Heinrich Heine Universität Düsseldorf

Transparent high-speed networking for low latency data center Java applications

Inaugural-Dissertation

Zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Fabian Ruhland

geboren in
Mettmann

Düsseldorf, 16.01.2024

aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Berichterstatter:

1. Prof. Dr. Michael Schöttner

2. Prof. Dr. Stefan Conrad

Tag der mündlichen Prüfung: 18.04.2024

Abstract

Many big-data frameworks are written in Java, ranging from disk-based processing systems like Hadoop MapReduce to in-memory solutions like Apache Spark. The scalability of any distributed cloud system is limited by the network speed. As a result, high-speed networks, like InfiniBand, have become dominant in data centers, providing throughput rates of up to 400 GBit/s and round-trip latency of less than 2 µs.

While it is possible to use InfiniBand and High-Speed Ethernet over traditional Java sockets, this approach is not very efficient. Several projects addressed this problem in the past with either transparent or non-transparent solutions for Java applications. Transparent network acceleration libraries redirect data sent via traditional sockets over high-speed networks, while non-transparent libraries offer an alternative, but more direct programming interface to the network hardware. Naturally, the best performance can be achieved by directly programming the hardware, but there are many pitfalls and often performance may suffer from poor programming. Furthermore existing applications need to adapt their (often complex) network code in order to integrate non-transparent libraries.

This thesis proposes *hadroNIO*, a novel transparent approach, accelerating Java NIO sockets by replacing the Java NIO library. The architecture of hadroNIO is designed for highly concurrent applications using asynchronous communication. On the lower level, hadroNIO relies on the *Unified Communication X* (UCX) framework, which provides a unified networking API and supports several transports (e.g. InfiniBand or Ethernet), making hadroNIO network agnostic.

hadroNIO has been evaluated using microbenchmarks and real-world applications on different hardware setups with 100 GBit/s high-speed networks and compared with *libvma*, a highly efficient socket acceleration library developed by Mellanox, as well as traditional Java sockets. First evaluations have shown, that the minimum achievable latency using blocking NIO communication is around 3 µs, imposing only half a microsecond of overhead compared to directly working with UCX in Java. Since then, hadroNIO has been heavily optimized, leading to even less overhead.
Later measurement results with real-world frameworks and applications show, that full end-to-end round-trip times of 5 µs can be achieved using asynchronous communication with the popular event-driven networking framework *netty* (used by many big-data frameworks). Experiments with *gRPC* show a throughput increase of more than 50% over classic sockets, while request latencies for *Apache ZooKeeper* have been reduced by 50%. Connection scalability experiments show, that hadroNIO can efficiently handle hundreds of connections. A microbenchmark based on netty demonstrates, that it is able to provide average round-trip times of less than 10 µs with 80 connections all sending/receiving messages in parallel. hadroNIO outperforms libvma in each benchmark and works reliably in scenarios where libvma shows stability issues and fails to finish benchmark runs.

Contents

1 Introduction **1**
- 1.1 Motivation and Background . . . 1
  - 1.1.1 Context . . . 1
  - 1.1.2 Infiniband & Java . . . 2
- 1.2 Research Contributions . . . 3
  - 1.2.1 Publications . . . 3
  - 1.2.2 Software . . . 4
- 1.3 Project Hermes . . . 6
- 1.4 Thesis Structure . . . 7

2 Observatory benchmark **8**
- 2.1 A unified InfiniBand benchmark . . . 8
- 2.2 InfiniBand monitoring . . . 9
- 2.3 Neutrino . . . 9
- 2.4 Contributions . . . 10

3 hadroNIO - Transparent Java network acceleration **39**
- 3.1 Java NIO and Netty overview . . . 39
- 3.2 OpenUCX overview . . . 41
- 3.3 Transparency challenges . . . 43
- 3.4 Contributions . . . 44

4 Accelerating big-data applications with hadroNIO **61**
- 4.1 Optimizations . . . 61
- 4.2 Contributions . . . 64

5 Conclusions and Outlook **81**
- 5.1 Achievements . . . 81
- 5.2 Future Work . . . 82

Bibliography **86**

Chapter 1

Introduction

1.1 Motivation and Background

1.1.1 Context

Digital networks have become such an integral part of our life and daily routines, that it is hard to imagine a world without the degree of connectivity we have today. The applications seem almost endless, with new technologies being innovated each year. Some are obvious, like computers and smartphones, that we use to surf the web, connect with other people around the world, track our health condition or manage our finances. Others integrate seamlessly into everyday life, so that we do not notice them most of the time. Examples for that are smart home appliances, which can regulate inside temperatures, automatically close the blinds or even order food, when the fridge is empty.

Most of these applications, whether it being large scale social networks, or something as simple as a smart light switch, need some form of data processing. This often happens in cloud data centers, hosting hundreds or thousand of servers. Of course, these are not only used to perform mundane tasks, simplifying our everyday life, but also to perform big data computing assignments, like genome sequencing and simulating the impact of certain molecules on life forms to develop new medicine.

To leverage the full potential of these compute servers, they are connected via high-speed interconnects, such as InfiniBand [1] or High-Speed Ethernet, allowing them to exchange data at rates of up to 400 GBit/s with latencies as low as 1-2 µs. This is achieved by moving the network protocol stack onto the network interface device (NIC) itself, instead of letting the operating system's kernel handle the network traffic. While it is possible to use the traditional socket API with High-Speed Ethernet, and a Linux kernel module achieves the same for InfiniBand, this involves using the kernel's network stack, requiring context switches which introduce a large overhead. To leverage the full potential of modern high-speed NICs, the *ibverbs* library must be used. However, the programming model, introduced by this library, differs vastly from sockets and introduces many challenges and pitfalls.

Simultaneously, Java has become one of the most popular programming languages for big-data frameworks and applications, with projects like Apache Spark [2], Apache Flink

[3], or Apache Cassandra [4] being implemented in Java. Low network latency and high throughput is crucial for these types of applications, since distributed instances need to communicate with each other as fast as possible. A slow network can easily become a bottleneck, drastically limiting scalability. While modern interconnects like InfiniBand offer the type of low latency communication desired in such scenarios, Java has no official support for ibverbs. However, there have been several attempts at providing InfiniBand support for Java. These can generally be divided into non-transparent libraries, which port the ibverbs interface to Java, and transparent solutions, which redirect socket traffic over high-speed interconnects, allowing programmers to profit from modern networking hardware, while still using the traditional socket API.

The main focus of this thesis is on developing a new transparent acceleration library for Java, outperforming existing solutions, while not requiring changes to the applications. This new approach offers both low latency and high scalability regarding hundreds of connections. Furthermore, it supports multiple modern interconnects.

1.1.2 Infiniband & Java

While transparency generally comes at the cost of some performance, using a non-transparent InfiniBand library for Java forces programmers to learn the ibverbs API. Implementing a scalable network system based on ibverbs requires a lot of experience [5]. On the other hand, the JDK already comes with its own scalable network library, called *Java NIO*. This library offers an asynchronous network API, based on classic sockets. Traditionally, send and receive operations on sockets are blocking, meaning that the current thread cannot do any other work, until the operation has finished. To support multiple connections, each connection has to be managed by its own thread. Java NIO introduces non-blocking operations, allowing threads to perform other tasks, while waiting for a message to arrive, for example. Furthermore, a single thread can handle an arbitrary amount of connections, giving the programmer control over how many network threads should run. This allows for scalable applications, that do not overwhelm the CPU with hundreds of threads, each handling only a single connection.

Today Java NIO is the de-facto standard for network programming in Java, partly because of the popularity of *Netty*, an asynchronous event-driven network application framework [6]. While Java NIO solves the problem of blocking socket connections, it also introduces some complexity. For example, network buffers are not guaranteed to be processed fully at once and may require multiple send operations to be transferred to their receivers. Additionally, as with blocking sockets thread management is still left to the programmer, and buffer allocation also has to be done manually, which may cause performance penalties when done wrong (i.e. repeated allocations of small buffers). The Netty library solves these problems by automatically starting enough threads to saturate, but not overwhelm, the CPU and introducing a sophisticated buffer management, based on reference counting. Many modern distributed Java applications and frameworks are based on Netty, for example Apache BookKeeper [7] and gRPC [8]. Another example is Apache ZooKeeper, a highly reliable key-value store, used to coordinate distributed cloud services [9]. It is based on Java NIO directly by default, but can also be configured to use an experimental Netty-based networking subsystem.

Java NIO, and thus Netty, still have one major drawback: With NIO being based on traditional sockets, all network operations are still done via the kernel's network stack, rendering it unable to take full advantage of modern low latency networking hardware. The solutions are transparent acceleration libraries, intercepting socket traffic and completely bypassing the kernel. However, most of these solutions are not being maintained anymore and some of them come with unattractive caveats like being closed-source or requiring root privileges. In this thesis, a new approach at transparent network acceleration for Java applications has been developed. While existing libraries target traditional sockets, offering high compatibility with native and Java applications, our approach is specifically tailored towards accelerating Java NIO. This allows us to focus on non-blocking asynchronous communication and omit some of the legacies coming from the traditional socket interface, like working with raw `byte` arrays instead of more modern `ByteBuffer` instances.

1.2 Research Contributions

The key contributions of this thesis are:

- An extensive overview of existing (transparent and non-transparent) InfiniBand solutions for Java applications.
- *Observatory*, an extensible benchmarking framework, designed to evaluate and compare different InfiniBand libraries for Java.
- Evaluation results of different high-speed networking solutions for Java using the Observatory benchmark.
- *hadroNIO*, a transparent acceleration library for Java applications, based on the popular NIO API.
- Evaluation results of hadroNIO, using a set of tests comprised of a synthetic microbenchmark and real-world applications, showing that hadroNIO outperforms existing solutions in synthetic and real-world loads.

1.2.1 Publications

All publications are full papers of 8 pages each.

International Conferences

- **Fabian Ruhland**, Filip Krakowski, Michael Schöttner. *Performance analysis and evaluation of Java-based InfiniBand Solutions.* In Proceedings of the 19th IEEE International Symposium on Parallel and Distributed Computing (ISPDC). 2020.
- Filip Krakowski, **Fabian Ruhland**, Michael Schöttner. *Neutrino: Efficient InfiniBand Access for Java Applications.* In Proceedings of the 19th IEEE International Symposium on Parallel and Distributed Computing (ISPDC). 2020.

- **Fabian Ruhland**, Filip Krakowski, Michael Schöttner. *hadroNIO: Accelerating Java NIO via UCX*. In Proceedings of the 20th IEEE International Symposium on Parallel and Distributed Computing (ISPDC). 2021.
- Filip Krakowski, **Fabian Ruhland**, Michael Schöttner. *Infinileap: Modern High-Performance Networking for Distributed Java Applications*. In Proceedings of the 27th IEEE International Conference on Parallel and Distributed Systems (ICPADS). 2021.
- **Fabian Ruhland**, Filip Krakowski, Michael Schöttner. *Transparent network acceleration for big data computing in Java*. In Proceedings of the 26th IEEE International Conference on Computational Science and Engineering (CSE). 2023.

Technical Reports

- Stefan Nothaas, **Fabian Ruhland** Michael Schöttner. *A Benchmark to Evaluate InfiniBand Solutions for Java Applications*. Published on arXiv e-prints. arXiv:1910.02245. 2019.
- **Fabian Ruhland**, Filip Krakowski, Michael Schöttner. *Accelerating netty-based applications through transparent InfiniBand support*. Published on arXiv e-prints. arXiv:2209.14048. 2022.

1.2.2 Software

Over the course of this thesis, the author was involved in developing the following libraries and applications:

Observatory is a novel benchmarking framework, designed to compare different network libraries for Java. It specifically targets InfiniBand libraries with support for RDMA, but also contains an implementation using traditional sockets to analyse transparent acceleration solutions. The benchmark supports uni- and bidirectional connections and uses a blocking communication pattern to evaluate the lowest possible latencies. To compare InfiniBand performance of Java and native applications, the framework has additionally been implemented in C++, with a binding based on the *ibverbs* library providing baseline results. The project is open-source and available on GitHub [10].
Contributors (in chronological order): Fabian Ruhland (main development), Peter Rudenko (JUCX benchmark).
Size and language(s): ~11000 lines of code, Java and C++.

hadroNIO is a Java library, reimplementing parts of Java NIO. It replaces the default implementation, based on traditional sockets with a new one, specifically designed to route network traffic over high-speed networks using the *Unified Communication X* (UCX) framework [11]. In contrast to classic socket implementations, the kernel is not involved in any parts of the communication, fully alleviating the need for context switches, if an appropriate NIC is available. The project is open-source and available on GitHub [12].
Contributors (in chronological order): Fabian Ruhland (main development), Edwin Stang (fixed two possible null pointer exceptions).
Size and language(s): ~7000 lines of code, Java.

neutrino is a new Java library, leveraging the Java Native Interface to expose native *ibverbs* functionality to Java applications. Its `core` package provides wrapper classes for native structs and functions, as well as methods to manipulate native buffers. The project is open-source and available on GitHub [13].
Contributors (in chronological order): Filip Krakowski (main development), Fabian Ruhland (worked on `core` package, bringing native functionality to Java).
Size and language(s): ∼11000 lines of code (`core` package), Java and C++.

Infinileap is a new Java library, providing efficient access to the native UCX framework and thus allows leveraging high-speed networks with Java. It is based on the new Foreign Function & Memory API (Project Panama) [14], which are developed to supersede the Java Native Interface. Like neutrino, it has a `core` package, consisting of wrapper classes around native structs and functions. The project is open-source and available on GitHub [15].
Contributors (in chronological order): Filip Krakowski (main development), Fabian Ruhland (bugfixes and improvements to `core` package).
Size and language(s): ∼8400 lines of code (`core` package), Java.

Detector is a native library, offering a simple API to read statistics from InfiniBand devices. It uses the libraries *libibmad* and *libibnetdisc* to discover all devices in a network and read their performance counters. This way, the full data size, including protocol overhead, can be measured on a per-device basis. Furthermore, it is possible to read error counters, indicating problems with specific NICs or the network setup. The project is open-source and available on GitHub [16].
Contributors (in chronological order): Fabian Ruhland.
Size and language(s): ∼1300 lines of code, C++.

jDetector is Java library, using the Java Native Interface to call native code from *Detector* and expose its functionality to Java applications. The project is open-source and available on GitHub [17].
Contributors (in chronological order): Fabian Ruhland.
Size and language(s): ∼1400 lines of code, Java and C++.

ib-scanner is a terminal frontend for *Detector*, based on ncurses [18]. It provides a simple UI, which can monitor up to 4 network devices at once and is fully controllable via the keyboard. The project is open-source and available on GitHub [19]. Contributors (in chronological order): Fabian Ruhland.
Size and language(s): ∼1500 lines of code, C++.

1.3 Project Hermes

Project Hermes is a research project, sponsored by *Oracle*, which aims at providing a network agnostic ultra-fast communication solution for Java. While existing solutions for high-speed networking in Java are either transparent or require using their own API, Project Hermes targets both, by incorporating two separate libraries, aiming to work in conjunction ultimately.

The aforementioned library *Infinileap* provides fast access to the native UCX framework [11], which offers network-agnostic communication, supporting high-speed networks such as InfiniBand. To access native functionality from Java, the new Foreign Function & Memory APIs [14] are used.

The second project, *hadroNIO*, aims at transparently accelerating Java NIO traffic over high-speed networks by using UCX. Development on hadroNIO started on the basis of JUCX, the official Java binding for UCX, which is based on the traditional Java Native Interface, which is soon to be replaced by the new the new Foreign Function & Memory APIs. This allowed Infinileap and hadroNIO to be developed in parallel and replace JUCX with Infinileap later on.

Ultimately, Project Hermes has been evaluated on high performance bare-metal nodes in the Oracle Cloud Infrastructure, connected via 100 GBit/s High-Speed Ethernet. Access to these resources has been granted by the *Oracle for Research* program.

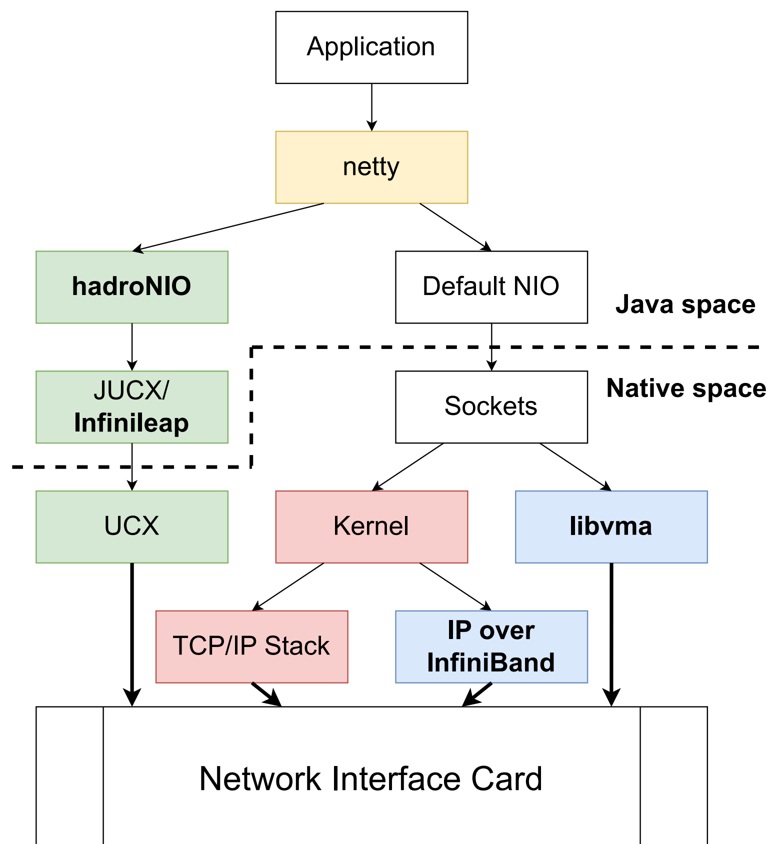

Figure 1.1: Project Hermes architecture compared to existing solutions

1.4 Thesis Structure

In Chapter 1 the context and motivation of the research behind this thesis are presented. Chapter 2 introduces the *Observatory* benchmark, elaborates on existing transparent and non-transparent network acceleration solutions for Java and discusses the benchmark results of the different libraries. A novel approach at redirecting Java NIO traffic over high-speed networks, called *hadroNIO*, is presented in Chapter 3. Furthermore, benchmark results using microbenchmarks based either directly on Java NIO or Netty are shown and discussed. This constitutes the main research contribution of this thesis together with Chapter 4, which presents optimizations to hadroNIO and results from real world scenarios. A lot of effort has been put into optimizing hadroNIO for applications using hundreds of connections, as well as keeping round-trip times as low as possible. The thesis is concluded by Chapter 5, which also presents ideas for future work.

Chapter 2

Observatory benchmark

This chapter describes *Observatory*, a novel benchmarking framework to evaluate different InfiniBand libraries for Java. Furthermore, it provides an overview of the benchmarked libraries and elaborates on monitoring InfiniBand traffic for overhead calculations.

2.1 A unified InfiniBand benchmark

Observatory was developed as a successor to the *Java InfiniBand Benchmark (JIB)*, a suite consisting of several standalone microbenchmarks [20]. It contains a socket-based benchmark to evaluate transparent acceleration solutions, as well as one benchmark for each of the non-transparent libraries. The JIB suite supports the native ibverbs library through a microbenchmark written in C, as well as *jVerbs* by IBM [21], which accesses libibverbs via JNI. While the Java InfiniBand Benchmark was successfully used to analyse InfiniBand libraries, it was hard to expand with support for new InfiniBand libraries, since each new non-transparent solution requires its own standalone application in the JIB suite. Additionally, there is a large portion of duplicated code between all microbenchmarks.

To this end, the Observatory benchmark was developed, aiming to unify the JIB's standalone applications. Furthermore, Observatory simplifies the benchmark configuration and result evaluation. The JIB applications must be configured via console arguments and can only perform one benchmark run per invocation. A complicated bash-script is used to perform a set of benchmark runs and gather results inside a CSV-file. In contrast, Observatory is able to perform a set of benchmark runs, described by a JSON-file, per invocation and appends results consecutively to a CSV-file, omitting the need for a bash-script.

Observatory defines a clear interface for network libraries, requiring only communication connection setup routines to be implemented. The process of benchmarking and gathering results is controlled by the framework, in contrast to the JIB suite, where each benchmark application measures results by itself. This heavily simplifies the development of extending Observatory with support for new InfiniBand solutions. As a result, not only jVerbs is supported, but also its open-source counterpart the *Direct Storage and Networking Interface (DiSNI)* [22]. Furthermore, Observatory also includes a binding for *neutrino*, a JNI-based library aiming at fast access to native ibverbs, while bypassing the performance problems that existing solutions are showing [23].

2.2 InfiniBand monitoring

Both the JIB suite and Observatory support calculating the data overhead, produced by different InfiniBand libraries. This is achieved by reading the performance counters from the used network cards. Each InfiniBand NIC counts outgoing and incoming packets, bytes and other statistics like different types of errors. These can be read via filesystem nodes for local devices, or queried from the subnet manager to get statistics from remote devices. To read these counters, the author implemented the C++-library *Detector*, which supports both ways of reading the performance counters. A Java binding called *jDetector* has been developed as a separate project, allowing to access Detector functionality via JNI.

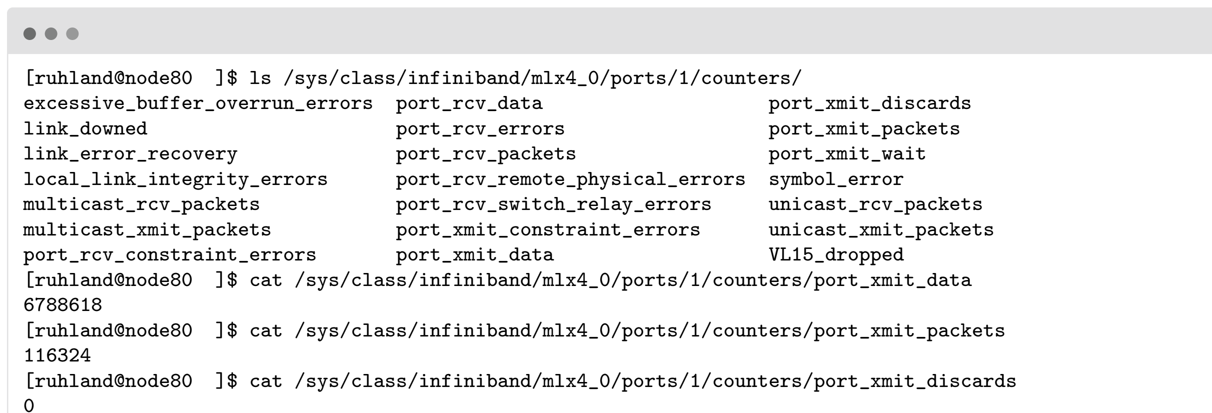

Figure 2.1: InfiniBand performance counters accessed via `sysfs`

Furthermore, the author has developed a monitoring tool based on Detector. This application, called *ib-scanner*, scans the network for compatible devices and periodically queries their performance counters. Each device's port can be monitored separately and up to 4 devices/ports may be displayed simultaneously in a terminal-based UI. While ib-scanner is not mentioned in any of the author's publications, it has proven to be a valuable tool for debugging and error detection during development.

2.3 Neutrino

Neutrino is a non-transparent InfiniBand library for Java, developed at the operating systems department at Heinrich Heine University Düsseldorf. The in initial goal for Neutrino was to provide access to native ibverbs functionality via a thin JNI layer. While similar solutions, like jVerbs [21] and its open-source successor DiSNI [22] by IBM exist, they both suffer from performance problems in certain scenarios, as shown by the JIB suite and Observatory [20] [24]. Furthermore, performing calls to native functions with these libraries requires serializing parameters into native structs, adding an overhead to network operations. They try to alleviate this by implementing *Stateful Verbs Methods (SVM)*, which cache the serialized version of a native function call, eliminating the need for subsequent serialization when the same parameters are used multiple times (e.g. the same buffer is used for sending/receiving messages). However, theses stateful objects are not flexible and calls with different parameters require serialization. Real world applications

would probably only be able to reuse SVM objects for a part of their network communication, but not all of it. Fig 2.2 shows, that not using SVM at all adds an additional microsecond of latency to each RDMA write operation on average, compared to using a single SVM for all operations. While this might not sound like a lot, it means a 50% latency increase for small payloads. Furthermore, only a single connection was used for this measurement. The overhead may grow with an increasing amount of connections and threads, putting more burden on the garbage collector. The measurement was performed as part of the following paper about Observatory and in the same environment as all the other measurements, but not included due to space constraints.

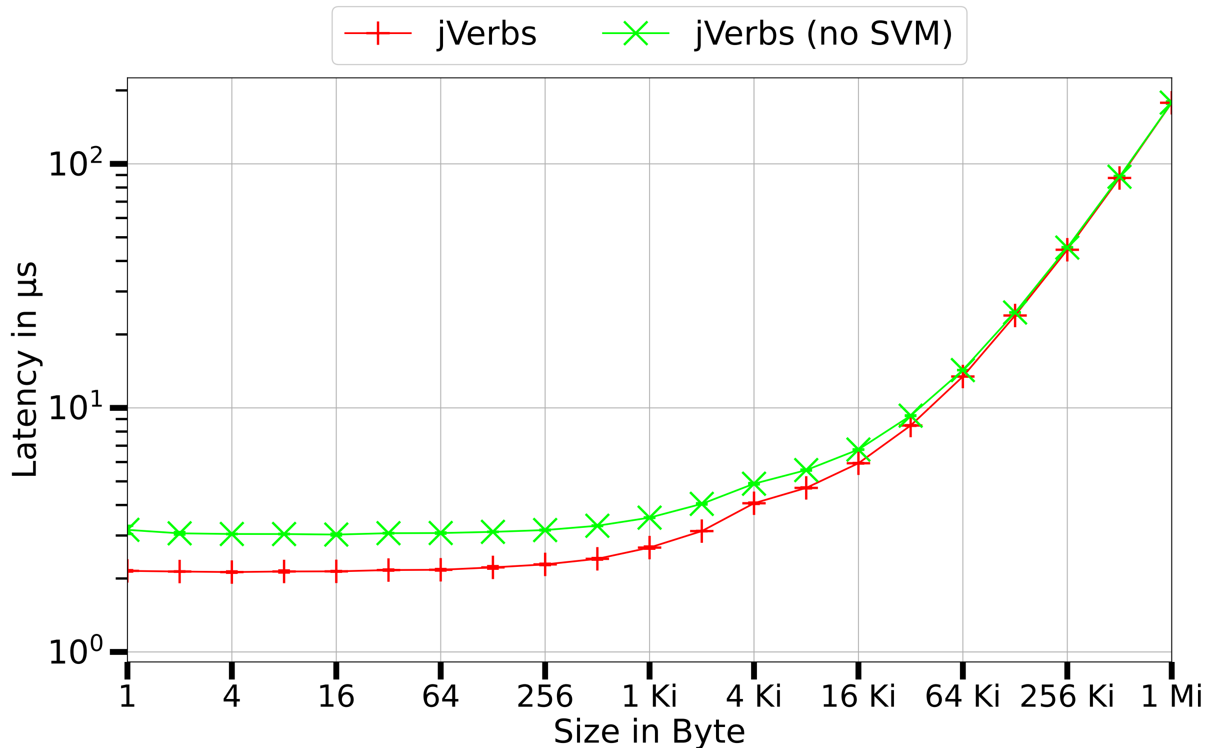

Figure 2.2: RDMA write average latency with jVerbs, measuring the overhead of not using Stateful Verbs Methods

Neutrino uses a different approach, that does not mirror native struct state in Java objects, and serializes it back to native space when needed. Instead, Neutrino leverages the Unsafe API, which allows working with off-heap memory, to manipulate native structs. Neutrino offers getter- and setter-methods on each object, that directly influence the underlying ibverbs structure. This way, serialization is not necessary when calling a native function, because all objects used as parameters already have the required state in native space.

2.4 Contributions

The author has first developed throughput benchmarks and automation/evaluation scripts for the JIB suite, supporting native ibverbs, IBM jVerbs and transparent acceleration libraries via a socket benchmark. The idea for an InfiniBand monitoring tool came from Stefan Nothaas, but the author was fully responsible for developing Detector, jDetector

and ib-scanner and integrated data overhead measurement into the JIB suite by himself. The JIB applications have then been enhanced with latency measurements, supporting percentile calculation, by Stefan Nothaas. The corresponding paper was written by Stefan Nothaas, while the author and Michael Schöttner reviewed the paper and provided feedback in several discussions.

Since Observatory is a successor to the JIB suite, its round-trip latency benchmark implementation is based on the latency benchmark by Stefan Nothaas, especially the percentile calculation. The benchmarking process itself works fine in the JIB applications, but the challenge was to create a unified, but extensible, benchmarking framework. To this end, the Observatory architecture was designed and implemented. The main contributions of this publication are the benchmark design and implementation in Java and C++, as well as providing an overview of existing InfiniBand solutions for Java and evaluating them using Observatory. It was written by the author of this thesis, while Filip Krakowski and Michael Schöttner took part in many valuable discussions about the design and implementation of Observatory.

Regarding Neutrino, the author was involved in developing the project's core functionality, which is offering the ibverbs API as a Java library. This was done in cooperation with Filip Krakowski, who is the main developer of Neutrino. Once the main part of the native functionality was available in the project and it was stable enough for use in larger projects, the author went on to develop Observatory and compare Neutrino to other InfiniBand libraries for Java. Filip Krakowski continued his work on Neutrino, aiming to build a higher-level networking framework around it. The associated paper was written by Filip Krakowski, while the author and Michael Schöttner were in involved in the form of proofreading and various discussions.

A Benchmark to Evaluate InfiniBand Solutions for Java Applications

Stefan Nothaas
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
stefan.nothaas@hhu.de

Fabian Ruhland
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
fabian.ruhland@hhu.de

Michael Schoettner
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
michael.schoettner@hhu.de

arXiv:1910.02245v1 [cs.NI] 5 Oct 2019

***Abstract*—Low-latency network interconnects, such as InfiniBand, are commonly used in HPC centers and are even accessible with todays cloud providers offering equipped instances for rent. Most big data applications and frameworks are written in Java. But, the JVM environment alone does not provide interfaces to directly utilize InfiniBand networks.**

In this paper, we present the "Java InfiniBand Benchmark" to evaluate the currently available (and supported) "low-level" solutions to utilize InfiniBand in Java. It evaluates socket- and verbs-based libraries using typical network microbenchmarks regarding throughput and latency. Furthermore, we present evaluation results of the solutions on two hardware configurations with 56 Gbit/s and 100 Gbit/s InfiniBand NICs. With transparency often traded for performance and vice versa, the benchmark helps developers with studying the pros and cons of each solution and support them in their decision which solution is more suitable for their existing or new use-case.

***Index Terms*—High-speed networks, Distributed computing**

I. Introduction

RDMA capable devices have been providing high throughput and low-latency to HPC applications for several years [18]. With todays cloud providers offering instances equipped with InfiniBand for rent, such hardware is available to a wider range of users without the high costs of buying and maintaining it [25]. Many application domains such as social networks [20], [29], [31], search engines [24], [36], simulations [37] or online data analytics [21], [41], [42] require large processing frameworks and backend storages. Many of these are written in Java, e.g. big data batch processing frameworks [28], [33], databases [1], [2] or backend storages/caches [3], [4], [7], [35].

These applications benefit from the rich environment Java offers including automatic garbage collection and multi-threading utilities. But, the choices for inter-node communication on distributed applications are limited to Ethernet-based socket-interfaces (standard ServerSocket or NIO) on the commonly used JVMs OpenJDK and Oracle. They do not provide support for low-latency InfiniBand hardware. But, there are external solutions available each with pros and cons.

This raises questions if a developer wants to chose a suitable solution for a new use-case or an existing application: What's the throughput/latency on small/large payload sizes? Is the performance sufficient when trading it for transparency requiring less to no changes to the existing code? Is it worth considering developing a custom solution based on the native API to gain maximum control with chances to harvest the performance available by the hardware?

In this paper, we address these questions by presenting a "Java InfiniBand (JIB) benchmark" to evaluate existing solutions to leverage the performance of InfiniBand hardware in Java applications. The modular benchmark currently provides implementations to evaluate three socket-based libraries and implementations, IP over InfiniBand, libvma and JSOR, as well as two verbs-based implementations, native C-verbs and jVerbs. This paper focuses on the fundamental performance metrics of low-level interfaces and *not* on higher-level network subsystems with connection management, complex pipelines and messaging primitives, e.g. MPI. We discuss and evaluate these in a separate publication [34]. We used our benchmark to evaluate the listed solutions on two hardware configurations with 56 Gbit/s and 100 Gbit/s InfiniBand NICs. The contributions of this paper are:

- An overview of existing Java InfiniBand solutions
- An extensible and open source benchmark to easily evaluate solutions to use InfiniBand in Java applications
- Extensive evaluation of existing Java libraries with 56 Gbit/s and 100 Gbit/s hardware

The remaining paper is structured as follows: Section II discusses related work with socket-based (§II-A) and verbs-based (§II-B) libraries. Section III presents the JIB Benchmark Suite which is used to evaluate two verbs-based solutions and three socket-based solutions in the following Section IV regarding overhead (§IV-A), uni-directional (§IV-B) and bi-directional (§IV-C) throughput, as well as one-sided latency (§IV-D) and full round-trip-time using a ping-pong benchmark (§IV-E). Conclusions are presented in Section V.

II. Related Work

This section elaborates on existing "low-level" solutions/libraries that can be used to leverage the performance of InfiniBand hardware in Java applications. This does not include network or messaging stacks/subsystems implementing higher-level primitives such as the Massage Passing Interface, e.g. Java-based FastMPJ [22] providing a special transport to use InfiniBand hardware. To the best of our knowledge, there is no benchmark available to evaluate InfiniBand solutions in Java.

A. Socket-based Libraries

The socket-based libraries redirect the send and receive traffic of socket-based applications transparently over InfiniBand host channel adapters (HCAs) with or without kernel bypass depending on the implementation. Thus, existing applications do not have to be altered to benefit from improved performance due to the lower latency hardware compared to commonly used Gigabit Ethernet. The following three libraries are still supported to date and evaluated in Section IV.

IP over InfiniBand (IPoIB) [27] is not a library but actually a kernel driver that exposes the InfiniBand device as a standard network interface (e.g. *ib0*) to the user space. Socket-based applications do not have to be modified but use the specific interface. However, the driver uses the kernel's network stack which requires context switching (kernel to user space) and CPU resources when handling data. Naturally, this solution trades performance for transparency.

libvma [10] is a library developed by Mellanox and included in their OFED software package [11]. It is pre-loaded to any socket-based application (using *LD_PRELOAD*). It enables full bypass of the kernel network-stack by redirecting all socket-traffic over InfiniBand using unreliable datagram with native C-verbs. Again, the existing application code does not have to be modified to benefit from increased performance.

Java Sockets over RDMA (JSOR) [40] redirects all socket-based data traffic in Java applications using native verbs, similar to libvma. It uses two paths for implementing transparent socket streams over RDMA devices. The "fast data path" uses native verbs to send and receive data and the "slow control path" manages RDMA connections. JSOR is developed by IBM on only available in their proprietary J9 JVM.

The following libraries are also known in literature but are not supported or maintained anymore.

The **Sockets Direct Protocol (SDP)** [23] redirects all socket-based traffic of Java applications over RDMA with kernel-bypass. It supported all available JDKs since Java 7 and was part of the OFED package until it was removed with OFED version 3.5 [12].

Java Fast Sockets (JFS) [39] is an optimized Java socket implementation for high speed interconnects. It avoids serialization of primitive data arrays and reduces buffering and buffer copying with shared memory communication as its main focus. However, JFS relies on SDP (deprecated) for using InfiniBand hardware.

Speedus [17] is a native library that optimizes data transfers for applications especially on intra-host and inter-container communication by bypassing the kernel's network stack. It is also advertised to support low-latency networking hardware for inter-node communication. But, the latest available version to date (2016-09-08) does not include such support.

B. Verb-based Libraries

Verbs are an abstract and low-level description of functionality for RDMA devices (e.g. InfiniBand) and how to program them. Verbs define the control and data paths including RDMA operations (write/read) as well as messaging (send/receive). RDMA operations allow reading or writing directly from/to the memory of the remote host without involving the CPU of the remote. Messaging follows a more traditional approach by providing a buffer with data to send and the remote providing a buffer to receive the data to.

The programming model differs heavily from traditional socket-based programming. Using different types of asynchronous queues (send, receive, completion) as communication endpoints. The application uses different types of work-requests for sending and receiving data. When handling data to transfer, all communication with the HCA is executed using these queues. The following libraries are verbs implementations that allow the user to program the RDMA capable hardware directly. The first two libraries presented are evaluated in Section IV.

