NetIO and LibFabric

Jörn Schumacher, CERN
Jorn.Schumacher@cern.ch

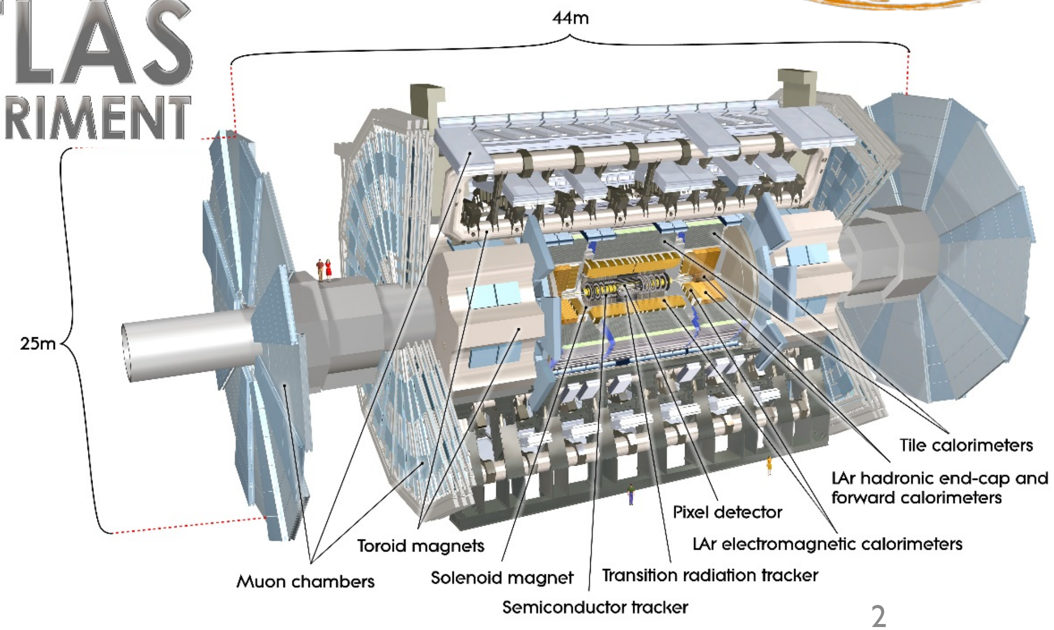
The ATLAS Experiment

Large Hadron Collider (LHC)
27km circular collider in Geneva, Switzerland

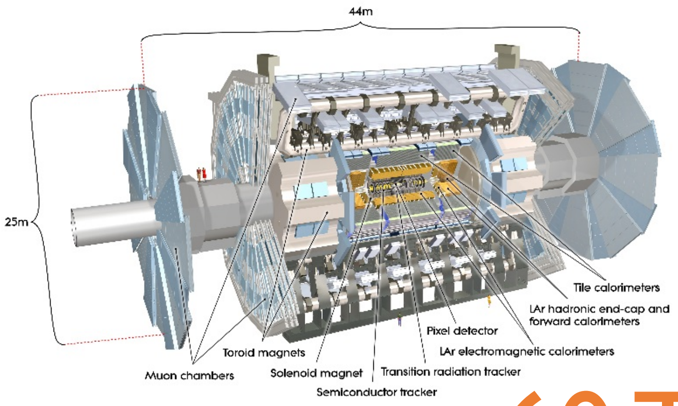
Particle colliders used in HEP study physics processes on a microscopic scale

Length (m)	46
Diameter (m)	25
Weight (t)	7000
Number of electronic channels	$100 \cdot 10^6$

ATLAS EXPERIMENT



Data Acquisition



60 TB/s

Custom electronics and
a server farm with 40,000 cores

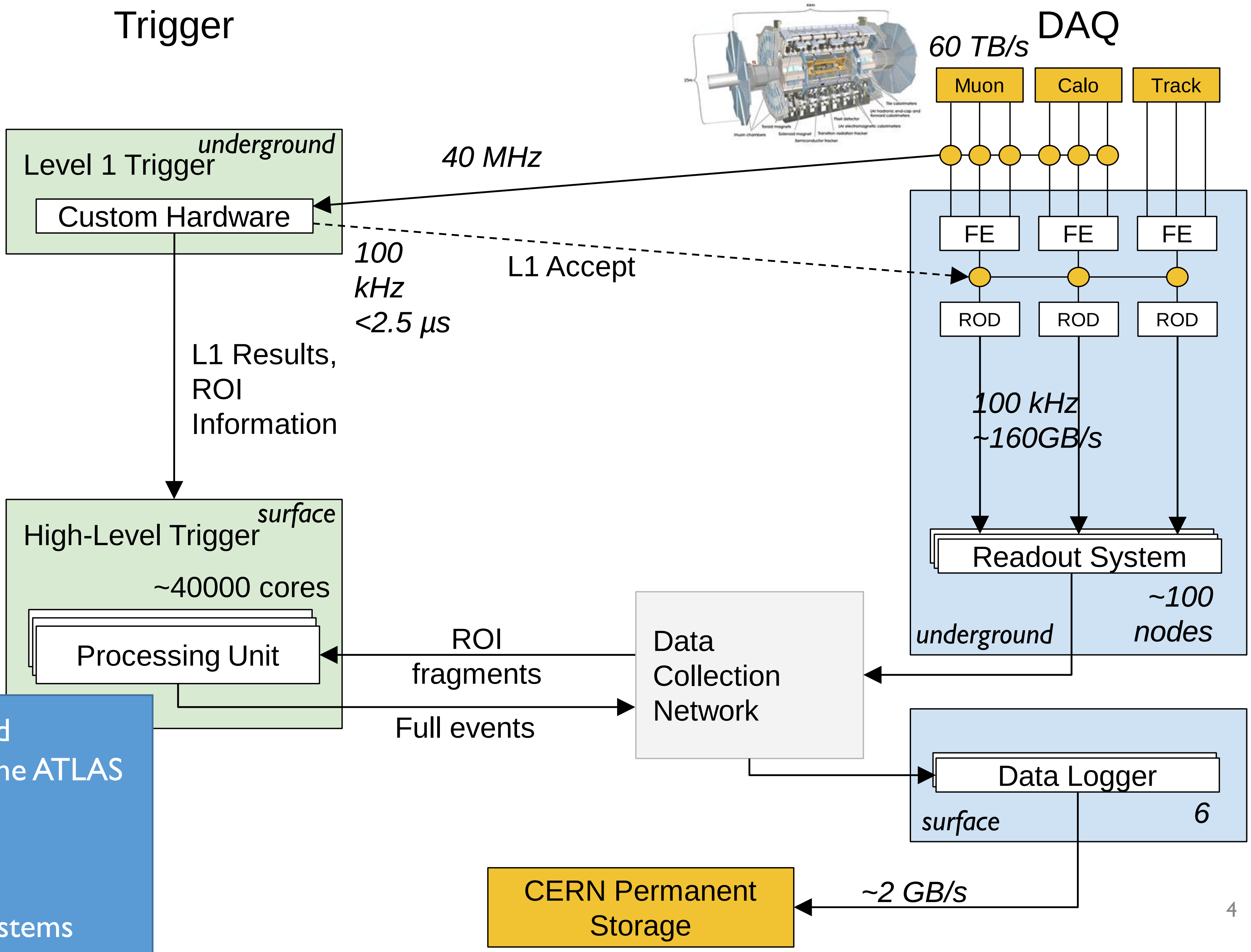
Data Filtering:
Order of
10000x reduction
in **real-time**

