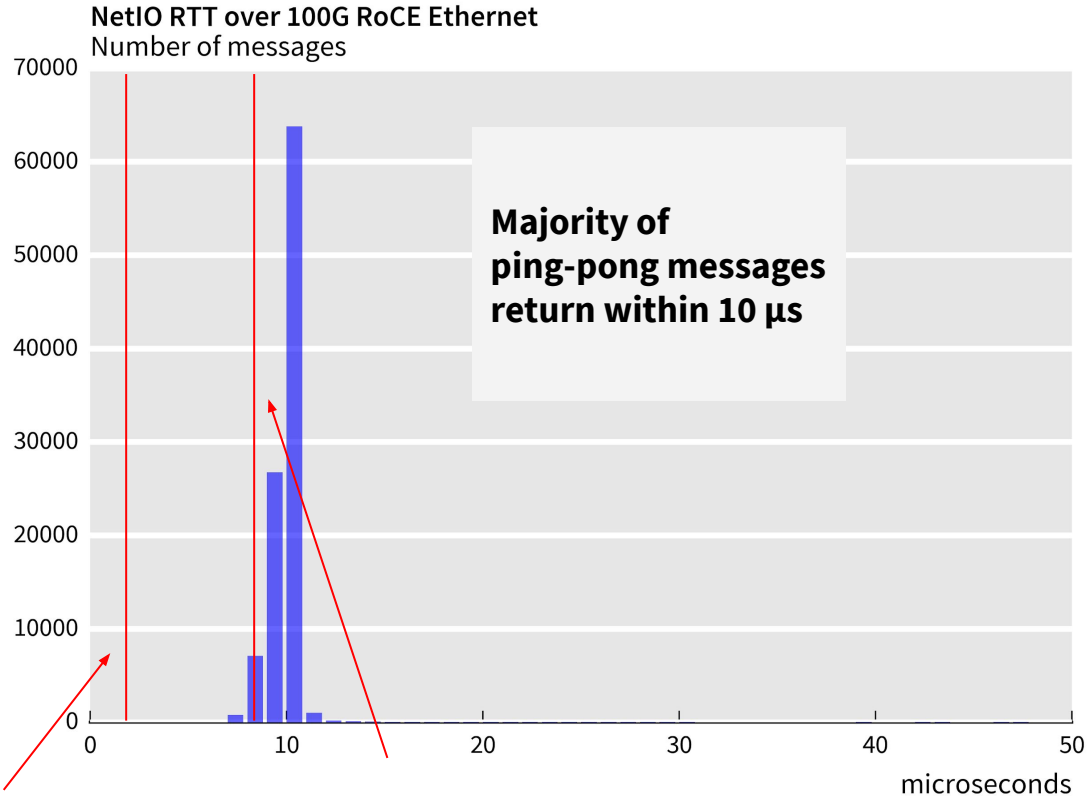


The optimized NetIO (unbuffered) is operating at peak performance, about the same speed as the native benchmark

The optimized NetIO (buffered) improves performance for smaller message sizes (at the cost of some CPU resources)

**A single thread can saturate the link**

# Performance - Latency / RTT (Round Trip Time)



Ideal avg. RTT when polling (3.2  $\mu$ s)

Ideal avg. RTT when sleeping for events (8  $\mu$ s)

## Network Limits

Average **latency** when **polling**:

**1.6  $\mu$ s**

(measured with *ib\_send\_lat*)

Average **latency** when **sleeping for events**:

**4  $\mu$ s**

(measured with *ib\_send\_lat -e*)

NetIO is sleeping for events (this is given by the event-driven architecture)


The ideal expected RTT is then roughly  
 $2 \times 4 \mu\text{s} = 8 \mu\text{s}$