C-verbs are the native verbs implementation included in the OFED package [13]. Using the Java Native Interface (JNI) [30], this library can be utilized in Java applications as well in order to create a custom network subsystem [22] [34]. Using the Unsafe class [32] or Java DirectByteBuffers, memory can be allocated off-heap to use it for sending and receiving data with InfiniBand hardware (buffers must be registered with a protection domain which pins the physical memory).

jVerbs [38] are a proprietary verbs implementation for Java developed by IBM for their J9 JVM. Using a JNI layer, the OFED C-verbs implementation is accessed. "Stateful verb methods" (*StatefulVerbsMethod* Java objects) encapsulate the verb to call including all parameters with parameter serialization to native space. Once the object is prepared, it can be executed which actually calls the native verb. These objects can be re-used for further calls with the same parameters to avoid repeated serialization to native space and creating new objects which would burden garbage collection.

Jdib [26] is a library wrapping native C-verbs function calls and exposing them to Java using a JNI layer. According to the authors, various methods, e.g. queue pair data exchange on connection setup, are abstracted to create an easier to use API for Java programmers. The fundamental operations to create protection domains, create and setup queue pairs, as well as posting data-to-send to queues and polling the completion queue seem to wrap the native verbs and do not introduce additional mechanisms like jVerbs's stateful verb calls. We were not able to obtain a copy of the library for evaluation.

III. A Benchmark for Evaluating InfiniBand Libraries for Java

To evaluate and compare the different libraries available, a common set of benchmarks had to be implemented for two programming languages (C and Java) and two programming models (sockets and verbs). Existing solutions like the iperf [8] tools for TCP/UDP or the ibperf tools included in the OFED package [13] do not cover all libraries we want to evaluate and do not implement all necessary benchmark types.

In this paper, we want to evaluate most of the available and still maintained libraries (§II) in a fundamental point-to-point setup regarding throughput and latency. Like other benchmarks

(e.g. OSU [14]), we want to determine the maximum throughput on uni-directional and bi-directional communication (e.g. application pattern asynchronous "messaging"), as well as one-sided latency and full round-trip-time (RTT) with a ping-pong communication pattern (e.g. application pattern "request-response"). These benchmarks allow us to determine the fundamental performance of the presented solutions and are commonly used to evaluate network hardware or applications [8], [13], [14]. The evaluation of higher-level primitives, e.g. collectives, and all-to-all communication is not possible with fundamental low-level interfaces. These require a higher-level networking stack with connection management and a complex pipeline which is not the focus of this paper.

The Java InfiniBand Benchmark (JIB) provides implementations of the listed benchmarks for two verbs-based solutions (C-verbs, jVerbs) and three socket-based solutions (IPoIB, libvma, JSOR). It is open source and available at Github [9]. Each benchmark run is configurable using command line parameters such as the benchmark type (uni-/ bi-directional, one-sided latency or ping-pong), the message size to send/receive and the number of messages to send/receive. For convenience, we refer to the payload size sent as messages independent of how it is transferred (e.g. sockets, verbs RDMA or verbs messaging). The context and all buffers are initialized before the benchmark is started. Afterwards, the current instance connects to the remote specified via command line parameters. Once the connection is established, a dedicated thread is started for sending data and another thread for receiving. Today, we can expect this to run on a multicore system with at least two physical cores to ensure that the send and receive thread can be run simultaneously to avoid blocking one another. The benchmark instance sends the specified number of messages to the remote and measures the time it takes to send the messages. Furthermore, we utilize the performance counters of the InfiniBand HCA to determine the overhead added by any software defined protocol which is especially relevant for the socket-based libraries (§IV-A).

For socket-based libraries, the benchmark is implemented in Java using TCP sockets with the ServerSocket, DataInputStream and DataOutputStream classes. Sending and receiving data is executed synchronously in a single loop on each thread. No further adjustments are required because all libraries redirect the normal send and receive calls of the socket libraries. With IPoIB, we use the address of the exposed *ib0* device. The libvma library is pre-loaded to the benchmark using *LD_PRELOAD*. In order to use JSOR, we run the benchmark in the J9-JVM and provide a configuration file specifying IP-addresses whose traffic is redirected over the RDMA device.

The verbs-based benchmarks are implemented in C and Java. Both implementations use RC queue pairs for RDMA and message operations. UD queue pairs can also be used for message operations but this option is currently not implemented. On RDMA operations, we did not include immediate data with a work request which would require a work completion on the remote (optional for signalling incoming RDMA operations on the remote). When sending RDMA operations to the HCA to determine the maximum throughput, we do not repeatedly add one work request to the send queue, then poll the completion queue waiting for that single work request to complete. This pattern is commonly used in examples [16] and even larger applications [15] but does not yield optimal throughput because the queue of the HCA runs empty very frequently. To keep the HCA busy, the send queue must be kept filled at all times. Thus, we fill up the send queue to the maximum size configured, first. Then, we poll the completion queue and once at least one completion is available and processed, we immediately fill the send queue again. Naturally, this pattern cannot be applied to the ping-pong latency benchmark executing a request-response pattern.

This data path is implemented in both, the C-verbs and jVerbs implementation. The C implementation uses the verbs implementation included in the OFED package and serves as a reference for comparing the maximum possible performance. To establish a remote connection, queue pair information is exchanged using a TCP socket. The jVerbs implementation has to implement the operations of the data path using the previously described stateful verbs methods. Thus, the sending of data on the throughput benchmark had to be altered slightly. A single stateful verb call for posting work requests to the send queue always posts 10 elements. Hence, new work requests are added to the send queue once at least 10 work completions were polled from the completion queue. We create all stateful verbs calls before the benchmark and re-use them to avoid performance penalties. On connection creation, queue pair information is exchanged with the API provided by jVerbs which is using the RDMA connection manager.

IV. EVALUATION

In this Section, we present the results of the evaluation of the socket-based libraries/implementations **IPoIB**, **libvma** and **JSOR** and the verbs-based libraries **C-verbs** and **jVerbs** using our benchmark suite (§III). We analyze and discuss basic performance metrics regarding throughput and latency using typical benchmarks with a two node setup with 56 Gbit/s and 100 Gbit/s interconnects. A summary of the benchmarks executed with each library/implementation is given in Table I. Due to space constraints, we limit the discussion of the results to selected conspicuities of the plotted data.

Library/Benchmark	OV	Uni-dir	Bi-dir	Lat	PingPong
C-verbs rdmaw		x	x	x	
C-verbs rdmar		x	x	x	
C-verbs msg	x	x	x	x	x
jVerbs rdmaw		x	x	x	
jVerbs rdmar		x	x	x	
jVerbs msg	x	x	x	x	x
IPoIB	x	x	x	x	x
JSOR	x	x	error	x	x
libvma	x	x	x	x	x

TABLE I: Overview of libraries evaluated with benchmarks available. Abbreviations: OV = Overhead, rdmaw = RDMA write, rdmar = RDMA read, msg = messaging verbs

We use the term “message” (msg) to refer to the unit of transfer which is equivalent to the data payload. The size of a message does refer to the payload size only and does not include any additional protocol or network layer overhead. Each throughput focused benchmark run transfers 100 million messages and each latency focused benchmark run transfers 10 million messages. Starting with 8 kB message size, the amount of messages is incrementally halved to avoid unnecessary long running benchmark runs. We evaluated payload sizes of 1 byte to 1 MB in power-of-two-increments. When discussing the results, we focus on the message rate on small messages with payload sizes less than 1 kB and on the throughput on middle sized and large messages starting at 1 kB.

The throughput results are depicted as line plots with the left y-axis describing the throughput in million messages per second (mmps) and the right y-axis describing the throughput in MB/s. For the latency results, the left y-axis describes the latency in µs and the right y-axis the throughput in mmps. The dotted lines always represent the message throughput while the solid lines represent either the throughput in MB/s or the latency in µs, depending on the benchmark. For the overhead results, a single y-axis describes the overhead in percentage in relation to the amount of payload transferred on a logarithmic scale. On all plot types, the x-axis depicts the size of the payload in power-of-two increments from 1 byte to 1 MB. Each benchmark run was executed three times and the min and max as well as average of the three runs are visualized using error bars.

The following releases of software were used for compiling and running the benchmarks: Java 1.8, OFED 4.4-2.0.7, libvma 8.7.5, IBM J9 VM version 2.9, gcc 8.1.0. We ran our experiments on the following two setups with two nodes each:

1) Mellanox ConnectX-3 HCA, 56 Gbit/s InfiniBand, MTU size 4096, Bullx blade with Dual socket Intel Xeon E5-2697v2 (2.7 GHz) 12 core CPUs, 128 GB RAM, CentOS 7.4 with the Linux Kernel version 3.10.0-693, SBE-820C with built-in switch
2) Mellanox ConnectX-5 HCA, 100 Gbit/s InfiniBand, MTU size 4096, Supermicro blade with Dual socket Intel Xeon Gold 6136 (3.0 GHz) 12 core CPUs, 128 GB RAM, CentOS 7.4 with the Linux Kernel version 3.10.0-693, Super Micro EDR-36 Chassis with built-in switch

Flow steering must be activated for libvma to redirect all traffic over InfiniBand by setting the parameter *log_num_mgm_entry_size* to *-1* in the configuration file */etc/modprobe.d/mlnx.conf* for the InfiniBand kernel module. Otherwise, libvma falls back to sockets over Ethernet.

In the following subsections, we focus the analysis and discussion on the results with 100 Gbit/s hardware. Selected Figures depicting the results with 56 Gbit/s hardware are also included if they provide additional insights and value for discussion and comparision. Due to automated execution of the benchmarks, the naming in the figures differs slightly, e.g. ”JSocketBench(msg)“ refers to IPoIB. The other names are self-explanatory.

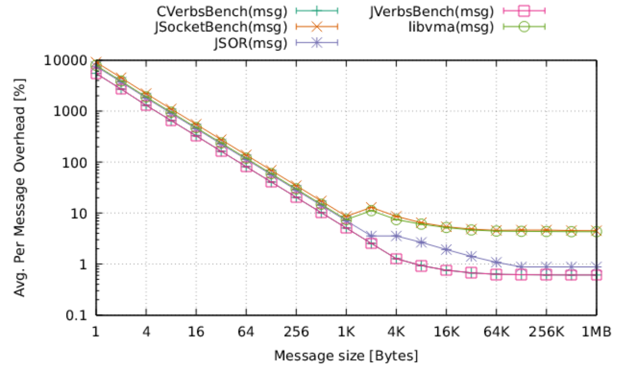

Fig. 1: Average per message overhead in percentage in relation to the payload size transferred.

A. Overhead

In this Section, we present the results of the overhead measurements of the described libraries/implementations. As overhead, we consider the additional amount of data that is sent along with the payload data of the user. This includes any data of any network layer down to the HCA. We measured the amount of data emitted by the port using the performance counter *port_xmit_data* of the HCA.

IPoIB and libvma are implementing buffer/message aggregation when sending data. Applications on high load sending many small messages benefit highly from increased throughput and the overall per message overhead is lowered. However, in order to determine the general per message overhead, we used the pingpong benchmark which does not allow aggregation due to its nature. The results of both types (sockets/verbs) are depicted in a single figure (see Figure 1).

We try to give a rough breakdown of the overhead involved with each method evaluated. A precise breakdown is rather difficult with just the raw amount of data captured from the ports as re-transmission of packages are also captured (e.g. RC queue pairs or custom protocols based on UD queue pairs).

The results in Figure 1 show that the overhead for msg operations of C-verbs and jVerbs are identical. For a single byte of payload, an additional 54 bytes are required which corresponds to two 27 byte headers which are part of the low-level InfiniBand protocol. Required by the RC protocol, one package is used for sending the ping and the other package to receive the pong. The metadata consists of a local routing header (8 bytes), base transport header (12 bytes), invariant CRC (4 bytes) and variant CRC (2 bytes) [19]. This makes a total of 26 bytes which is close to the measured 27 bytes (errors due to *port_xmit_data* yielding values in octets). For RDMA operations, an additional RDMA extended transport header (16 bytes) is added which makes a total of 42 bytes of “metadata“ for such a packet. Naturally, this overhead cannot be avoided. As expected with jVerbs using the native verbs directly without adding another software protocol layer, the overhead added is identical to C-verbs’s. The overhead stays constant which leads to an overall decreasing per message

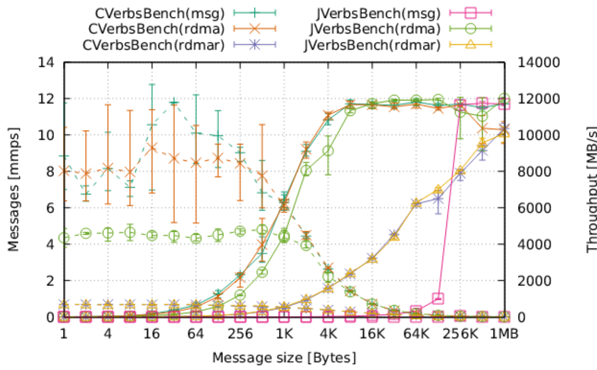

Fig. 2: **Uni-directional** throughput, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

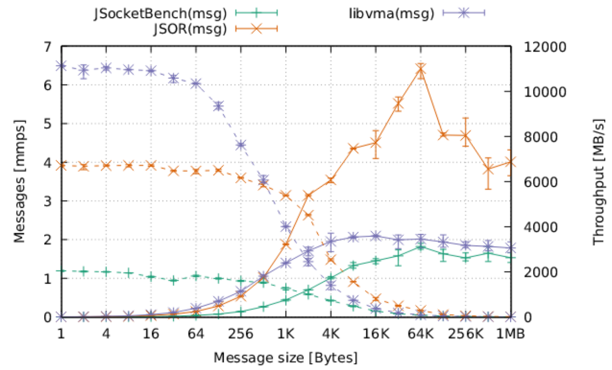

Fig. 3: **Uni-directional** throughput, **socket-based** libraries, increasing message size, **100 Gbit/s**.

overhead with increasing payload size. Starting with 8 kB payload size, the overhead ratio drops below 1%.

The overhead of the socket-based solutions is overall slightly higher. Again, considering 1 byte messages, JSOR adds an additional 7500 %, libvma 7900 % and IPoIB 9100 % overhead to each pingpong transfer. libvma and IPoIB rely on UD messaging verbs which add a datagram extended transport header (8 bytes) to the InfiniBand header and include additional information to allow IP-address based routing of the packages. The IPoIB specification describes an additional header of 4 octets (4 bytes) and IP header (e.g. IPv4 20 bytes + 40 bytes optional) which are added alongside the message payload [27]. libvma adds an IP-address (4 bytes) and Ethernet frame header (14 bytes) [10]. Remaining data is likely committed towards a software signalling protocol. Regarding JSOR, we could not find any information about the protocol implemented (closed source).

B. Uni-directional Throughput

This section presents the throughput results of the uni-directional benchmark. Starting with the verbs-based results depicted in Figure 2, jVerbs RDMA write message throughput (4.3 - 4.6 mmps) is about half of C-verbs's RDMA write throughput (7.9 - 9.3 mmps) for small payload sizes up to 512 byte. The RDMA write performance of C-verbs is nearly double the throughput of C-verbs messaging but with high jitter. Starting at 1 kB payload size, jVerbs's RDMA write throughput stays clearly below both C-verbs's RDMA write and message send throughput. Interesting to note that C-verbs's messaging is significantly better, though highly jittery, on small messages up to 512 bytes and middle sized messages up to 4 kB. Both C-verbs operations saturate throughput with 8 kB payload size and low jitter at around 11.7 GB/s. We could not determine the reason for the very poor performance of jVerbs's msg verbs on both 56 Gbit/s and 100 Gbit/s hardware.

The results of socket-based libraries are depicted in Figure 3. On small payload sizes up to 256 bytes, IPoIB achieves a throughput of approx. 1 - 1.2 mmps. With increasing payload size, the throughput starts saturating at 32 kB message size and peaks at 64 kB with 3.1 GB/s throughput. The results of libvma show an highly increased throughput of 6.0 to 6.5 mmps for up to 64 byte messages. Overall throughput for middle and large sized messages surpasses IPoIB's peaking at 3.5 GB/s with 8 kB messages but also starting to decrease down to 3.0 GB/s when increasing the message size up to 1 MB. JSOR achieves a significantly lower throughput of 3.8 - 3.9 mmps for up to 128 byte messages. However, JSOR provides a much higher throughput starting at 1 kB message size compared to IPoIB and libvma. Throughput peaks at 64 kB message size with 11 GB/s but drops down to approx 6.5 GB/s with 512 kB messages afterwards. As described in Section IV, we determined that JSOR's performance degrades considerable on payload sizes of 128 kB and greater which required us to increase the RDMA buffer size (to 1 MB). However, this problem could not be resolved entirely.

The results on 56 Gbit/s hardware for both, verbs and sockets, show an overall and expected lower throughput but not any notable differences. Thus and due to space constraints, the figures are omitted. But, results for libvma were not available because the benchmarks failed repeatedly with a non fixable ”connection reset“ error by libvma.

C. Bi-directional Throughput

This section presents the throughput results of the bi-directional benchmark. With full-duplex communication supported, we expect roughly double the throughput of the uni-directional results in general. Figure 4 depicts the results of the verbs-based implementations and, as expected, all implementations show roughly double the message rate on small messages and roughly double the throughput on large messages compared to the uni-directional results (§IV-B).

C-verbs RDMA writes are still jittery but yield the best performance with 15.8 - 19.7 mmps for 1 - 128 byte payload size, 23.1 GB/s peak performance at 32 kB payload size. This is followed by C-verbs messages with 11.5 - 18.6 mmps for 1 - 512 bytes, 23.3 GB/s peak performance at 8 kB payload size. jVerbs RDMA writes perform worse on payload sizes up to 1 kB (8.3 - 9.3 mmps) but with less jitter than C-verbs RDMA writes. Saturation on large messages starts at around around 16

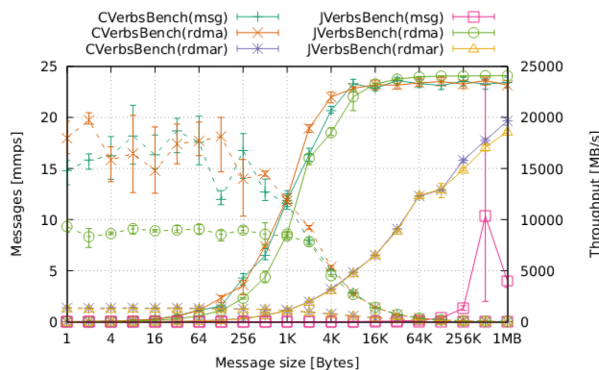

Fig. 4: **Bi-directional** throughput, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

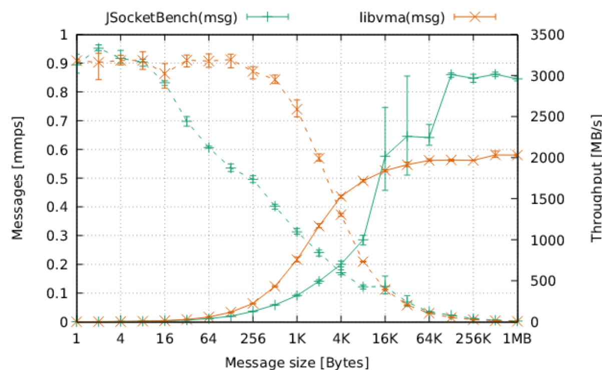

Fig. 6: **Bi-directional** throughput, **socket-based** libraries, increasing message size, **100 Gbit/s**.

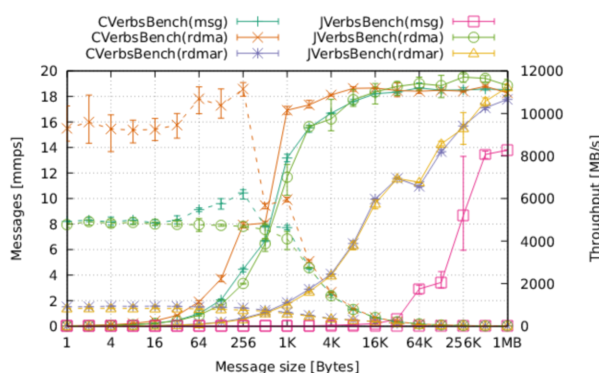

Fig. 5: **Bi-directional** throughput, **verbs-based** libraries with different transfer methods, increasing message size, **56 Gbit/s**.

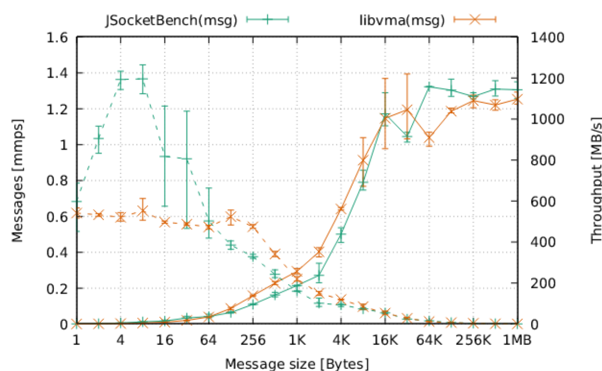

Fig. 7: **Bi-directional** throughput, **socket-based** libraries, increasing message size, **56 Gbit/s**.

kB with 23.7 GB/s with a peak performance of 24.0 GB/s at 128 kB message size. RDMA reads of both verbs interfaces are nearly on par. The incomprehensible poor performance of jVerbs msg verbs, as already seen on the uni-directional benchmark results (§IV-B), is also present here.

The 56 Gbit/s results are depicted in Figure 5 and show an overall similar but as expected lower performance regarding throughput. On small messages, the RDMA write performance of C-verbs is less jittery and jVerb's not significantly lower compared to the 100 Gbit/s results. The RDMA write performance of C-verbs clearly outperforms jVerb's sometimes slight jittery performance on 128 byte to 1 kB messages.

Figure 6 depicts the socket-based results of the bi-directional benchmark. Due to unresolvable errors causing disconnects, especially on message sizes smaller 512 bytes, we cannot provide results for JSOR. This seems to be a known problem [6] but increasing the send and receive queue sizes as described does not resolve this issue. Furthermore, the benchmark does not terminate anymore on message sizes greater than 32 kB. The proposed solution to increase the buffer size does not resolve this problem either [5].

The results available show a very low overall performance for IPoIB and libvma compared to their results on the uni-directional benchmark (§IV-B). IPoIB achieves an aggregated throughput of 0.89 to 0.95 mmps for just up to 16 byte messages with a considerable drop in performance for small messages afterwards. But, throughput increases with increasing message size starting with middle sized messages saturating and peaking at 128 kB message with 3.0 GB/s aggregated throughput. Further notable are high fluctuations for 16 kB to 64 kB. On small messages, libvma can at least provide a constant performance for small messages up to 128 bytes with 0.9 mmps. Throughput increases with increasing message size with saturation starting at approx. 32 kB messages with 1.9 GB/s aggregated throughput. A lower peak performance than IPoIB is reached at 512 kB messages with 2.0 GB/s.

The results on 56 Gbit/s hardware in Figure 7 show high fluctuations on up to 64 byte messages with IPoIB and also on 8 kB to 32 kB messages with libvma.

D. One-sided Latency

This section presents the results of the one-sided latency benchmark to determine the latency of a single operation. Section IV-E further discusses full RTT latency for a ping-pong communication pattern. Results are separated by socket-based and verbs-based, and include the average latency as well as the 99.9th percentiles.

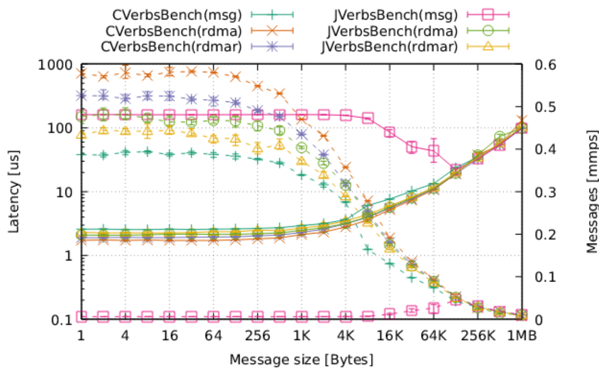

Fig. 8: Average **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

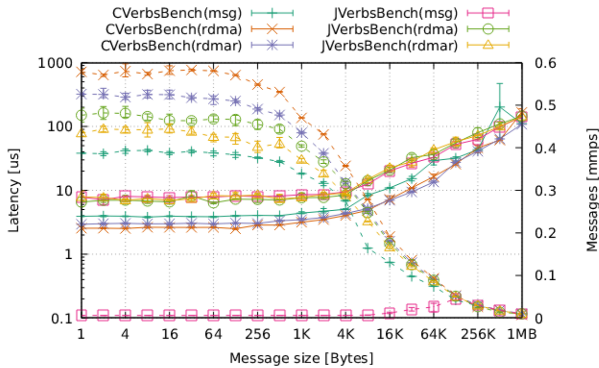

Fig. 9: 99.9th percentile **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

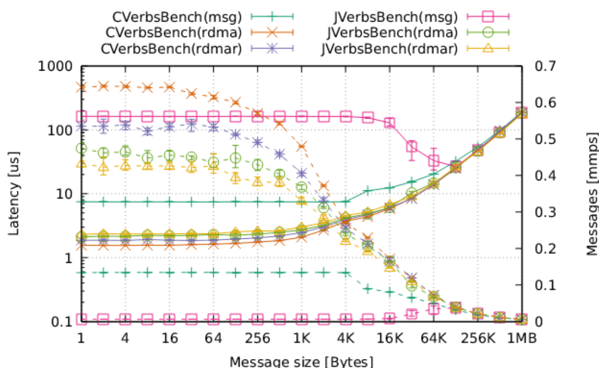

Fig. 10: Average **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **56 Gbit/s**.

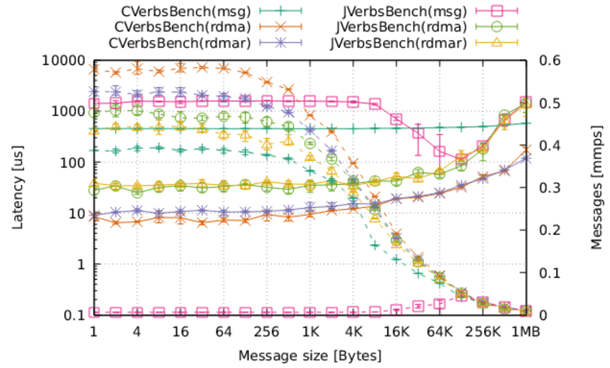

Fig. 11: 99.99th percentile **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

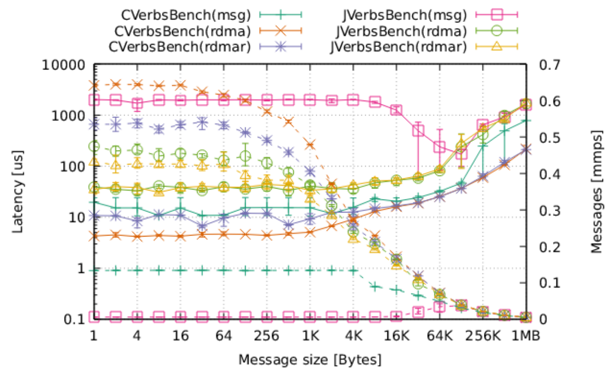

Fig. 12: 99.99th percentile **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **56 Gbit/s**.

Figure 8 shows the average latencies and Figure 9 the 99.9th percentiles of the verbs-based benchmarks. The results of all native verbs-based communication as well as jVerbs RDMA write and read are as expected providing an average close to 1 µs latency. From lowest to highest: C-verbs RDMA write, C-verbs RDMA read, jVerbs RDMA write, jVerbs RDMA read and C-verbs msg. As expected, jVerbs adds some overhead leading to a minor increase in latency.

But, jVerbs msg shows unexpected average latency results. Up to 4 kB message size, which equals the used MTU size, the latency is very high and constant at approx. 160 µs. With further increasing message size, the latency is lowered and approximates the average latencies of the other transfer methods. At 128 kB message size, it goes along with the other results. This is also present on the results on 56 Gbit/s hardware for jVerbs msg (see Figure 10).

To further analyze this issue, we depicted the 99.9th percentiles in Figure 9. The other transfer methods are showing expected behaviour and overall low latency. But, jVerbs msg also shows very low latencies around 8 µs latency for 99.9th of all messages transferred. Thus, just a small amount of messages yields latencies higher than 8 µs. This can be verified

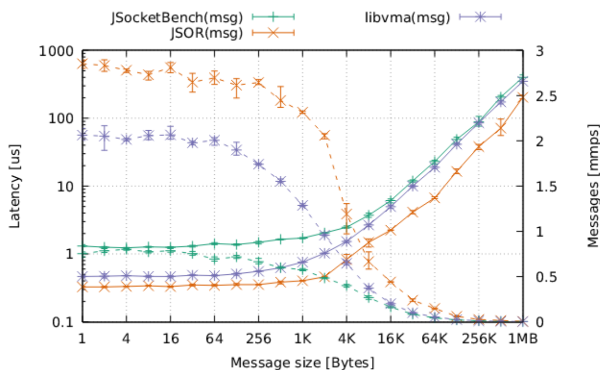

Fig. 13: Average **one-sided latency**, **socket-based** libraries, increasing message size, **100 Gbit/s**.

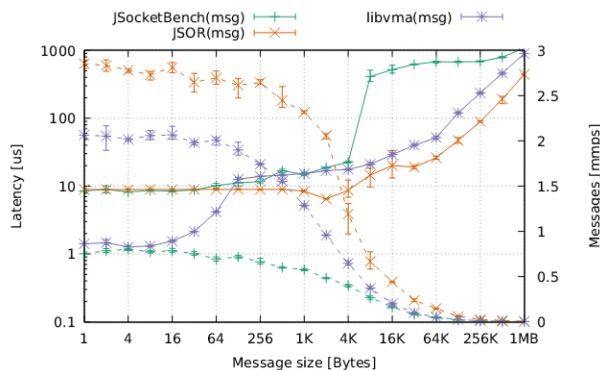

Fig. 14: 99.9th percentile **one-sided latency**, **socket-based** libraries, increasing message size, **100 Gbit/s**.

by analyzing the 99.99th percentiles (i.e. 1,000 worst out of 10 million) depicted in Figure 11. The results show a latency of approx. 1500 µs confirming that there is a very small amount of messages with very high latency causing a rather overall high average latency. This conclusion can also be drawn for jVerbs's results on 56 Gbit/s hardware (see Figure 12).

Two further issues regarding C-verbs msg can be observed. The first is a nearly constant 450 µs on the 99.99th percentile results on 100 Gbit/s hardware (see Figure 11). But, the average and 99.9th percentile results of C-verbs msg are as expected. This means that 1,000 out of 10,000,000 messages show a latency of 450 µs or worse. This is caused by the RC protocol yielding occasional RNR NAKs on high loads when sending data and no corresponding work requests for receiving are queued on the remote at that moment. This is proven by the value of the hardware counter *rnr_nak_retry_err* which shows that about 3,100 send requests are NAK'd during one benchmark run. For the sender, each NAK enforces a small delay to wait for resources to become available again. This behaviour cannot be avoided in such a benchmark scenario.

The second issue with C-verbs msg is a rather high average latency of 7.5 µs for up to 4 kB messages on 56 Gbit/s hardware. However, the RDMA write/read latency results on 56 Gbit/s hardware are similar to the ones on 100 Gbit/s hardware. The 99.9th and 99.99th percentiles (10 to 15 µs) prove that the base latency on that hardware configuration with 56 Gbit/s speed is unexpectedly high.

For the socket-based solutions, the average latencies in Figure 13 show that JSOR performs best with an average per operation latency of 0.3 - 0.4 µs for up to 1 kB messages. With further increasing payload size, latency increases as expected. libvma shows similar results with a slightly higher latency of 0.4 - 0.5 µs for small messages. IPoIB follows with a further increased average latency of 1.3 to 1.5 µs for small messages. These results, especially JSOR's, seem unexpected low at first glance. But, when considering the socket-interface, it does not provide means to return any feedback to the application when data is actually sent to the remote. With verbs, one polls the completion queue and as soon as the work completion is

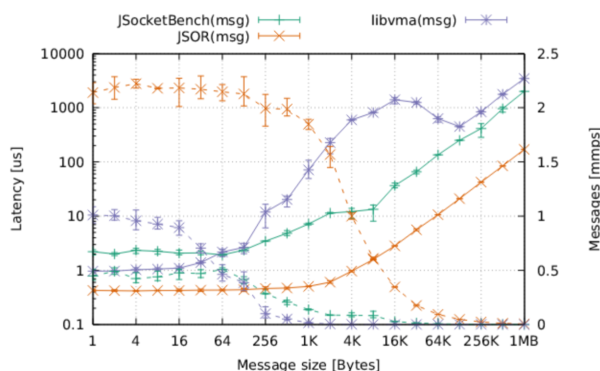

Fig. 15: Average **one-sided latency**, **socket-based** libraries, increasing message size, **56 Gbit/s**.

received, it is guaranteed that the local data is sent and received by the remote. A socket send-call however, does not guarantee that the data is sent once it returns control to the caller. Typically, a buffer is used to allow aggregation of data before putting it on the wire. JSOR, libvma and IPoIB implement message aggregation before actually sending out any data. This is further proven by the ping-pong benchmark which does not allow any aggregation to be applied by the backend (§IV-E). On 56 Gbit/s hardware (see Figure 15), JSOR and IPoIB show similar results with a slightly higher latency but libvma shows significantly worse results for 256 byte to 64 kB message sizes compared to running on 100 Gbit/s hardware.

The 99.9th percentiles in Figure 14 show further interesting aspects not just limited to message aggregation. libvma starts with very low 99.9th latencies for up to 16 byte messages indicating a rather low threshold for aggregation benefitting small messages by keeping the total of worst latencies low. However, with 16 to 128 byte messages, the latency increases considerably. JSOR and IPoIB show similar results for small messages up to 1 kB. JSOR's stays even constant with 9 µs up to 512 byte messages. This indicates a flush threshold based on the number of messages instead of a total buffer size. But, the latency of IPoIB starts to increase already starting with 64 byte message size and even jumps significantly up to over 400

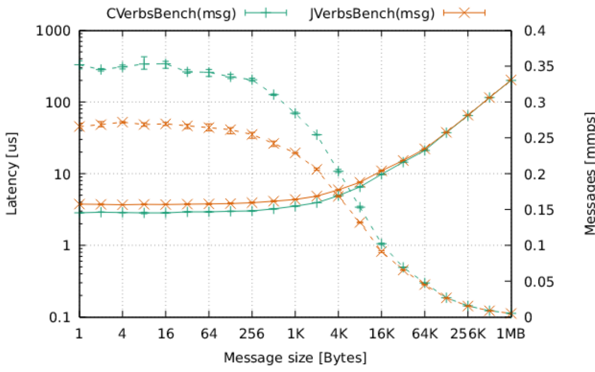

Fig. 16: Average **ping-pong** latency, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

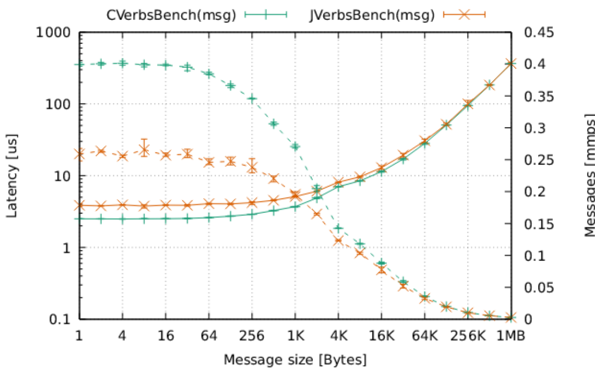

Fig. 17: Average latency **ping-pong**, **verbs-based** libraries with different transfer methods, increasing message size, **56 Gbit/s**.

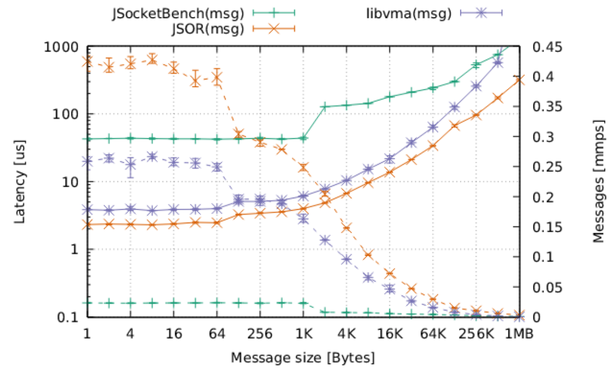

Fig. 18: Average **ping-pong** latency, **socket-based** libraries, increasing message size, **100 Gbit/s**.

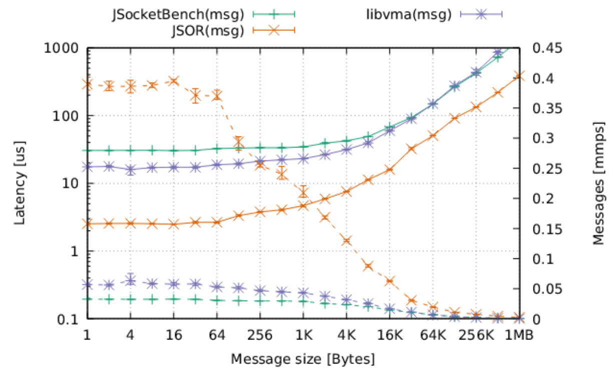

Fig. 19: Average **ping-pong** latency, **socket-based** libraries, increasing message size, **56 Gbit/s**.