2 GB/s

Needle in a haystack: Looking for extremely rare events with a probability of 10^{-13}

Need **High Performance Networks** to move data at high rates under real-time conditions

ATLAS Data Acquisition 2017



In total 10,000s of distributed applications are running on the ATLAS DAQ system

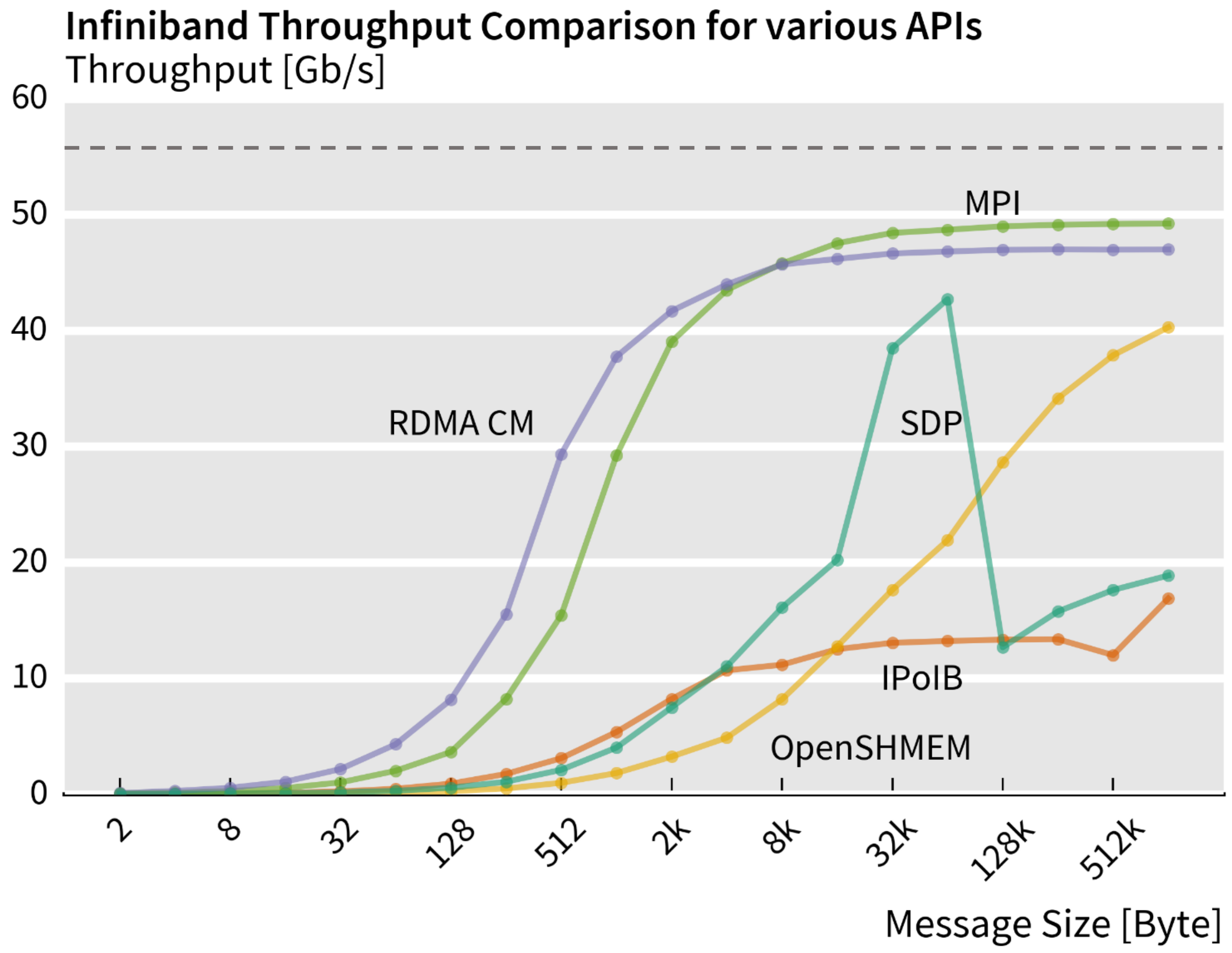
Not displayed: Monitoring, Infrastructure, Calibration systems

Requirement for Network API

- High Throughput (ATLAS Data Acquisition system has to transport more than100 GB/s)
- Low Latency connections for detector control and calibration applications
- High level communication patterns like client/server and publish/subscribe
- Technology agnostic

Via libfabric

Infiniband API Performance



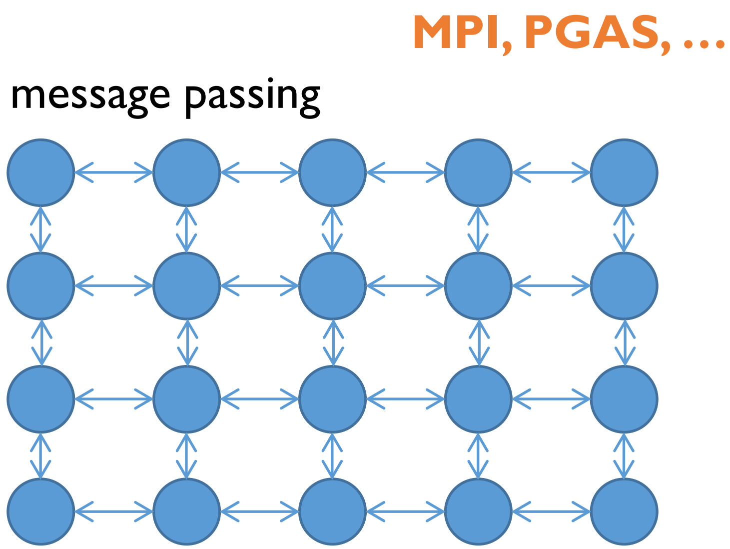
Benchmark on
56G Infiniband FDR

Data-transfer between two nodes (Intel Haswell, 8-core and 10-core systems)

Connected via a single switch

Why not MPI?

