Project Hermes

Fabian Ruhland, Filip Krakowski, Michael Schöttner

Department of Computer Science
Heinrich-Heine-University Düsseldorf

18.04.2023

Motivation

Motivation

Java NIO provides a modern and easy-to-use API for blocking and non-blocking communication $\rightarrow$ Many projects are built using NIO

Networking Frameworks **Distributed Databases** **Computing Frameworks**

Netty **Apache Cassandra** **Apache Spark**

Motivation

Java NIO provides a modern and easy-to-use API for blocking and non-blocking communication $\rightarrow$ Many projects are built using NIO

Networking Frameworks **Distributed Databases** **Computing Frameworks**

Netty **Apache Cassandra** **Apache Spark**

Drawback: NIO relies on classic Java Sockets $\rightarrow$ Ethernet
On many HPC and cloud systems, you are stuck with Gigabit Ethernet and not able to benefit from high speed transports (e.g. InfiniBand)

Motivation

Solution: Build our own NIO implementation (including `SelectorProvider`, `Selector`, `SelectionKey`, `SocketChannel` and `ServerSocketChannel`) -> Use a high speed transport, such as InfiniBand

Motivation

Solution: Build our own NIO implementation (including `SelectorProvider`, `Selector`, `SelectionKey`, `SocketChannel` and `ServerSocketChannel`) -> Use a high speed transport, such as InfiniBand

Unified Communication X (UCX) provides a single API for many transports (e.g. InfiniBand, Shared Memory, NVLink, ...)

$\Longrightarrow$ Use UCX as communication backend for our NIO implementation (Project title: **hadroNIO**)

Related Work

Related Work

Alternative solutions accelerate traditional sockets → NIO relies on sockets and can also be accelerated by these solutions

Related Work

Alternative solutions accelerate traditional sockets $\rightarrow$ NIO relies on sockets and can also be accelerated by these solutions

IP over InfiniBand (IBoIP)

- Kernel driver, exposing InfiniBand devices as standard network interfaces
- Transparently usable by applications
- Uses the kernel's network stack (Context switching, CPU resources)

Related Work

Alternative solutions accelerate traditional sockets $\rightarrow$ NIO relies on sockets and can also be accelerated by these solutions

IP over InfiniBand (IBoIP)

- Kernel driver, exposing InfiniBand devices as standard network interfaces
- Transparently usable by applications
- Uses the kernel's network stack (Context switching, CPU resources)

libvma

- Open source library, developed by Mellanox
- Preloaded to socket-based applications (LD_PRELOAD)
- Full kernel bypass using native ibverbs
- Requires elevated privileges (CAP_NET_RAW or root)

Related Work

Java Sockets over RDMA (JSOR)

- Developed by IBM $\rightarrow$ Only available in proprietary J9 JVM
- Full kernel bypass via RDMA
- Has shown promising results, but has problems in multi-threaded applications and stuck connections
- Not supported anymore in IBM SDK 11 (uses OpenJ9)

Related Work

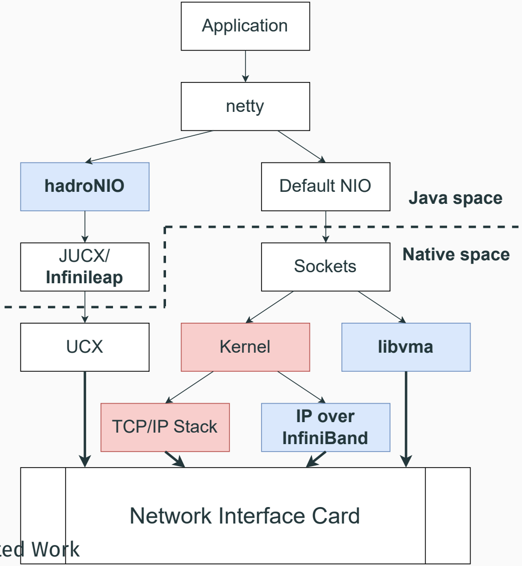
Java Sockets over RDMA (JSOR)

- Developed by IBM $\rightarrow$ Only available in proprietary J9 JVM
- Full kernel bypass via RDMA
- Has shown promising results, but has problems in multi-threaded applications and stuck connections
- Not supported anymore in IBM SDK 11 (uses OpenJ9)

Sockets Direct Protocol (SDP)