µs with 8 kB message size. This might indicate that additional allocations are involved for large(r) messages increasing the overall worst latencies significantly. On 56 Gbit/s hardware, JSOR's and IPoIB's results are similar with a slightly higher latency. But, libvma's 99.9th percentile latency is significantly increasing with 256 byte messages and even reaching 600 ms on multiple sizes greater than 4 kB.

E. Ping-pong Latency

In this section, we present the results of the ping-pong latency benchmark. Due to the nature of the communication pattern, the methods of transfer are limited to messaging operations for verbs-based implementations. Using RDMA operations is also possible but requires additional data structures and control logic which is currently not implemented. The average latencies, i.e. full round-trip-times, are depicted in Figure 16 (Figure 17 for 56 Gbit/s results). C-verbs messaging achieves an average latency of 2.8 to 3.2 µs for up to 512 byte messages. jVerbs shows similar behaviour but with a slightly higher latency of 3.7 to 4.1 µs. Further increasing the message size of both, C-verbs and jVerbs increases the latency as expected. The 99.9th percentiles results on 56 Gbit/s and 100 Gbit/s hardware do not show any abnormalities and their Figures are omitted due to space constraints.

The average latencies of the socket-based methods are depicted in Figure 18 (Figure 19 for 56 Gbit/s results). Both, JSOR and libvma show low average latencies of 2.3 to 3.5 µs and 3.7 to 5.3 µs for message sizes up to 512 bytes. With increasing message sizes, the average latency also increases as expected. A small "latency jump" of around 1 µs is notable on both libraries from 64 byte to 128 byte message size. IPoIB shows a similar behaviour but with a significant higher average latency of 42.7 to 43.2 µs up to 1 kB message size. The same "latency jump" can be seen from 1 kB to 2 kB message size. This increase of latency might be caused due to switching to a different buffer size which might involve additional buffer allocation. The 99.9th percentile results show a similar behaviour without further abnormalities. Their figures are omitted due to space constraints.

V. Conclusions

We presented our JIB-Benchmark to evaluate three socket-based and two verbs-based solutions to leverage InfiniBand in Java applications. We believe that such a benchmark, not available thus far, will help the community and developers interested in using InfiniBand in their Java applications to

find a suitable solution for their applications. The benchmark is open source and can be extended to evaluate further libraries. We evaluate the available solutions on two hardware configurations with 56 Gbit/s and 100 Gbit/s InfiniBand NICs. As expected, the socket-based solutions provide a transparent solution requiring low effort to get additional performance from InfiniBand hardware for existing socket-based software without requiring any changes. But, this comes at the price that the full potential of the hardware cannot be exploited, especially on bi-directional communication. Compared to the performance of Gigabit Ethernet, latency is at least halved on 56 Gbit/s hardware and can even be as low as 2-5 µs for small messages. Regarding throughput, one can get an at least ten-fold increase and it is even possible to saturate 56 Gbit/s hardware on uni-directional communication.

To leverage the true potential of the hardware, the verbs-based solutions are a must. Overall, jVerbs is performing very well and brings nearly native performance on RDMA operations to the Java space with a few minor performance differences. But, the inexplicable limited performance of jVerbs messaging verbs does not allow any meaningful usage in applications. With C-verbs, the full potential of the hardware can be exploited on all communication methods. Thus, one has to decide whether to stay entirely in Java space but having to rely on the proprietary JV9 JVM or having the freedom to write a custom network subsystem using C-verbs with JNI which is more time consuming and challenging.

Our personal recommendations regarding the evaluation: we consider libvma a good solution to benefit from some of the performance of InfiniBand hardware without having to invest a significant amount of time and work and not depending on a proprietary JVM. But, we think that it is worth spending additional work and time on implementing a custom network subsystem based on C-verbs to leverage the true performance of InfiniBand hardware if required for a target application.

REFERENCES

[1] Apache ignite. https://ignite.apache.org/.
[2] Cassandra. https://cassandra.apache.org/.
[3] Gemfire. https://pivotal.io/pivotal-gemfire.
[4] Hazelcast. https://hazelcast.com.
[5] Ibm. rdma communication appears to hang. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/diag/problem_determination/rdma_jsor_hang.html.
[6] Ibm. rdma connection reset exceptions. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/diag/problem_determination/rdma_jsor_connection_reset.html.
[7] Infinispan. http://infinispan.org/.
[8] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.
[9] Jib-benchmarks github. https://github.com/hhu-bsinfo/jib-benchmarks/.
[10] libvma github. https://github.com/Mellanox/libvma/.
[11] Mellanox. https://www.mellanox.com/.
[12] Ofed 3.5 release notes. https://downloads.openfabrics.org/OFED/release_notes/OFED_3.5_release_notes.
[13] Openfabrics alliance. https://openfabrics.org/.
[14] Osu micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.
[15] Ramcloud source code. https://github.com/PlatformLab/RAMCloud.
[16] Rdmamojo. blog by dotan barak. https://www.rdmamojo.com.
[17] Speedus. http://speedus.torusware.com/index.html.
[18] Top500 list.
[19] Infiniband architecture specification volume 1, release 1.3. http://www.infinibandta.org/, 2015.
[20] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.
[21] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 1094–1095, 2005.
[22] R. R. Exposito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Fastmpj: a scalable and efficient java message-passing library. *Cluster Computing*, 17:1031–1050, Sept. 2014.
[23] D. Goldenberg, T. Dar, and G. Shainer. Architecture and implementation of sockets direct protocol in windows. *2006 IEEE International Conference on Cluster Computing*, pages 1–9, 2006.
[24] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 902–903, 2005.
[25] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.
[26] W. Huang, H. Zhang, J. He, J. Han, and L. Zhang. Jdib: Java applications interface to unshackle the communication capabilities of infiniband networks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 596–601, 10 2007.
[27] V. Kashyap. Ip over infiniband (ipoib) architecture. https://www.ietf.org/rfc/rfc4392.txt, April 2006.
[28] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB 2011: 6th Workshop on Networking meetsDatabases*, 2011.
[29] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, 2010.
[30] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.
[31] X. Liu. Entity centric information retrieval. *SIGIR Forum*, 50:92–92, June 2016.
[32] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. *SIGPLAN Not.*, 50:695–710, Oct. 2015.
[33] S. Mehta and V. Mehta. Hadoop ecosystem: An introduction. In *International Journal of Science and Research (IJSR)*, volume 5, June 2016.
[34] S. Nothaas, K. Beineke, and M. Schoettner. Ibdxnet: Leveraging infiniband in highly concurrent java applications. *CoRR*, abs/1812.01963, 2018.
[35] Oracle. Oracle coherence. https://www.oracle.com/technetwork/middleware/coherence/overview/index.html.
[36] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, November 1999. Previous number = SIDL-WP-1999-0120.
[37] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. sson, D. van der Spoel, B. Hess, and E. Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29:845–854, 2013.
[38] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14. ACM, 2013.
[39] G. L. Taboada, J. Touriño, and R. Doallo. Java fast sockets: Enabling high-speed java communications on high performance clusters. *Comput. Commun.*, 31:4049–4059, Nov. 2008.
[40] S. Thirugnanapandi, S. Kodali, N. Richards, T. Ellison, X. Meng, and I. Poddar. Transparent network acceleration for java-based workloads in the cloud. https://www.ibm.com/developerworks/library/j-transparentaccel/, January 2014.
[41] X. Wu, X. Zhu, G. Q. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26:97–107, Jan. 2014.
[42] P. Zhao, Y. Li, H. Xie, Z. Wu, Y. Xu, and J. C. Lui. Measuring and maximizing influence via random walk in social activity networks. pages 323–338, Mar. 2017.

Neutrino: Efficient InfiniBand Access for Java Applications

Filip Krakowski
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
filip.krakowski@hhu.de

Fabian Ruhland
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
fabian.ruhland@hhu.de

Michael Schöttner
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—Fast networks like InfiniBand are important for large-scale applications and big data analytics. Current InfiniBand hardware offers bandwidths of up to 200 Gbit/s with latencies of less than two microseconds. While it is mainly used in high performance computing, there are also some applications in the field of big data analytics. In addition, some cloud providers are offering instances equipped with InfiniBand hardware. Many big data applications and frameworks are written using the Java programming language, but the Java Development Kit does not provide native support for InfiniBand. To this end we propose neutrino, a network library providing comfortable and efficient access to InfiniBand hardware in Java as well as epoll based multithreaded connection management. Neutrino supports InfiniBand message passing as well as remote direct memory access, is implemented using the Java Native Interface, and can be used with any Java Virtual Machine. It also provides access to native C structures via a specially developed proxy system, which in turn enables the developer to leverage the InfiniBand hardware's full functionality. Our experiments show that efficient access to InfiniBand hardware from within a Java Virtual Machine is possible while fully utilizing the available bandwidth.**

***Index Terms*—InfiniBand, Java Native Interface, Remote Direct Memory Access**

I. Introduction

RDMA capable devices are providing high throughput and low latency to HPC applications for several years [1]. With todays cloud providers offering instances equipped with InfiniBand for rent, such hardware becomes available to a wider range of users without the high costs of buying and maintaining it [2]. Many big data systems are written in Java [3], [4] benefitting from the strong type system, the rich libraries and the automatic garbage collection.

Distributed Java applications are limited to Ethernet-based socket-interfaces (standard ServerSocket or NIO) on the commonly used JVMs OpenJDK and Oracle. These JVMs do not provide support for low-latency InfiniBand hardware. But, there are third-party solutions like for example *DiSNI* [5], *Ibdxnet* [6], and *jverbs* [7] available each with pros and cons.

Ibdxnet is an InfiniBand message passing transport we developed in the past for *DXNet* [8] both for distributed and parallel Java applications. While our previous efforts are based on transparent serialization of messaging objects we are now developing the successor *neutrino* aiming at providing RDMA for native data which is managed by Java applications and can be accessed efficiently and easily. The latter is realized by automatically generated proxy objects which are linked to native C structs. This allows us to provide the full functionality of the ibverbs library within the Java space and consequently implement all logic that previously had to be implemented in native code in Java. Similarly, these capabilities allow us to provide an application library for developing RDMA-enabled Java applications.

II. Related work

In the past, several attempts have been made to use the ibverbs library from Java, such as *jVerbs* and the Direct Storage and Networking Interface (*DiSNI*) library developed at the IBM Research Lab [5], [7]. *jVerbs* is a proprietary library, while *DiSNI* is an open source solution based on *jVerbs* [9]. The authors also emphasize that native method calls are expensive and therefore they need a solution that minimizes these costs. To this end, the authors use a procedure which they call "Stateful Verb Calls". The core function of this procedure is to serialize operations allocated in Java space into the format expected by ibverbs and to cache them for further calls. After this step it is possible to execute the operation as often as desired by passing the serialized state to the corresponding ibverbs method using the Java Native Interface.

From our point of view, this approach has some disadvantages. First, serialization logic as well as the memory layout of the native structures for each operation must be laboriously created by hand in Java. Second, ordinary Java objects are serialized into a format understandable to ibverbs, resulting in additional copies of the required structures. In addition, a memory layout must also be adapted when changes are made within the native library, otherwise it can lead to write or read accesses at incorrect memory offsets and thus to undefined behavior.

Jdib [10] is another library wrapping native ibverbs function calls and exposing them to Java using a JNI layer. According to the authors, various methods, e.g. queue pair data exchange on connection setup, are abstracted to create an easier to use API for Java programmers. The fundamental operations to create protection domains, create and setup queue pairs, as well as posting data-to-send to queues and polling the

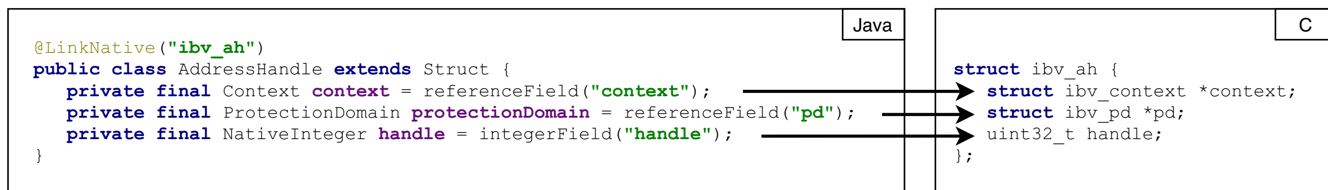

Fig. 1. Example mapping between automatically generated Java proxy object and native C struct.

completion queue seem to wrap the native verbs and do not introduce additional mechanisms like jVerbs's stateful verb calls. Unfortunately, we were not able to obtain a copy of the library for further investigation.

III. Efficient Structured Access to ibverbs

The key objectives of *neutrino* include efficient access to the functionality provided by ibverbs on any JVM. For this reason, the idea of adapting the source code of one specific JVM was not an option and we have developed a universally applicable solution.

The approach we propose for a structured access to ibverbs is a concept that allows programmers to link native structures with automatically generated *proxy objects* in Java space and pass them as efficiently as possible through the Java Native Interface (JNI).

Interfacing with native methods from Java space is known to be costly and can be measured on a per invocation basis [11]. To keep these costs as low as possible, we aim at minimizing the number of border crossing calls and keep them as simple as possible. This is achieved by passing only primitive data types to the native part of *neutrino*. For this purpose, we use automatically generated Java proxy objects in order to write and read memory outside the Java managed heap in a structured way. Since native memory is not managed by the JVM, it is safe to share it with native code without having to fear object movements by the garbage collector.

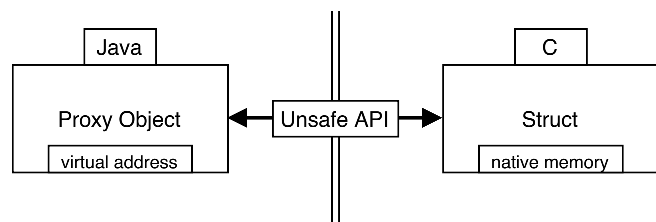

Fig. 2. Components of the structured native memory access.

As shown in Figure 2, each proxy object encapsulates the virtual address of the corresponding native structure. This approach allows direct access to native structures using Java's Unsafe class and its intrinsic methods [12]. Furthermore, our proxy objects allow selective access to individual fields of native structures. Special access objects for various native data types are available to implement this property.

Figure 1 shows an example of a generated Java proxy object, which includes references to two other generated proxy objects (source code is not shown) and one access object for an integer field. As can also be seen, the individual fields of the proxy object use the names of the corresponding fields within the native structure and the enclosing class has an annotation containing the native structure's name. Our system uses this information to automatically create a mapping between each pair of fields. To achieve this, the offsets of the individual fields within the native structure must be known at runtime.

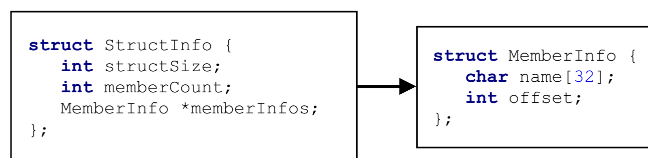

Fig. 3. Metadata structs used to map proxy object fields onto native C struct fields.

Our solution stores the metadata shown in Figure 3 in the native code and makes it available to the Java space through the JNI. This allows us to lookup and cache the names and offsets of each field of a native structure. More importantly, the metadata is stored in a form in which each field query can be completed in a constant time. All metadata is automatically generated in native code using macro functions that extract the required information. This allows proxy objects to easily retrieve the storage layout of their associated native structures and configure their access objects accordingly.

structSize	memberCount	memberInfos
20	3	• →

name	offset
context	0
pd	8
handle	16

Fig. 4. Generated metadata for the `ibv_ah` C struct.

Figure 4 shows an exemplary setup of the metadata for the native structure `ibv_ah` shown in Figure 1. We need to know the size of the structure (in this case 20 bytes) in advance in order to allocate correspondingly large memory blocks in Java space. Similarly, we need to know the number of fields (in

this case 3) contained within the structure so that the list of metadata generated for it can be traversed from Java space.

Because both data structures in Java and native space share the same memory layout, we can safely and efficiently access the Java space from native code. This is done by passing the pointer encapsulated in a proxy object to a native method, which in turn is now able to read and write to the referenced memory in a structured way using a typecast. Since the referenced memory exists on both sides, changes can be seen immediately without copying data.

We use this concept for automatically generating Java classes for all native structures contained in ibverbs. For this purpose we have implemented a custom code generator, which processes header files of native libraries and then creates the corresponding Java classes. In this way, we are able to use the full functionality of the library from within the Java space and consequently implement all logic that previously had to be implemented in native code in Java. Since the memory addresses of the created objects do not change at runtime, it is also possible for us to cache the created proxy objects in Java space and keep them ready for future access. Thus, no unnecessary instances of proxy objects are created and the garbage collector is not burdened.

IV. Neutrino's architecture

Developing an application using ibverbs and our JNI access layer alone requires considerable effort and careful programming. This is particularly the case for applications aiming at high performance. In this section we propose neutrino, a network library aiming at simplifying the development of RDMA-enabled applications in Java. The provided functionalities include connection management, concurrent messaging and operations on remote storage. The core idea behind neutrino is to use small messages to control the system and remote direct memory accesses to transfer large amounts of data.

A. Connection management

Within the ibverbs library connections are abstracted in the form of *queue pairs*. To manually establish a reliable connection between two queue pairs, certain information must be exchanged in advance. This includes the InfiniBand device port's local id and number and the local queue pair's number. Using this information the queue pairs can be configured and transitioned into a state in which they can be used for sending and receiving messages on both sides. Neutrino handles this procedure transparently by using a TCP connection for the exchange of all necessary information. In this way, the connection between two endpoints is established by using an IP address and a port. The RDMA Communication Manager library [13] offers similar functionality and is therefore also supported for connection establishment. While being supported, we decided against its usage, because it sets some parameters independently during the connection setup. Configuration from the application side is therefore limited.

B. Threading model

To make optimal use of the available resources, neutrino makes use of a thread pool and works event-based in a non-blocking fashion. Besides this, as seen in Figure 5 the processing of messages to be sent and received is handled by separate threads which are created based on the available number of CPU cores.

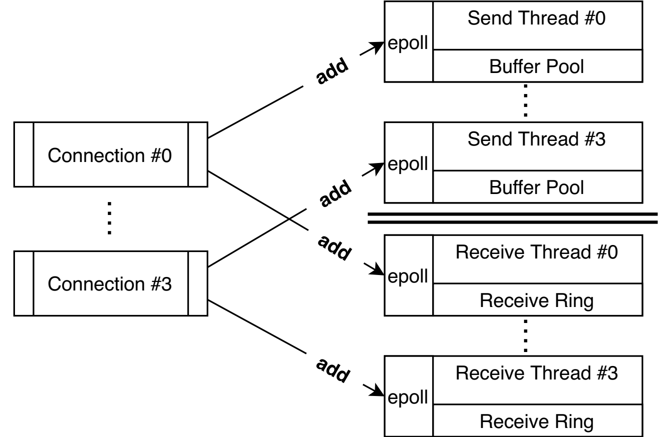

Fig. 5. Adding individual connections to sender and receiver threads.

Each connection is assigned to exactly one receive and one send thread in a round robin fashion, which perform the processing of the outgoing and incoming messages from this point on. This architectural design decision offers the opportunity to better configure individual endpoints in the network based on their tasks. For example, an endpoint that is intended to collect data can specify a greater weighting when creating receive threads and thus process more received messages in parallel. Similarly, an endpoint that only distributes data can use more sender threads than receiver threads and therefore process more outgoing messages in a concurrent fashion.

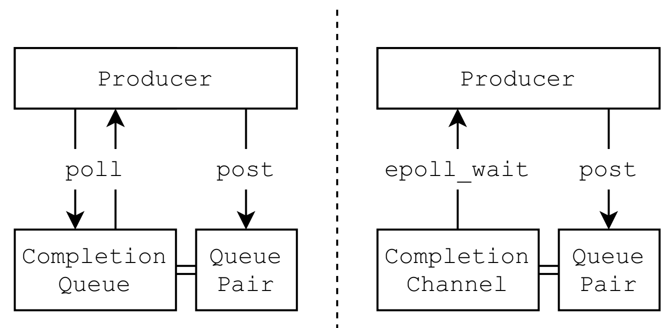

Fig. 6. Continuous polling of the completion queue (left) and notification based waiting on completions (right).

The execution of operations such as accessing remote memory must be triggered within the underlying ibverbs library by placing so-called *work requests* on the corresponding queue

pair. Each completed work request optionally generates a *work completion*, which the application can query to find out the request's status. For this purpose, each queue pair is assigned a *completion queue* for sent and received messages. Whenever a pending work request completes the network controller places a work completion on the corresponding completion queue. The application is then able to query these work completions and use their metadata to call up the appropriate processing function. By default, the query of completed requests is based on polling. Since continuous polling of completion queues results in high CPU usage while potentially not processing any work completions, ibverbs provides also *completion channels*. These contain a file descriptor which can be used with existing IO multiplexing approaches like `select`, `poll` and `epoll` as illustrated in Figure 6.

We decided to use epoll because of its good scalability with many connections. Each thread within the system receives its own epoll file descriptor, which is used to monitor the connections assigned to it for corresponding events. In this way it is possible to distribute connections to different threads for load balancing purposes. At the same time we avoid synchronization issues, because the data structures for sending and receiving messages of a connection are accessed only by a single thread. This also minimizes the necessary number of atomic operations on data structures and allows to avoid context switches.

C. Send request processing

InfiniBand offers two possibilities to exchange data between two network participants. On the one hand, it is possible to send data as messages, which must be actively processed by the other side. Alternatively, it is also possible to read or write remote memory using RDMA operations without including the CPU of the other node. A pre-requisite for both modes is the registration of so-called *memory regions*, which can then be used for the above operations. This is necessary since the InfiniBand hardware must know the physical addresses of the memory to be used. Furthermore, the mapping of virtual to physical memory addresses within the registered memory must not change during the runtime of the application. The corresponding pages are therefore additionally pinned by the operating system.

Neutrino aims at supporting both modes and therefore needs an abstraction layer that allows applications to easily send messages and work with remote memory without the need to perform the mentioned steps. For this purpose, certain data structures are created within connections as well as within the send threads, which enable easier handling of registered memory and facilitate the creation of work requests.

Each send thread allocates a configurable contiguous block of memory at the beginning of its execution. This memory block is registered with the InfiniBand hardware and then divided into smaller slices. The default size for each slice is the maximum MTU supported by the network card. Each slice is assigned a unique identifier and put into a *send buffer array* of memory blocks using the identifier as the index. A work request allows setting user-defined data for recognizing the corresponding work completion only within the id field, which is a 64 bit number. We therefore use this id field to store the index of the buffer belonging to the request. This later helps to release buffers processed or sent by the network controller. Finally, each slice is placed in a bounded multi-producer multi-consumer queue[14], the *send buffer queue*, which is used for borrowing memory blocks.

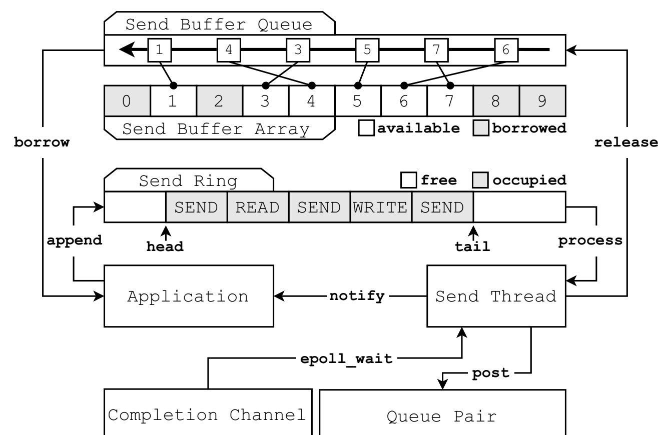

Fig. 7. Processing of outgoing operations using a ring buffer for requests and a queue of buffers for writing messages.

As shown in Figure 7, the send buffer array contains a fixed number of memory blocks ready for application requests. Each buffer may be available (white), and thus enqueued within the send buffer queue for borrowing, or borrowed (grey) and in the process of being accessed by the InfiniBand hardware. Sorting within the queue (`1,4,3,...`) can be arbitrary, as we cannot guarantee in which order an application will pass its borrowed buffers to a send thread for processing. However, this is not a problem because the buffers can be used in any order. An application borrows a buffer by polling the send buffer queue's next element.

We decided to register one memory region per thread instead of one memory region per connection as registering many scattered memory regions consumes additional resources of the InfiniBand hardware. The hardware needs to copy the registered memory regions using direct memory access. By using many scattered memory regions the hardware's access pattern is unpredictable, which can seriously affect performance. Also, caching within the hardware benefits of less memory regions, because there are only a few resources to be cached. In addition, we also align memory areas so that the network controller may transfer them using as few as possible direct memory accesses. Using our interface for accessing native memory, we are also able to wrap the buffers borrowed from the send thread with the help of a proxy object and thus write directly and in a structured way into the memory intended for sending. This way copies of the messages or data to be sent within the Java managed heap are avoided.

The network controller accesses borrowed buffers using information (virtual memory address, size and access key) contained within work requests. These work requests are stored in a modified version of Agrona's ring buffer[15] by the application. We call this ring buffer *send ring* since it does only contain work requests. Furthermore, each connection has its own send ring. Our modification to the original version was needed as the standard implementation only allows consuming messages or events written to the ring buffer isolated from each other. Since ibverbs offers the possibility to post requests in batches by linking work requests together, we needed a way to access successive requests within the send ring in order to chain them. As a first step, the application reserves an area large enough for storing its work request. This is done by atomically incrementing the send ring's tail index. Afterwards a work request is written directly into the reserved area. In the case of a message to be sent, this work request contains a reference to the borrowed buffer so that it can be released after processing. Finally, the written work request is committed to the send ring so that the send thread can consume it.

The send thread is responsible for posting pending work requests within the connection's send ring to the queue pair associated with the connection the send ring belongs to. To do this, the send thread first identifies and extracts the readable area of the send ring. Afterwards the work requests contained within the extracted area are chained together so that they can be transferred to the hardware in one batch. After the work requests have been transmitted, the send thread increments the head index of the send ring, freeing the extracted area for new work requests. The work requests can be released immediately after they are posted because ibverbs copies them into an internal representation for the hardware.

The other task of the send thread is the notification of completed work requests. For this purpose, the completion channel belonging to the connection is monitored using the epoll file descriptor of the send thread. As soon as a work completion is generated for a connection, the corresponding send thread is woken up. At this point it starts polling the completion queue of the associated connection and notifies the application of each completed work request. After processing is complete, the send thread waits for further notifications using the `epoll_wait` call.

D. Receive request processing

Just like the execution of outgoing requests, the receipt of messages requires the creation of work requests. Within these work requests the registered memory area in which data is received is referenced. It is important to provide large enough buffers so that the InfiniBand hardware is able to process incoming messages. For example, it is not sufficient to post several small buffers to receive one large message, because the hardware consumes exactly one work request for each incoming message. Likewise, work requests must also be provided to the hardware in order to receive messages, otherwise the network controller does not know in which memory areas it should place the incoming data. In case no work request is provided or the memory region is not large enough, the network controller of the receiving side sends a so-called RNR (receiver not ready) NACK, whereupon the sender waits a certain time until the message is transmitted again. This can lead to a severe drop in performance.

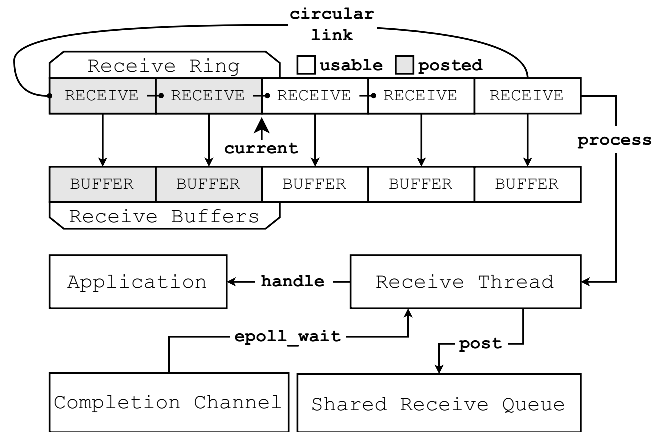

Fig. 8. Processing of incoming messages using a circular linked list of pre-allocated work requests.

Similar to the concept of the send ring owned by each connection, the receive thread creates a data structure, which bundles work requests and their corresponding buffers for the received data. We call this data structure the *receive ring* (see Figure 8). Within the receive ring, all work requests are connected to their successor and the last to the first. These preallocated work requests are later used for receiving messages.

In normal mode, work requests for receiving messages as well as for sending messages are posted to the queue pair assigned to the connection. To avoid having to fill each queue pair individually with new work requests for receiving messages, ibverbs provides the *shared receive queue*. It can be assigned to several queue pairs, whereupon these can consume the work requests on it collectively when receiving messages. This helps to reduce the total number of work requests on the recipient side. Each receive thread creates its own shared receive queue, which is associated with its assigned connections. Since a connection is associated with exactly one receive thread, it can therefore fill the shared receive queue assigned to it when it receives work completions.

The handling of incoming messages is implemented in a way in which the shared receive queue is refilled as quickly as possible, because missing work requests can lead to the before-mentioned loss of performance. Similar to the send thread, the receive thread first waits for new notifications regarding new work completions via the `epoll_wait` call. After a notification is received, the work completions on the completion queue belonging to the connection are polled but not yet processed. Immediately after polling, the number of existing work completions is determined and the same number of work requests are refilled in the shared receive queue.

This is done by maintaining an index within the receive ring, which indicates the position from which new work requests can be used. Starting from this index, the index of the last work request needed for the required amount is calculated and the connection to its successor is removed. The resulting list of work requests is then passed to the shared receive queue for consumption. After posting the list of work requests, the connection of the last element to its successor is restored and the index of the next free work request is set to this successor. To guarantee that the shared receive queue can always be completely filled, we choose twice the capacity of the shared receive queue as the size of the receive ring. It should also be noted that this data structure does not require synchronization since it is only used within the receive thread. As a last step, the receive thread calls a handler function of the application to notify it of the incoming messages.

V. Evaluation

To give an idea of what is possible with neutrino, we examine the system for different aspects with the help of implemented benchmarks. We present results on messaging and operations on remote memory using one or two connections. In case of two connections, the mentioned operations (sending messages or remote memory access) are performed concurrently using separate threads.

CPU	Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz (15 MB Cache)
RAM	4x Samsung 16GB DDR4-2400 CL17
NIC	Mellanox Technologies MT27500 Family [ConnectX-3] (56Gbit/s)

Fig. 9. System specifications of the hardware used in all experiments.

Within each experiment, two nodes are used equipped with the hardware shown in Figure 9. Each node uses two send and two receive threads. Since, to the best of our knowledge, no other system provides such an abstraction layer over the ibverbs library as neutrino, we cannot make a comparison with other systems at this point. The systems mentioned at the beginning of this paper are designed to work by putting the user in control of posting work requests and handling work completions directly while our system accepts buffers and automatically creates work requests for them. In our opinion a comparison would therefore not be meaningful.

A. Messaging

In our messaging benchmark we measure the average number of messages sent per second, the average network throughput achieved and the average latency per message in microseconds. To determine the throughput, a message of fixed size ranging between 16 bytes and 4 kilobytes is created per connection and then sent continuously over the network. The number of messages to be sent was set to one million. In addition, this number of messages is sent in several runs, so we have several measurements for each message size. We choose 10 runs for warmup and 30 runs for measurement. The warmup runs are necessary because the Java Virtual Machine analyzes and optimizes the executed code at runtime. To provide a suitable long time for the analysis we use the warmup runs. Within each measurement run, the time between sending the specified number of messages and the arrival of all corresponding work completions is measured. In case of two connections, the number of messages is divided between both connections and the time between sending the messages and receiving all work completions is waited on both connections. Using the measured time of a run, we then calculate how many messages were sent in this run and what network throughput this corresponds to.

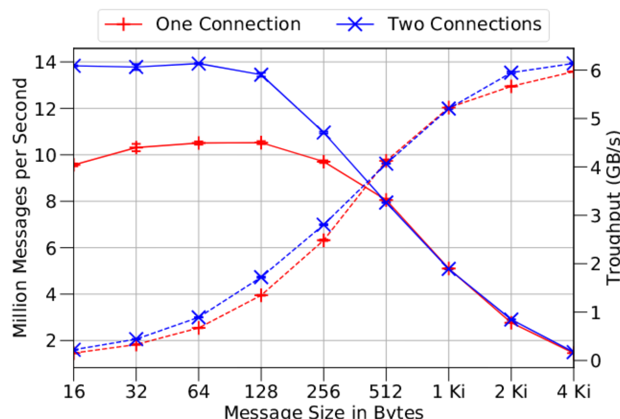

Fig. 10. Average message (solid line) and network (dashed line) throughput by message sizes using one and two connections.

As can be seen in Figure 10, neutrino achieves an average message throughput of about 10 million messages per second using a message size up to 256 bytes with one connection. When using two connections working in parallel, up to a message size of 128 bytes, an average of about 14 million messages per second is possible. This shows that the parallel processing of connections by multiple send and receive threads can result in a big improvement for small message sizes. From a message size of 512 bytes on, the use of one and two connections are almost equal regarding average messages sent per second. Only in terms of network throughput there is still an improvement in the area of larger messages between 2 and 4 kilobytes.

Similar to message throughput, latency is also measured by sending multiple messages using one and two connections. We use the same number of messages as well as warmup and measurement runs like in the previous experiment. The difference here lies in the measurement of time. Instead of waiting for the work completions of all messages, the benchmark waits for the corresponding work completion for each individual message and connection until the next message is sent and measures the time between these two events. Figure 11 shows the results as average values for different

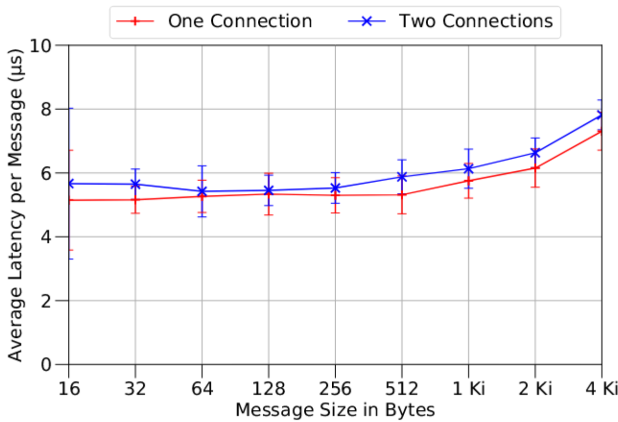

Fig. 11. Average latency by message sizes using one and two connections.

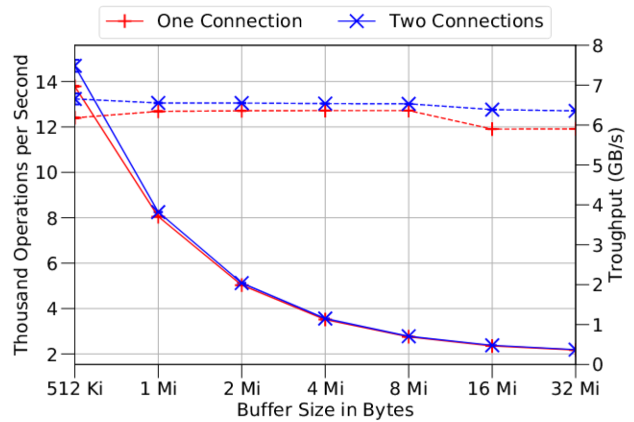

Fig. 12. Average read operation (solid line) and network (dashed line) throughput by buffer sizes using one and two connections.

message sizes. In contrast to a single connection, the latency increases minimally when using two connections. Overall, the latency remains below 6 microseconds in both cases up to a message size of 512 bytes. InfiniBand hardware is known to deliver latencies below 2 microseconds. However, this can only be achieved if the processing of messages is done by active polling to minimize latency. Neutrino, on the other hand, uses Linux's IO multiplexing system `epoll`, which introduces additional latency by notifying threads and using system calls for such purposes. This therefore explains the increased latencies within our experiments.

B. Remote memory access

Exchanging memory between two nodes is one of neutrino's core functions and should therefore work reliably and fast. For this purpose we implement a second experiment showcasing the average operation (`RDMA_READ` or `RDMA_WRITE`) throughput as well as the average network throughput. As in the Messaging Benchmark, all measurements are collected in several runs, consisting of 10 warmup phases and 30 measurement phases. In each phase a buffer ranging from 512 kilobytes to 32 megabytes is read or written 100 times by means of remote memory access. In this experiment the two nodes are divided into the roles of an initiator and a responder. The initiator first asks the responder to create a buffer of sufficient size via messaging. The responder then returns the information necessary for remote memory access (virtual address and access key) to the initiator by sending a message. After receiving this information from the responder, the benchmark proceeds similarly to the messaging benchmark. The received information is used to continuously execute remote read or write accesses. The time until completion of all operations within each run is also measured based on the received work completions.