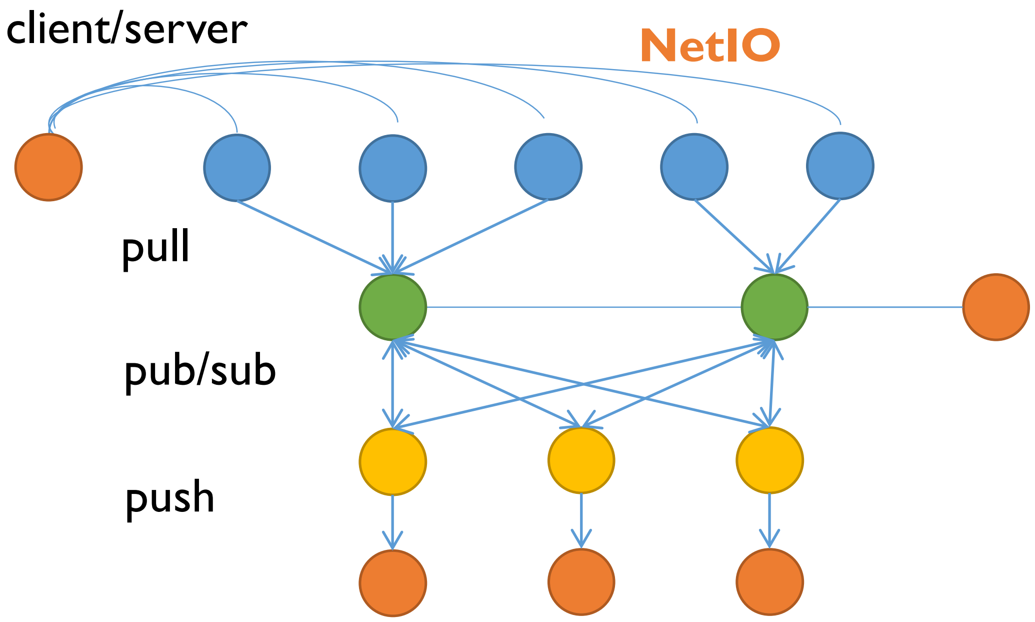
HPC



Regular topology
SPMD Pattern
No Real-Time Requirements
No Failure Tolerance
Static Resource Management

HEP

(High Energy Physics)



Complex topology
Complex distributed system
Real-Time Requirements
Some Failure Tolerance
Dynamic Resource Management

NetIO

A high-level, general-purpose API for HPC networks

NetIO was designed with High Energy Physics experiments in mind, but it is not restricted to this use case

Design Goals:

- Native support for HPC interconnects via a back-end system
- Different operation modes tuned for high-throughput communication or low-latency communication
- High-level communication patterns including publish/subscribe

User-level sockets

Provide a simple interface for users

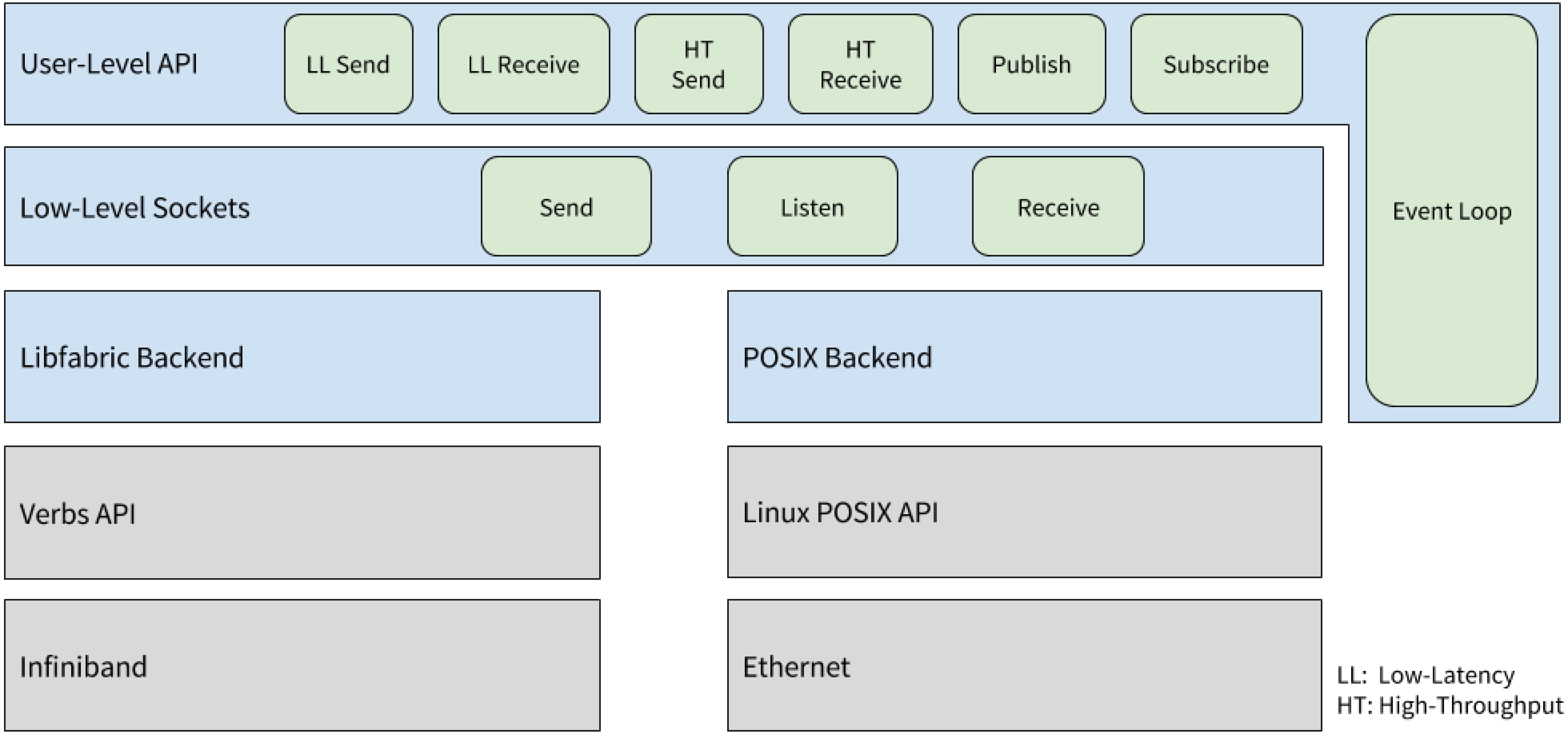
High-level communication patterns:
- Send/Receive
- Publish/Subscribe