- Full kernel bypass via RDMA
- Included in OFED and introduced into the JDK with Java 7
- Support has officially ended (not included in OFED since version 3.5)

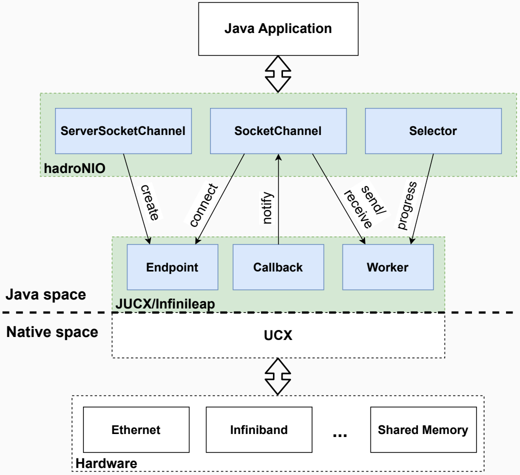
Related Work



- Many (successful) attempts in the past, but only IPoIB and libvma are still actively supported
- libvma is the only socket-based solution offering kernel bypass but requires **elevated rights** and can be **complex to configure**
- hadroNIO offers kernel bypass via UCX and works completely in **user space** (no special privileges needed)

Implementation

Architecture



- UCX is written in C/C++, but provides **JUCX**, a Java binding via JNI
- **Endpoint** abstracts one destination of a connection → Connects to a remote Endpoint
- **Worker** can represent multiple network resources with their *Progress Engine*
- `Worker.progress()` needs to be called for send/receive requests to be finished (→ Callback)

Interfacing between NIO and JUCX

Problem:

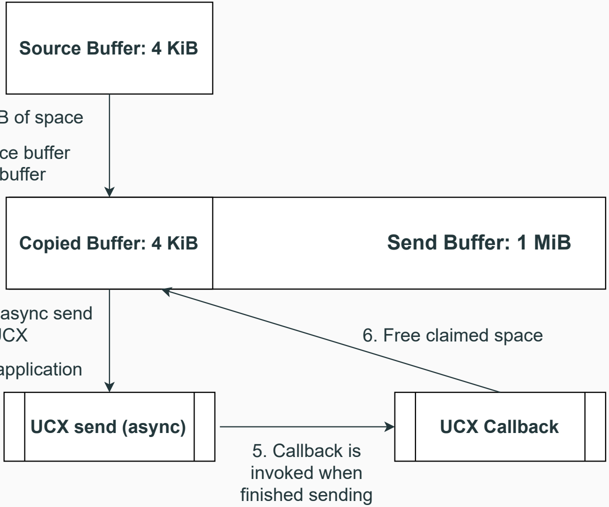
- UCX communication is asynchronous $\rightarrow$ Buffers may not be altered by the application, while a read/write is in progress
- After a call to `SocketChannel.write(ByteBuffer buffer)`, the buffer may be altered by the application

Solution: Use an intermediate buffer

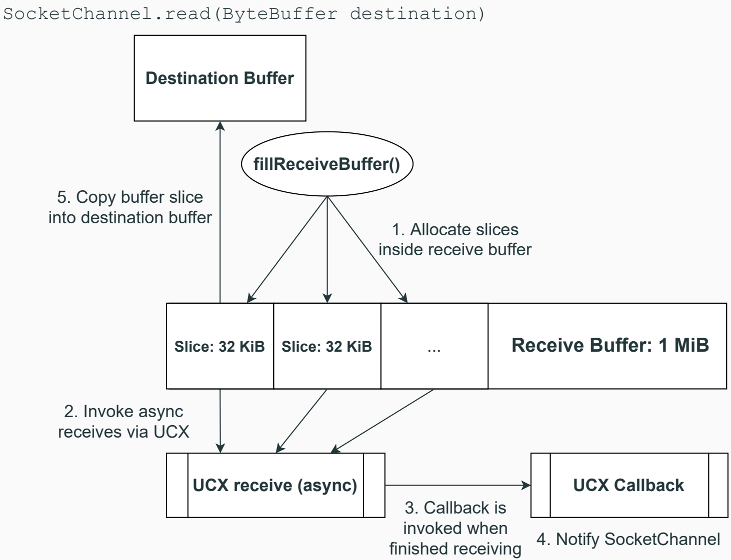
- `SocketChannel.write(ByteBuffer buffer)` copies the the data into the intermediate buffer
- UCX read/write methods only work on the intermediate buffer