Figure 12 shows the measured results as the average number of send operations per second and the network throughput using remote read accesses. Up to a buffer size of 8 megabytes, the average network throughput always remains above 6 GB/s when using a single connection. After this, the average throughput drops to just under 6 GB/s. In contrast, the average network throughput remains relatively stable at all buffer sizes when using two connections. This shows that the use of two connections is more suitable for accessing larger amounts of data through reading remote memory. In terms of the average number of operations per second, the use of one and two connections is quite similar and there are hardly any differences.

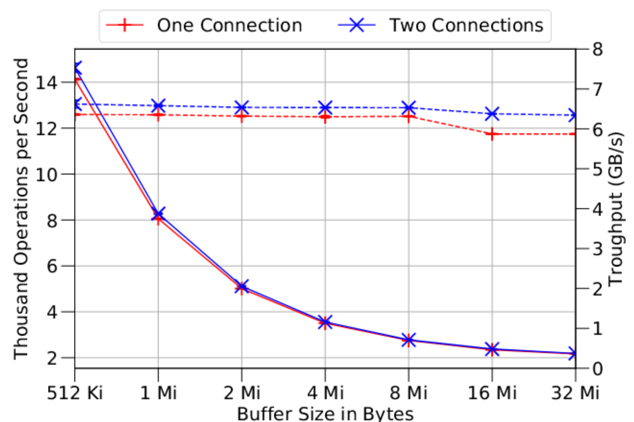

Fig. 13. Average write operation (solid line) and network (dashed line) throughput by buffer sizes using one and two connections.

With respect to the results shown in Figure 13 regarding the average write operations per second as well as the average network throughput when using remote write accesses, it can be said that they perform similarly well as read accesses. Using two connections, the same more stable average network throughput can be observed as with read accesses. Remote

write accesses can thus also be used to exchange large amounts of data between two nodes.

VI. Conclusion

The Java Development Kit and the Java Virtual Machine do not yet offer an official solution to use InfiniBand hardware for the implementation of network applications. In this paper we propose neutrino, a system aiming at providing efficient means for accessing InfiniBand hardware from Java space through usage of the Java Native Interface as well as building an abstraction layer above the native ibverbs library. This system works in a multithreaded non-blocking fashion using thread pools for performing work and grants users access to messaging and remote direct memory access functionalities through a simple programming interface. Examples for the usage of our system can be found in the public GitHub repository.[16]. Our experiments show that neutrino is well suited for use with InfiniBand hardware and reaches saturation in case of network throughput of remote memory accesses. When sending small messages we can also show that neutrino benefits from the multithreading architecture and with its help reaches up to about 14 million messages per second on average. In summary, it can be said that the use of InfiniBand hardware within the Java Virtual Machine is quite possible and practical in terms of performance and usability as shown within our experiments and examples.

VII. Outlook

In our future work we plan on focusing neutrino on the use with Apache Arrow [17], which provides an platform independent columnar memory format for representing in-memory data sets. Since each column's data is stored in contiguous memory areas, they are very well suited for remote memory accesses. In the long term, we hope to enable integration with Apache Flight [18], which is designed to transport Arrow in-memory data. The core idea is to implement control messages for looking up data locations via messaging and the retrieval of the actual data via remote memory accesses. We assume that applications which transfer and process large amounts of data should benefit greatly from these efforts.

References

[1] *TOP500 Supercomputer Sites*. [Online]. Available: https://top500.org (visited on 04/15/2020).

[2] I. Hashem, I. Yaqoob, N. Anuar, S. Mokhtar, A. Gani, and S. Khan, "The rise of "Big Data" on cloud computing: Review and open research issues," *Information Systems*, vol. 47, pp. 98–115, Jul. 2014. DOI: 10.1016/j.is.2014.07.006.

[3] S. Mehta and V. S. Mehta, "Hadoop Ecosystem : An Introduction," 2016.

[4] J. Kreps, "Kafka : a Distributed Messaging System for Log Processing," 2011.

[5] P. Stuedi. "Direct Storage and Networking Interface (DiSNI)." (2018), [Online]. Available: https://developer.ibm.com/open/projects/direct-storage-and-networking-interface-disni (visited on 04/15/2020).

[6] S. Nothaas, K. Beineke, and M. Schöttner, "Leveraging InfiniBand for Highly Concurrent Messaging in Java Applications," *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pp. 74–83, 2019.

[7] P. Stuedi, B. Metzler, and A. Trivedi, "JVerbs: Ultra-Low Latency for Data Center Applications," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, Santa Clara, California: Association for Computing Machinery, 2013, ISBN: 9781450324281. DOI: 10.1145/2523616.2523631.

[8] K. Beineke, S. Nothaas, and M. Schöttner, "Efficient Messaging for Java Applications Running in Data Centers," *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 589–598, 2018.

[9] *DiSNI GitHub repository*. [Online]. Available: https://github.com/zrlio/disni (visited on 04/15/2020).

[10] W. Huang, H. Zhang, J. He, J. Han, and L. Zhang, "Jdib: Java Applications Interface to Unshackle the Communication Capabilities of InfiniBand Networks," in *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, 2007, pp. 596–601.

[11] D. Kurzyniec and V. Sunderam, "Efficient cooperation between Java and native codes–JNI performance benchmark," Jan. 2001.

[12] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at Your Own Risk: The Java Unsafe API in the Wild," *SIGPLAN Not.*, vol. 50, no. 10, pp. 695–710, Oct. 2015, ISSN: 0362-1340. DOI: 10.1145/2858965.2814313.

[13] *RDMA communication manager*. [Online]. Available: https://www.ibm.com/support/knowledgecenter/ssw_aix_72/communicationtechref/rdma_cm.html (visited on 04/15/2020).

[14] *Agrona ManyToManyConcurrentArrayQueue*. [Online]. Available: https://github.com/real-logic/agrona/blob/master/agrona/src/main/java/org/agrona/concurrent/ManyToManyConcurrentArrayQueue.java (visited on 04/16/2020).

[15] *Agrona RingBuffer*. [Online]. Available: https://github.com/real-logic/agrona/blob/master/agrona/src/main/java/org/agrona/concurrent/ringbuffer/RingBuffer.java (visited on 04/16/2020).

[16] *Neutrino github*. [Online]. Available: https://github.com/hhu-bsinfo/neutrino (visited on 06/21/2020).

[17] *Apache Arrow Explained by Dremio*. [Online]. Available: https://www.dremio.com/apache-arrow-explained (visited on 04/19/2020).

[18] W. McKinney. "Introducing apache arrow flight: A framework for fast data transport." (2019), [Online]. Available: https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight (visited on 04/19/2020).

Performance analysis and evaluation of Java-based InfiniBand Solutions

Fabian Ruhland
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
fabian.ruhland@hhu.de

Filip Krakowski
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
filip.krakowski@hhu.de

Michael Schöttner
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—Low-latency network interconnects, such as InfiniBand, are widely used in HPC centers and are becoming available in public cloud offerings, too. For MPI applications accessing InfiniBand is transparent, but many big-data applications are written in Java, which does not provide direct access to InfiniBand networks, but relies on thid-party libraries. In this paper, we present *Observatory*, a benchmark for evaluating *low-level* libraries, providing InfiniBand access for Java applications. Observatory can be used for evaluating and comparing socket- and verbs-based libraries regarding throughput and latency. With transparency often traded for performance and vice versa, the benchmark helps developers with studying the pros and cons of each solution and supports them in their decision which solution is more suitable for their existing or new use-case. We also give an overview of existing and maintained InfiniBand libraries for Java and evaluate them with the proposed benchmark.**

***Index Terms*—High-speed Networks, InfiniBand, Remote Direct Memory Access**

I. Introduction

RDMA capable devices have been providing high throughput and low-latency to HPC applications for several years [13]. With todays cloud providers offering instances equipped with InfiniBand (IB) for rent, such hardware is available to a wider range of users without the high costs of buying and maintaining it [18]. Many big data frameworks these days are written in Java, e.g. batch processing frameworks [23], databases [1] or backend storages/caches [5].

These applications benefit from the rich environment Java offers, including automatic garbage collection and multi-threading utilities. But, the choices for inter-node communication on distributed applications are limited to Ethernet-based socket-interfaces (standard ServerSocket or NIO) on the commonly used JVMs OpenJDK and Oracle. They do not provide support for low-latency IB hardware. However, there are external solutions available each with pros and cons.

This raises questions if a developer wants to choose a suitable solution for a new use-case or an existing application: What's the throughput/latency on small/large payload sizes? Is the performance sufficient when trading it for transparency requiring less to no changes to the existing code? Is it worth considering developing a custom solution based on the native API to gain maximum control with chances to harvest the performance available by the hardware?

In this paper, we address these questions by proposing the *Observatory* benchmark to evaluate existing libraries to leverage the performance of IB hardware in Java applications. Existing benchmark tools like iperf [6] for TCP/UDP or the ibperf included in the OFED package [11] do not support Java libraries. Observatory has a modular design and currently supports implementations to evaluate four verbs-based libraries (ibverbs, jVerbs, DiSNI and neutrino), as well as socket-based implementations, of which we evaluated IP over IB, libvma and JSOR. This paper focuses on the fundamental performance metrics of low-level interfaces and *not* on higher-level network subsystems with connection management, complex pipelines and messaging primitives like for example provided by MPI. The proposed benchmark is used to evaluate the libraries mentioned above with 56 Gbit/s IB NICs. The contributions of this paper are:

- An overview of existing Java IB solutions
- The design and implementation of Observatory, an extensible and open-source benchmark to evaluate Java-based IB solutions
- Evaluation results using Observatory

The paper is structured as follows: Section II discusses related work, Section III presents existing IB solutions with socket-based (§III-A) and verbs-based (§III-B) libraries. Section IV presents the Observatory benchmark, followed by Section V with the evaluation results. Conclusions are presented in Section VI.

II. Related work

Java networking performance with and without IB networks has been evaluated in literature. However, to the best of our knowledge, there is no benchmark aiming at comparing both socket- and verbs-based libraries for Java.

In 2007 *Jnetperf* has been implemented analog to the netperf utility to evaluate Gigabit Ethernet and 20 Gbit/s IB in Java [31]. Jnetperf and netperf were then used to analyze the throughput and round-trip latency achievable in Java and native applications with TCP/IP, IP over IB and the now discontinued Socket Direct Protocol. Regarding latency, the native ibverbs API has also been evaluated to set a baseline for the remaining solutions. While insightful results could be achieved with Jnetperf, many new solutions for leveraging IB

in Java have been developed since then. Especially in the field of making ibverbs available in the JVM, several attempts were made, which will be evaluated in this paper.

In 2012 Vienne et al. evaluated IB and RoCE (RDMA over Converged Ethernet) for HPC and Cloud Computing scenarios [30]. While they evaluated raw network level performance, their main focus was on MPI and the impact, that different hardware solutions have on middleware applications in the cloud. With Observatory, we focus solely on network level performance and solutions that work on the network level, instead of the application level.

In 2014 Ekanayake et al. have shown, that MPI performance in Java has vastly improved over the preceeding years and concluded, that the gap between Java and native performance is decreasing continuosly [15]. However, their focus was completely on MPI, which is not what we intend to evaluate with our benchmark.

III. InfiniBand libraries

This section elaborates on existing *low-level* solutions/libraries that can be used to leverage the performance of InfiniBand hardware in Java applications. This does not include network or messaging systems, implementing higher-level primitives such as the Message Passing Interface, e.g. Java-based FastMPJ [16] providing a special transport to use InfiniBand hardware.

A. Socket-based libraries

The socket-based libraries redirect the send and receive traffic of socket-based applications transparently over InfiniBand host channel adapters (HCAs) with or without kernel bypass depending on the implementation. Thus, existing applications do not have to be altered to benefit from improved performance due to the lower latency hardware compared to commonly used Gigabit Ethernet. The following three libraries are still supported to date and evaluated in Section V.

IP over InfiniBand (IPoIB) [20] is not a library but actually a kernel driver that exposes the InfiniBand device as a standard network interface (e.g. *ib0*) to the user space. Socket-based applications do not have to be modified but use the specific interface. However, the driver uses the kernel's network stack which requires context switching (kernel to user space) and CPU resources when handling data. Naturally, this solution trades performance for transparency.

libvma [7] is a library developed by Mellanox and included in their OFED software package [8] and is preloaded to any socket-based application (using *LD_PRELOAD*). It enables full bypass of the kernel network-stack by redirecting all socket-traffic over InfiniBand using unreliable datagram with native ibverbs. Again, the existing application code does not have to be modified to benefit from increased performance.

Java Sockets over RDMA (JSOR) [29] redirects all socket-based data traffic in Java applications using native verbs, similar to libvma. It uses two paths for implementing transparent socket streams over RDMA devices. The "fast data path" uses native verbs to send and receive data and the "slow control path" manages RDMA connections. JSOR is developed by IBM and only available in their proprietary J9 JVM.

The following libraries are also known in literature but are not supported or maintained anymore.

The **Sockets Direct Protocol (SDP)** [17] redirects all socket-based traffic of Java applications over RDMA with kernel-bypass. It supported all available JDKs since Java 7 and was part of the OFED package until it was removed with OFED version 3.5 [10].

Java Fast Sockets (JFS) [28] is an optimized Java socket implementation for high speed interconnects. It avoids serialization of primitive data arrays and reduces buffering and buffer copying with shared memory communication as its main focus. However, JFS relies on SDP (deprecated) for using InfiniBand hardware.

B. Verbs-based Libraries

Verbs are an abstract and low-level description of functionality for RDMA devices (e.g. InfiniBand) and how to program them. Verbs define the control and data paths including RDMA operations (write/read) as well as messaging (send/receive). RDMA operations allow reading or writing directly from/to the memory of the remote host without involving the CPU of the remote. Messaging follows a more traditional approach by providing a buffer with data to send and the remote providing a buffer to receive the data to.

The programming model differs heavily from traditional socket-based programming. Using different types of asynchronous queues (send, receive, completion) as communication endpoints. Applications use different types of work requests to send and receive data. When handling data transfers, all communication with the HCA is executed using these queues. The following libraries are verbs implementations that allow programming RDMA capable hardware directly. The first four libraries presented are evaluated in Section V.

ibverbs are the native verbs implementation included in the OFED package [11]. Using the Java Native Interface (JNI) [21], this library can be utilized in Java applications as well in order to create a custom network subsystem [16] [24]. Using the Unsafe class [22] or Java DirectByteBuffers, memory can be allocated off-heap to use it for sending and receiving data with InfiniBand hardware (buffers must be registered with a protection domain which pins the physical memory).

jVerbs [27] is a proprietary verbs implementation for Java, developed by IBM for their J9 JVM. Using a JNI layer, the OFED ibverbs implementation is accessed. "Stateful verb methods" (*StatefulVerbsMethod* Java objects) encapsulate the verb to call including all parameters with parameter serialization to native space. Once the object is prepared, it can be executed, which actually calls the native verb. These objects are reusable for further calls with the same parameters, to avoid repeated serialization and creating new objects which would burden garbage collection. However, if a program works with constantly changing buffer addresses, thus calling verbs with different parameters, repeated serialization is inevitable.

DiSNI [26] is an open source solution based on jVerbs [2]. It utilizes the same “Stateful verb method” mechanism as jVerbs.

neutrino [9] is our own approach at making verbs accesible from within the JVM. It allows structured access to native structures with automatically generated *proxy objects* in Java space. This allows manipulating native structures and calling native methods without any form of serialization or copying. neutrino aims to be more flexible than jVerbs and DiSNI, while still offering high throughput rates and low latency.

Jdib [19] is a library wrapping native ibverbs function calls and exposing them to Java using a JNI layer. According to the authors, various methods, e.g. queue pair data exchange on connection setup, are abstracted to create an easier to use API for Java programmers. The fundamental operations to create protection domains, create and setup queue pairs, as well as posting data-to-send to queues and polling the completion queue seem to wrap the native verbs and do not introduce additional mechanisms like jVerbs's stateful verb methods. We were not able to obtain a copy of the library for evaluation.

IV. OBERVATORY BENCHMARK

In this section we describe the architecture and implementation aspects of the Observatory benchmark which aims at allowing to compare different Java-based IB solutions (§III) with each other, as well as comparing them to C-based libraries. The latter include the ibverbs library to provide a baseline for performance measurements.

A. Communication patterns

Observatory aims at evaluating a fundamental point-to-point connection regarding throughput and latency. Like other benchmarks (e.g. OSU [12]), we want to determine the maximum throughput on unidirectional and bidirectional communication (e.g. application pattern asynchronous “messaging”), as well as one-sided latency and full round-trip-time (RTT) with a ping-pong communication pattern (e.g. application pattern “request-response”). These communication patterns are commonly used to evaluate network hardware or applications [6], [11], [12] and allow us to determine the fundamental performance of a Java-based IB library. Complex communication patterns, like for example all-to-all and multi-threading are planned, but not implemented so far.

B. Architecture

The work on Observatory began as continuation of our *Java InfiniBand Benchmark* [25], which consisted of multiple standalone micro benchmarks for each library. Our goal with Observatory is to develop a coherent benchmark architecture for Java libraries and C/C++ solutions, see Fig. 1. This led us to an architecture, that is easier to extend and results in less duplicate code compared to the *Java InfiniBand Benchmark*. The benchmark needs to support two programming languages (C and Java) and two programming models (sockets and verbs), as well as two different forms of network communication (messaging and RDMA). The benchmark provides a flexible interface with default implementations for standard message passing and RDMA operations, so it is not necessary to always implement all methods for each library.

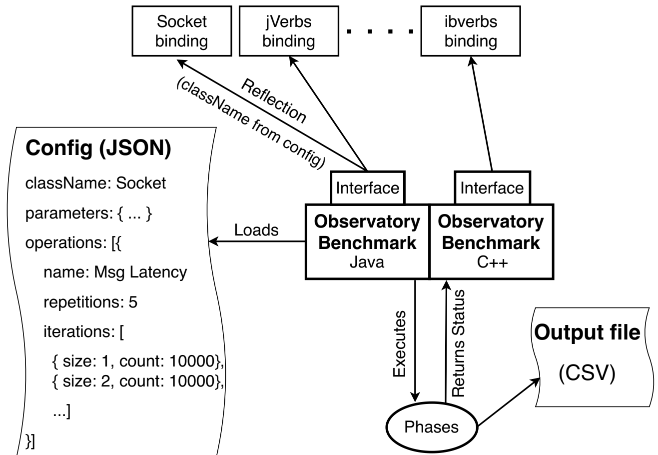

Fig. 1: Observatory architecture design

Observatory can be configured through a JSON file, including the communication pattern (uni-/bidirectional throughput, latency, etc.), buffer sizes, the number of repitions, and a potential warmup phase.

C. Benchmark phases

Each benchmark run is made up of the following six phases, which call methods that need to be implemented by each library binding:

1) *Initialization*. During this phase, the client should allocate any needed resources (e.g open an IB context and allocate a Protection Domain). Client-specific configuration parameters, that are defined in the configuration file (JSON), are passed to the client as key-value tuples. This can be used to pass IB related parameters to the client (e.g. the device and port number).
2) *Connection*. A connection is setup, after IB connection information has been exchanged (e.g. via TCP sockets).
3) *Preparation*. The operation size, which dictates the size of the messages being sent, respectively the size of RDMA writes/reads being performed, is passed to the client, allowing it to allocate matching buffers to use in the benchmark. It is also reasonable to preallocate reusable Work Requests during this phase.
4) *Warmup*. A configurable amount of operations are executed as a warmup, allowing the JVM and its JIT to optimize the benchmark code.
5) *Operation*. This is the main phase of the benchmark, executing the configured amount of operations. If a bidirectional benchmark run is performed, dedicated threads for sending and receiving are started. If a throughput benchmark is being performed, two timestamps will be taken right before the first operation starts and right after the last one has finished. Otherwise, if a latency measurement is performed, the time needed for each

operation is measured and stored in an array. This allows calculating percentiles afterwards.
Furthermore, the benchmark utilizes the performance counters of the IB HCA to determine the raw amount of data being sent/received. This enables us to calculate the overhead added by any software defined protocol which is especially relevant for the socket-based libraries (§V-B).

6) *Cleanup*. The benchmark is finished, resources shall be freed and all connections shall be closed.

The benchmark automatically fills up the receive queue before the warmup and operation phases in order to avoid *Receiver Not Ready* (RNR) timeouts, which would force the sender to wait for a short amount of time, before retrying to send a message.

After a benchmark run has finished successfully, the measured results are appended to a CSV-file, which can later be plotted with a Python script, that is bundled with Observatory.

V. Evaluation

In this Section, we present the evaluation results using Observatory (§IV). An overview of all experiments is shown in the following Table I.

Library/Benchmark	OV	Unidir	Bidir	Lat	PingPong
ibverbs RDMA write		x	x	x	
ibverbs messaging	x	x	x	x	x
jVerbs RDMA write		x	x	x	
jVerbs messaging	x	x	x	x	x
DiSNI RDMA write		x	err	x	
DiSNI messaging	err	err	err	err	err
neutrino RDMA write		x	x	x	
neutrino messaging	x	x	x	x	x
IPoIB messaging	x	x	x	x	x
JSOR messaging	x	x	err	x	x
libvma messaging	x	x	x	x	x

TABLE I: Overview of all experiments; OV = overhead.

The verbs-based libraries showed similar behaviors regarding RDMA write and read, so that no additional insights could be gained by analyzing both. For this reason, we decided to only discuss RDMA write results.

In the following text we use the terms “operation” (op) and “message” (msg) for referring only to the payload, excluding overhead of the network protocols. Each throughput focused benchmark run executes 100 million operations and each latency focused benchmark run executes 10 million operations. Starting with 8 KiB payload size, the amount of operations is incrementally halved to avoid unnecessary long running benchmark runs. We evaluated payload sizes of 1 byte to 1 MiB in power-of-two increments. When discussing the results, we focus on the operation rate on small operations, with payload sizes less than 1 KiB and on the throughput on middle sized and large operations, starting at 1 KiB.

The throughput results are depicted as line plots with the left y-axis showing the throughput in million operations per second (Mop/s) and the right y-axis showing the throughput in GB/s. For the latency results, the left y-axis shows the latency in µs and the right y-axis the throughput in Mop/s. The dotted lines always represent the operation throughput while the solid lines represent either the throughput in GB/s or the latency in µs, depending on the benchmark. For the overhead results, a single y-axis describes the overhead in percentage in relation to the amount of payload transferred on a logarithmic scale. On all plot types, the x-axis depicts the size of the payload in power-of-two increments from 1 byte to 1 MiB. Each benchmark run was executed five times and the average is used to depict the graph, while the error bars visualize the standard deviation.

A. Configuration

We ran all experiments on two servers with the following hardware: Intel Xeon CPU E5-1650 v3 @ 3.50GHz (6 cores, 12 threads), 64 GB RAM, Mellanox ConnectX-3 HCA, 56 Gbit/s IB (Link width 4x), MTU size 4096. Both nodes run CentOS 8.1 with the Linux Kernel version 4.18.0-151
The software used included OpenJDK 11.0.6, IBM SDK 8.0.6.6 with the J9 JVM 2.9, rdma-core v28.0, libvma 9.0.2, gcc 8.3.1.

libvma. Flow steering must be activated for libvma to redirect all traffic over IB, by setting the parameter *log_num_mgm_entry_size* to *-1* in the configuration file */etc/modprobe.d/mlnx.conf* for the IB kernel module. Otherwise, libvma falls back to sockets over Ethernet.

JSOR. For JSOR, we set the send and receive buffer sizes to 1 MiB, to avoid hanging connections [3]. However, the bidirectional throughput benchmark did not terminate for buffer sizes greater than 32 KiB. Furthermore, sudden disconnects occurred for buffer sizes smaller than 512 byte. This seems to be a known problem [4], but increasing the send and receive queue size did not solve this issue.

DiSNI. There seems to be a problem with memory management in DiSNI, which causes the JVM to crash during the benchmark. When looking at the stacktrace after a crash, we observed that the last method call was either a *malloc()* or *free()*. We tried to compile and run the benchmark with different JDKs/JVMs (OpenJDK 8, OpenJDK 11, IBM SDK 8), but the problem could not be fixed. The crashes did not occur after a certain amount of operations or time. However, most of the times, we were not able to run Observatory with DiSNI for more than a few minutes. The only benchmark type that finished successfully was the unidirectional RDMA write benchmark.

B. Overhead

In this Section, we present the results of the overhead measurements of the described libraries/implementations. As overhead, we consider the additional amount of data that is sent along with the payload data of the user. This includes any data of any network layer down to the HCA. We measured the amount of data emitted and received by the port using the performance counters *port_xmit_data* and *port_rcv_data* of the HCA. These counters contain the amount of byte sent/received per lane, which means the values need to be multiplied with the link width to get the correct amount of data. The cards in

our test systems have a link width of 4, leading to a granularity of 4 byte for the measured values.

IPoIB and libvma implement buffer/message aggregation when sending data, which allows increasing throughput and reducing overhead when sending many small messages in a row. However, in order to determine the general per message overhead, we used the pingpong benchmark which does not allow aggregation due to its nature. The results of both types (sockets/verbs) are shown in Fig. 2. Since all verbs-based libraries generate the identical amount of overhead, we only include the results of ibverbs.

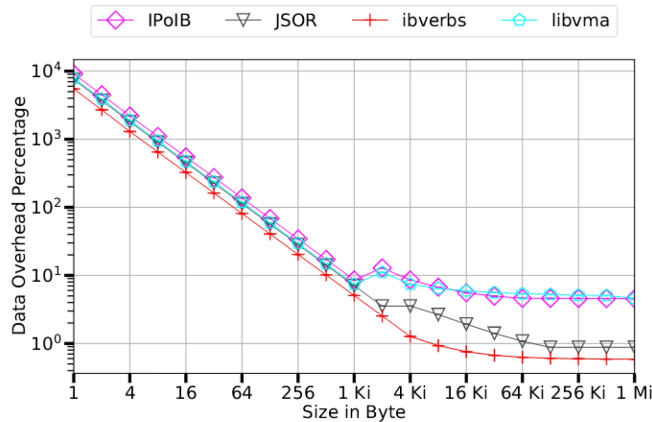

Fig. 2: Avg. overhead (%) in relation to the payload size.

We try to give a rough breakdown of the overhead involved with each method evaluated. A precise breakdown is rather difficult with just the raw amount of data captured from the ports as re-transmission of packages are also captured (e.g. RC queue pairs or custom protocols based on UD queue pairs).

The results in Fig. 2 show that the overhead for messaging operations of verbs-based libraries is 5500%, connotating that for a single byte of payload in each of the ping and pong messages, a total of 112 byte are sent and received (2 byte payload, 110 byte overhead). When using the RC protocol each package starts with a local routing header (8 byte), followed by a base transport header (12 byte) and ends with an invariant CRC (4 byte), as well as a variant CRC (2 byte) [14], which makes a total of 26 byte of metadata. Sent packages must be acknowledged, typically one ACK/NACK for multiple messages. Each ACK/NACK message contains an additional acknowledge extended transport header, which is 4 byte long. Acknowledging multiple packages is however prevented by the ping-pong pattern used in the benchmark, see Fig. 3.

Each ping-pong iteration requires two messages to be sent and both of them need to be acknowledged, so that the total amount of metadata sums up to $2 * 26$ byte $+ 2 * 30$ byte $=$ 112 byte. The 2 byte of payload are still missing, probably due to the 4 byte granularity of the hardware counters. The overhead stays constant, which leads to an overall decreasing per message overhead with increasing payload size. Starting with 1 KiB payload size the overhead drops below 10% and with 8 KiB below 1%.

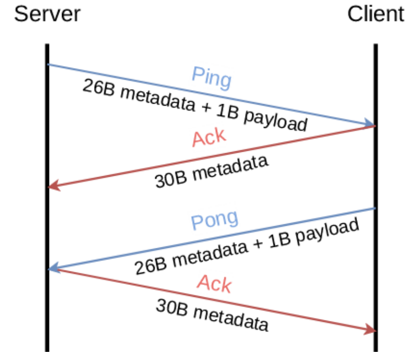

Fig. 3: Verbs-based ping-pong communication pattern.

The overhead of the socket-based solutions is overall slightly higher. Again, considering 1 byte messages, JSOR adds an additional overhead of 7500%, libvma 7900% and IPoIB 9100%. IPoIB and libvma rely on UD messaging verbs which add a datagram extended transport header (8 byte) to the IB header and include additional information to allow IP-address based routing of the packages. The IPoIB specification describes an additional header of 4 octets (4 byte) and IP header (e.g. IPv4 20 byte + 40 byte optional) which are added alongside the message payload [20]. libvma adds an IP-address (4 byte) and Ethernet frame header (14 byte) [7]. Remaining data is likely committed towards a software signaling protocol. Regarding JSOR, we could not find details of the protocol as it is closed source.

For IPoIB and libvma the overhead drops below 10% starting with message sizes of 4 KiB and decreases further to around 4%-6% with increasing message sizes. JSOR manages to keep the overhead below 10% starting with 1KiB messages and below 1% with 128 KiB, which is closer to the verbs-based libraries than IPoIB and libvma.

C. Unidirectional Throughput

This section presents the throughput results of the unidirectional benchmark. Starting with the messaging results depicted in Fig. 4, neutrino and ibverbs are mostly on par regarding operation throughput for small messages (< 1 KiB), with ibverbs having slight advantages but also showing higher signs of jitter. However, jVerbs is yielding very poor performance with only 6000 operations per second for message sizes of up to 4 KiB. Although we cannot provide results for DiSNI due to the stability issues we experienced V-A, we observed the same behavior as with jVerbs, during the few benchmark runs that would complete. Starting with 8 KiB messages, there is virtually no difference between the three libraries.

Looking at the socket-based libraries, we can see that on small payload sizes up to 64 byte, IPoIB achieves a throughput of approximately 1.1 Mop/s. With increasing payload size, the throughput stagnates at 128 KiB message size with 4.1 - 4.2 GB/s. The results of libvma show a highly increased throughput of 5.0 to 5.4 Mop/s for up to 64 byte messages. Overall throughput for middle and large sized messages ini-

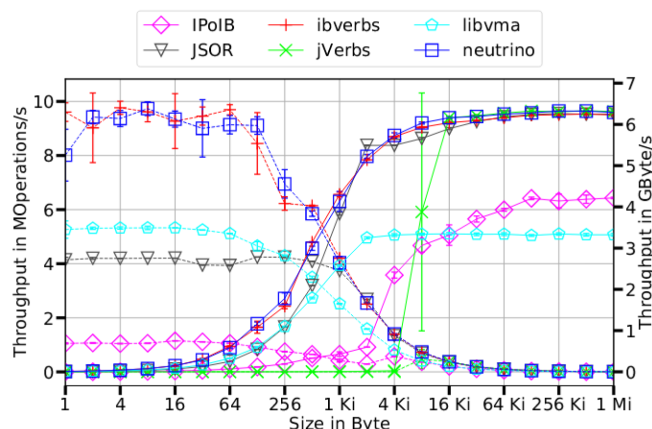

Fig. 4: Unidir. throughput (msg), increasing message size,

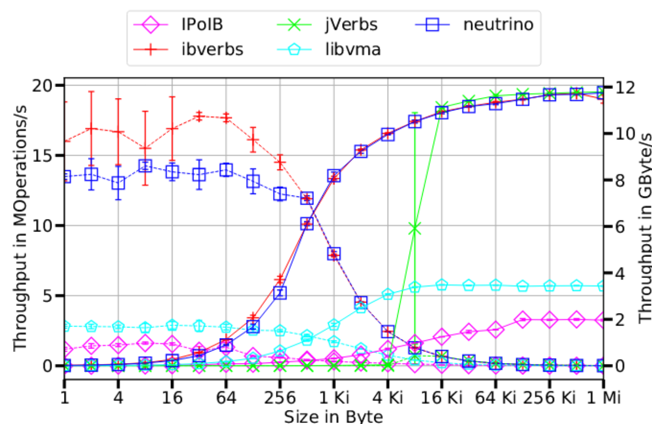

Fig. 6: Bidir. throughput (msg), increasing message size.

tially surpasses IPoIB's, but stagnates at 3.3 GB/s starting with 2 KiB messages. JSOR achieves a significantly lower throughput of 3.2 - 4.0 Mop/s for up to 256 byte messages. However, it provides a much higher throughput starting at 512 KiB message size compared to IPoIB and libvma. Throughput saturates at 64 KiB message size with approx. 6.2 GB/s, which is on par with the verbs-based libraries.

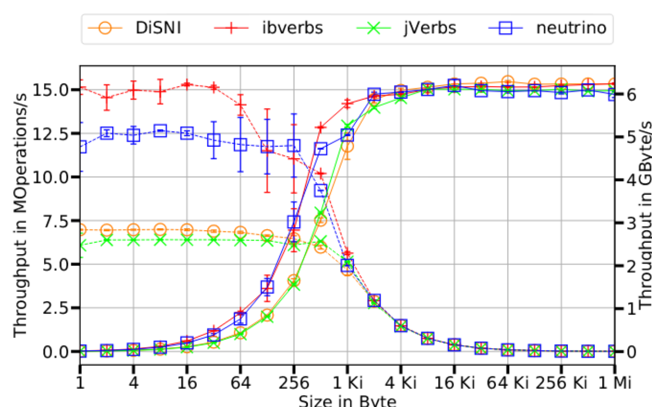

Fig. 5: Unidir. throughput (write), increasing buffer size.

The results in Fig. 5 show, that the RDMA write throughput of jVerbs and DiSNI (6.0 - 7.0 Mop/s) is less than half of ibverbs's RDMA write throughput (approximately 15.0 Mop/s) for small payload sizes up to 64 byte, with DiSNI yielding approximately 500 Kop/s more than jVerbs. However, neutrino achieves an operation rate much closer to ibverbs with 12.5 Mop/s. Starting with 128 byte, ibverbs's throughput abruptly decreases to 10 Mop/s with high jitter. For large sized buffers, all three libraries yield a similar throughput, saturating at 8 KiB with 6.0 GB/s for jVerbs and neutrino and 16 KiB with 6.2 GB/s for the other two libraries.

D. Bidirectional Throughput

This section presents the throughput results of the bidirectional benchmark. For full-duplex communication we expect roughly doubled throughput.

Fig. 6 depicts messaging results and as expected, all verbs-based implementations show an increased message rate on small messages and roughly double the throughput on large messages compared to the unidirectional results (§V-C). However, the socket-based libraries did not scale with two nodes, but even degraded in most cases, with JSOR not able to even finish the benchmark for all payload sizes (§V-A).

The message rate of ibverbs is roughly 17.5 Mop/s, though highly jittery, for payload sizes up to 64 Byte and constantly decreasing afterwards, saturating the bandwidth at 256 KiB with 11.7 - 11.8 GB/s. As with the unidirectional benchmark, jVerbs is again showing a very poor message rate (8000 Op/s) for payload sizes smaller than 8 KiB, but manages to achieve a high bandwidth, on par with ibverbs and neutrino, starting with 16 KiB messages. neutrino did not manage to fully double its message rate from the unidirectional results, but achieves a respectable 14.0 Mop/s for payload sizes up to 64 Byte. At 512 byte and onwards, it yields the same message rate and data throughput as ibverbs.

Regarding the socket-based libraries, libvma performs best with a message rate of roughly 2.7 - 3.0 Mop/s for payload sizes up to 256 byte, saturating at 16 KiB with a throughput of 3.4 - 3.5 GB/s. IPoIB shows a slightly improved message rate of 1.3 - 1.6 Mop/s for messages smaller than 128 byte, but does not manage to yield a throughput higher than 2 GB/s, stagnating at 128 KiB payload size.

Fig. 7 depicts the results of the bidirectional RDMA write benchmark, with ibvebs roughly doubling its operation rate to 30.0 Mop/s for payload sizes up to 64 byte. The operation rates of neutrino and jVerbs have not fully doubled with 20.0 - 21.0 Mop/s and 11.0 - 11.1 Mop/s respectively for buffer sizes smaller than 512 byte. Starting at 4 KiB, there is virtually no difference between the three libraries, with saturation reached at 32 KiB and 11.8 GB/s.

E. One-sided latency

Next we are evaluating the latency of a single operation. Section V-F further discusses full RTT latency for a ping-pong communication pattern. Results are separated by socket-based

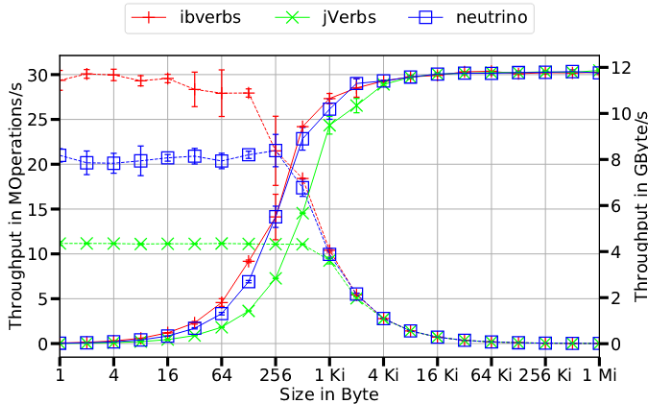

Fig. 7: Bidir. throughput (write), increasing buffer size.

and verbs-based. Due to space constraints, we try to limit the discussion to the most interesting values, and only depict the 99.99th percentiles for the verbs-based and the average values for the socket-based libraries.