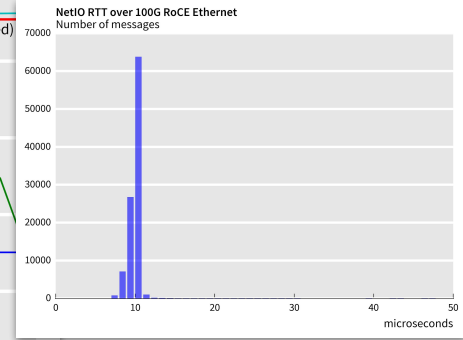
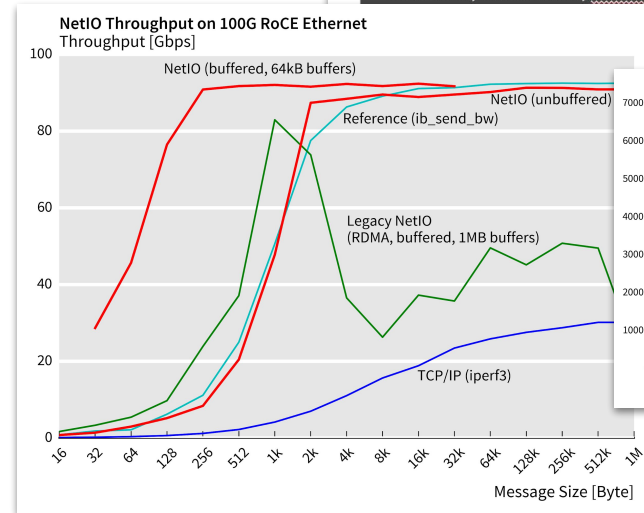
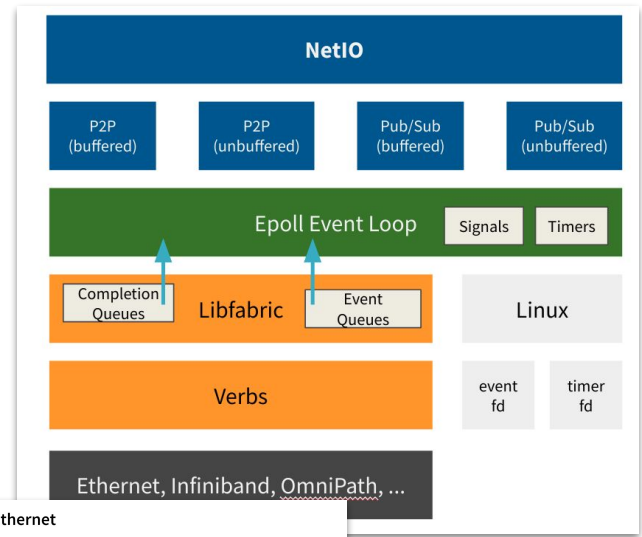
# Summary

Reimplementing NetIO based on a fully event-driven architecture lead to significant performance gains

Performance gains are mostly due to significantly reduced synchronization overhead

NetIO will be released under an Open Source licence in the coming months

<http://cern.ch/atlas-project-felix>  
 jorn.schumacher@cern.ch





# Backup

# Libfabric

NetIO is based on libfabric, a technology-agnostic low-level API for various high-performance fabrics

Libfabric supports a variety of RDMA-hardware (Infiniband, OmniPath, ...) and provides a uniform API to user applications

Libfabric is very thin wrapper around native APIs and quite efficient (low penalty compared to direct use of native APIs)

<https://ofiwg.github.io/libfabric/>

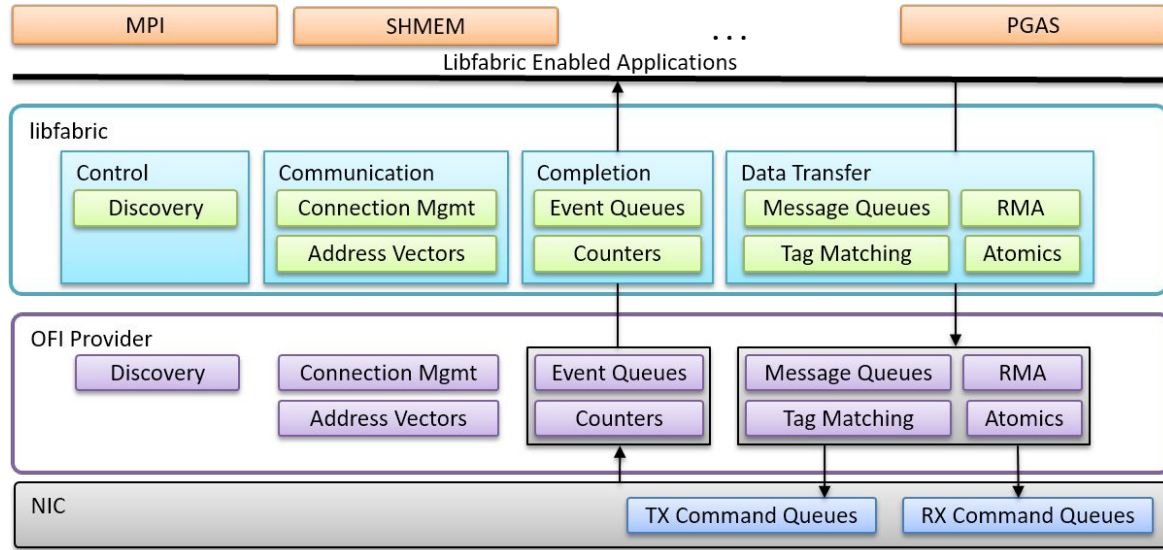


Image source: libfabric manual