Send

```
SocketChannel.write(ByteBuffer source)
```



Receive


SocketChannel.read(ByteBuffer destination)
Destination Buffer
fillReceiveBuffer()
5. Copy buffer slice into destination buffer
1. Allocate slices inside receive buffer
Slice: 32 KiB
Slice: 32 KiB
...
Receive Buffer: 1 MiB
2. Invoke async receives via UCX
UCX receive (async)
UCX Callback
3. Callback is invoked when finished receiving
4. Notify SocketChannel

Busy Polling vs. Epoll

- Busy polling UCX workers offers best performance with `thread count` $<=$ `CPU count`, but does not scale well
- Selector can use `epoll()` to let thread sleep while no event is incoming
- Only using `epoll()` causes a high increase in latency
- **Solution:** Use busy polling for a short time (e.g 20 µs) and fallback to `epoll()` if no event happens

Selector can be configured to use 1 of 3 modes: **Busy Polling**, **Epoll** and **Dynamic**

Current project state

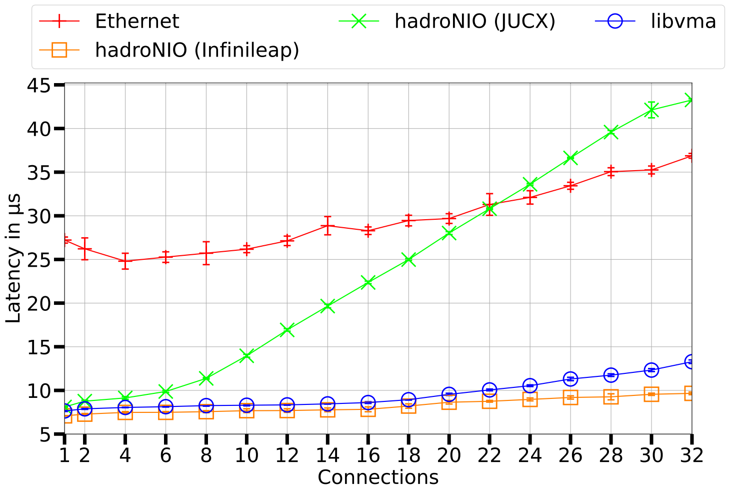
- ☑ Accelerating netty works
- ☑ Accelerating gRPC works
- ☑ Accelerating Apache Ratis works
- ☑ Busy Polling & Epoll support
- ☑ Works with JUCX and Infinileap

Evaluation

Cluster setup (OCI)

CPU	2x Intel Xeon Gold 6154 CPU (3.00GHz, 18 Cores/36 Threads)
RAM	384 GiB
NIC	Mellanox MT28800 (ConnectX-5) 100 Gbit/s Ethernet
OS	Oracle Linux 8
OFED	MLNX 5.4
Java	OpenJDK 19.0.1
UCX	1.13.1
libvma	9.8.1

Latency (Netty RTT with 16-byte messages - Average values)

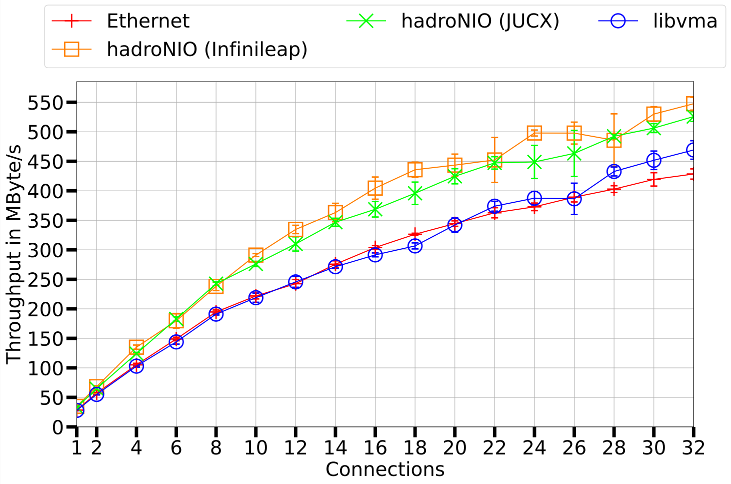


- Ethernet cannot reach below 25 µs
- hadroNIO (Infinileap) 0.5 µs **faster than libvma** with few connections → gap grows for many connections
- hadroNIO (JUCX) starts well, but latency increases fast with rising connection count → ends slower than Ethernet