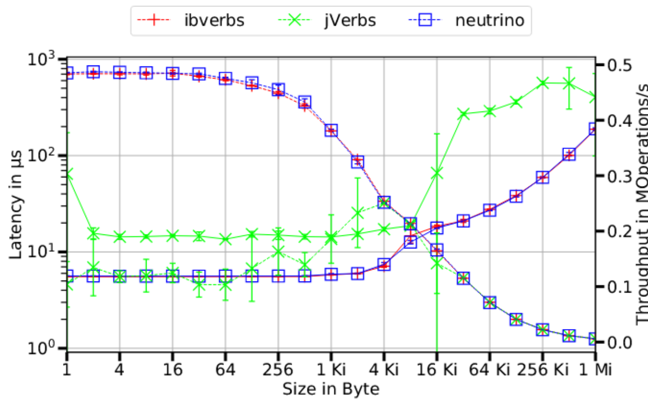

Fig. 8: 99.99% latency (msg), increasing message size.

The average latency of ibverbs and neutrino is on par at 2.0 µs for message sizes up to 256 byte, with ibverbs only showing a slight advantage of less than 0.1 µs. However, jVerbs shows unexpected average latency results. Up to 4 KiB message size, which equals the used MTU size, the latency is high and fluctuating at approx. 7 - 11 µs with high signs of jitter. At 4 KiB and beyond it equals the average latencies of the other transfer methods.

To further analyze this issue, we looked at the 99.9th and 99.99th percentiles. While ibverbs and neutrino are showing expected behavior and overall low latency regarding the 99.9th percentiles, jVerbs is now on par with them. However, looking at the 99.99th percentiles (i.e. 1000 worst out of 10 million, depicted in Fig. 8), jVerbs is again showing poor results, even for large payload sizes, indicating that only a small amount of messages yield high latencies, raising the average latency results. The maximum latency (i.e. single worst out of 10 million) for buffer sizes up to 1 KiB is extraordinary high, with 2.13 seconds, confirming our assumption. However, ibverbs and neutrino are still showing similar and stable results of 5.5 - 6.0 µs for small messages.

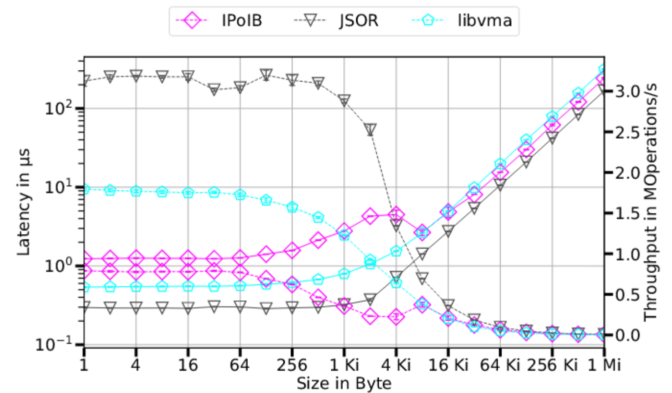

Fig. 9: Avg. latency (msg), increasing message size.

For the socket-based solutions, the average latencies in Fig. 9 show that JSOR performs best with an average per operation latency of roughly 0.3 µs for up to 1 KiB messages. With further increasing payload size, latency increases as expected. libvma shows similar results with a slightly higher latency of 0.5 - 0.9 µs for small messages. IPoIB follows with a further increased average latency of 1.0 to 1.2 µs for small payload sizes. These results, especially JSOR's, seem unexpectedly low at first glance. However, when considering the socket interface, it does not provide means to return any feedback to the application when data is actually sent. With verbs, one polls the completion queue and as soon as the work completion is received, it is guaranteed that the local data is sent and received by the remote. A socket send-call however, does not guarantee that the data is sent once it returns control to the caller. Typically, a buffer is used to allow aggregation of data before sending it. JSOR, libvma and IPoIB implement message aggregation, which can also be cross-checked by the ping-pong benchmark, which prevents aggregation (§V-F).

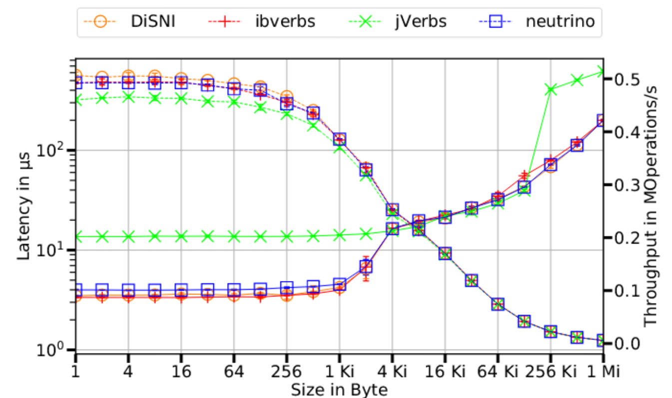

Fig. 10: 99.99% latency (write), increasing message size.

Regarding RDMA write latencies, all four libraries yield average values very close to each other. For buffer sizes

smaller than 256 byte, the average latency is around 2 µs, with DiSNI and ibverbs managing to stay slightly under 2 µs and jVerbs yielding latencies slightly higher than the rest. However, looking at the 99.99th percentile values (depicted in Fig. 10), we observed highly increased latency values of roughly 14 µs for jVerbs. From 4 KiB to 128 KiB the latencies are similar to the other libraries, but then abruptly jump to over 400 µs and rise even more, reaching over 600 µs for 1 MiB message sizes. The other libraries manage to not rise over 200 µs.

F. Ping-Pong latency

In this section, we present the results of the ping-pong latency benchmark. Due to the nature of the communication pattern, the methods of transfer are limited to messaging operations for verbs-based implementations. Using RDMA is also possible, but requires additional data structures and control, currently not implemented in Observatory.

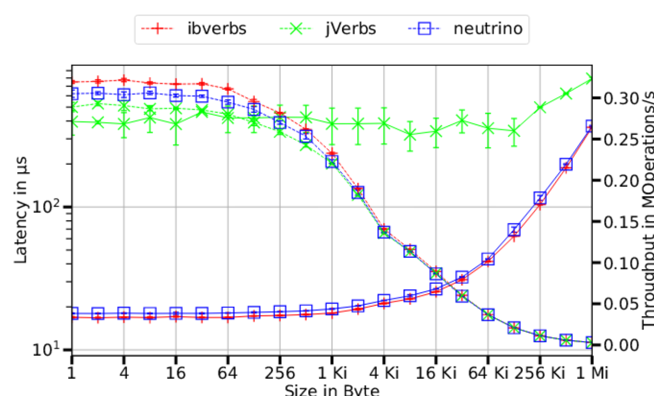

Fig. 11: 99.99% ping-pong latency, increasing message size.

Regarding the average latencies, i.e. full round-trip-times, of the verbs-based libraries, all three perform very similar to each other, yielding values between 3 and 4 µs for message sizes up to 1 KiB, with ibverbs showing the best results and neutrino performing slightly better than jVerbs. However, looking at the 99.99th percentiles (depicted in Fig. 11), jVerbs already starts with more than 400 µs for 1 byte messages, while ibverbs and neutrino manage to yield latencies of 16 - 18 µs for payload sizes up to 1 KiB. At 8 KiB message size, jVerbs reaches its lowest latency at roughly 320 µs. However, this is still extraordinary high, compared to ibverbs and neutrino.

Regarding the average latencies of the socket-based methods, depicted in Fig. 12, JSOR shows low average latencies of 2.1 to 3.5 µs for message sizes up to 512 byte. A small "latency jump" of around 1 µs is notable from 64 byte to 128 byte message size. The results of libvma are slightly higher with 3.7 to 5.5 µs for payload sizes up to 512 byte and the same "latency jump" from 128 byte to 256 byte. IPoIB's latency is distinctly higher, being constantly at 18.2 - 19.0 µs up to 512 byte message size. Starting with 128 KiB, libvma's latency values are abruptly rising faster than IPoIB's and make a large "jump" from 512 KiB to 1 MiB.

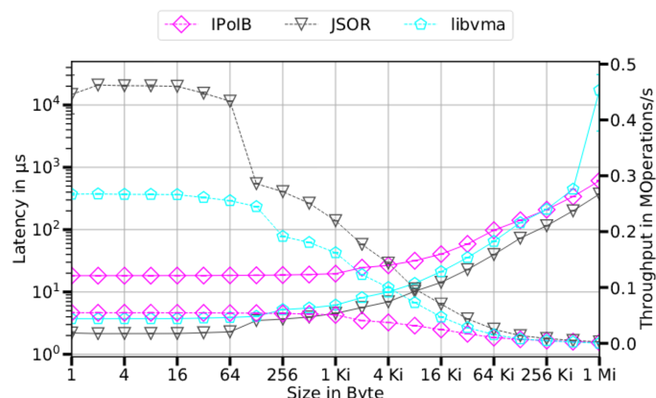

Fig. 12: Avg ping-pong latency, increasing message size.

VI. Conclusions

InfiniBand is transparently available for HPC applications using MPI. However, many big-data applications are developed in Java and need to be adapted towards a specific verbs-based library in order to benefit from the full potential of an IB network. In this paper we have proposed the Observatory benchmark, which aims at comparing different existing IB solutions for Java. The benchmark is open source and has a lean interface, allowing to easily add other IB libraries.
Socket-based solutions are transparent for the application, but the evaluation results show that they cannot exploit the full hardware potential, especially regarding bidirectional communication. The latency is at least half on 56 Gbit/s hardware compared to Gigabit Ethernet and sometimes is even as low as 2-5 µs for small messages. The throughput is at least ten-fold faster and it is possible to saturate 56 Gbit/s on unidirectional communication. libvma is a good choice providing transparency, while not requiring a proprietary JVM.
Verbs-based solutions are not transparent for the application but are a must for exploiting the full potential of IB networks. jVerbs performs well and brings nearly native performance on RDMA operations to the Java space. However, message passing is slow with jVerbs and it can only be used with IBM's SDK (limited to Java 8). For DiSNI, the RDMA write results look promising, but we observed the same messaging issues as with jVerbs. neutrino, our own open source IB library shows overall very good results and is compatible with new Java versions.
Future work includes extending Observatory with more communication patterns, integrate multi-threading support and evaluations on 100 GBit/s IB. Also, other IB connection types, such as "Unreliabe Datagram" (UD) and "Dynamic Connected Transport" (DCT) are planned to be evaluated.

References

[1] Apache ignite. https://ignite.apache.org/.
[2] Disni github. https://github.com/zrlio/disni.
[3] Ibm. rdma communication appears to hang. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/diag/problem_determination/rdma_jsor_hang.html.

[4] Ibm. rdma connection reset exceptions. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/diag/problem_determination/rdma_jsor_connection_reset.html.
[5] Infinispan. http://infinispan.org/.
[6] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.
[7] libvma github. https://github.com/Mellanox/libvma/.
[8] Mellanox. https://www.mellanox.com/.
[9] neutrino github. https://github.com/hhu-bsinfo/neutrino.
[10] Ofed 3.5 release notes. https://downloads.openfabrics.org/OFED/release_notes/OFED_3.5_release_notes.
[11] Openfabrics alliance. https://openfabrics.org/.
[12] Osu micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.
[13] Top500 list.
[14] Infiniband architecture specification volume 1, release 1.3. http://www.infinibandta.org/, 2015.
[15] S. Ekanayake and G. Fox. Evaluation of java message passing in high performance data analytics. 03 2014.
[16] R. R. Exposito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Fastmpj: a scalable and efficient java message-passing library. *Cluster Computing*, 17:1031–1050, Sept. 2014.
[17] D. Goldenberg, T. Dar, and G. Shainer. Architecture and implementation of sockets direct protocol in windows. *2006 IEEE International Conference on Cluster Computing*, pages 1–9, 2006.
[18] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.
[19] W. Huang, H. Zhang, J. He, J. Han, and L. Zhang. Jdib: Java applications interface to unshackle the communication capabilities of infiniband networks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 596–601, 10 2007.
[20] V. Kashyap. Ip over infiniband (ipoib) architecture. https://www.ietf.org/rfc/rfc4392.txt, April 2006.
[21] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.
[22] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. *SIGPLAN Not.*, 50:695–710, Oct. 2015.
[23] S. Mehta and V. Mehta. Hadoop ecosystem: An introduction. In *Int. Journal of Science and Research (IJSR)*, volume 5, June 2016.
[24] S. Nothaas, K. Beineke, and M. Schoettner. Ibdxnet: Leveraging infiniband in highly concurrent java applications. *CoRR*, abs/1812.01963, 2018.
[25] S. Nothaas, Ruhland. A benchmark to evaluate infiniband solutions for java applications. Technical report, 8 2019.
[26] P. Stuedi. Direct storage and networking interface (disni). https://developer.ibm.com/open/projects/direct-storage-and-networking-interface-disni/, 2018.
[27] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14. ACM, 2013.
[28] G. L. Taboada, J. Touriño, and R. Doallo. Java fast sockets: Enabling high-speed java communications on high performance clusters. *Comput. Commun.*, 31:4049–4059, Nov. 2008.
[29] S. Thirugnanapandi, S. Kodali, N. Richards, T. Ellison, X. Meng, and I. Poddar. Transparent network acceleration for java-based workloads in the cloud. https://www.ibm.com/developerworks/library/j-transparentaccel/, January 2014.
[30] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *IEEE 20th Ann. Symposium on High-Performance Interconnects*, pages 48–55, 2012.
[31] H. Zhang, W. Huang, J. Han, J. He, and L. Zhang. A performance study of java communication stacks over infiniband and giga-bit ethernet. In *2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007)*, pages 602–607, 2007.

Chapter 3

hadroNIO - Transparent Java network acceleration

This chapter introduces *hadroNIO*, a transparent network acceleration library, that replaces the JDK's default NIO implementation. Using *OpenUCX* for data transmission enables it to leverage modern high-speed network interconnects, such as InfiniBand.

3.1 Java NIO and Netty overview

As described in Section 1, Java NIO offers an asynchronous (non-blocking) socket-based network API, as opposed to the classic Java sockets, which only offer synchronous (blocking) communication. It was designed with scalability in mind and allows developers to implement applications that are adaptable to the target platforms amount of available CPU cores. To achieve this, Java NIO introduces socket channels with similar semantics to traditional sockets. In the default JDK implementation, these channels actually work on an underlying socket, but have the ability to switch between blocking and non-blocking mode. In blocking mode, they work exactly like sockets, with each write operation blocking until all bytes are processed and read operations only returning, once at least on byte has been received or the connection is canceled. In non-blocking mode however, a write request only processes as many bytes as the underlying socket can handle at the moment, while read operations may return without any bytes read, if the underlying socket has no bytes in its internal receive buffer (i.e. no bytes have been received since the last read request).

The key part of Java NIO is the selector, which is used to query multiple socket channels for operation *readiness*. Each channel may be registered at exactly one selector at any given time. When registering a socket channel, the selector returns a *selection key*, which indicates the associated channels readiness. This way, a channel can signal that it is ready to perform a certain operation. Each type of operation is represented by one of the following flags:

- `OP_WRITE`: The channel is ready to perform a write operation (i.e. the underlying socket's internal write buffer is not full).
- `OP_READ`: Data has been received from the remote side and may now be retrieved by the application using a read operation.

- `OP_CONNECT`: Either a connection has been established successfully and may now be finalized by calling *finishConnect()*, or a connection error has occurred.
- `OP_ACCEPT`: This flag is only valid for server socket channels, and indicates either a pending connection request from a remote client, or an error.

By registering multiple socket channels at a single selector, it is possible to query the readiness of all these channels at once. This is called a *select* operation and it allows for multiplexing socket channels in a single thread (see Fig. 3.1). This way, a single thread is able to handle an arbitrary number of channels, instead of just one socket, as it is the case with the classic blocking API. If, for example, a select operation results in `OP_READ` readiness signals on multiple channels, the thread can read the incoming data from all these channels one after another, before calling `select()` once again. Contrary to this, when using traditional sockets, one might miss incoming data on other sockets, while waiting for a *read()* call to return. Furthermore, NIO based applications can be implemented to scale with a system's CPU by starting one thread per CPU core, with each thread handling its own selector. Channels can then be distributed over the selectors (i.e. CPU cores), utilizing the CPU's full capacity without overwhelming it with too many threads.

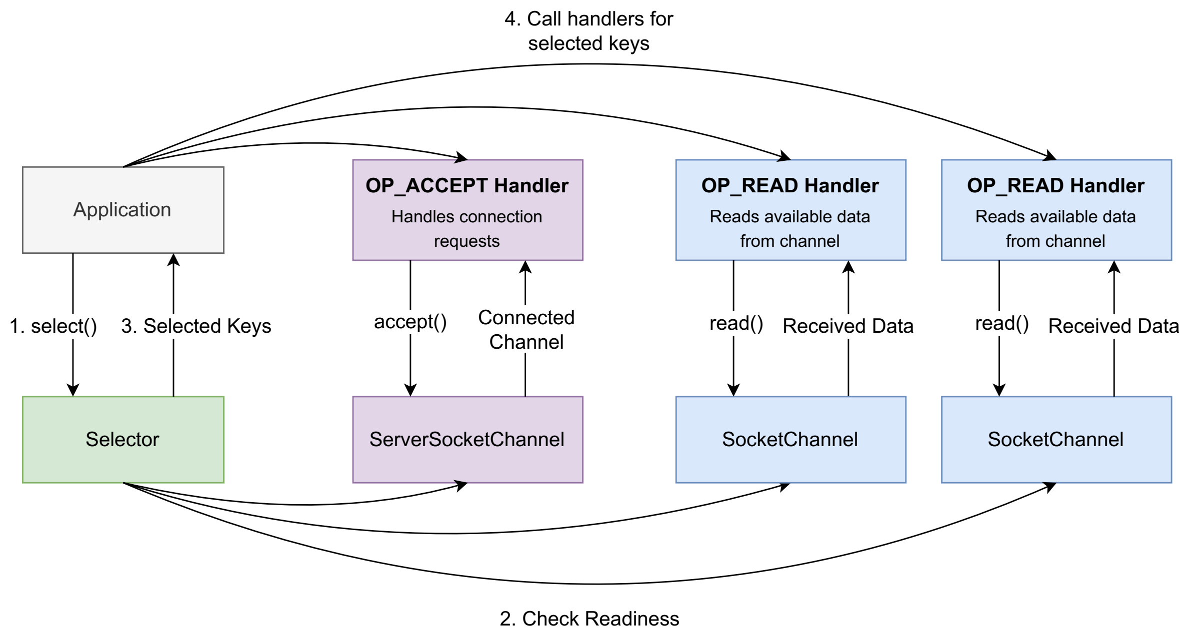

Figure 3.1: Example diagram of a Java NIO setup with two active connections

While Java NIO allows for a flexible connection handling, it is rather complicated to use. Programmers are still responsible for managing threads and buffers manually. To this end, many applications do not use NIO directly, but are instead based on *Netty*. This asynchronous networking framework allows applications to handle network communication in an even-driven manner. It manages threads automatically, according to the amount of available CPU cores, but also allows programmers to use their own thread pool implementations. This makes it scalable out of the box, but still flexible enough to be adapted to specific use cases. Furthermore, Netty implements a pooling mechanism for buffers to lessen the pressure on the garbage collector. It also comes with pre-implemented protocols

like HTTP/2 and TLS [6].
These features render Netty a viable choice for modern network development in Java. Setting up an application with Netty is much easier than using raw NIO methods. Developers can choose to implement low-level protocols based on TCP/UDP (e.g. Apache ZooKeeper [9]) or benefit from a high-level protocol with encryption support (e.g. gRPC [8]).

3.2 OpenUCX overview

Unified Communication Framework X (UCX) is a networking framework, providing a unified API for different communication patterns like message passing, streaming, or remote direct memory access (RDMA). It supports several transports, such as InfniBand or High-Speed Ethernet, but can also use TCP as a fallback [11]. The fastest transport is automatically chosen and it is even possible to combine multiple transports in a multi-rail setup. UCX is written in C/C++, but supports Java via a binding called JUCX, based on the Java Native Interface (JNI).

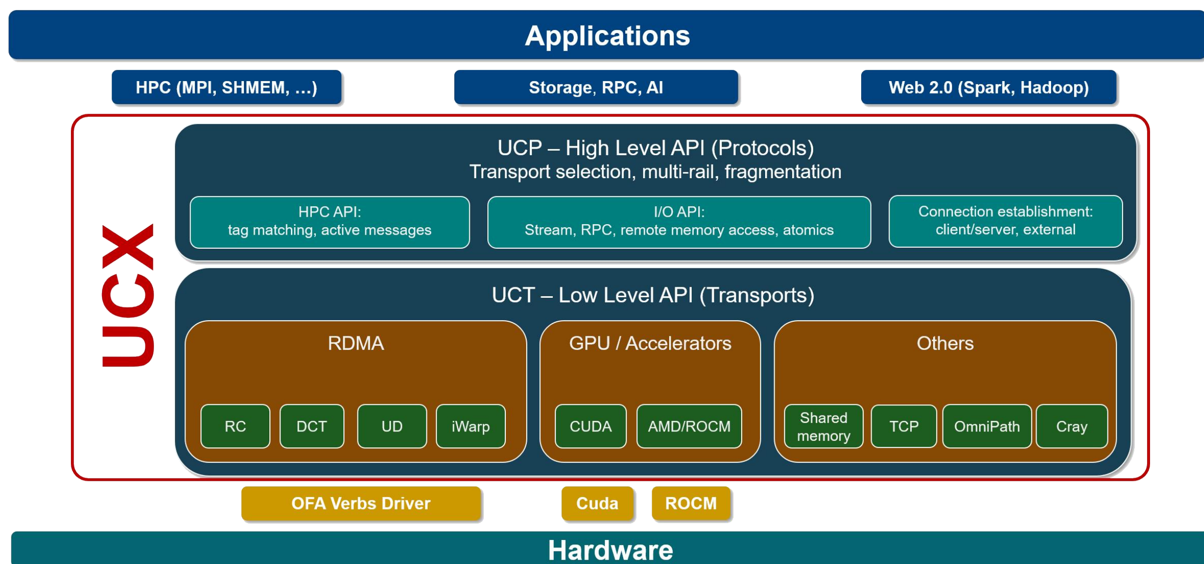

Figure 3.2: UCX architecture layers [25]

UCX is divided into a low-level and a high-level package. The low-level library is called UCT, providing a thin layer over libraries like ibverbs. It offers functions that are still close to the hardware, with all implying advantages and disadvantages. It serves as the base foundation for the UCX architecture. While it may be useful for some cases, most projects based on UCX, including this dissertation, use the high-level package UCP. It provides the main API, abstracting the low-level programming interfaces of the supported transports.

In UCP, connections are abstracted by endpoints. Each endpoint represents one side of a connection. Connection setup works similar to sockets, with the server side setting up a listener and the client side creating an endpoint with the server's address as a param-

eter. This works using traditional IP-addresses. Once a listener receives a connection request, it can create an endpoint, connected to the client. Now, both sides have created an endpoint instance that is configured to communicate with the remote side. However, to finish connection setup, a message must be exchanged manually. The first data exchange between two endpoints triggers UCP's internal wire-up protocol. On success, the connection setup is finished, but if the operation fails, the endpoints may not be regarded as connected and the application has to handle this connection setup error appropriately.

The most important UCP structure is the worker. Workers can abstract multiple network interfaces and manage the operations of multiple endpoints. They play a key part in connecting the high-level UCP API with the low-level network interfaces, by implementing a progress engine. UCP operations like messaging or RDMA are initiated via requests. A request (e.g. sending/receiving a message) can be handled directly by the UCX framework, if the requested operation is small and the appropriate resources are available. However, most requests are handled asynchronously, in which case UCP creates a unique handle for each request. While the operation can be executed in the background by the network hardware, the associated worker's progress engine needs to be advanced in order for the application to be notified about the success or failure of a request (see Fig. 3.3). This needs to be done manually by the application, by calling a `progress()` method, which updates the state of each finished request. An application can either poll the state of a request to see if it is finished, or associate a callback with it, which will be called by the progress engine, once the request has been processed.

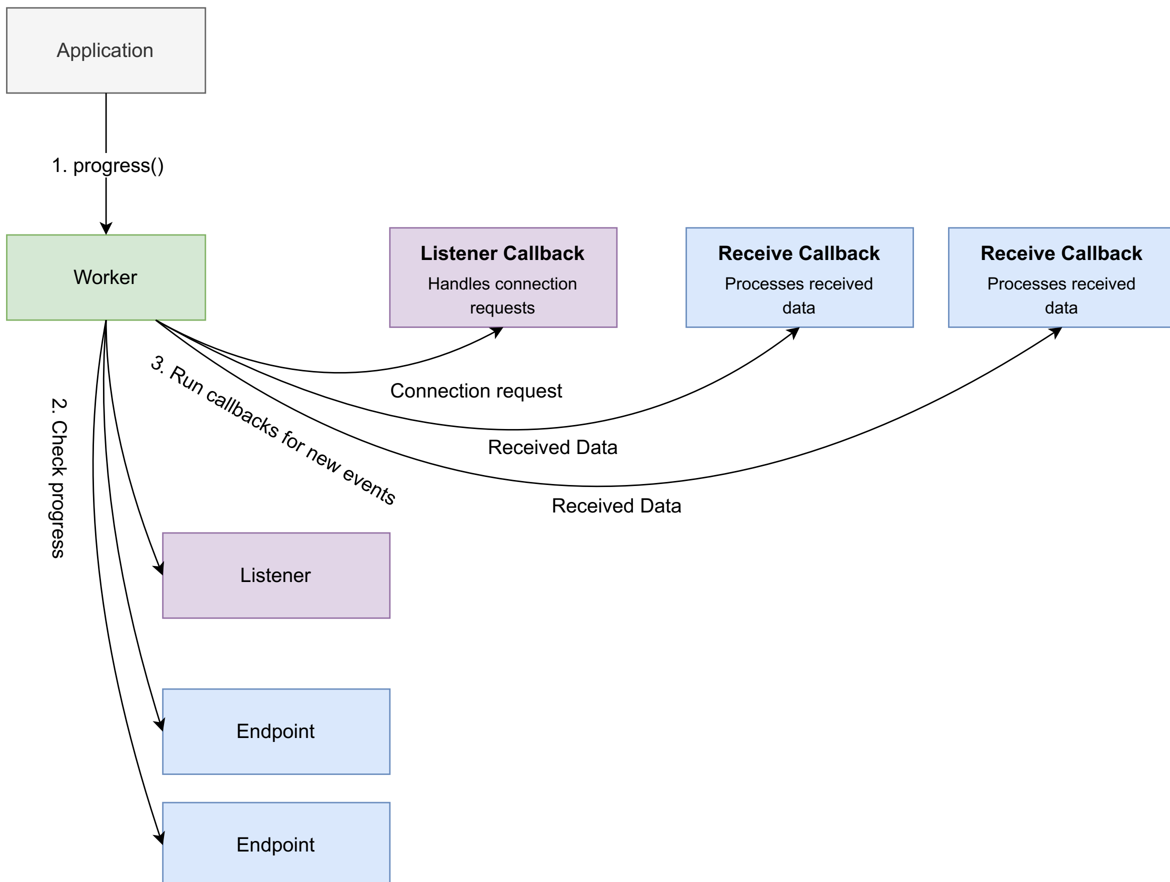

Figure 3.3: Example diagram of a UCP setup with two active connections

3.3 Transparency challenges

It would also be possible to reimplement the classic Java Socket API to accelerate both, applications build upon that, as well as applications using Java NIO. However, basing hadroNIO on Java NIO was a deliberate choice by the author, since it provides a more modern API, designed around asynchronous communication and is used by most modern Java network applications already. Furthermore, it is a much better match for UCX than the traditional Socket API, with both NIO and UCX offering an asynchronous programming model, with a central part that allows multiplexing connections, in form of the UCP worker and the NIO selector, respectively. However, there are some challenges, that need to be addressed.

Specifically, NIO allows socket channels to change the selector they are registered at. While this is not a common practice, hadroNIO needs to be aware of that to be fully compliant with the Java NIO specification. In UCP, the connection between an endpoint and a worker is fixed and cannot be changed later on. To solve this problem, hadroNIO creates one worker per connection (i.e. per endpoint) and a selector needs to poll multiple workers. While this sounds more complicated than just creating one UCP worker per selector, it actually solves another problem, caused by UCX: The amount of endpoints per worker is limited, which means in order to support an arbitrary number of socket channels per selector, a single worker does not suffice.

Furthermore, there is a considerable incompatibility between the send and receive functions of NIO and UCP. Both APIs work in an asynchronous manner, meaning that the functions will not block until the network operation is finished. Instead, they will just initiate the operation and return. In Java NIO, socket channels offer `write()` and `read()`, both taking a buffer as parameter.

When `SocketChannel.write()` returns, the passed buffer can be reused directly, even though the send operation might not have been finished yet. This is the case, because the buffer's content has been copied into the underlying socket's internal buffer, from where it is then being processed by the network card. UCP on the other hand, works directly with the buffer, handed over as parameter. This means, when sending a message via UCX, the user may not alter the buffer's content, until the request has been finished. Elsewise, data corruption might occur on the receiving side.

A similar incompatibility exists receiving data: `SocketChanne.read()` copies received data from the underlying socket's internal buffer into the target buffer, specified by the user. When it returns, the target buffer contains the received data and the user can directly work with it. When working with UCP, the buffer's content may only be considered complete, after the receive request has been finished. Reading from the target buffer after initiating a receive operation, but without checking the request's status, may result in incomplete or corrupted data being read.

To address these problems, hadroNIO implements intermediate buffers for sending and receiving data. This means, that a single copy operation is needed for each network operation, which causes a small overhead, compared to using UCX directly. A detailed description of how the intermediate buffer's work and how large of an overhead is imposed by hadroNIO is presented in the following paper.

3.4 Contributions

The author is the main developer of hadroNIO and has implemented most of the code by himself. This project is the main research contribution of this dissertation, with two papers and a technical report. However, the ring buffer implementation used for the intermediate buffering mechanism is largely based on the `OneToOneRingBuffer` class from the *Agrona* library [26]. It has been altered by Filip Krakowski to support a two-step write mechanism, where a slice of the buffer may first be reserved and then written to later. The original implemenation only supported directly writing to the ring buffer's current position.

Both of the following papers were written by the author, while Filip Krakowski and Michael Schöttner took part in many discussions about the design and implementation of hadroNIO.

Furthermore, there has been a minor external contribution by Edwin Stang on GitHub, fixing some possible null pointer exceptions when closing a UCP endpoint or listener [27]. Additionally, Peter Rudenko from Mellanox has extended the Observatory benchmark to support JUCX [28].

hadroNIO: Accelerating Java NIO via UCX

Fabian Ruhland
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
fabian.ruhland@hhu.de

Filip Krakowski
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
filip.krakowski@hhu.de

Michael Schöttner
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—InfiniBand networks with bandwidths up to 400 Gbit/s and sub-microsecond latencies are more and more popular in HPC and cloud data centers. Many big-data frameworks, such as Apache Spark and Cassandra, are written in Java and use Java NIO socket channels, which are designed for Ethernet networks. Rewriting network code for such complex systems is typically not an option and thus, transparent solutions like IP over InfiniBand are used.**
In this paper, we present *hadroNIO*, a Java library, that transparently replaces the default NIO implementation, providing support for InfiniBand (as well as Ethernet and other transports) through the *Unified Communication X* (UCX) library. We compare our library against other transparent network acceleration solutions in an InfiniBand environment and also evaluate the overhead, that is introduced by using hadroNIO versus directly accessing UCX. We show that it is possible to achieve latencies as low as 3.1 µs, while also being able to leverage the full bandwidth of InfiniBand hardware with our fully transparent acceleration solution. In the future we aim at extending hadroNIO, and thus the NIO API, with RDMA directives.

***Index Terms*—High-speed Networks, InfiniBand, OpenUCX, Java, Remote Direct Memory Access**

I. Introduction

Java NIO is the standard for modern network development on the Java platform for many years now. With its elegant API for asynchronous communication, it empowers application developers to handle several connections with just a single thread, while still being flexible to scale with large thread counts. Additionally, it supports blocking communication, resembling the traditional Java socket API. Its success is underlined by the amount of projects based on NIO (or *netty*, building upon NIO [6]), such as Spark [16] or Cassandra [2].

However, since the NIO implementation relies on classic sockets, applications are limited to using Ethernet for communication. While there are several successful approaches mitigating this problem by transparently offloading socket traffic to fast networks like InfiniBand [5] [10] [15], our past research shows, that none of them are capable of leveraging the full potential of the underlying InfiniBand hardware [13].

Unified Communication X (UCX) is a native framework, aiming to provide a unified API for multiple transport types [14]. The UCX API offers several forms of communication, such as tagged messaging, active messaging, streaming or RDMA. Application developers do not need to target a specific network interconnect, since UCX automatically scans the system for available transports and chooses the fastest one (e.g. Ethernet or InfiniBand). Since it also provides a Java-binding called JUCX (based on JNI), this framework can also be used in Java applications [4].

In this paper, we propose *hadroNIO*, aiming at accelerating Java communication using JUCX. Instead of offloading the traffic to a specific type of transport, we leverage UCX to benefit from several transports. We aim at enabling NIO based applications to transparently use the full potential of the available hardware, regarding both high throughput with NIO's non-blocking API, as well as low latencies with blocking socket channels. Developers do not have to specifically build their applications against hadroNIO, but can just use the standard NIO API. In the future, we plan to provide RDMA functionality by extending the NIO API with new directives for remote reading and writing. This would allow applications, that are aware of hadroNIO, to use RDMA features, without requiring developers to learn a new API from the ground up. The contributions of this paper are:

- An overview of existing socket acceleration solutions for Java applications
- The design and implementaion of hadroNIO, a library to transparently accelerate Java NIO using UCX, enabling developers to benefit from several types of interconnects without learning a new API
- Evaluation of hadroNIO against IP over InfiniBand and directly using JUCX with blocking and non-blocking socket channels

The paper is structured as follows: Section II discusses related work and presents some of the existing acceleration solutions. Section III discusses hadroNIO's architecture followed by Section IV with the evaluation results. Conclusions are presented in Section V.

II. Related work

To the best of our knowledge, there are no alternative NIO implementations, but there are multiple solutions (some of them still actively maintained) aiming to accelerate traditional Java sockets by offloading the send and receive traffic of socket-based applications to InfiniBand host channel adapters (HCAs). Since NIO is based on classic Java sockets, these solutions also work with applications based on Java NIO. In the past, we have already evaluated some of them using *Observatory*, our networking micro-benchmark suite, tailored towards evaluating InfiniBand solutions for Java applications.

IP over InfiniBand (IPoIB) [10] is a kernel module, that exposes the InfiniBand device to the user space as a standard network interface (e.g. *ib0*). Applications can just bind their sockets to an IP-address associated with such a network interface, making IPoIB transparent to use. However, since it uses the kernel's network stack, thus requiring context switching between user and kernel space, there is a relatively high performance penalty (especially regarding latency).

libvma [5] is a native open source library, developed by Mellanox, that can be preloaded to any socket-based application (using *LD_PRELOAD*). It enables full bypass of the kernel's network stack by redirecting all socket traffic over InfiniBand using a custom protocol based on unreliable datagram communication. While existing application code does not have to be modified to benefit from increased performance, libvma requires the *CAP_NET_RAW* capability, as well as flow steering to be enabled, which might not be available depending on the cluster environment.

Java Socket over RDMA (JSOR) is a java library, developed by IBM, which redirects all socket traffic over high-speed networks using RDMA. It is included in IBM's Java SDK and requires their proprietary J9 JVM, thus only supporting Java versions up to 8, so far. While JSOR has shown promising results, there are known problems with connections getting stuck [8] and exceptions [9]. Additionally, when evaluating JSOR, we were not able to perform a full benchmark run with a bidirectional connection, using separate threads for sending and receiving [13]. These problems and its dependency on proprietary technology limit its usability.

The **Sockets Direct Protocol (SDP)** leverages RDMA with full kernel bypass to accelerate all socket traffic of Java applications. It was part of the OFED and introduced into the JDK starting with Java version 7. However, support has officially ended and it has been removed from the OFED since version 3.5 [7].

Java Fast Sockets is an optimized Java socket implementation for high-speed interconnects. It avoids serialization of primitive data arrays and reduces buffering and buffer copying with shared memory communication as its main focus. While JFS provides InfiniBand access, it relies on SDP, which is deprecated.