Come in two flavors:
- Low-latency
- High-Throughput

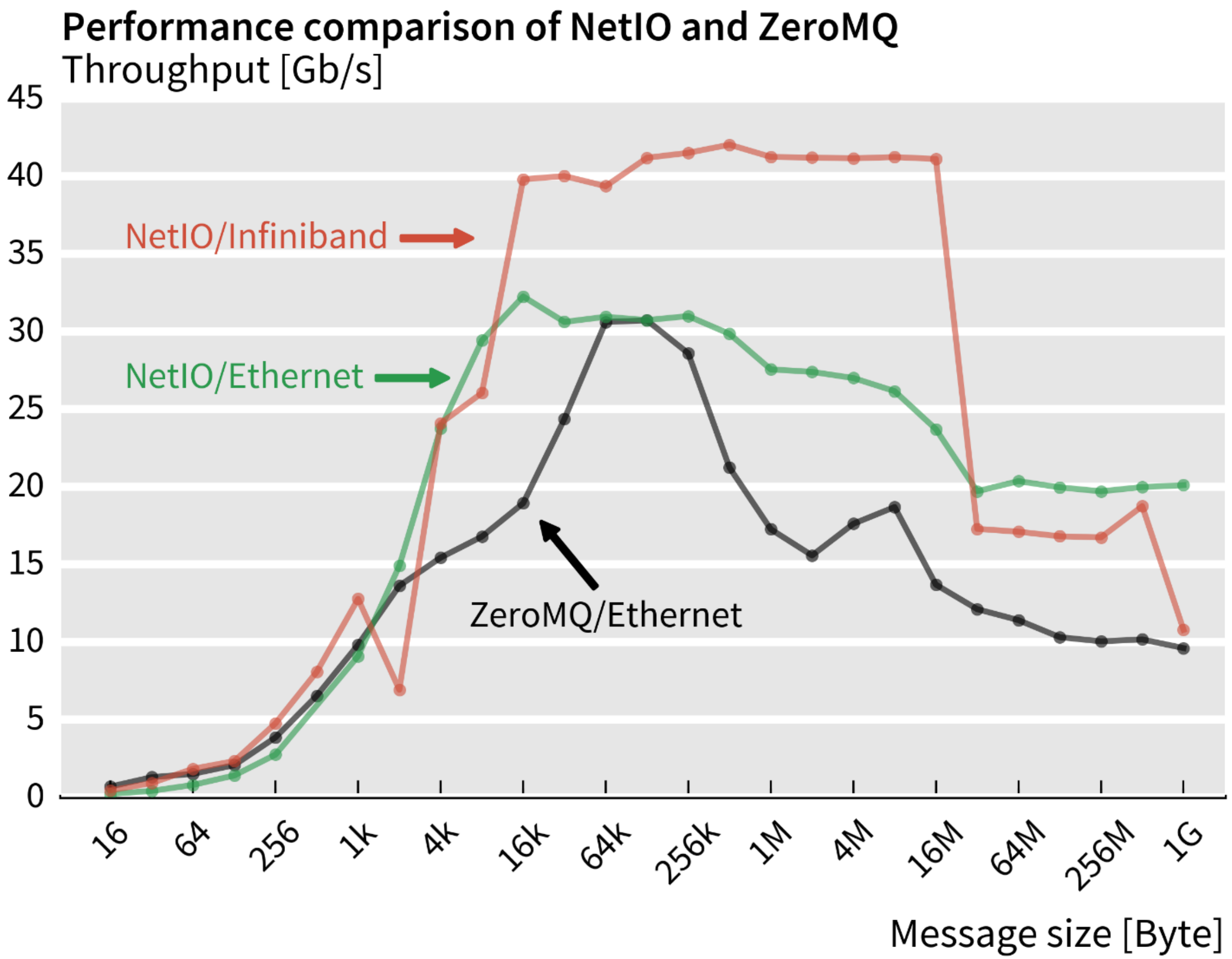
Addressing based on IPv4 or IPv6

Low-Latency • **Callback-based** • **No buffering**	**High-Throughput** • **Queue-based** • **Buffering**
LL Send socket	HT Send socket
LL Receive socket	HT Receive socket
LL Subscribe socket	HT Subscribe socket
Publish socket *Both high-throughput and low-latency subscribe sockets can connect*	