JUCX vs. Infinileap

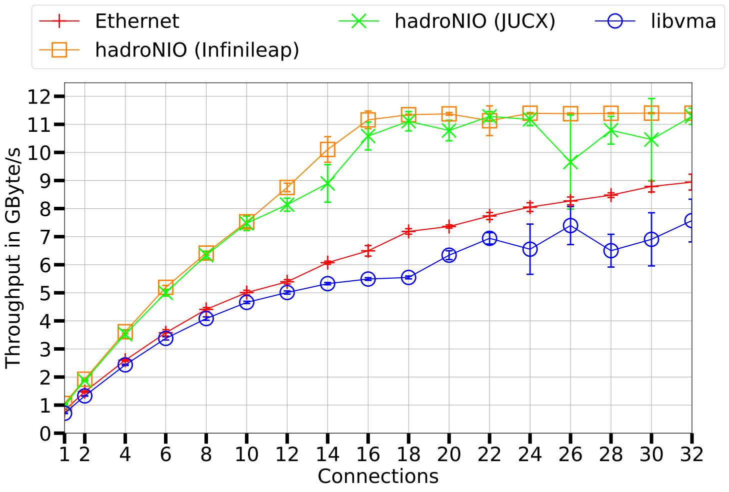
- Why does Infinleap scale so much better?
  - Minimal time spent in native code
  - No allocation of objects or GlobalRefs in native code
  - Less upcalls
    - UCX may process small messages directly (blocking), without calling a callback
    - JUCX calls the callback manually in these cases
  - JNI vs FFI performance differences

Throughput (Netty with 16-byte messages)



- **hadroNIO yields best throughput** → performs well with Infinileap and JUCX
- libvma does not offer a huge advantage over Ethernet

Throughput (Netty with 1-KiB messages)



- Only hadroNIO can saturate the hardware
- Infinileap offers more stable performance than JUCX
- libvma performs **worse than Ethernet**

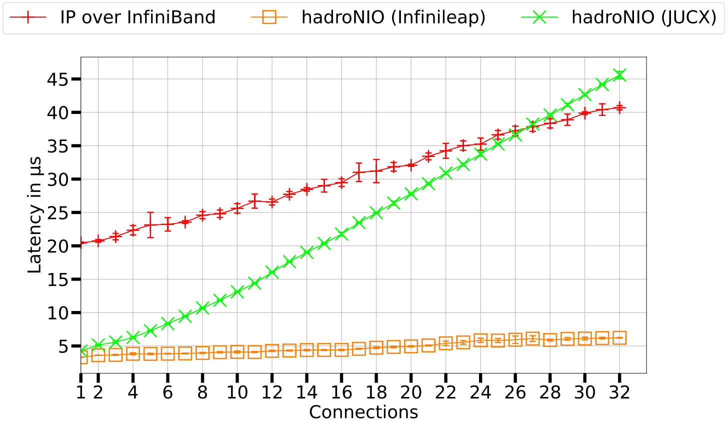
Evaluation

Cluster setup (HHU)

CPU	Intel Xeon Silver 4216 (2.10GHz, 16 Cores/32 Threads)
RAM	64 GiB
NIC	Mellanox MT27800 (ConnectX-5) 100 Gbit/s InfiniBand
OS	CentOS Stream 8
rdma-core	42.0
Java	OpenJDK 19.0.1
UCX	1.13.1

Back-to-back connection (no switch)

Latency (Netty RTT with 16-byte messages - Average values)



- Minimum latency: 3.2 µs
- hadroNIO (Infinileap) offers less than 5 µs RTT with **up to 20 connections**

Conclusion & Future Work

Conclusion & Future Work

- hadroNIO accelerates NIO completely in user space
- Offers better latency and throughput than libvma
- 100 GBit/s hardware can be saturated by NIO applications
- Infinileap scales much better than JUCX

Future Work:

- Scalability tests in OCI (Epoll overhead?)
- Benchmarks with applications and libraries based on NIO
- Successful tests have been done with **gRPC** and **Apache Ratis**

Infinileap and hadroNIO are sponsored by Oracle and supported by Oracle Cloud credits provided by the Oracle for Research program.