III. hadroNIO Architecture

This section presents the architecture of hadroNIO and the challenges we solved when interfacing between Java NIO and UCX, as well as the benefits of using UCX.

A. Replacing the default NIO implementation

Per JDK specification, a socket channel may either be configured to be blocking or non-blocking [3]. In blocking mode, each `write()` operation will block until all bytes from the source buffer have been processed. This does not imply, that all bytes have been received by the remote side, but that the data has at least been copied to the underlying socket's buffer. A similar norm applies to the `read()` method, as in blocking mode it must block until at least one byte may be read from the underlying socket's buffer.

NIO's non-blocking API works quite different from that, since a call to `write()` or `read()` is not obligated to block, but is allowed to return after processing only part of the source buffer and in fact may not process any data at all (e.g. if the underlying socket's buffer is full or empty). To check which operations are eligible to be performed on a socket channel, NIO introduces the concept of selectors and selection keys. Each socket channel may be registered with one selector. This registration is represented by a selection key, which signals the associated channel's current state (e.g. if the channel is readable/writeable). To refresh the information held by a selection key, the `select()` method of the appropriate selector must be called. The selector will then check the state of each associated channel and refresh the selection keys accordingly. Additionally, an object may be attached to each key. Typically, applications attach `Runnable` instances and execute each attachment (commonly called *handler*) after the selection operation, to handle the associated channel's state (e.g. perform a read or write operation).

To transparently accelerate existing NIO applications, hadroNIO needs to fully substitute the involved classes, including `SocketChannel`, `ServerSocketChannel`, `Selector` and `SelectionKey`. The Java platform provides a comfortable way of exchanging the default NIO implementation through a class called `SelectorProvider`. This class offers methods to create instances of the different NIO components (e.g. `SocketChannel` or `Selector`). To accelerate an NIO based application, users simply need add to the hadroNIO JAR file to the classpath and configure the JVM to use the hadroNIO selector provider by setting the property `java.nio.channels.spi.SelectorProvider`.

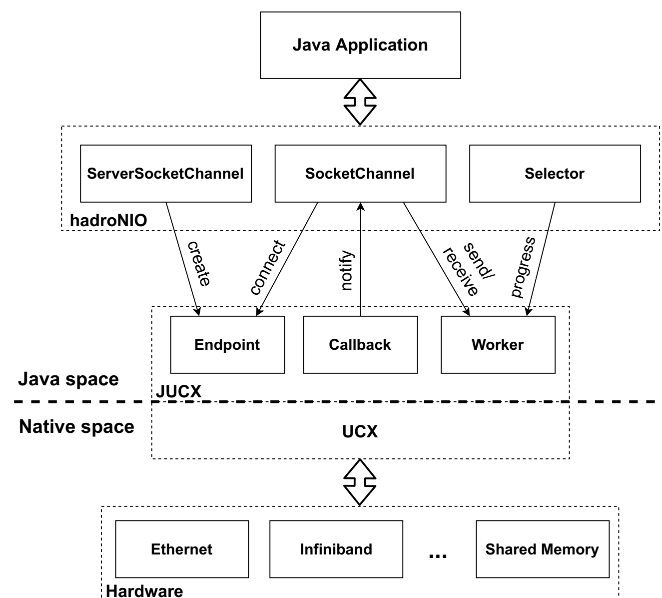

Fig. 1. Architecture overview

B. UCX request processing

UCX's tagged messaging API, which we used to build hadroNIO, generally works in a non-blocking fashion. While operations with small buffers may be completed directly, the majority of requests are executed asynchronously. To keep track of a request's state, a handle is created for each asynchronous request. These handles may be used to check if a request is completed, still in progress or was aborted with an error. Asynchronous requests do not get executed automatically, but are processed by so called workers. In UCX, a worker abstracts one or multiple network resources (e.g. HCA ports). To complete an asynchronous request, the worker, associated with the network device to which the request has been issued, needs to be progressed manually. Optionally, a callback can be associated with each request and be automatically executed once the request has been finished or aborted. This asynchronous communication concept fits well with NIO's non-blocking API, since UCX requests can be issued within the `read()` and `write()` methods of the `SocketChannel` class, while the responsible workers can be progressed in the `select()` method of the `Selector` class (see Fig 1). However, mapping this concept to blocking socket channels proved to be more challenging (see Section III-E).

C. Buffer management for writing

Buffers are managed differently in UCX and NIO: In the default NIO implementation, calling `write()` will copy the the source buffer's content into the underlying socket's buffer and return. Even though the actual process of sending the data is then performed asynchronously, the source buffer may be reused and altered by the application. UCX's behaviour differs from that by not allowing the source buffer to be modified until the request is completed.

We address this by introducing an intermediate buffer to our `SocketChannel` implementation. In its `write()` method, the source buffer's content is copied into the intermediate buffer and all UCX send requests will only operate on the copied data. Since we want to be able to handle multiple active send requests, a simple yet thread-safe memory management is needed to manage the space inside the intermediate buffer. To achieve this, the buffer is implemented as a ring buffer, based on Agrona's *OneToOneRingBuffer*. *Agrona* is a library providing multiple lock-free thread-safe data structures [1]. The full write mechanism can be divided into the following steps (also depicted in Fig. 2):

1) Allocate the needed amount of space inside the intermediate buffer.
2) Copy the source buffer's content into the newly allocated space.
3) Issue a send request via UCX.
4) Return to the application. The source buffer may now be reused and the actual process of sending the data to a remote receiver is performed asynchronously.
5) Once the request has been completed by UCX, a callback is invoked.
6) The space inside the intermediate buffer is not needed anymore and is freed by the callback routine.

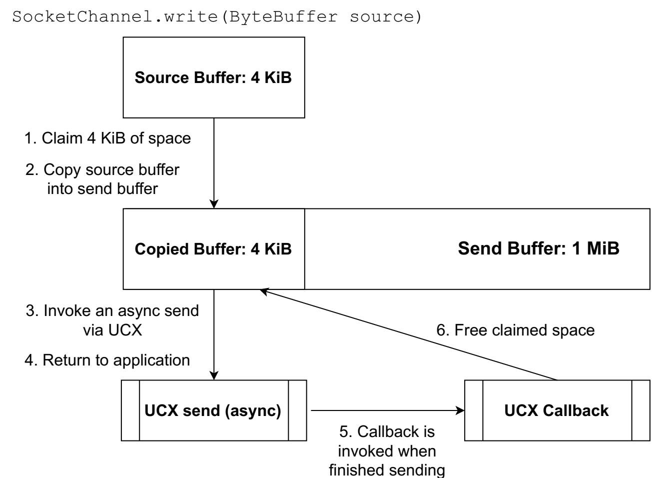

Fig. 2. Write mechanism with a 4 KiB source buffer and a 1 MiB intermediate buffer

D. Buffer management for reading

In the traditional NIO implementation, all received data is first being stored in the underlying socket's internal buffer and the `read()` method copies this data into the application's target buffer. A similar technique is applied in hadroNIO's `read()` implementation: Equivalent to the `write()` method, an intermediate buffer is used to store asynchronously received data and `read()` just needs copy this data. To issue receive requests to UCX, the method `fillReceiveBuffer()` is introduced to the `SocketChannel` class. This method allocates several *slices* of the same length inside the intermediate buffer and creates a receive request for each of these slices. This implies, that send requests, issued by `write()`, may not be larger than the slices created by `fillReceiveBuffer()`. To accommodate for that, `write()` divides larger buffers into multiple smaller send requests, that fit into the slices inside the remote's receive buffer. To ensure that hadroNIO never runs out of active receive requests, `fillReceiveBuffer()` is called once a connection has been established, and afterwards inside each selection operation. The full read mechanism can be divided into the following steps (also depicted in Fig. 3):

1) Slices inside the intermediate receive buffer are allocated by `fillReceiveBuffer()`.
2) A receive request is issued for each of the newly allocated slices.
3) Once a request has been completed by UCX, a callback is invoked.
4) The callback routine notifies the socket channel, that a new buffer slice has been filled with data. The channel keeps an internal counter of how many of the allocated slices contain valid data.

5) When the application calls `read()`, the content of a buffer slice is copied into the destination buffer. If a slice has been read fully, the allocated space is freed and reused the next time `fillReceiveBuffer()` is called.

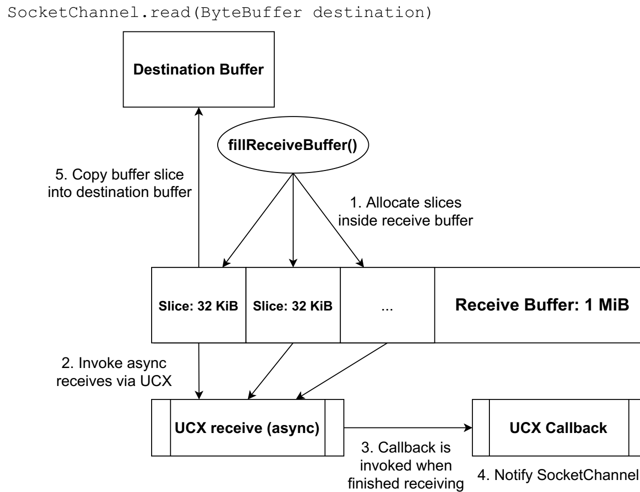

Fig. 3. Read mechanism with 32 KiB buffer slices and a 1 MiB intermediate buffer.

Additionally, each buffer slice is preceded by an 8-byte long header, consisting of two 4-byte fields. The first field indicates the length of valid data inside the slice and is needed to accommodate for the case that the remote side sends a buffer smaller than the length of a slice. The second field keeps track of the amount of data, that has already been copied to an application buffer by `read()`. In case the destination buffer is not large enough to fit a whole slice, only part of it may be copied and this header field gets updated.

The size of the send and receive intermediate buffers, as well as the slice length, have a large impact on hadroNIO's performance. For example, too small slices will result in lower bandwidths, while extremely large slices can lead to a lot of wasted memory, since each slice will probably not be filled completely. It is also important to keep in mind how many slices fit inside an intermediate buffer, since this number limits the amount of active requests and thus directly affects performance. However, large slice lengths do not affect latency and for the evaluation in section IV, we found that 64 KiB slices suffice to saturate bandwidth on our test hardware. Nevertheless, all three of these values are configurable via Java properties, allowing hadroNIO to be tuned to specific application scenarios.

E. Blocking vs. non-blocking socket channels

As mentioned before, to actually send or receive data with UCX, the appropriate worker instance needs to be progressed. In non-blocking mode, this is done inside the associated selector's `select()` method. However, in blocking mode no selector is involved, which means that the worker has to be progressed elsewhere.

For `write()`, this is done right after the send request for the last buffer slice has been issued, implying that in contrary to non-blocking mode, the data to send has already been processed by UCX, once `write()` returns. Naturally, this approach favours latency over throughput. An alternative might be to constantly progress the worker in a separate thread. While this would benefit throughput, it would also have a negative impact on latency.

For `read()`, the worker is progressed and `fillReceiveBuffer()` called every time there are no slices left to be read from the intermediate receive buffer.

F. Sender throttling

While evaluating hadroNIO, we noticed that receiving messages in non-blocking mode caused high memory usage. UCX buffers received data, that can not be directly processed by a receive request. For that purpose, a pool of memory, that grows as needed, is used. In our test case, data was sent faster than the receiving side could process it, causing the buffer pool to grow, thus resulting in high memory usage. We address this by introducing a flush mechanism into our implementation. In fixed intervals (e.g. every 1000 requests), a socket channel waits for the remote side to finish processing the received data. During that time, a socket channel will no longer indicate to be writeable. Once the remote side has finished receiving, it will send a short acknowledgment message, causing the waiting channel to be writeable again. After implementing this mechanism, we did not see an increased memory usage anymore, implying the receiving side is no longer unable to cope with the amount of incoming messages. This mechanism works fully transparent and does not affect application developers in any way.

Since the interval size may have a huge impact on performance, it is also configurable via a Java property. However, we found 1024 to be a good size, since with that, we did not see any negative effect on performance and in fact even saw an increased bandwidth, compared to not using any flush mechanism at all. This is probably due to the fact, that UCX no longer needs to allocate memory for the growing buffer pool, causing the receiving side to slow down even more.

IV. Evaluation

This section presents the evaluation results, comparing hadroNIO against IPoIB with blocking and non-blocking socket channels, as well as directly using JUCX on 56 GBit/s InfiniBand hardware.

A. Evaluation setup

To evaluate hadroNIO, we used *Observatory*, our micro-benchmark suite targeting InfiniBand solutions for Java. It aims at evaluating both messaging and RDMA performance with single point-to-point connections, regarding throughput and latency. It has already proven its usability with verbs-based libraries (i.e. directly programming the InfiniBand hardware), as well as socket-based solutions using traditional Java sockets [13]. For the purpose of evaluating hadroNIO, we enhanced

Observatory with a new NIO binding, configurable to use both blocking and non-blocking socket channels. To evaluate the overhead introduced by hadroNIO, compared to directly programming with JUCX, Observatory includes a binding for JUCX, provided by Mellanox [11] [12]. We also compared hadroNIO to IPoIB, probably being the most common acceleration solution, available in many environments. Unfortunately, we were not able to generate results for libvma with our NIO benchmark, since it yielded an error message about flow steering not being enabled. Looking into libvma's debug logs, we saw, that the error was caused by `ibv_create_flow()`, even though we have flow steering enabled in the driver. Nevertheless, libvma's performance using traditional sockets from has been published before [13]. In IV-B, we evaluated hadroNIO with different buffer sizes and slice lengths to find the optimal configuration for the following benchmarks.

The benchmarks have been performed on two identical nodes with the following setup:

CPU	Intel(R) Xeon(R) CPU E5-1650 v4 (6 Cores/12 Threads @3.60 GHz)
RAM	64 GB DDR4 @2400 MHz
NIC	Mellanox Technologies MT27500 Family [ConnectX-3] (56 GBit/s)
OS	CentOS 8.1-1.1911 with Linux kernel 4.18.0-151
JDK	OpenJDK 1.8.0_265
UCX	1.10.0 stable

Fig. 4. Hardware and software specification of the benchmark systems.

For evaluating throughput, we executed 100 million operations per benchmark run, while we used 10 million operations to evaluate round trip latencies. Starting with 8 KiB payload size, the amount of operations is incrementally halved, to avoid unnecessary long running benchmarks. We evaluated unidirectional throughput, as well as round trip latencies with payload sizes from 1 Byte to 1 MiB in power-of-two increments. When discussing the throughput results, we focus on the operation rate for small buffers up to 1 KiB and on the data rate for larger buffers.

In IV-B, the results are depicted as single line plots, while in IV-C and IV-D, we combined two line plots at a time: When looking at the throughput results, the left y-axis shows the operation rate in million operations per second (Mop/s) and the right axis the data throughput in GB/s. For the latency results, the left y-axis shows the latency in µs and the right y-axis the operation throughput in Mop/s. The dotted lines always depict the operation throughput, while the solid lines represent either the throughput in GB/s or the latency in µs, depending on the benchmark. Each benchmark run was executed five times and the average values are used to depict the graph, while the error bars visualize the standard deviation.

B. Evaluation of different buffer configurations

We evaluated hadroNIO with 4 KiB, 16 KiB, 32 KiB, 64 KiB and 128 KiB large buffer slices and configured the intermediate buffers to be large enough to fit 128 slices. Furthermore, we used blocking socket channels for evaluating latency and non-blocking socket channels for the throughput benchmarks.

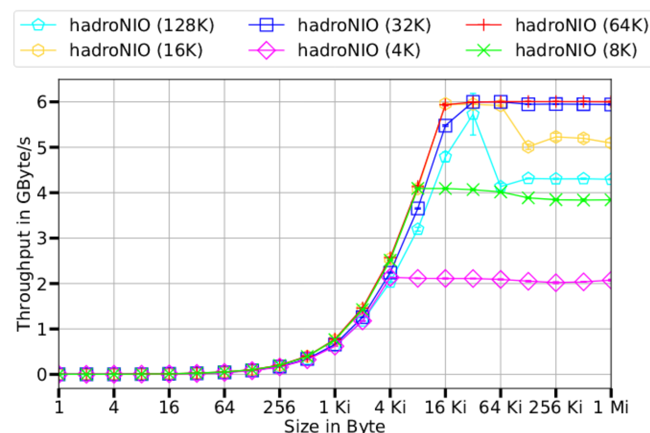

Fig. 5. Throughput results with non-blocking socket channels and different buffer configurations.

As depicted in Fig. 5, 4 KiB and 8 KiB buffer slices do not suffice to saturate the hardware, with a maximum bandwidth of 2.1 GB/s and 4.1 GB/s respectively. With 16 KiB slices, 5.9 GB/s can be reached using a payload size of 32 KiB. However, for larger messages, the throughput drops to around 5 GB/s. Using a slice length of 32 KiB, it is possible to achieve a throughput of 6 GB/s starting with payload sizes of 32 KiB. The best results were achieved using 64 KiB buffer slices, with a slight advantage over 32 KiB slices. Unexpectedly, the throughput is worse using a slice length of 128 KiB. A significant drop from 5.8 GB/s to 4.1 GB/s can be observed between payload sizes of 32 KiB and 64 KiB. This issue needs further investigation in the future.

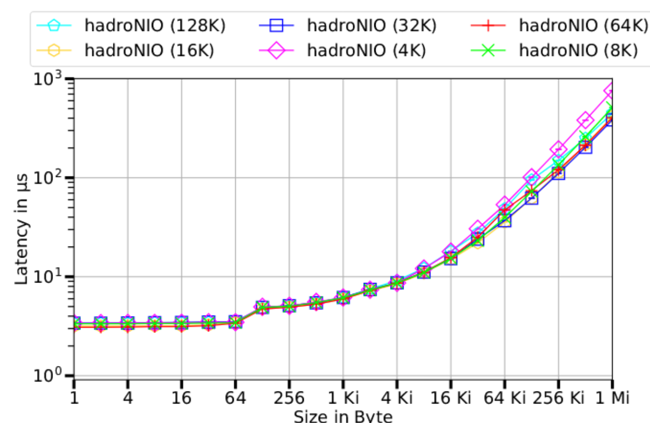

Fig. 6. Latency results with blocking socket channels and different buffer configuration.

Fig. 6 shows, that the slice length has virtually no impact on the round trip time using small payload sizes. This was expected, since hadroNIO does not send a full slice, if it is not fully used. Only the part of a buffer slice, that contains

relevant data (i.e. user data), is processed. Starting with 8 KiB payloads, the latencies of the 4 KiB configuration rise faster, which is expected since multiple send and receive operations are needed to perform a full iteration.

Based on these results, we configured hadroNIO to use 8 MiB large send and receive buffers with 64 KiB slices, since this configuration has shown the highest throughput and we did not see any negative impact on the latency. In the future, we need to investigate the performance drop using 128 KiB slices and reevaluate the different configurations on more modern hardware.

C. Evaluation of blocking socket channels

When using blocking socket channels, it is not necessary (and in fact even impossible) to use a selector and selection keys. The channels behave similar to traditional sockets, with each `write()` call only returning after writing all bytes of the source buffer and each `read()` call blocking until at least one byte has been read.

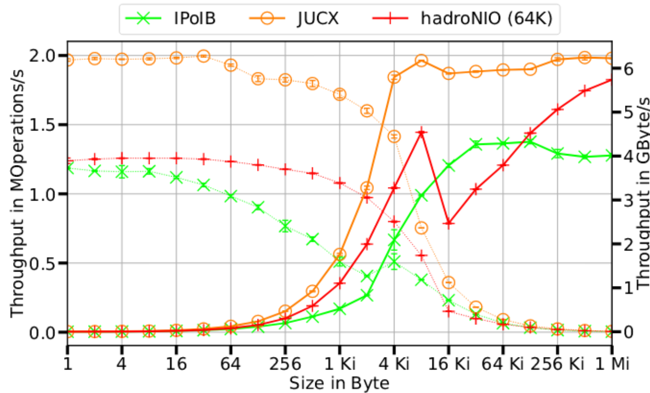

Fig. 7. Throughput results with blocking socket channels.

As depicted in Fig. 7, JUCX sets the baseline with almost 2 Mop/s for buffer sizes up to 32 byte and slowly decreasing from there, in the throughput benchmark. As expected, both hadroNIO and IPoIB stay well under that mark, but still yield around 1.2 Mop/s with hadroNIO having a slight advantage. However, with increasing buffer sizes, IPoIB's operation rate constantly decreases, while hadroNIO manages to still push over 1 million operations per second with 1 KiB buffers. From that point on, JUCX's data throughput rapidly increases, reaching the 6 GB/s mark using a payload size of 8 KiB. While hadroNIO manages to yield 4.5 GB/s at that point, its throughput drops to around 2.5 GB/s with 16 KiB buffers. This might look surprising, but can be explained by the different ways UCX handles small and large message sizes.

Up to 8 KiB, send requests are typically processed instantly, while with larger buffers, asynchronous request processing is used, which should, in theory, be beneficial for data throughput. However, hadroNIO's `write()` implementation waits until UCX has processed all requests associated with the current operation, when blocking mode is configured. This results in only a single asynchronous request being processed at a time for buffers smaller than the configured slice length, limiting throughput. We tried to mitigate this by using 8 KiB slices instead of 64 KiB and while that alleviated the data rate drop, the throughput did not rise above 4.5 GB/s, even for larger payload sizes. As explained in Section III-E, the best way to solve this problem would be to use another thread for progressing the UCX worker, but that would come at the cost of an increased latency. We plan to address this issue, by providing both implementations, one focussed on maximum throughput and one targeting minimum latencies, letting the user decide which configuration meets the application requirements best.

While IPoIB provides a higher throughput for buffer sizes from 16 KiB to 64 KiB, reaching its maximum of around 4.3 GB/s at 32 KiB, it is outpaced again by hadroNIO, starting at 128 KiB. With constantly increasing data rates, hadroNIO reaches 5.7 GB/s with 1 MiB buffers.

Although hadroNIO had problems with specific buffer sizes regarding throughput with blocking socket channels, it shows its strength looking at the round trip latencies, depicted in Fig. 8

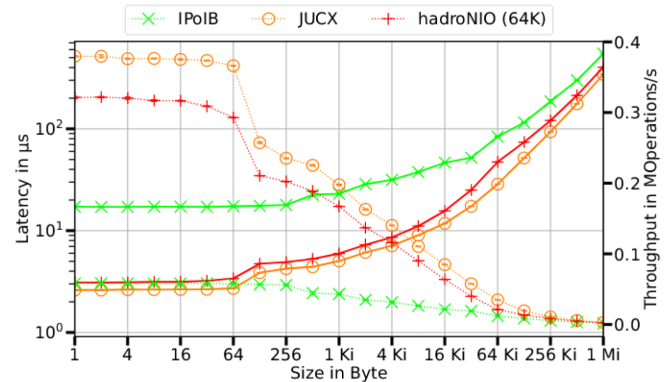

Fig. 8. Average round trip latency with blocking socket channels.

Compared to directly programming with JUCX, hadroNIO introduces only a small latency overhead. Up to 64 byte buffer sizes, JUCX yields average round trip times of 2.6 µs, while hadroNIO delivers latencies of 3.1 µs, indicating that hadroNIO's buffer management has an overhead of just 500 ns. Contrary, IPoIB provides results more than 5 times worse with latencies over 17 µs and an operation rate of 58 Kop/s vs hadroNIO's 320 Kop/s. Both JUCX and hadroNIO manage to yield single digit microsecond round trip times for buffer sizes up to 4 KiB buffers. At that point, IPoIB already passed the 30 µs mark. Naturally, with growing payloads copying data between the application and hadroNIO's internal buffers takes more time, but even at 1 MiB the difference is only around 60 µs, with JUCX needing just over 340 µs for a full round trip iteration and hadroNIO around 405 µs.

D. Evaluation of non-blocking socket channels

For benchmarking non-blocking socket channels, we need to use a selector and introduced different types of runnable

handler objects (as commonly used with NIO), attached to a selection key. For the throughput benchmark, the handler just calls `write()` or `read()` respectively one time per invocation. For the latency benchmark, the handler switches from writing to reading and vice-versa, once a buffer has been fully processed. After setting up the connection, the benchmark enters a loop, calling the selector's `selectNow()` method and running the selected key's handler in each iteration. We expect higher latencies in this scenario compared to using blocking socket channels, since the selector's logic induces an overhead and processing a buffer completely may take several handler invocations. However, data throughput should benefit from this approach, since hadroNIO is not forced to wait until UCX has finished processing, allowing requests to accumulate.

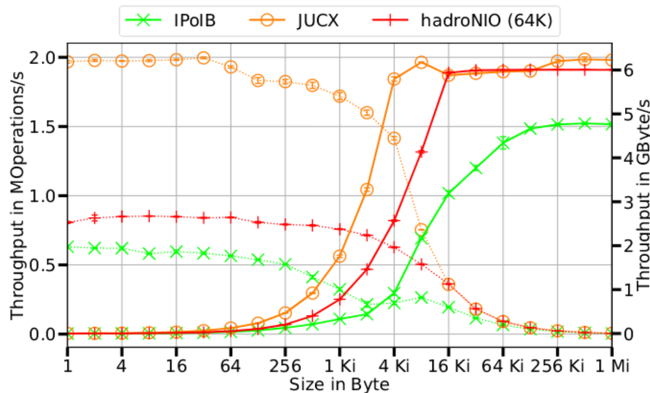

Fig. 9. Throughput results with non-blocking socket channels.

As can be seen in Fig. 9, the operation throughput with small buffers has decreased, compared to using blocking socket channels, for both hadroNIO and IPoIB. This was expected and is caused by the overhead introduced by the selector's logic. However, hadroNIO still manages to process more operations per second than IPoIB (ca. 850 Kop/s vs. ca. 620 Kop/s using 4 byte buffers). With larger buffers, hadroNIO's data throughput increases rapidly, reaching 6 GB/s at 16 KiB. In contrast to using blocking socket channels, there is no performance drop from 8 KiB to 16 KiB and the the throughput stays stable at 6 GB/s going further, almost matching the maximum throughput of 6.2 GB/s, reached by the JUCX benchmark. IPoIB's data throughput also increases with larger buffers, but at a slower pace and stagnating at a maximum speed of 4.7 - 4.8 GB/s, starting with 256 KiB payload sizes.

As expected, both hadroNIO and IPoIB yield higher latencies using non-blocking socket channels (see Fig. 10). Nevertheless, hadroNIO manages to yield round trip times as low as 5 µs and staying within single digit microsecond latencies up to 2 KiB buffer sizes. With 16 to 19 µs, IPoIB's latency results in that range are more than 3 times as high. This is also reflected by the operation throughput, with hadroNIO reaching 200 Kop/s and IPoIB maxing out at around 60 Kop/s.

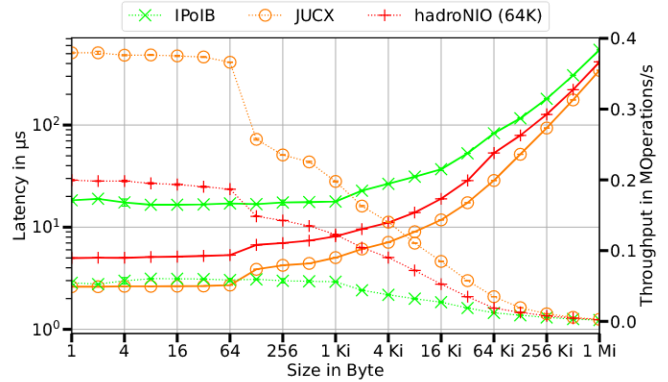

Fig. 10. Average round trip latency with non-blocking socket channels.

V. Conclusions & Future Work

In this paper, we propose hadroNIO, a novel approach at transparently accelerating network communication for Java applications. Instead of implementing a communication library for traditional sockets, we chose to provide an implementation for Java NIO, based on UCX, which is a very attractive choice as communication backend. It provides a straightforward API and supports multiple types of network interconnects, allowing us to offer applications and application developers with new possibilities regarding networking hardware without the need to learn a new API from the ground up. We have shown, that our implementation only adds a minimal overhead regarding latency, providing round trip times as low as 3.1 µs (using blocking socket channels), while still being able to saturate 56 GBit/s hardware starting with 16 KiB buffers (using non-blocking socket channels).

In the future, we plan to improve the performance of blocking socket channels by providing two modes, leaving the decision whether to focus on throughput or latency to the user. Furthermore, we plan to integrate RDMA directives into hadroNIO, thus augmenting the NIO API and providing developers with an easy way of directly accessing remote memory in a familiar environment. As a next step, we aim to evaluate hadroNIO in a setting with multiple 100 GBit/s InfiniBand connections, as well as accelerating existing applications. Further plans include using *Infinileap*, our Java binding for UCX, based on Project Panama, as an alternative way of accessing UCX.

VI. Acknowledgment

We thank Oracle for their sponsorship in the context of this work. We also especially thank Peter Rudenko for providing the JUCX binding for Observatory.

References

[1] Agrona GitHub. https://github.com/real-logic/Agrona.
[2] Cassandra. https://cassandra.apache.org/.
[3] Java Platform Standard Edition 8: SocketChannel. https://docs.oracle.com/javase/8/docs/api/java/nio/channels/SocketChannel.html.
[4] JUCX GitHub. https://github.com/openucx/ucx/tree/master/bindings/java.

[5] libvma GitHub. https://github.com/Mellanox/libvma/.
[6] Netty related projects. https://netty.io/wiki/related-projects.html.
[7] OFED 3.5 release notes. https://downloads.openfabrics.org/OFED/release_notes/OFED_3.5_release_notes.
[8] IBM. RDMA communication appears to hang. https://www.ibm.com/docs/en/sdk-java-technology/7?topic=problems-rdma-communication-appears-hang.
[9] IBM. RDMA connection reset exceptions. https://www.ibm.com/docs/en/sdk-java-technology/7?topic=problems-rdma-connection-reset-exceptions.
[10] V. Kashyap. IP over InfiniBand (IPoIB) Architecture. https://www.ietf.org/rfc/rfc4392.txt, April 2006.
[11] P. Rudenko. Observatory pull request 1. https://github.com/hhu-bsinfo/observatory/pull/1.
[12] P. Rudenko. Observatory pull request 2. https://github.com/hhu-bsinfo/observatory/pull/2.
[13] F. Ruhland, F. Krakowski, and M. Schöttner. Performance analysis and evaluation of Java-based InfiniBand Solutions. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 20–28, 2020.
[14] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, et al. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.
[15] S. Thirugnanapandi, S. Kodali, N. Richards, T. Ellison, X. Meng, and I. Poddar. Transparent network acceleration for Java-based workloads in the cloud. https://www.ibm.com/developerworks/library/j-transparentaccel/, January 2014.
[16] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59:56–65, Oct. 2016.

Accelerating netty-based applications through transparent InfiniBand support

Fabian Ruhland
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
fabian.ruhland@hhu.de

Filip Krakowski
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
filip.krakowski@hhu.de

Michael Schöttner
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
michael.schoettner@hhu.de

Abstract—**Many big-data frameworks are written in Java, e.g. Apache Spark, Flink and Cassandra. These systems use the networking framework *netty* which is based on Java NIO. While this allows for fast networking on traditional Ethernet networks, it cannot fully exploit the whole performance of modern interconnects, like InfiniBand, providing bandwidths of 100 Gbit/s and more.**
In this paper we propose netty support for *hadroNIO*, a Java library, providing transparent InfiniBand support for Java applications based on NIO. hadroNIO is based on *UCX*, which supports several interconnects, including InfiniBand. We present hadroNIO extensions and optimizations for supporting netty. The evaluations with microbenchmarks, covering single- and multi-threaded scenarios, show that it is possible for netty applications to reach round-trip times as low as 5 µs and fully utilize the 100 Gbit/s bandwidth of high-speed NICs, without changing the application's source code. We also compare hadroNIO with traditional sockets, as well as libvma and the results show, that hadroNIO offers a substantial improvement over plain sockets and can outperform libvma in several scenarios.

Index Terms—**High-speed networks, Cloud computing, Ethernet, InfiniBand, OpenUCX, Java**

I. Introduction

Modern big-data applications need to operate on large data sets, often using well-known big-data frameworks, such as Apache Spark [37], Flink [1] or Cassandra [11]. Many of these systems are written in Java, relying on Java NIO. Java NIO provides developers with the tools for building large-scale networking applications, by allowing a single thread to handle multiple connections asynchronously, thus being able to scale with the amount of CPU cores available in a system.

However, its API has a steep learning curve compared to traditional Java sockets, thread management is still being left to the programmer and buffers need to be allocated manually, requiring a sophisticated buffer management to prevent performance penalties by repeated allocations. Thus, many applications do not use Java NIO directly, but are based on *netty*, an asynchronous event-driven network application framework [22]. It abstracts the complexity introduced by Java NIO, implements buffer pooling based on reference counting, and automatically uses as many worker threads, as there are CPU cores available. It is also highly configurable, rendering it a powerful, yet easy-to-use networking library.

Netty is widely adopted in the Java community as the standard framework for fast and scalable networking and is used in many projects, e.g. Apache BookKeeper [9] or Ratis [30], which implements the Raft [27] algorithm in Java. Additionally, it serves as the base for other networking libraries, like the widely used RPC framework gRPC [2], as well as many more projects [23]. Its relevance is further underlined by the amount of organizations, that incorporate netty into their projects, such as Google, Facebook and IBM [24].

However, since netty is based on Java NIO, which relies on traditional sockets, it cannot use the full potential of modern network interconnects, like InfiniBand or high-speed Ethernet. While the socket API is compatible with high-speed Ethernet NICs and can be used with InfiniBand cards via the kernel module *IP over InfiniBand* [10], it uses the kernel's network stack, involving context switches between user and kernel space, for exchanging network data, thus imposing a high performance penalty, especially regarding latency.

This problem has been addressed in the past, with different native and Java-based solutions, which came in form of user space TCP-stacks, transparent libraries offloading traffic to high-speed NICs or kernel modules, replacing the traditional TCP implementation. However, many of these solutions are not supported anymore and introduce their own sets of problems, which we discuss in Section II.

We proposed *hadroNIO* in 2021 [32], a Java library, which transparently replaces the default NIO implementation, offloading traffic via the *Unified Communication X* framework (UCX) [33]. UCX is a native library, providing a unified API for multiple transport types (including InfiniBand) and offering a multitude of communication models, such as streaming, tagged messaging, active messaging and RDMA. It automatically detects all available transports and chooses the fastest one, but can also be configured to use a specific NIC or utilize multiple interconnects in a multi-rail setup. It officially supports Java via a JNI-based binding called JUCX. We already have shown that hadroNIO provides huge performance improvements over using traditional sockets in a single-connection setup, using a microbenchmark based directly on Java NIO [32].

In this paper, we present the extensions and optimizations, introduced in hadroNIO and evaluate its performance with

netty-based microbenchmarks using multiple connections on high-speed networking hardware, capable of 100 Gbit/s bandwidth.

The contributions of this paper are:

- An overview of existing netty-compatible acceleration approaches
- Design and implementation of hadroNIO extensions for supporting netty
- Evaluations using microbenchmarks on 100 GBit/s hardware, showing the benefits of the proposed solution

The paper is structured as follows: Section II presents related work, discussing alternative acceleration solutions. Section III elaborates on updates to hadroNIO, followed by Section IV, which presents the architecture of our microbenchmarks. Evaluation results are discussed in Section V, while Section VI concludes this paper and provides ideas for future work.

II. Related work

Modern high-speed NICs from Mellanox can be configured to use either InfiniBand or Ethernet as link layer protocol. Choosing Ethernet makes these cards fully compatible with the standard socket API, while still being programmable via the ibverbs library. Regardless of the link layer protocol, traditional sockets do not suffice for using the full potential of such a NIC.

While we are not aware of any alternative NIO implementations, there are several solutions for accelerating traditional sockets, with only few being still actively maintained. Typically, these can come in three different shapes: kernel modules, native libraries and Java libraries. Since the default NIO implementation is based on classic sockets, these solutions can be used to accelerate Java NIO applications. We have already evaluated some of these solutions, using socket-based microbenchmarks [31] and compared them to hadroNIO with another microbenchmark, directly using the NIO API [32].

A. Kernel modules

IP over InfiniBand [10] exposes InfiniBand devices as standard network interfaces, enabling applications to use them by simply binding to an IP address, associated with such a device. This solution does not require any preloading of libraries, making it the easiest to use. However, it relies on the kernel's network stack, thus requiring context switches which impose a large performance overhead, rendering it unattractive for applications requiring low latency.

Fastsocket [16] replaces the Linux kernel's TCP implementation, aiming to provide better scaling with multiple CPU cores. It has been evaluated using up to 24 cores using 10 Gbit/s Ethernet NICs, showing much better scalability than the default TCP implementation. Fastsocket consists of kernel level optimizations, a kernel module and a user space library. It requires a custom kernel, based on Linux 2.6.32 and officially only supports CentOS 6.5, which is outdated by now. While it would be interesting to see how such an integrated solution would perform on modern high-speed Ethernet hardware, it does not seem to be in active development anymore.