NetIO Architecture



NetIO Throughput: Push/Pull



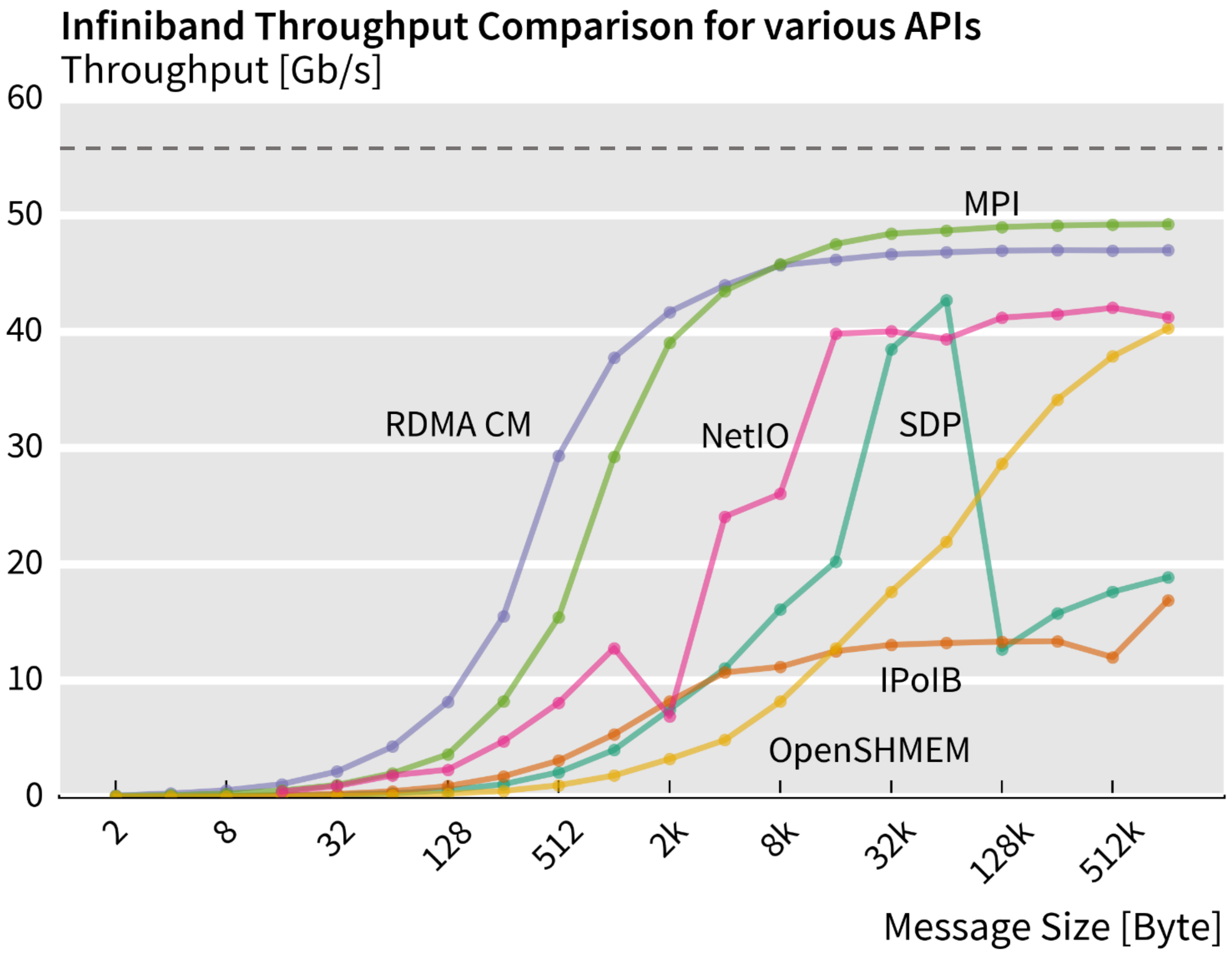
Push/Pull benchmarks

56G FDR Infiniband
40G Ethernet
1 MB pagesize

NetIO outperforms ZeroMQ in nearly all uses cases

Using the Infiniband mode of the underlying hardware allows a performance boost that we can leverage with NetIO – without changing our software

NetIO compared to other Infiniband APIs



Some room for improvement compared to MPI and RDMA CM benchmarks

LibFabric is great

- Documentation is much better than, e.g., Verbs
- Asynchronous
- File Descriptors for completion and event queues – easy integration with epoll
- Technology agnostic
  - Enables us to explore new technologies without fear of vendor lock-in
  - Or even make better use of our current hardware (e.g. RoCE)

Some ideas from the NetIO perspective

- NetIO requires ordering of messages, i.e. Reliable Connection (RC) mode, for deserialization
  - Limits choice of providers
  - Might be able to work around that, but it would be nice if the providers took care of that by providing RC
  - Or a generic compatibility mode for non-RC providers. How efficient could that be implemented?

- A written performance tuning guide would be useful
  - Parameter settings etc. can be difficult for non-industry experts. Would be good for us to learn about best practices

Summary

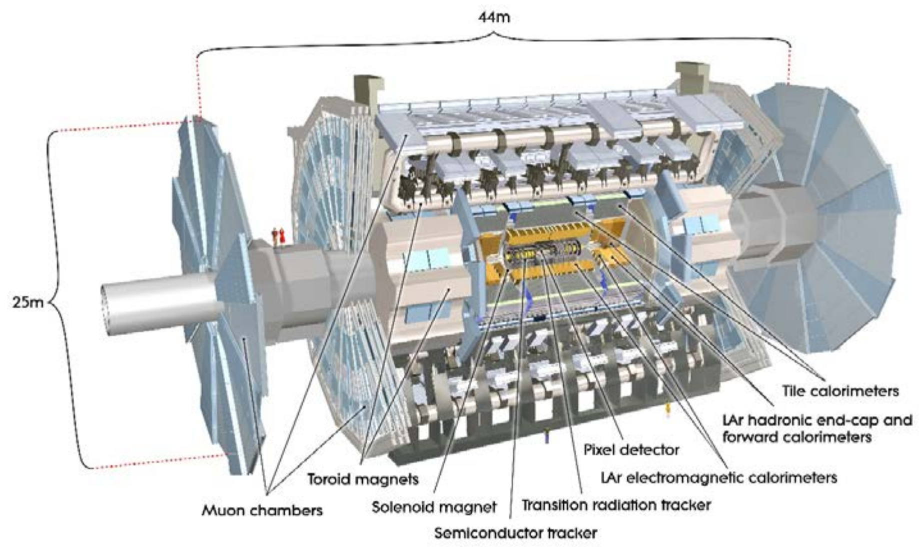
High Energy Physics has different requirements than typical HPCs applications

What we need:

APIs with high-level interfaces for datacenter-like applications that support high-performance fabrics

-> NetIO + LibFabric

NetIO is not yet in production-ready state, but getting there. The plan is to release NetIO as OpenSource software ~end of 2017/beginning of 2018



Backup

Memory Management

Messages are packed into **pages** (buffering for higher efficiency)

Typical max. page size is 1 MB

Event loop drives a timeout to send out partial pages and avoid connection starvatation

NetIO maintains a list of pre-allocated, free pages per connection

Default: up to 256 pages per connection

Pages are recycled after having been processed (i.e. fully sent or received)

Low-level sockets

Uniform interface used by user-level sockets

Abstract interface that is implemented by back-ends

Pages: Buffers that contain one or multiple messages

Listen Sockets: Listen to incoming connection requests, create receive sockets

Receive Sockets: Receive pages from remote endpoints, deserialize into messages

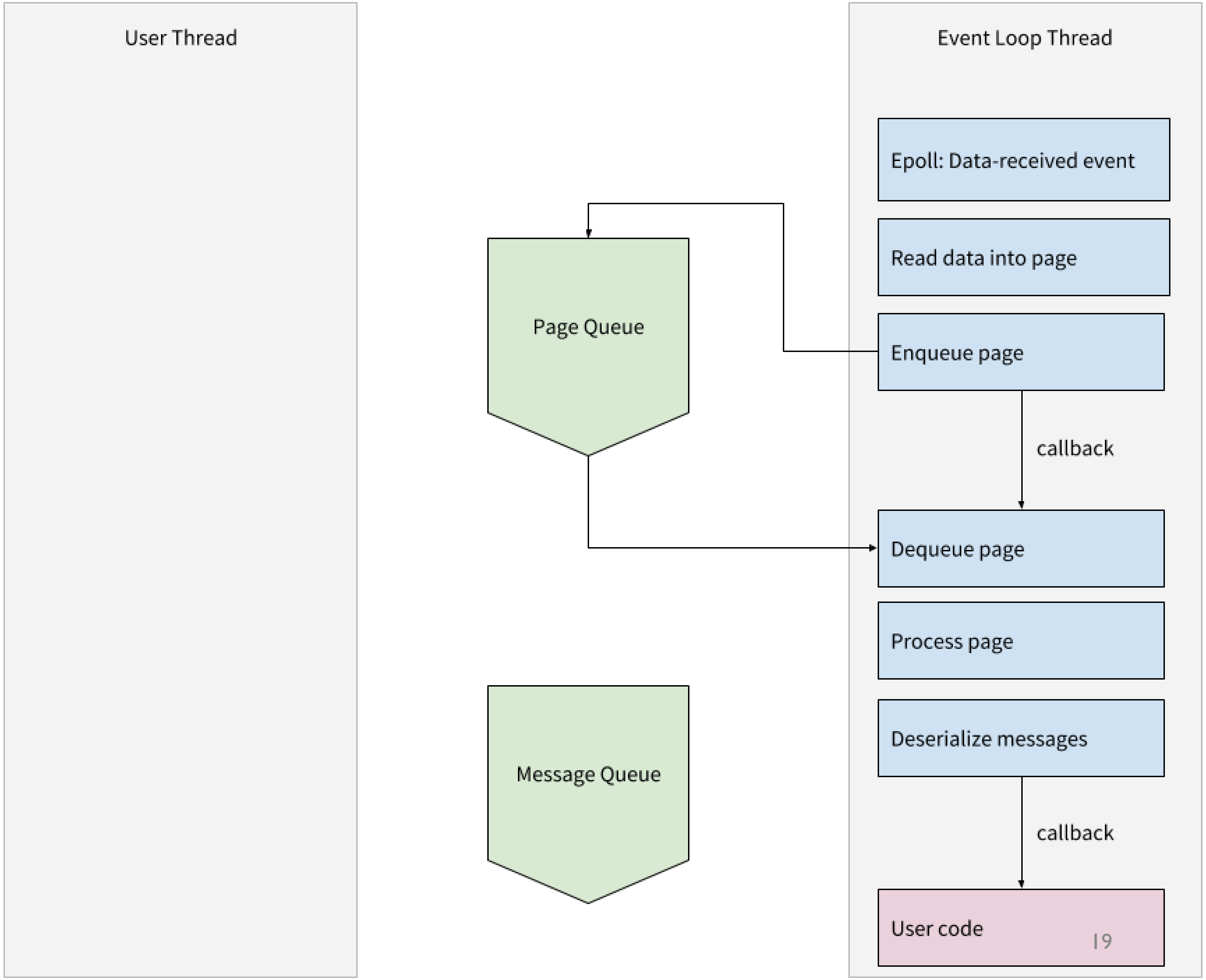
Send Sockets: Send pages to remote endpoint.

No distinction between high-throughput and low-latency communication (this is done in the user-level sockets)

Configuration interface to enable fine-tuning of connection parameters

Low Latency Mode

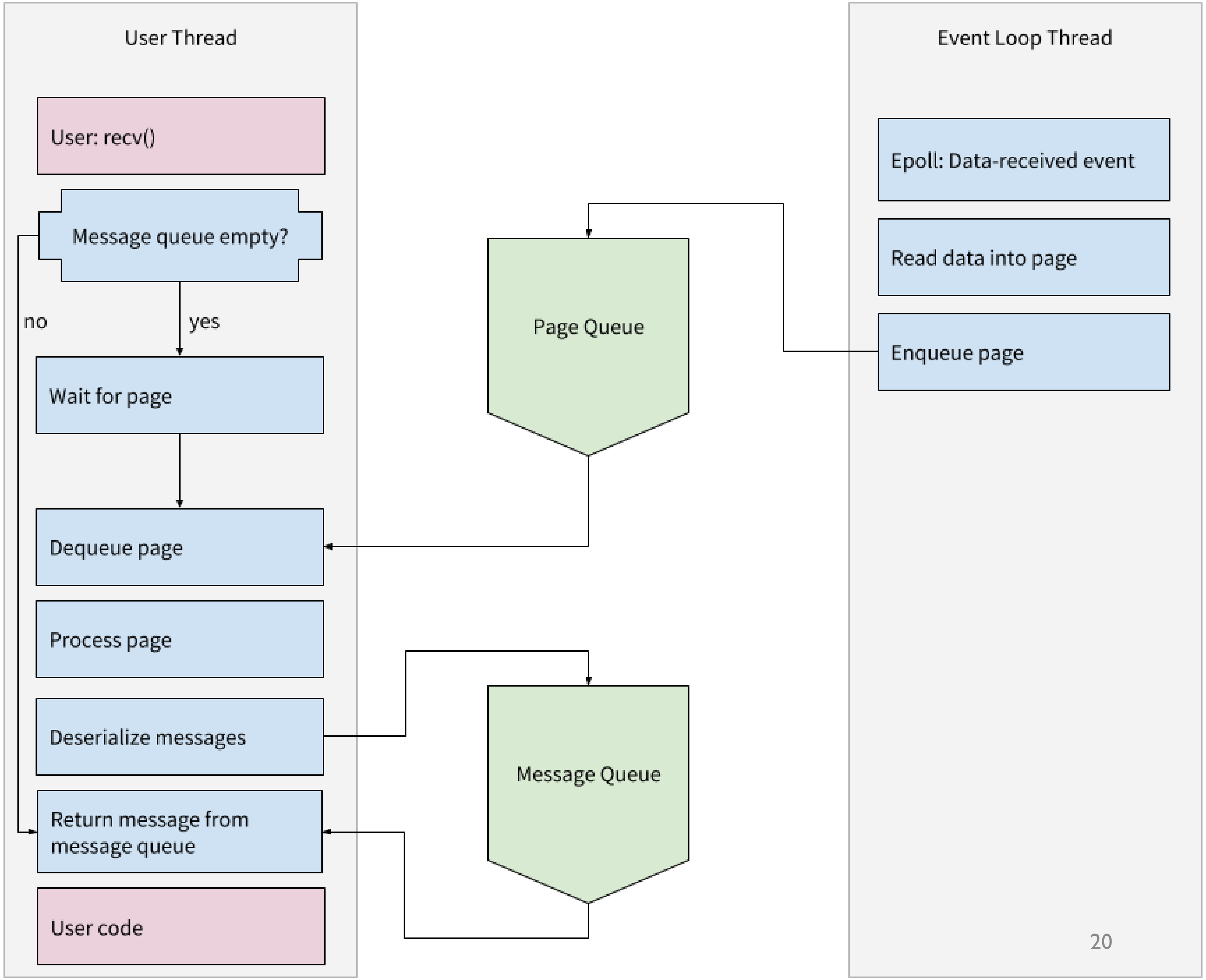
Low latency
- No thread synchronization
- No buffering, pages contain a single message
- Skipping message queue
- Immediate handling of messages via callbacks
- In the future: also skip page queue