B. Native libraries

mTCP [8] is a TCP-stack, running completely in user space. As Fastsocket, it primarily aims at high scalability, which it achieves by being independent from the kernel's network stack, alleviating the need for context switches in network applications. Contrary to the other solutions, it is not transparent and requires rewriting parts of an application's network code. It has no official support for Java, but there is an unofficial binding called JmTCP, based on the Java Native Interface (JNI). However, it does not seem to be actively maintained, probably requiring Java applications to manually access mTCP via JNI or the experimental Foreign Function & Memory API (Project Panama) [6]. Since it is neither transparent, nor officially supports Java, mTCP does not fit our use case of accelerating netty-based applications.

libvma [13] is a library developed in C/C++ by Mellanox, transparently offloading socket traffic to high-speed Ethernet or InfiniBand NICs. It can be preloaded to any socket-based application (using *LD_PRELOAD*), enabling full kernel bypass without the need to modify an application's code. However, libvma requires the *CAP_NET_RAW* capability, which might not be available, depending on the cluster environment.

While it is highly configurable by exposing many parameters, allowing users to tune the library to the needs of a specific applications, the resulting performance can actually be worse compared to using the traditional socket implementation, as we show in Section V. Additionally, the default configuration is only suited to basic use cases (e.g. single threaded applications), requiring some time being spent on finding the right configuration for complex applications, using multiple threads and connections.

SocksDirect [12] is a closed source library from Microsoft, written in C/C++. Like libvma, it works by preloading it to socket-based applications, redirecting socket traffic via a custom protocol based on RDMA. It also supports acceleration of intra-host communication via shared memory. It achieves low latencies and a high throughput by removing large parts of the synchronization and buffer management involved in traditional socket communication, while being fully compatible with linux sockets, even when process forking is involved.

We were able to get access to the source code from the authors and have successfully tested it with native applications, but so far we could not get the library working with Java applications. Additionally, SocksDirect uses the experimental verbs API, only available in the Mellanox OFED up to version 4.9 [21].

C. Java libraries

The **Sockets Direct Protocol) SDP** [20] provided transparent offloading of socket traffic via RDMA, fully bypassing the kernel's network stack. It was part of the OFED package and introduce into the JDK starting with Java 7. However, support

has officially ended and it has been removed from the OFED in version 3.5 [19]

Java Sockets over RDMA (JSOR) [3] has been developed by IBM with the goal to offload all socket traffic of Java applications to high-speed NICs using RDMA. It is included in the IBM SDK up to version 8, requiring their proprietary J9 JVM. JSOR is not available in newer SDK versions and while the old SDK still receives security updates, applications using features not available in Java 8 cannot be used with JSOR.

While it has shown promising results in our benchmarks, there are known problems with connections getting stuck [4] and exceptions [5]. Additionally, we were not able to evaluate JSOR using a bidirectional connection with separate threads for sending and receiving. These problems and its reliance on on proprietary technology limit its usability, especially for modern applications.

D. Application-specific solutions

Other approaches aim at accelerating network performance of a specific application or framework. In 2014, a successful attempt at redesigning Spark's shuffle engine for RDMA usage has been made [17] and refined in 2016 [18]. Similar solutions have been implemented for Apache Storm: In 2019, RJ-Netty has been proposed as a replacement for netty in Apache Storm [36], while in 2021 another approach at integrating RDMA into Storm, based on DiSNi [34] (formerly jVerbs [35]) has been implemented [38].

While these solutions show, that the performance benefit for using high-speed networking hardware can be huge, they are specific to a single framework only and can not be used for general purpose network programming, like transparent acceleration libraries.

III. Supporting netty in hadroNIO

While hadroNIO has been working with applications directly using Java NIO, we encountered new challenges with netty-based applications. This section presents these challenges and their solutions, as well as changes in the design of hadroNIO. For a general overview of our architecture and the Java NIO API, as well as UCX, we refer to our original paper [32].

The full application stack for hadroNIO, libvma, IP over InfiniBand and traditional sockets from netty to NIC is depicted by Fig. 1.

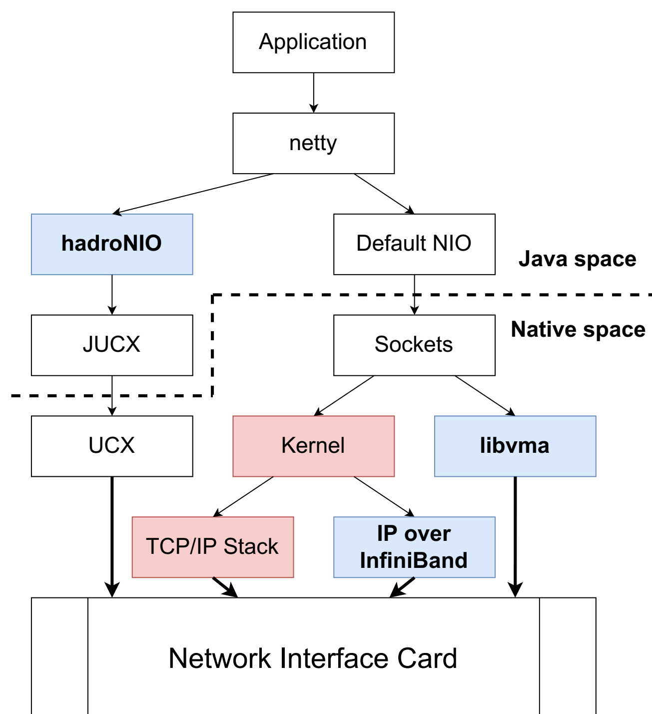

Fig. 1. Application stack overview

A. Providing a direct socket reference for netty

A Java NIO `SocketChannel` provides access to its underlying socket via the `socket()` method. Since this would defeat the purpose of NIO, accessing a socket directly via a channel, is generally not done. However, netty keeps a reference to the socket of each channel to access its configuration (e.g. buffer size).

Since hadroNIO directly replaces the default NIO implementation, there is no underlying socket. In contrast to the aforementioned transparent acceleration solutions, we consciously chose to intercept traffic at the NIO level, instead of the socket level, since it fits well with the UCX API and has a more modern interface than traditional Java sockets. In its initial version, hadroNIO would throw an `UnsupportedOperationException` when `socket()` is called. However, for compatibility with netty, we needed to provide a workaround for the socket access, which we implemented in the form of two classes called `WrappingSocket` and `WrappingServerSocket`, extending the JDK's `Socket` and `ServerSocket` classes. They wrap an instance of a `SocketChannel`, or `ServerSocketChannel` respectively, and implement methods to access connection attributes, such as IP addresses and buffer sizes.

Once a connection has been terminated, the respective socket channel becomes readable, indicated by the `OP_READ` flag. However, each attempt at actually reading data from the channel will return `-1`, signalling a closed connection. This behaviour was not implemented in earlier version of hadroNIO, since it only affects connection termination and our benchmarks would run without it. However, for full compatibility with the NIO specifications, we retrofitted it.

B. UCX worker management

UCX uses so called *endpoints* to represent connections. However, these endpoints cannot send/receive data on their own. Instead UCX introduces the concept of *workers*, which serve as an abstraction between endpoints and network resources (i.e. NICs). Each worker can be associated with multiple endpoints.

In the original design of hadroNIO, we used a single worker for all connections. However, since there is a limit on the maximum amount of connections a worker can handle, we

refined our architecture to use multiple workers. Originally, we planned to use one worker per selector, which appeared as a natural fit, because a selector is used to query multiple channels, while a worker can progress multiple endpoints. However, NIO allows reassigning of channels to different selectors, which is not possible with UCX endpoints and workers. Ultimately, we settled on using a single worker per connection. This added complexity to our selector implementation, since it now has to poll multiple workers, but makes channels independent from selectors and allows reassignments.

C. Supporting netty write aggregation

Java NIO offers two methods for sending data via a socket channel: One only takes a single buffer, while the other one is prescribed by the interface `GatheringByteChannel` [7], thus capable of gathering write operations, accepting an array of buffers to send. Gathering writes are used heavily by netty (see chapter IV-B) to bundle multiple send requests into a single method call, in order to achieve higher throughputs. However, in the initial hadroNIO version, we implemented the gathering write method by simply looping over all buffers, sending each one separately using the single buffer write method. While this implementation worked correctly, it did not offer any performance improvements, which is why we reimplemented it. Now, as many buffers as possible are merged into a single contiguous space inside hadroNIO's outgoing ring buffer, requiring only a single UCX write request to send. This massively improved throughput rates with netty-based applications.

IV. Benchmark architecture

To evaluate the performance of different acceleration solutions with netty-based applications, we designed and implemented two microbenchmarks, using netty for connection establishment and data exchange: One is focussed on throughput while the other implements a ping-pong pattern to measure round-trip times. The benchmarks are designed to work on two nodes of a cluster environment with one acting as a server and one acting as a client. Both support on or multiple connections between server and client and each connection is handled by a separate thread. Measurements are taken per connection, and a final result, taking all measurements into account, is calculated at the end.

A. Connection setup

The connection setup is similar for both benchmarks: On startup, the server sets up a server channel to listen for incoming connections. It then waits until a specified amount of connections has been established. Once the amount is reached, all threads start sending messages at the same time (throughput benchmark) or send a single message to kick off the ping-pong pattern (latency benchmark). Before the actual benchmark starts, a tenth of the operations are executed as warm up, without taking any measurements.

The client on the other side just needs to establish the specified amount of connections and wait for the server to start the benchmark.

B. Throughput benchmark

Once all connections are set up, the server starts a separate thread for each connection, responsible only for sending messages through the respective channel. Once all warmup messages are sent, the thread waits for a synchronization message from the client, signalling that all messages have been received successfully. Each thread then needs to pass a barrier, ensuring that all threads start the benchmark at the same time. After all benchmark messages have been sent, the client once again sends a signal to server, finishing the benchmark. Times are measured once after the warmup barrier has been passed and after the second signal from the client has been received. allowing us to calculate the average data and operation throughput rates.

When sending a buffer via netty, it is not transmitted directly, but first stored in an instance of a class called `ChannelOutboundBuffer` [25], which accumulates outgoing write requests. To make sure, that data is actually transmitted, applications need to manually flush the respective channel. The data, contained in a buffer, is not copied, but only references to all outgoing buffers stored. Once netty is requested to perform a flush operation, all buffers are send with a minimal amount of write operations, using the gathering write method described in chapter III-C. This aggregation strategy allows netty to reach high throughputs without requiring any buffer copies. Our throughput benchmark can be configured to use a specific interval (e.g. every 64 buffers) for flushing a channel, allowing us to analyse performance with different amounts of aggregated buffers.

C. Latency benchmark

The latency benchmark does not start threads on its own, but makes use of netty's worker threads. Each time data is received, a worker thread invokes a method in the respective handler (instance of `ChannelInboundHandlerAdapter` [26]) and once our handler implementation has received a full message, it issues a write request, following a ping-pong pattern. We configure netty to start as many worker threads, as there are connections, with each thread opening its own selector and connections being assigned to these selectors in a round-robin fashion. This ensures, that each connection has its own thread, responsible only for handling requests on that specific connection. Times are measured before each send call and after each received message, allowing us to gather the round-trip latencies of all operations.

V. Evaluation

This section presents and discusses the evaluation results, comparing default netty performance using sockets via Ethernet versus accelerating netty with hadroNIO and libvma using 100 GBit/s high-speed NICs.

A. Evaluation setup

We used the microbenchmarks described in chapter IV for evaluating messaging performance with netty, regarding throughput, as well ass round-trip latency in two different

cluster environments. To test the scalability of each solution, we increased the connection count step-wise from 1 to 16.

Our benchmark environment consisted of two identical bare-metal nodes, provided by the Oracle Cloud Infrastructure, using the *HPC Cluster* Terraform stack [28]:

CPU	2x Intel(R) Xeon(R) Gold 6154 CPU (18 Cores/36 Threads @3.00 GHz)
RAM	384 GB DDR4 @2933 MHz
NIC	Mellanox Technologies MT28800 Family [ConnectX-5] (100 GBit/s) Ethernet
OS	Oracle Linux 7.9 with Linux kernel 3.10.0-1160
OFED	MLNX 5.3-1.0.0.1
Java	OpenJDK 17.0.2
UCX	1.12.1
hadroNIO	0.3.2
libvma	9.5.0

Fig. 2. Hardware specification of the OCI systems.

We evaluated throughput and latency with small (16 byte) mid-sized (1 KiB) and large (64 KiB) messages. For evaluating throughput, we sent 100 million messages per benchmark run, while 10 million round-trip operations were executed during each latency benchmark run. For the large buffers, we used 10 million and 1 million messages respectively and evaluated with up to 12 connections, to avoid unnecessary long running benchmarks. The amount of connections is always depicted by the y-axis, while the x-axis shows the data throughput in MB/s or GB/s when looking at throughput results, and the round-trip time in µs when evaluating latency. Each benchmark run was executed five times and the graph depicts the average values, while the error bars show the standard deviation.

B. Configuration and Optimizations

Each of the OCI nodes had two CPUs with 18 cores and 36 threads each at its disposal. To optimize performance, we used the tool *numactl* to bind the JVM process to the processor, that the network card is connected to. Since a single CPU has 18 cores, it should not be overwhelmed by 16 connections at once. The ConnectX-5 NICs were configured to use Ethernet as the link layer protocol, making them fully compatible with traditional sockets.

To improve performance regarding the throughput benchmarks, we did not flush the channels after each written message, but gave netty the chance to gather multiple message and send them at once. For small messages, we flushed each time 64 messages were written and for mid-sized and large messages, we used intervals of 16 and 4 messages respectively.

To work correctly, libvma needs to either be executed by the root user or with the CAP_NET_RAW privilege. We tried granting CAP_NET_RAW as described in libvma's README file [15], but could not get it to offload traffic. Fortunately, running as the root user worked in the OCI environment.

Additionally, we set the amount of hugepages to 800 and shmmax to 1000000000, as recommended [15]. Furthermore, libvma exposes a lot of configuration parameters, settable via environment variables. As endorsed by the libvma wiki, we set VMA_RING_ALLOCATION_LOGIC_RX and VMA_RING_ALLOCATION_LOGIC_TX to 20, which should improve multithreading performance [14]. We also needed to increase the amount of receive buffers via VMA_RX_BUFS to 800000, otherwise the benchmark would sometimes not finish with 12 or more connections, because libvma ran out of buffers. For the round-trip measurements, we set VMA_SPEC to *latency*.

For hadroNIO, we used the default configuration with 8 MiB large ring buffers and a slice length of 64 KiB.

C. Small messages (16 byte)

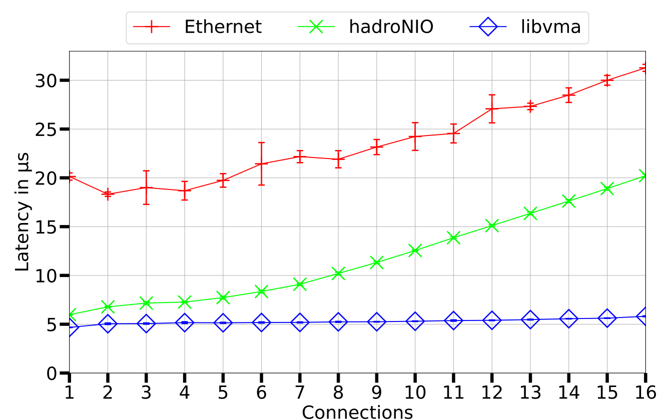

Fig. 3. Average round-trip times with 16-byte messages

Starting with 16-byte messages, Fig. 3 shows the round-trip times for traditional Ethernet, hadroNIO and libvma. As can be seen, libvma offers the best latency, with almost no overhead being generated by using multiple connections. Starting with 4.7 µs using a single connection, it still manages to yield round-trip times of 5.8 µs with 16 parallel connections.

While hadroNIO offers similarly low latencies with few connections, starting with 6 µs, it breaks the 10 µs mark using 8 connections. From there on, each additional connection adds around 1 µs of latency.

However, both acceleration solutions offer a substantial performance improvement over plain Ethernet, which starts at 20 µs using a single connection. Curiously, round-trip times fall to around 18 µs for 2-4 connections but constantly rise starting with 5 connections.

The throughput values, depicted by Fig. 4, paint a different picture. When using only one connection, all three solutions offer similar performance between 28 and 35 MB/s, with hadroNIO having a slight advantage. However, with a rising connection count, the gap between hadroNIO and Ethernet/libvma grows larger, with libvma even offering slightly lower throughput values than plain Ethernet. Starting with 13 connections, libvma almost completety stops scaling, reaching around 250 MB/s, while hadroNIO scales further to 380 MB/s using 16 connections.

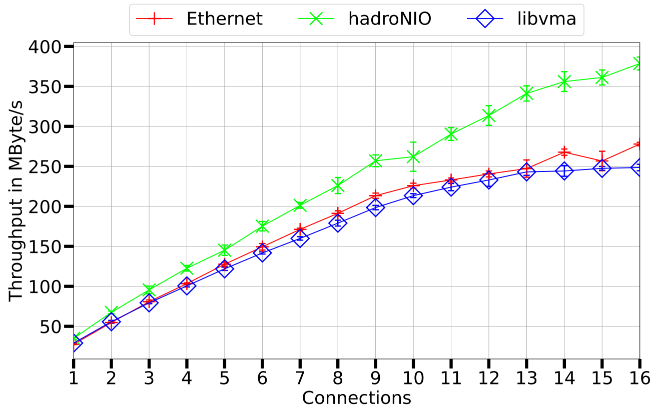

Fig. 4. Average throughput with 16-byte messages

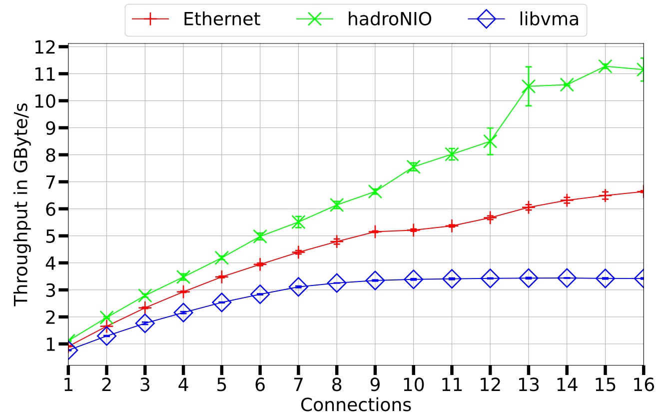

Fig. 6. Average throughput with 1 KiB messages

While libvma offers the smallest round-trip times with small messages, its throughput rates are slower than using Ethernet, whereas hadroNIO scales much better than the other candidates in our throughput benchmark.

D. Mid-sized messages (1 KiB)

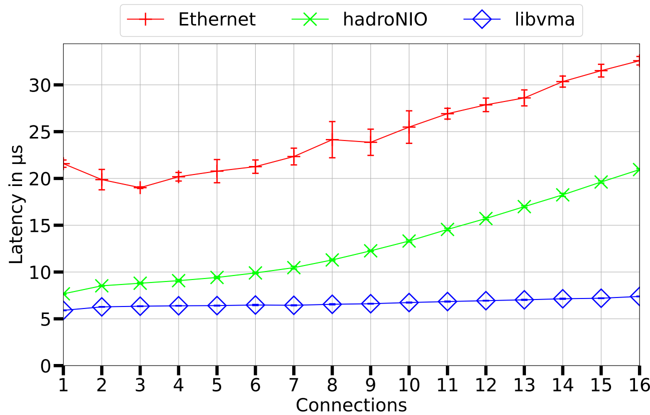

Fig. 5. Average round-trip times with 1 KiB messages

Looking at the round-trip times for 1 KiB payloads (see Fig. 5), the three solutions perform almost the same compared to the 16-byte results, apart from an offset being added to all latencies. Again, libvma scales almost perfectly, starting with 5.9 µs for a single connection and only rising to 7.4 µs using 16 connections, while hadroNIO starts with 7.6 µs, with slowly rising latencies up to 10.5 µs using 7 connections and linear increasing values from there on.

The throughput values, shown in Fig. 6, demonstrate that hadroNIO again scales well with an increasing amount of connections, reaching more than 11 GB/s at the end, thus almost saturating the 100 GBit/s hardware. On the other side, libvma scales much slower and reaches its top speed of just 3.4 GB/s with 10 parallel connections. The same throughput can be achieved using hadroNIO with only 4 connections and even using no acceleration solution at all is substantially faster, surpassing libvma's maximum throughput using 5 threads and reaching around 6.6 GB/s with 16 threads.

To conclude the evaluation of mid-sized messages, libvma continues to offer the best performance with regards to round-trip times, but comparing the results of the throughput benchmark, it falls behind hadroNIO and even plain Ethernet by far.

E. Large messages (64 KiB)

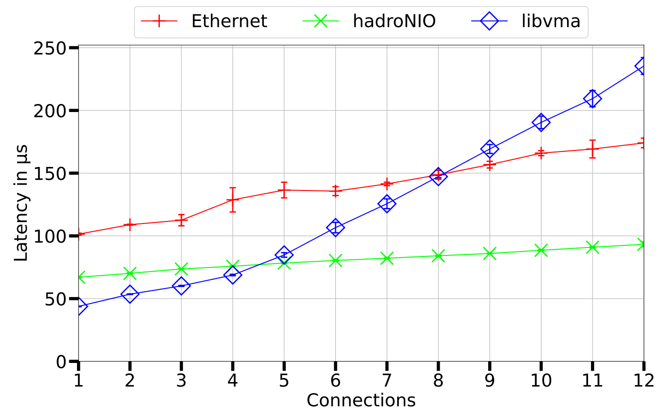

Fig. 7. Average round-trip times with 64 KiB messages

Continuing with large 64 KiB payloads, the latency results, depicted by Fig. 7, differ from the previous ones. While libvma yields the lowest round-trip times for up to 4 connections (44-69 µs), values increase faster from there on, rising by around 20-25 µs per additional connection. Starting with 9 parallel connections, libvma performs worse than plain Ethernet and the gap grows further with an increasing thread count. While hadroNIO yields higher latencies than libvma for 1-4 connections (67-76 µs), it offers the best performance using 5 or more parallel connections, reaching round-trip times of only 94 µs using 12 threads, while libvma is 2.5 times slower with around 235 µs.

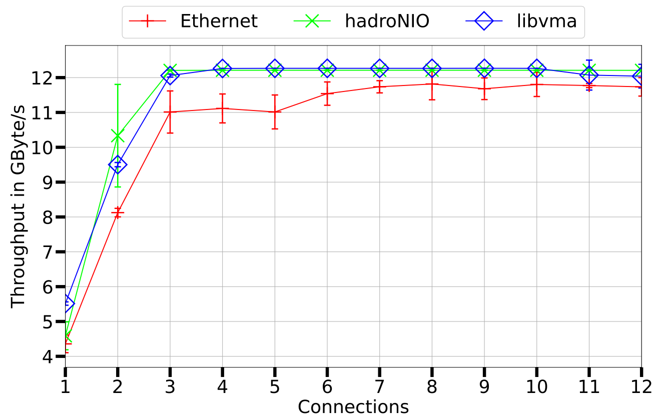

Fig. 8. Average throughput with 64 KiB messages

We close the evaluation, by looking at the throughput values using 64 KiB messages. Both acceleration solutions offer similar performance, managing to saturate the NIC with more than 12 GB/s using 3 or more connections. For a single connection, libvma is faster with 5.5 GB/s versus 4.6 GB/s, but with 11 and 12 connections, libvma becomes somewhat unstable and falls slightly behind hadroNIO. Using plain Ethernet offers acceptable performance, but 12 GB/s cannot be reached and the results are not stable, with standard deviations sometimes as high as 1 GB/s.

Concluding the large payload results, both libvma and hadroNIO are able to saturate the hardware, but regarding round-trip times, it depends on the amount of connections, which solution performs best.

VI. Conclustions & Future Work

In this paper, we presented hadroNIO extensions to support netty and compared the performance of netty based on hadroNIO versus libvma and traditional sockets over Ethernet using two microbenchmarks, for evaluating round-trip times and throughput. Our results show, that hadroNIO offers a substantial performance improvement over Ethernet on the same NIC, without needing elevated privileges or complex configurations. All results were achieved using hadroNIO's default configuration values. While libvma offers the lowest latency with small and mid-sized messages, preloading it to a netty-based application can actually worsen performance and it may not be usable in every environment due to it being dependent on CAP_NET_RAW or root privileges..

Future work includes evaluating hadroNIO with large netty-based applications and frameworks, such as Apache Cassandra and gRPC. We also aim to improve our selector implementation, by leveraging epoll, since it is currently based on busy polling. Additionally, we are working on integrating Infinileap , a UCX binding for Java, based on the experimental Foreign Function & Memory API (Project Panama) [6], into hadroNIO to see if the overhead introduced by JNI calls can be alleviated. Furthermore, we want to evaluate hadroNIO with GraalVM [29], offering low-cost interoperability between Java and native code.

VII. Acknowledgment

We thank Oracle for their sponsorship in the context of this work.
This work was supported in part by Oracle Cloud credits and related resources provided by the Oracle for Research program.

References

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.
[2] gRPC. https://grpc.io/.
[3] Java Sockets over Remote Direct Memory Access (JSOR). https://www.ibm.com/docs/en/sdk-java-technology/7?topic=networking-java-sockets-over-remote-direct-memory-access-jsorl.
[4] IBM. RDMA communication appears to hang. https://www.ibm.com/docs/en/sdk-java-technology/7?topic=problems-rdma-communication-appears-hang.
[5] IBM. RDMA connection reset exceptions. https://www.ibm.com/docs/en/sdk-java-technology/7?topic=problems-rdma-connection-reset-exceptions.
[6] Project Panama. https://openjdk.java.net/projects/panama/.
[7] Javadoc: GatheringByteChannel. https://docs.oracle.com/javase/7/docs/api/java/nio/channels/GatheringByteChannel.html.
[8] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, Apr. 2014. USENIX Association.
[9] F. P. Junqueira, I. Kelly, and B. Reed. Durability with bookkeeper. *SIGOPS Oper. Syst. Rev.*, 47(1):9–15, jan 2013.
[10] V. Kashyap. IP over InfiniBand (IPoIB) Architecture. https://www.ietf.org/rfc/rfc4392.txt, April 2006.
[11] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.
[12] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *ACM SIGCOMM Conference (SIGCOMM)*, August 2019.
[13] libvma GitHub. https://github.com/Mellanox/libvma/.
[14] VMA Parameters. https://github.com/Mellanox/libvma/wiki/VMA-Parameters.
[15] libvma README. https://github.com/Mellanox/libvma/blob/master/README.
[16] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable kernel tcp design and implementation for short-lived connections. *SIGPLAN Not.*, 51(4):339–352, mar 2016.
[17] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda. Accelerating spark with rdma for big data processing: Early experiences. In *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, pages 9–16, 2014.
[18] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance design of apache spark with rdma and its benefits on various workloads. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 253–262, 2016.
[19] OFED 3.5 release notes. https://downloads.openfabrics.org/OFED/release_notes/OFED_3.5_release_notes.
[20] Sockets Direct Protocol. https://docs.oracle.com/javase/tutorial/sdp/sockets/index.html.
[21] Statement on support of experimental verbs. https://forums.developer.nvidia.com/t/verbs-exp-h-no-such-file-or-directory/206300/2.
[22] Netty. https://netty.io/index.html.
[23] Netty related projects. https://netty.io/wiki/related-projects.html.
[24] Netty adopters. https://netty.io/wiki/adopters.html.
[25] Netty Javadoc: ChannelOutboundBuffer. https://netty.io/4.1/api/io/netty/channel/ChannelOutboundBuffer.html.
[26] Netty Javadoc: ChannelInboundBuffer. https://netty.io/4.1/api/io/netty/channel/ChannelInboundHandlerAdapter.html.

[27] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[28] Oracle Marketplace: HPC Cluster Terraform Stack. https://cloudmarketplace..com/marketplace/en_US/listing/67628143.

[29] GraalVM. https://www.graalvm.org/.

[30] Apache Ratis. https://ratis.apache.org/.

[31] F. Ruhland, F. Krakowski, and M. Schöttner. Performance analysis and evaluation of Java-based InfiniBand Solutions. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 20–28, 2020.

[32] F. Ruhland, F. Krakowski, and M. Schöttner. hadronio: Accelerating java nio via ucx. In *2021 20th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 25–32, 2021.

[33] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, et al. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[34] P. Stuedi. Direct storage and networking interface (disni). https://developer.ibm.com/open/projects/direct-storage-and-networking-interface-disni/, 2018.

[35] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14. ACM, 2013.

[36] S. Yang, S. Son, M.-J. Choi, and Y.-S. Moon. Performance improvement of apache storm using infiniband rdma. *The Journal of Supercomputing*, 75:6804–6830, 2019.

[37] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59:56–65, Oct. 2016.

[38] Z. Zhang, Z. Liu, Q. Jiang, J. Chen, and H. An. Rdma-based apache storm for high-performance stream data processing. *International Journal of Parallel Programming*, 49:671–684, 2021.

Chapter 4

Accelerating big-data applications with hadroNIO

In this chapter, hadroNIO is evaluated using real-world applications. For this purpose, performance problems observed in the previous publications were addressed. To this end, the chapter starts with optimizations in hadroNIO, including performance improvements for handling many connections.

4.1 Optimizations

While hadroNIO has been tested successfully with Netty and was even able to outperform libvma in some scenarios, it has also shown performance degradation with a rising amount of connections when latency is important. To alleviate this, multiple optimizations have been implemented in hadroNIO. The biggest performance increase has been achieved by using *Infinileap*, which is an alternative Java binding for UCX. Instead of using the traditional *Java Native Interface* like the official JUCX binding, Infinileap leverages the new *Foreign Function Interface* and *Foreign Memory API*, allowing to interact with native code more efficiently. By switching from JUCX to Infinileap, the latency increase per additional connection, observed in the previous Netty benchmark results, has been shrinked drastically. More details on the benefits of Infinileap and the new foreign access APIs are given in the following paper. Infinileap has been developed by Filip Krakowski at the Operating Systems Group at Heinrich Heine University Düsseldorf [29].

Besides this large improvement, the author spent a lot of time with smaller optimizations, in order to get the most performance out of hadroNIO. Most of them aim at reducing the amount of heap allocations, since each chunk of allocated memory may increase the time needed by the garbage collector. Especially for latency critical applications, it is crucial to keep garbage collections times as low as possible. Switching to Infinleap, combined with the other improvements, allow hadroNIO to even outperform libvma in latency focused scenarios, which was not possible before. The following optimizations have been applied:

Remove anonymous (lambda) functions
Each time an anonymous function is called in Java, a wrapper object for this function is instantiated and occupies heap memory, burdening the garbage collector. In hadroNIO, such functions were used as callbacks, which are called each time a message has been sent

or received. This can easily be avoided by outsourcing these lambda functions into their own classes, which only get instantiated once per connection.

Reduce message header size
As mentioned in the original hadroNIO paper (see page 47), each message sent by hadroNIO is preceded by a header, consisting of the buffer length and an offset, each occupying 4 bytes. This header has been shrinked to 6 bytes, by only using 3 bytes per value. This reduces the data overhead, introduced by hadroNIO, especially for small buffers. For example, if a program sends 16 bytes, hadroNIO actually transfers 24 bytes including the header, resulting in a 50% data overhead. By shaving two bytes off of the header, only 22 bytes are transferred, posing a reduced data overhead of 37.5%. Of course, the benefit of this optimization becomes negligible with larger buffers, but is a welcome improvement for small messages.

Use optimized hash set from Agrona library
When iterating over a collection in Java, a new `Iterator` instance is allocated and used to iterate over each object, one after another. In hadroNIO, registered selection keys are stored inside hash sets by the selector. In fact, multiple hash sets are needed: One to store all registered keys and one to store all keys that are currently selected for an operation. Furthermore, the selector has two copies of each of these sets. One is designated for internal use only and the other represents the public view on the keys and is meant to be accessed by the application. The public key sets are not modifiable, so that the application is not able to manipulate the selector's internal state. There is also a fifth hash set, used to store keys which are due for removal. With a select operation iterating over the set of all keys and keys due for removal, and the application needing to iterate over the set of selected keys, at least three iterators are instantiated each time the application wants to query its socket channels for updates. These allocations can be bypassed, by using the `ObjectHashSet` class from Agrona [26], which caches and reuses a single iterator instance per set. Doing so is highly dangerous for many use cases, because iterators are not thread safe. However, since the selector is synchronized according to the NIO guidelines and NIO applications in general do not access the public key sets of one selector from multiple threads, this approach works well for hadroNIO.

Cache memory segments
This optimization only concerns Infinileap. When working with the new Foreign Memory API, addresses are not handled as primitive `long` values, but are wrapped in `MemorySegment` instances. This class is used to manage a chunk of memory, denoted by an address and a size. It provides convenient methods to copy data from other memory segments and to read and write primitive data types from/to memory. It also makes sure, that accesses are within the bounds of a specific chunk of memory. Infinileap uses memory segments to interact with native memory and requires instances of this class to be passed to its send and receive methods. This results in an allocation of a `MemorySegment` instance each time hadroNIO issues a network request to Infinileap. However, at least for receiving data, this can be alleviated. Since the receive buffer slices have a fixed size and are reused over and over again, because of the ring buffer mechanism, it is possible to cache the `MemorySegment` instances, eliminating the need to allocate a new one for each receive request. This does not work for send requests, because the size of the buffers slices varies

and depends on the data sent by the application.

Fix memory leak in JUCX

When working with the Java Native Interface, it is possible to create a *global reference* to a Java object from native code. This reference can be used to access and manipulate the object. An object addressed by such a reference is guaranteed to not be garbage collected until the reference is deleted manually. JUCX creates a global reference for each request and passes this reference to the native callback method, which is called when a request is finished. The reference is needed to manipulate the status of the Java object, representing the request. Once the native callback is called, it accesses the respective object and sets the requests status accordingly (i.e finished successfully or an error code). However, JUCX does not always delete the global reference, so that the garbage collector is never allowed to free the request object, although it is not needed anymore. This causes a memory leak on the Java heap, which can cause the JVM to crash over an extended period of time. Furthermore, garbage collection times are increased, since each of the leaked objects is still checked by the garbage collector. While it was hard to find the root of this problem, it could easily be fixed and the bugfix has already been merged into the official UCX source code [30].

Remove debug log messages

For debugging purposes, hadroNIO uses a lot of log messages, especially in the selector implementation and when sending/receiving data. These log messages can be deactivated by setting the log level of the logging framework, but the method for logging these messages sill gets called. It returns quickly after checking the log level, but still poses a small overhead, just by being called. Furthermore, the messages themselves are allocated as strings on the heap, increasing the burden put on the garbage collector. To avoid this overhead, a static boolean constant, indicating if debug log messages are activated, has been introduced. Each debug logger call is then moved inside an if-clause, checking if that static constant is `true`. At first glance, this sounds like even more overhead is introduced by first checking the constant. However, since the constant's value is known at compile-time, the compiler will just completely omit these logger calls, if the value is `true`.

With all these optimizations applied, hadroNIO itself does not allocate any heap memory during send, receive and select operations, besides the `MemorySegment` instances needed for Infinileap receive requests. However, JUCX and and Infinileap may still cause allocations (especially JUCX). For example, each time a request is issued via JUCX, an object used to keep track of the request's status is allocated and returned. Normally, this is not a huge problem, since the whole Java programming language is build around allocating object instances on the heap. Only primitive data type can be stored on the stack, everything else requires allocated heap memory. Avoiding such allocations is generally hard and not how Java is intended to be used. However, in the context of high-speed networks, delays of less than 1 µs are measurable. Thus, hadroNIO aims to be a hyper-thin layer between Java NIO and OpenUCX, requiring as little resources as possible and not causing any avoidable delays. Compared to JUCX, Infinileap puts less pressure on the garbage collector, since it does not allocate any additional objects when sending/receiving messages.

With the aforementioned optimizations and the following benchmarks, hadroNIO development has been focused on scalable applications, relying on asynchronous communication. Because of that, the throughput drops observed when using blocking socket channels with hadroNIO have not been fixed. Blocking socket channels are not widely used, since they cannot be multiplexed with a selector, like non-blocking socket channels, thus they do not fit well with large scale applications. They only provide a latency advantage for applications with only one or few connections and as the previous benchmarks have shown, hadroNIO works excellent in such scenarios.

4.2 Contributions

The author implemented all optimizations by himself. For the evaluation part, an existing benchmark was used (Yahoo! Cloud Serving Benchmark [31]). However, some wrapper code was needed to control the benchmark and convert results into into the CSV-format. This was also implented by the author. The benchmarked gRPC application is based on a simple key-value store, supporting only a single server, implemented by Carl Mastrangelo [32]. It has been modified and transformed into a distributed key-value store, supporting multiple servers.

The hadroNIO evaluation paper has been written by the author, while Filip Krakowski and Dr. Michael Schöttner took part in many discussions about the performance analysis of hadroNIO and provided valuable input regarding some of the applied optimizations.

Infinileap has been developed by Filip Krakowski, who has also written the corresponding paper. However, the author participated in many discussions about its development and contributed several bugfixes and improvements.