High Throughput Mode

High Throughput

- Minimal work in the event loop so it can return to process incoming pages
- Buffering: Multiple messages per page
- Event-loop drives buffer timeout to avoid connection starvation
- Explicit user call to retrieve messages



Back-ends

POSIX

Uses POSIX stream sockets (TCP), which translates naturally into the low-level socket API

Nagle's algorithm disabled
(buffering in user-level sockets)

Simple integration with epoll event loop

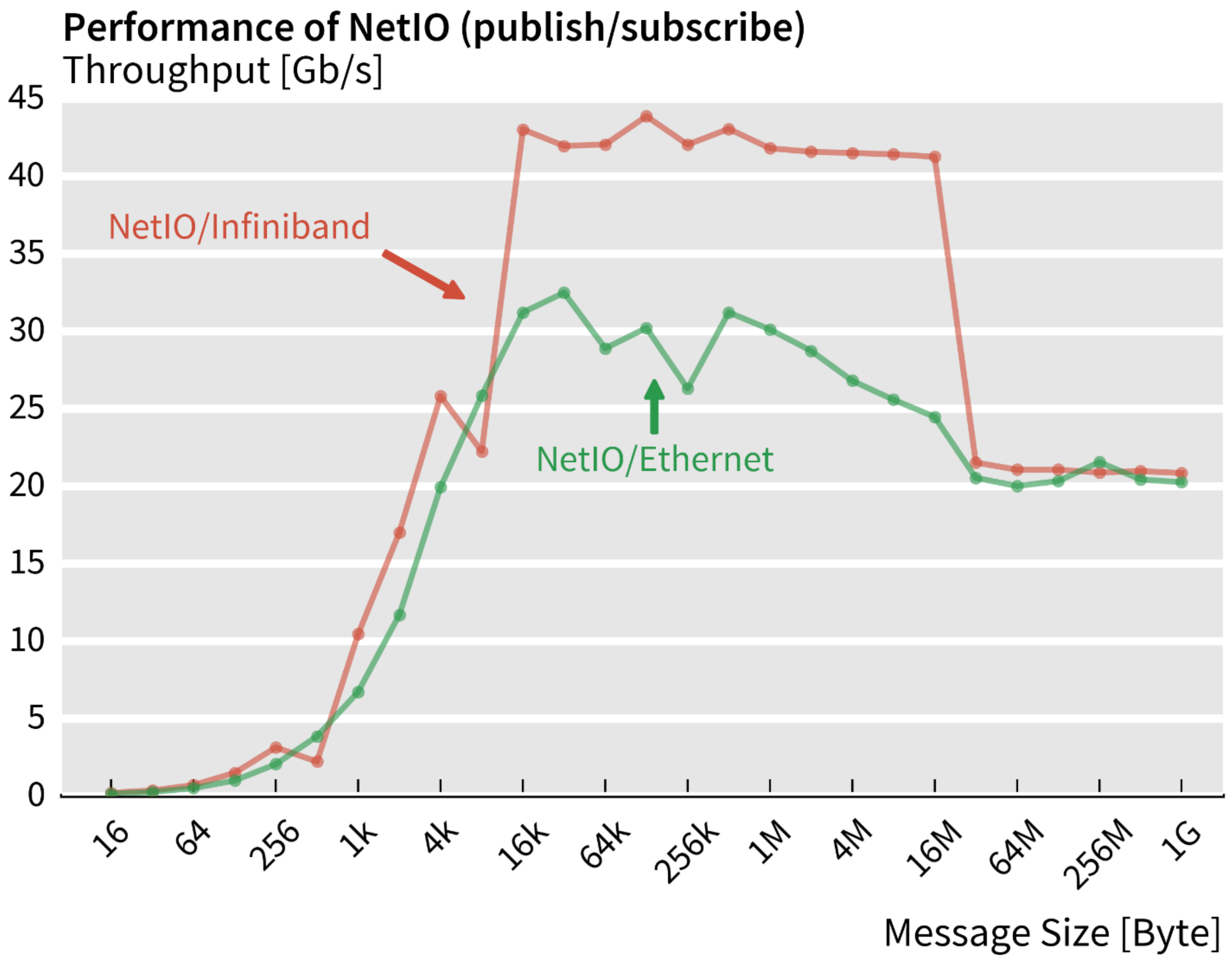
Libfabric

Uses libfabric Reliable Connection (RC)

RDM mode is not supported – ordering is needed to ensure proper deserialization
(That means currently RDM-based libfabric providers are not supported, for example the PSM provider for OmniPath. OmniPath is instead supported by the Verbs provider)
Send windows used to control data-flow for higher throughput

Uses file descriptors for asynchronous completion management – can be integrated in the epoll event loop

NetIO Throughput: Publish/Subscribe



Publish/Subscribe benchmarks

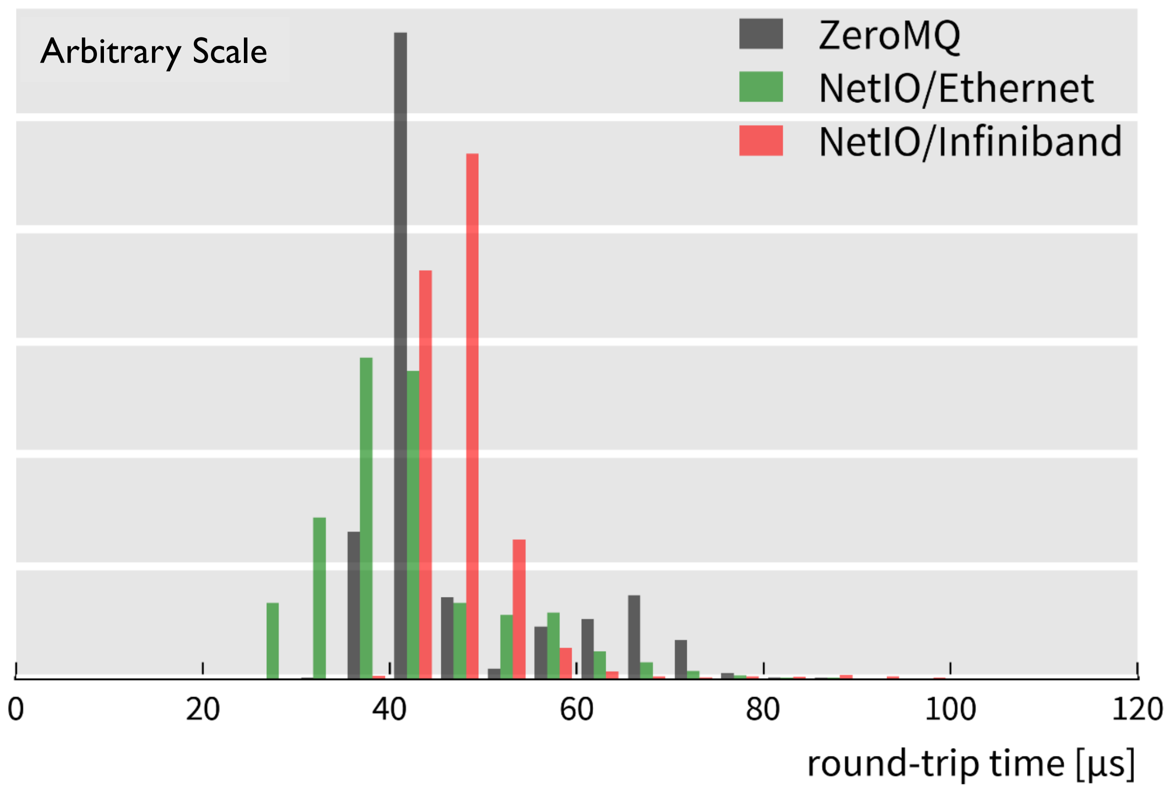
56G FDR Infiniband
40G Ethernet
1 MB pagesize

Similar to the push/pull benchmarks, using the Infiniband mode of the hardware yields a performance boost

ZeroMQ already discarded due to limited performance

NetIO Latency

Round-Trip Time (RTT) Comparison



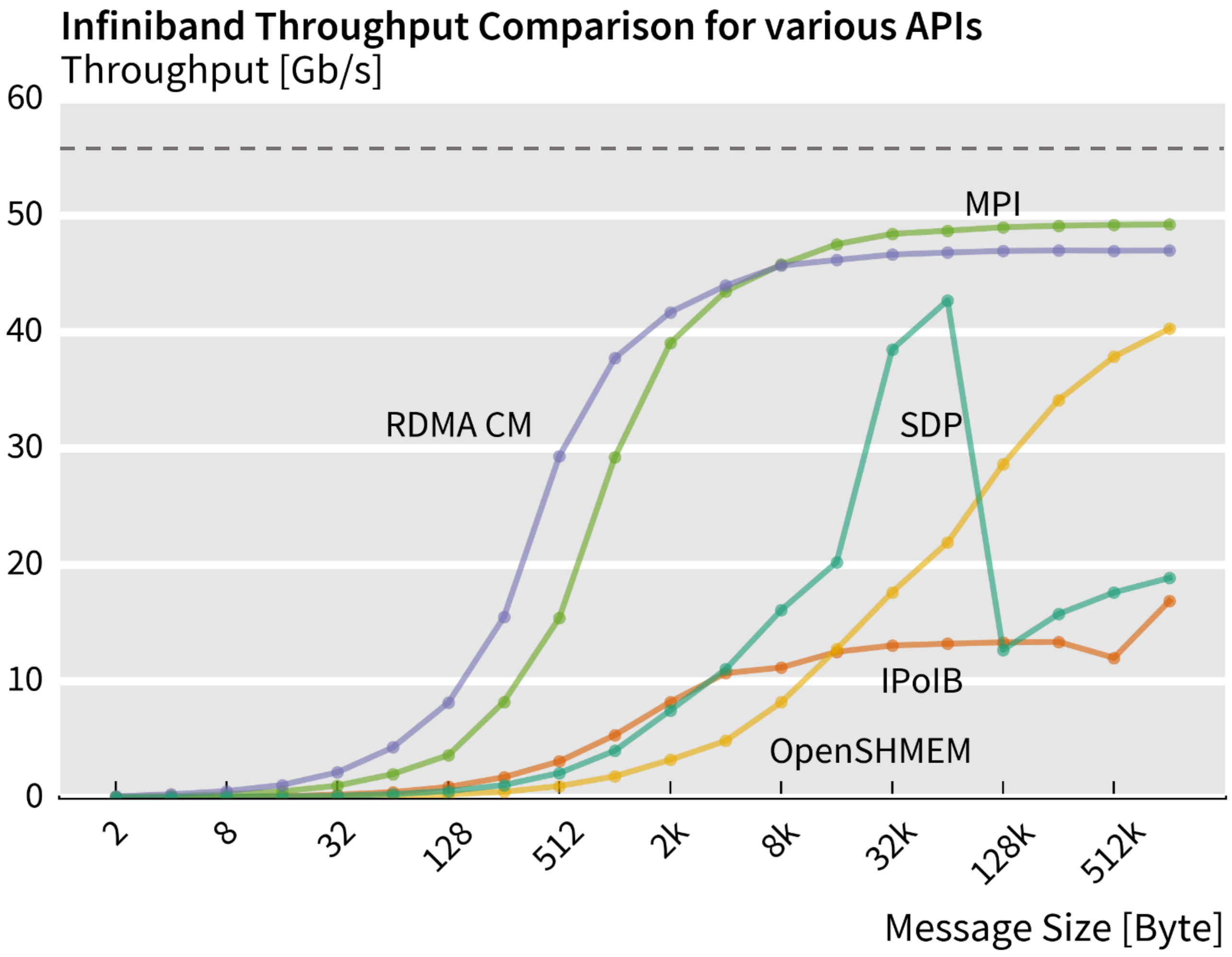
Point-to-Point benchmarks
No additional load on switch

56G FDR Infiniband
40G Ethernet

Latency is very similar for ZeroMQ and NetIO.

Lower latency expected for NetIO/Infiniband: room for improvement

NetIO compared to other Infiniband APIs



56G Infiniband FDR

Data-transfer between two nodes (Intel Haswell, 8-core and 10-core systems)

Connected via a single switch

NetIO performance exceeds the performance of emulation layers

Still some room for improvement compared to MPI/native APIs

NetIO Status & Outlook

Some performance improvements planned

- New ZeroCopy mode
- Improved queuing scheme

Status

- Small functional improvements needed
- Ongoing parameter studies
- User documentation being written
- OpenSource release planned this year
- NetIO going to be used in ATLAS data-taking beginning 2019