Infinileap: Modern High-Performance Networking for Distributed Java Applications based on RDMA

Filip Krakowski
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
filip.krakowski@hhu.de

Fabian Ruhland
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
fabian.ruhland@hhu.de

Michael Schöttner
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—In this paper, we propose Infinileap, a modern networking framework enabling high-performance memory transfer mechanisms like Remote Direct Memory Access (RDMA) for applications written in Java. Infinileap is based on the Open Communication X (UCX) framework, which is accessed from Java. This is accomplished through Oracle's Project Panama, which is currently in the preview phase and aims to significantly improve interoperability between Java and "foreign" languages, such as C. In contrast to often used internal and unsupported JDK APIs, Project Panama's APIs are explicitly intended for use and developers are encouraged to adapt their existing code accordingly. Using Project Panama, we implement an object as well as future-oriented framework based on UCX. Our experiments show that Infinileap and thus Project Panama's innovations work reliably and efficiently under heavy load and also, within benchmarks implemented for this purpose based on the Java Microbenchmark Harness (JMH), achieve very good performance results with over 110 million messages per second and round-trip latencies below two microseconds with a single ConnectX-5 InfiniBand (single-port) network interface controller.**

***Index Terms*—OpenUCX, Project Panama, Java, InfiniBand, Remote Direct Memory Access**

I. Introduction

Driven by the ever-increasing demands that modern distributed applications place on their underlying systems, RDMA-enabled hardware like *InfiniBand* is increasingly being adopted in more and more areas of cloud computing. Public platforms such as Amazon Web Services or Microsoft Azure already offer instances that support RDMA. In addition to the usual advantages such as low latency and high bandwidth, this type of hardware also offers offloading techniques such as tag matching or adaptive routing, which relieve the system's CPU and thus allow more computing time for application threads. Work is also in progress on so-called Data Processing Units (*DPUs*), which aim to perform data reception and transmission as well as programmable computations directly on a SmartNIC like NVIDIA's Bluefield-2 without having to use the PCI bus for larger data transfers. All these technologies have in common that, from the developer's point of view, the programming differs significantly from traditional socket programming.

On the one hand, low-level user-space libraries like *libibverbs* allow full and direct control over InfiniBand hardware, but on the other hand, these require a lot effort and expertise on hardware details in order to achieve reasonable results regarding network latency and throughput - especially for tuning configuration parameters.

Considering this background, the Unified Communicaton X (*UCX*) project was founded by leading industrial as well as academic institutions, addressing the mentioned challenges[1]. As the name suggests, the project aims to unify network communication between heterogeneous systems using different transport techniques (including RDMA), under a single abstract interface thus making the aforementioned advantages more accessible to the masses. Instead of different APIs for different transports, the developer is provided with one API for many transports. This allows programs based on the UCX framework to be executed on different computer and network architectures without changing the program code.

Many of the Big Data frameworks available today, such as Apache Spark or Apache Flink, continue to use ordinary socket communication for data exchange between cluster participants. This is no different for message broker services such as Apache Kafka or latency-critical coordination services such as Apache ZooKeeper. This is not because the developers behind these projects do not want to use fast interconnects, but rather because they cannot use them easily. All of the aforementioned projects are based on the Java platform, which does not yet offer the possibility of network communication outside the domain of ordinary sockets. This circumstance could be improved by the introduction of OpenJDK's Project Panama, which pursues the goal of being able to communicate with native libraries from Java as well as work with native memory outside the Java domain. Most of the functionality has already been rolled out with the release of OpenJDK 16 in incubator status and can therefore be tested with the official releases.

The contribution of this paper is *Infinileap*, a modern object-oriented networking framework based on UCX and purely written in Java. It enables Java-based distributed systems to use RDMA as well as other functionalities such as tag matching or atomic operations on remote memory. Unlike previous work, Infinileap relies on cutting-edge future-proof technologies, provides users with an easy-to-use API that greatly simplifies programming in the context of RDMA and is publicly available under an open-source license[2]. The core focus of our

framework lies within an easy integration into existing projects as well as the availability of the source code itself. Initial experiments also show that the use of this new technology is efficient and reliable under load. Similarly, we show by means of our benchmarks running on the Java Virtual Machine (JVM) that message rates of over 100 million messages per second between two Network Interface Controllers (NICs) are within the realm of possibility.

II. Related Work

The idea of accelerating applications in the Java domain by means of RDMA has already been studied in literature. Very prominent are custom implementations of the Apache Spark Shuffle Manager, which distributes the data to be processed within the Spark cluster, using RDMA for data transport[3]–[7]. Efforts have also been made to accelerate message broker services such as Apache Kafka using RDMA[8]. On the one hand, these implementations offer a high increase in performance, but on the other hand, they are not publicly available. Likewise, according to the architectures described, they are highly tailored to their intended use and thus cannot be readily deployed in third-party projects.

Another type of accelerated network communication was also investigated using socket-intercepting plugin mechanisms[9], [10]. In this case, ordinary socket calls were intercepted and passed on to the InfiniBand hardware by means of suitable operations. A major advantage of this approach is that the code of existing applications does not have to be modified. Likewise, however, the user is still bound to the semantics of ordinary sockets and thus can not explicitly perform RDMA operations or even control how they are executed by specifying configuration parameters available at the native layer.

Another research focus, which has formed in the area of RDMA within the Java domain, is the efficient connection of native libraries for the use of functionalities which would not be available otherwise. Many of the solutions implemented in this field establish a Java Native Interface-based connection to the native *libibverbs* library in order to be able to use functionalities of RDMA-capable network cards on the Java side[11]–[13]. Within the experiments carried out by the authors, very good results were also achieved in this case with regard to data throughput, latency as well as scalability. An important fact that should not be neglected is that previous projects have always relied on Java's Unsafe API for accessing off-heap memory that is not managed by the Java Virtual Machine, which is not well received within end-user software[14]. Lastly, there are frameworks providing a binding for the native libibverbs library based on the Java Native Interface as well as the Unsafe API, which aim at making native structs accessible in Java by means of so-called proxy objects and thus grant access to the functionality of the native library in a structured way[15].

III. Project Panama

The ability to call native functions from Java has been existing for a long time and has evolved from the Native Method Interface (NMI) in version 1.0 of the JDK, which was subsequently removed in version 1.2, to the Java Native Interface (JNI). The use of the JNI is challenging. As a basis for the native interface a wrapper around the native code that is to be called from Java must be implemented using the native programming language and compiled into a shared library for all targeted platforms. In addition, during the implementation of this wrapper code, some important properties of the JNI must be taken into account, otherwise the overhead of the interface can have a strong negative impact on the performance of the application[16], [17]. Likewise, there are limitations regarding the access to native memory. The `java.nio.ByteBuffer` class used for this purpose uses a 32-bit value as the offset for accessing individual values in the underlying memory block. Since Java also works exclusively with signed primitive data types, this results in the restriction that a single memory block can contain a maximum of two gigabytes of memory. To work around these limitations, many applications that need to work with native memory resort to using the JDK's internal Unsafe APIs, which offer no guarantees of support or future availability and may crash the JVM if handled wrong.

Oracle intends to solve these problems with its Project Panama in the form of a Foreign Linker API[18] as well as a Foreign Memory Access API[19]. Both components together allow the developer to call native functions from Java without adding an additional external layer, as well as manage native memory without practical size constraints. A central tool for linking native functionality in Java is Project Panama's `jextract` and its ability to parse C header files of existing native libraries and generate Java code from them that reflects the defined functions as well as data structures.

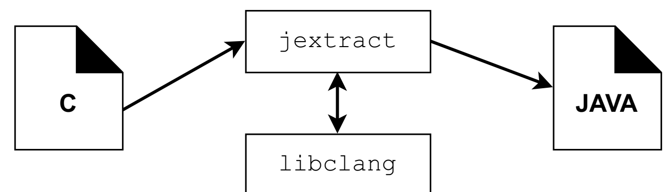

Fig. 1. Jextract's process of generating native bindings.

The foundation for this is the native `libclang` library, which, with the help of components provided by Project Panama, is used to generate an abstract syntax tree (AST) of the header file to be processed. Subsequently, the resulting AST is used to generate Java code that reflects the elements it contains. This process is depicted in Figure 1. The resulting source code can be used afterwards to call native functions and to allocate and manipulate native structs. It should also be noted that `jextract` is not a mandatory component, but merely assists the programmer in creating bindings. The following basic building blocks are provided by Project Panama for the integration of native functions as well as data structures.

- `jdk.foreign.incubator.MemoryAddress`

  A simple wrapper class which stores a memory address in the form of a primitive `long` value.
- `jdk.foreign.incubator.MemorySegment`

  A class that describes a memory segment including its access rights and owner or associated thread. An instance of `MemorySegment` can only be accessed by the associated thread, but ownership can be given to other threads.
- `jdk.foreign.incubator.MemoryAccess`

  A helper class which allows to read and write individual values within the memory area of a `MemorySegment` instance.
- `jdk.foreign.incubator.MemoryLayout`

  A class which is used to describe layouts within memory. This is primarily used to describe the layout of native structs.
- `jdk.foreign.incubator.CLinker`

  A class that allows to look up symbols within a shared library and link instances of the `java.lang.invoke.MethodHandle` class to them, in order to call them afterwards.

The code generated by `jextract` uses the preceding classes to establish the native interface. Building on this code, we then implement Infinileap.

IV. Infinileap Architecture

This section describes Infinileap's design and addresses which obstacles currently exist, how they are solved, and which opportunities will exist in the future.

A. Framework Design

Infinileap builds on top of the aforementioned `jextract` tool for generating the native interface and offers the developer an object-oriented API for using UCX. In addition, some helper functions exist to facilitate the processing of requests.

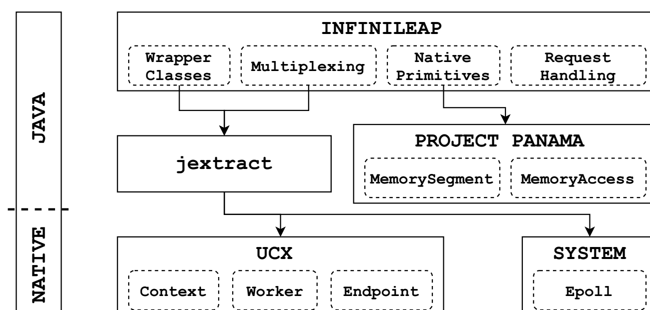

Fig. 2. Infinileap's architecture and dependencies.

The architecture as well as its individual components and dependencies are shown in Figure 2 and can be described as follows.

- **Wrapper Classes**
  The UCX library nearly always works with handles (i.e. memory addresses) within the high-level API, so that a high compatibility between the individual versions can be guaranteed. For these handles, Infinileap provides wrapper classes that bundle the functions belonging to the corresponding category (`Context`, `Worker`, `Endpoint`, etc.) using the bindings generated by `jextract`. These wrapper classes extend a common super class `NativeObject`, which, by implementing `java.lang.AutoCloseable`, allows resources, such as configuration parameters, to be temporarily created using try-with-resources statements and automatically released afterwards. The Infinileap API only accepts instances of these wrapper classes and thus prevents the incorrect use of memory addresses which can lead to segmentation faults and program crashes.

- **Multiplexing**
  The UCX API provides the developer with two mechanisms for asynchronous processing of requests. Both use the underlying `Worker` abstraction of the framework. The first and simpler variant is to use existing functions of the framework to wait for new events of the worker. Internally, the framework uses multiplexing functions of the operating system for this purpose. Since it is only possible to wait for a specific worker and several of these workers can exist, a filedescriptor belonging to the worker can also be queried and used for polling with `epoll`. To provide this second variant within the Infinileap API, the necessary `epoll` functions are also provided using the bindings generated by `jextract` at the Java level in the form of an object-oriented API.

- **Native Primitves**
  Project Panama provides with its `MemoryAccess` class the possibility to read and write single values at specific memory addresses. Since UCX provides an API for performing atomic operations on 4 and 8 byte values within remote memory, this class is an excellent foundation for it. To avoid errors in function calls, classes (referred to as `Native Primitives` in Figure 2) are developed that represent primitive values in the form of objects (similar to Java's boxing) and manipulate them using the `MemoryAccess` class. Just like the previously mentioned wrapper classes, instances of these classes are accepted within the Infinileap API to perform atomic operations on remote memory.

- **Request Handling**
  Latency-critical systems often work with synchronous network APIs, since the overhead caused by adding asynchronous mechanisms is unacceptable. UCX returns a handle (i.e. memory address) to a request for each network operation. This can subsequently be used to query the current status of the operation and thus achieve very low latency times by means of busy-polling. For this purpose Infinileap provides helper functions which wait for the completion of a request by means of busy-polling. This allows the developer to implement synchronous communication within an application and at the same time provides a simple abstraction for processing network operations.

B. Garbage Collector Overhead

One of the JVM's mechanisms which can have a negative impact on performance is garbage collection. In the software development context, Java's garbage collector offers a major advantage over languages without automatic memory management. For example, developers do not have to worry about freeing allocated memory, since it is automatically freed by the JVM as soon as it is no longer reachable. In performance-critical systems, however, this process can have a strong negative impact on the execution of the program. This is due to the fact that so-called "Stop the World" events occur, which stop the application threads in the course of cleanup.

Project Panama's `jextract` tool generates bindings which, if a pointer is returned from the native code, create an instance of the `MemoryAddress` class and store the returned value in it. In the case of a few calls, such as for configuration or establishing connections, this situation does not have a negative effect, since the garbage collector only has to release comparatively few objects. When network operations are executed, which can occur several million times per second, a large number of references are generated on the Java heap at the same time, which place a heavy load on the garbage collector.

We address this problem by using only primitive data types, which do not create objects in the heap managed by the JVM, for parameter and return types in the data path (i.e. sending and receiving data) of our framework. To achieve this we slightly modify the bindings generated by `jextract`, in the case of the data-path functions, so that they return values of type `long` (64 bit value) instead of `MemoryAddress` instances. This prevents the creation of references to objects that the garbage collector would otherwise have to clean up leading to "Stop the World" events.

Another way to address this problem is described by the Java Enhancement Proposal on so-called *Primitive Objects*[20]. Unlike ordinary Java objects, instances of primitive objects are treated just like primitive data types, and instead of the Java heap, the stack is used for storage. Since it often makes sense to encapsulate primitive data types in objects for abstraction reasons, such as in the case of Project Panama's `MemoryAddress` class, Primitive Objects provide a very good solution for avoiding garbage collector overhead due to the elimination of ordinary object overhead, while also being more memory efficient. After rolling out this new feature, Project Panama's `MemoryAddress` class would be a good candidate for adoption, as this would eliminate the need to allocate additional ordinary objects within `jextract`-generated bindings for returning memory addresses on the Java heap.

V. Experiments

In this chapter, we first present the architecture of our implemented benchmarks and show which problems have been solved. Afterwards, we take a closer look at the test setup and the subsequent results of all benchmarks and analyze them.

A. Benchmark Implementation

Since Java is a dynamically compiled programming language, the runtime behavior is often unpredictable and can change between individual program calls, which can lead to unexpected results, especially in the case of benchmarks. For example, the JVM uses a just-in-time compiler (JIT), which compiles the generated intermediate code (Java bytecode) into platform-dependent machine code at runtime. This has the great advantage that the intermediate code can be analyzed at runtime and thus optimizations can be made based on findings from real code behavior. In the context of benchmarks, however, this behavior can turn into a disadvantage, since the functions implemented by the developer can, in the worst case, be removed entirely by optimizations and results thus do not reflect expectations[21].

In order to address these challenges, the OpenJDK project provides the Java Microbenchmark Harness[22] (JMH), which is a framework for the development and execution of benchmarks written in Java. In addition to many configuration options as well as a simple API for third-party applications, it also provides a rich set of examples that present and explain best practices in benchmark development.

Since JMH's intended use lies primarily in the area of local microbenchmarks, we need to add a thin application layer enabling it for the use in distributed benchmarks over the network. JMH provides phases for initialization as well as release of resources, which are very suitable for establishing connections between network partners. Likewise, the already existing support for multithreaded benchmarks is used to create multiple connections in different threads and for utilizing the available processor cores.

As a counterpart to the client on which JMH is executed, we implement a server application that responds to the client's requests using a simple protocol and performs appropriate actions. The basic flow of a benchmark run can be described as follows.

- **SETUP** - Phase in which resources such as threads and buffers are initialized.
  1) Send a `START_RUN` command to tell the server to start the next or first run.

2) Send a configuration to the server which includes the number of threads, the buffer or message size, the number of operations and the type of operation which will be executed.

- **RUN** - Phase in which JMH invokes benchmark methods and makes measurements.
  1) Execute the specified number of benchmark method invocations until a configured time has expired.
  2) Synchronize with the server to let it receive new commands.
- **TEARDOWN** - Phase in which benchmark resources are released.
  1) Send a `END_RUN` command to tell the server to release its resources and terminate all worker threads.
  2) Send a `SHUTDOWN` command to tell the server it should terminate. Alternatively another benchmark run may be started by sending a `START_RUN` command and starting again from the beginning.

B. Benchmark Setup

CPU	1x Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz (22 MB Cache)
RAM	4x Micron Technology 36ASF2G72PZ-2G6E1 16GB
NIC	1x Mellanox Technologies MT27800 Family [ConnectX-5] (100Gbit/s)

Fig. 3. System specifications of the hardware used in all experiments.

OS	CentOS Linux release 8.1.1911 (Core)
JDK	OpenJDK 17-internal (commit 75329169a407)
UCX	UCX 1.9.0 stable (commit cd9efd3d80ec)

Fig. 4. Operating system and software versions used in all experiments.

All benchmarks are executed using two identical machines consisting of the hardware shown in Figure 3 and using the software specified in Figure 4. Both machines are connected back-to-back, which means that both InfiniBand network cards are directly connected to each other without adding a switch in between.

C. Latency Benchmark

In the case of the latency benchmark, each operation is executed exactly once and then waited for completion. The time measured in between represents the latency. The UCX framework considers some operations to be complete as soon as the associated buffer can be reused by the application. Since this time does not reflect the true network latency, these operations (`WRITE` and `SEND`) are measured using a ping-pong pattern and thus represent the round-trip time. Since both machines used in the benchmark are identical, the one-sided latency (**which is not shown in Figure 5**) can be determined by dividing by two.

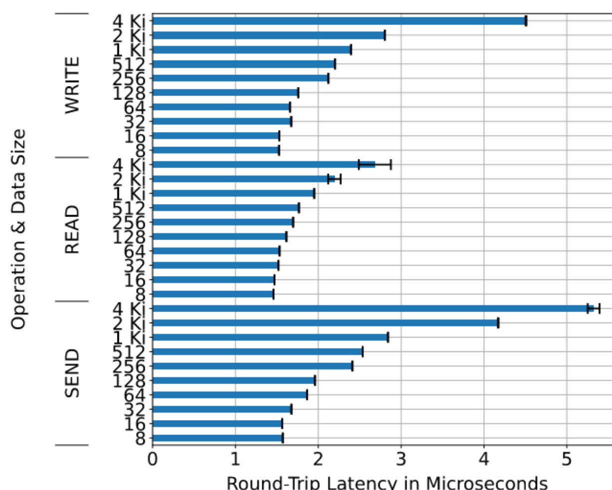

Fig. 5. Average Round-trip latency for RDMA write, RDMA read and send operations measured in microseconds per operation.

Figure 5 visualizes the average round-trip latency for read, send and write operations. It should be noted that, because of the aforementioned assumptions made by the UCX framework, in case of `SEND` and `WRITE` operations the message/buffer is sent/written to the destination and back again, while `READ` operations only send a small protocol message to the destination and finally receive the requested data. This explains the comparatively short round-trip latency when reading from remote memory.

As can also be seen, small amounts of data up to 128 bytes can be sent to the destination and back as well as written in under two microseconds, which is particularly beneficial in latency-critical applications. For data sizes above 128 bytes, it can be seen that the latency increases more significantly with each doubling of the size compared to sizes below 128 bytes. This can be explained by the fact that the UCX framework uses inlining for small messages, and the network interface controller therefore does not have to read them via the PCI bus, but can retrieve them directly from its integrated memory. Nevertheless, the latency times are always within a reasonable range and show that it is possible, for example, to send an entire memory page (4 Ki) to a remote destination in about 2.6 microseconds (RTT divided by two).

Since Infinileap also supports all available atomic operations on remote memory provided by the UCX framework, we measure their latencies as well. Unlike the operations mentioned so far, atomic operations can be performed with fetching semantics and thus wait for the result of the full operation. We use this mode to measure the true latency of all operations. In the case of a compare and swap (`CSWAP`) operation, the old value that was stored before the swap is thus returned within the operations result. Atomic operations are principally only possible with 4 or 8 byte values on the side of the UCX framework. The latencies of all supported atomic

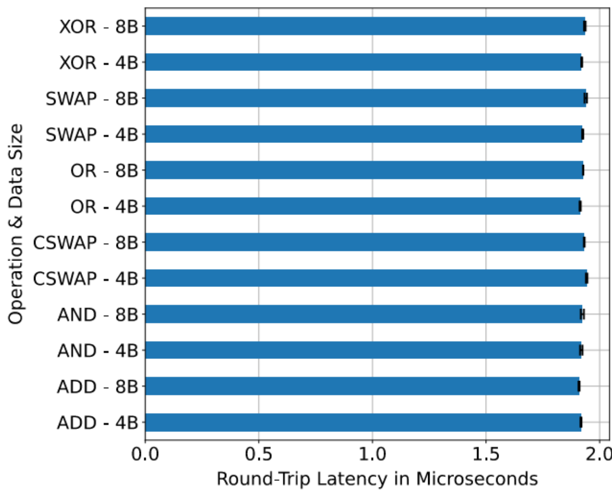

Fig. 6. Average operation latency for all UCX-supported atomic operations and data sizes.

operations are shown in Figure 6. What is remarkable in this regard is that all types of operations always require less than two microseconds to execute. At the time of this work, only two machines with the aforementioned network interface controllers were available, which means that an evaluation with the addition of lock contention was not possible. In principle, however, the results obtained suggest that the supported atomic operations can be very well used in, for example, coordination services.

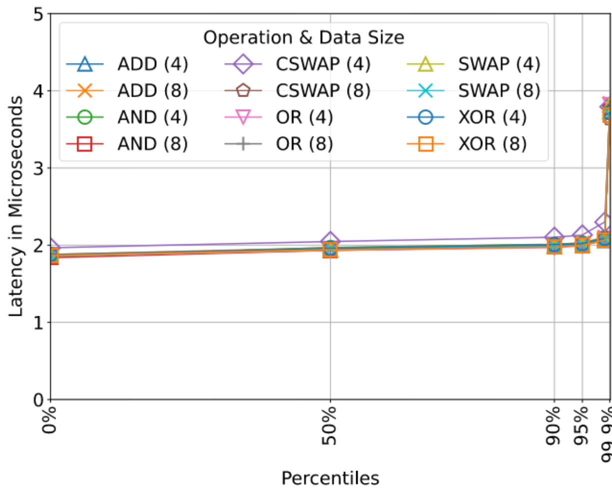

Fig. 7. Operation latency for all supported atomic operations by percentiles.

In addition to measuring the average latency of individual atomic operations, the distribution of times by percentiles is also measured. These are presented in Figure 7. Here it can be well observed that almost all kinds of atomic operations are performed within two microseconds in 99% of the cases (label omitted within the graph due to space constraints). Similarly, only 0.1% of atomic operations take longer than 3.7 microseconds to complete. An exception is the compare and swap operation on 4-byte data values, since it always has a small offset to all other measured latencies. Apart from this, it can finally be concluded that atomic operations can be used well due to their similar latency compared to ordinary operations and the stability of these times.

D. Throughput Benchmark

In addition to the latency benchmark, which measures average times related to individual operations, a throughput benchmark, which measures the number of operations per second, has been implemented, too. This involves continuously executing batches of operations and measuring how often these batches are completed per second. Due to the already mentioned assumptions of the UCX framework regarding completions, mechanisms for detecting true completions are also used here in the case of `WRITE` and `SEND` operations. Specifically, this means that the server sends a message back to the client after receiving all operations, so that the client can measure the time between the first operation and this received message. In addition, the benchmark is executed with different numbers of threads, each of which manages a single connection. For this, a thread pool is also used on the server side, which uses one thread per connection.

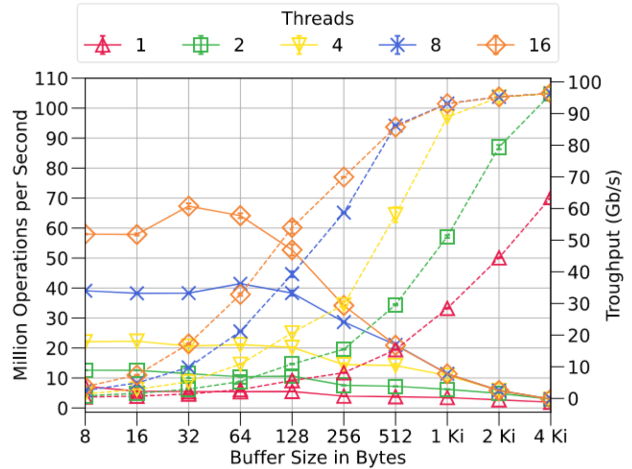

Fig. 8. Average RDMA write operation throughput in million operations (solid line) and gigabit per second (dashed line).

Figure 8 depicts the average write operation throughput in millions of operations as well as gigabits per second. It can be observed that the addition of connections leads to a high increase in throughput in each case, due to each thread working independently from another. For example, a single thread with a buffer size of four kilobytes achieves a throughput of about 63.5 gigabits per second while two threads perform

the same task with about 96.2 gigabits per second and are thus close to the theoretical maximum of 100 gigabits per second. Furthermore, it can be observed that writing many small buffers of sizes 8 and 16 bytes in parallel using 16 Threads leads to performance degradation.

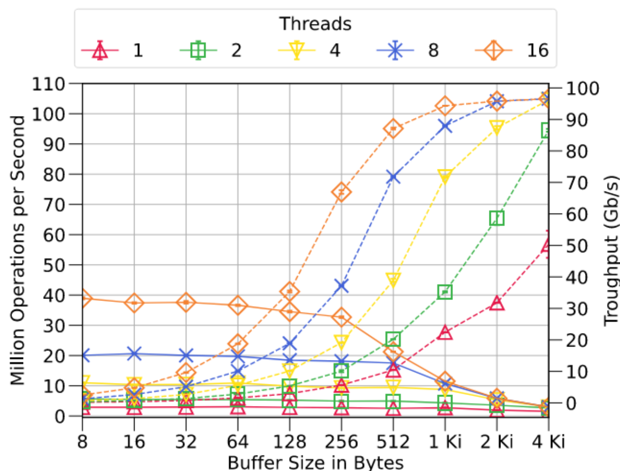

Fig. 9. Average RDMA read operation throughput in million operations (solid line) and gigabit per second (dashed line).

Unlike writing data to remote storage, reading involves overhead due to the protocol used and therefore results in comparatively lower throughputs. To complete each operation, a protocol message must first be sent to the remote NIC, which then sends back the requested data. The resulting overhead leads to 30 to 50 percent lower throughput compared to write operations in case of small messages. Figure 9 shows the measured results regarding read operation throughput. For messages smaller than 512 bytes, a plateau can be seen in the operation throughput due to the aforementioned overhead, while for messages 512 bytes and larger, the message throughput drops sharply as the bandwidth gradually becomes saturated.

The best results are achieved with send operations in the throughput benchmark. The results measured here are provided in Figure 10. While again saturating the bandwidth with multiple threads and large message sizes, in the case of small messages between 8 and 16 bytes, message throughputs of approximately 110 million messages per second are achieved when using 16 threads. This feature shows that the framework can be used particularly well in RPC systems with very small payload sizes. The temporary, comparatively easier increase in throughput when switching from 128 to 256 byte messages is also noticeable. This can be explained by the fact that the UCX framework uses different internal copy mechanisms (inlining, intermediate buffer and zero copy) for the user data, depending on the message size, and a change could have taken place at this point. In conclusion, all results are within a good

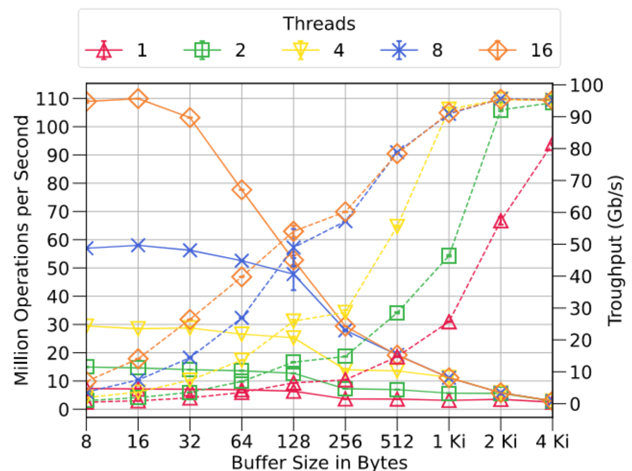

Fig. 10. Average send operation throughput in million messages (solid line) and gigabit per second (dashed line).

range, giving applications a lot of potential to speed up their communication.

VI. Conclusion & Future Work

In this paper we propose Infinileap, an easy-to-use and modern network communication framework for Java developers building on top of UCX that enables technologies previously unavailable in Java, such as RDMA. Instead of internal and not officially supported APIs, it relies on new and future-proof APIs developed within Oracle's Project Panama to connect native functionalities with Java. We also show that even in a dynamically compiled language such as Java, very good performance results are possible, such as 110 million messages per second as well as round-trip latencies below two microseconds using a single single-port InfiniBand NIC.

In the future, we plan to integrate Infinileap into existing larger Java-based distributed systems such as Apache ZooKeeper, Apache Spark, or Apache Kafka to accelerate network communication and leverage RDMA functionality at appropriate locations. We expect that the results obtained here will justify the use of high-performance interconnects such as InfiniBand within the Java programming language and thus lead one step closer to adoption.

VII. Acknowledgements

We thank the UCX developers as well as all developers involved in Project Panama for valuable discussions. We also especially thank Oracle for their sponsorship in the context of this work. Computational infrastructure and support were provided by the Centre for Information and Media Technology at Heinrich Heine University Düsseldorf.

REFERENCES

[1] P. Shamis, M. G. Venkata, M. G. Lopez, *et al.*, "Ucx: An open source framework for hpc network apis and beyond," (Aug. 26–28, 2015), IEEE, Aug. 26–28, 2015, pp. 40–43, ISBN: 978-1-4673-9160-3. DOI: 10.1109/HOTI.2015.13.

[2] F. Krakowski and F. Ruhland, *Infinileap GitHub Repository*. [Online]. Available: https://github.com/hhu-bsinfo/infinileap.

[3] B. Liu, F. Liu, N. Xiao, and Z. Chen, "Accelerating spark shuffle with RDMA," in *2018 IEEE International Conference on Networking, Architecture and Storage, NAS 2018, Chongqing, China, October 11-14, 2018*, IEEE, 2018, pp. 1–7. DOI: 10.1109/NAS.2018.8515724.

[4] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda, "High-performance design of apache spark with rdma and its benefits on various workloads," (Dec. 5–8, 2016), IEEE, Dec. 5–8, 2016, pp. 253–262, ISBN: 978-1-4673-9006-4. DOI: 10.1109/BigData.2016.7840611.

[5] H. Li, T. Chen, and W. Xu, "Improving spark performance with zero-copy buffer management and rdma," (Apr. 10–14, 2016), IEEE, Apr. 10–14, 2016, pp. 33–38, ISBN: 978-1-4673-9956-2. DOI: 10.1109/INFCOMW.2016.7562041.

[6] X. Lu, M. Wasi-ur-Rahman, N. S. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with RDMA for big data processing: Early experiences," in *22nd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2014, Mountain View, CA, USA, August 26-28, 2014*, IEEE Computer Society, 2014, pp. 9–16. DOI: 10.1109/HOTI.2014.15.

[7] Y. Wang, C. Xu, X. Li, and W. Yu, "Jvm-bypass for efficient hadoop shuffling," (May 20–24, 2013), IEEE, May 20–24, 2013, pp. 569–578, ISBN: 978-0-7695-4971-2. DOI: 10.1109/IPDPS.2013.13.

[8] M. H. Javed, X. Lu, and D. K. Panda, "Cutting the tail: Designing high performance message brokers to reduce tail latencies in stream processing," (Sep. 10–13, 2018), IEEE, Sep. 10–13, 2018, pp. 223–233, ISBN: 978-1-5386-8320-0. DOI: 10.1109/CLUSTER.2018.00040.

[9] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, "Sockets direct protocol over infiniband in clusters: Is it beneficial?," (Mar. 10–12, 2004), IEEE, Mar. 10–12, 2004, pp. 28–35, ISBN: 0-7803-8385-0. DOI: 10.1109/ISPASS.2004.1291353.

[10] IBM, *Java sockets over remote direct memory access*. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSYKE2_7.1.0/com.ibm.java.lnx.71.doc/diag/understanding/rdma_jsor.html.

[11] S. Nothaas, K. Beineke, and M. Schoettner, "Leveraging infiniband for highly concurrent messaging in java applications," (Jun. 3–7, 2019), IEEE, Jun. 3, 2019, pp. 74–83, ISBN: 978-1-7281-3802-2. DOI: 10.1109/ISPDC.2019.00013.

[12] P. Stuedi, B. Metzler, and A. Trivedi, "JVerbs: Ultra-Low Latency for Data Center Applications," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, New York, NY, USA: Association for Computing Machinery, 2013, ISBN: 9781450324281. DOI: 10.1145/2523616.2523631.

[13] W. Huang, H. Zhang, J. He, J. Han, and L. Zhang, "Jdib: Java applications interface to unshackle the communication capabilities of infiniband networks," (Sep. 18–21, 2007), IEEE, Sep. 18–21, 2007, pp. 596–601, ISBN: 978-0-7695-2943-1. DOI: 10.1109/NPC.2007.111.

[14] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: The java unsafe API in the wild," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, J. Aldrich and P. Eugster, Eds., ACM, 2015, pp. 695–710. DOI: 10.1145/2814270.2814313.

[15] F. Krakowski, F. Ruhland, and M. Schöttner, "Neutrino: Efficient infiniband access for java applications," (Jul. 5–8, 2020), IEEE, Jul. 5–8, 2020, pp. 12–19, ISBN: 978-1-7281-8947-5. DOI: 10.1109/ISPDC51135.2020.00012.

[16] D. Kurzyniec and V. Sunderam, "Efficient cooperation between java and native codes – jni performance benchmark," in *In The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[17] M. Dawson, G. Johnson, and A. Low. "Best practices for using the java native interface." (Jul. 7, 2009), [Online]. Available: https://developer.ibm.com/articles/j-jni.

[18] M. Cimadamore. "Jep 393: Foreign-memory access api (third incubator)." (Sep. 21, 2020), [Online]. Available: https://openjdk.java.net/jeps/393.

[19] M. Cimadamore. "Jep 389: Foreign linker api (incubator)." (Jul. 20, 2020), [Online]. Available: https://openjdk.java.net/jeps/389.

[20] D. Smith. "Jep 401: Primitive objects (preview)." (Aug. 13, 2020), [Online]. Available: https://openjdk.java.net/jeps/401.

[21] K. Shiv, R. Iyer, C. Newburn, J. Dahlstedt, M. Lagergren, and O. Lindholm, "Impact of jit/jvm optimizations on java application performance," IEEE, 2003, pp. 5–13, ISBN: 0-7695-1889-3. DOI: 10.1109/INTERA.2003.1192351.

[22] OpenJDK, *Java microbenchmark harness*. [Online]. Available: https://github.com/openjdk/jmh.

Transparent network acceleration for big data computing in Java

Fabian Ruhland
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
fabian.ruhland@hhu.de

Filip Krakowski
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
filip.krakowski@hhu.de

Michael Schöttner
Department Operating Systems
Heinrich Heine University
Düsseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—HPC and cloud data centers offer an increasing amount of cores per CPU, GPUs and high-speed networks like InfiniBand with up to 400 Gbit/s. Scaling out big-data computing is mostly limited by the network performance. However, many big data frameworks are written in Java (often using netty), which cannot fully exploit the performance of such networks. The reason is found in Java NIO, which is based on traditional sockets, while InfiniBand provides *ibverbs*, a totally different interface, to the operating system and applications. This challenge has been addressed by different approaches, providing transparent and non-transparent acceleration via high-speed NICs, many of them no longer maintained.**

In this paper, we propose *hadroNIO*, a Java library, providing transparent network acceleration for Java NIO applications, based on *Unified Communication X* (UCX). The latter is written in C, providing efficient access to different network technologies. hadroNIO has been extended to use Infinileap for efficiently accessing UCX. Infinileap is using the new Foreign Function & Memory APIs of Oracle's Project Panama to access native code.

Our evaluation results show, that hadroNIO allows netty to achieve round-trip times of less than 5 µs on a 100 GBit/s network and efficiently handle hundreds of connections per server. We compare the raw performance of hadroNIO with traditional Java sockets and libvma using a netty microbenchmark and include experiments with gRPC and Apache ZooKeeper. The measurements show, that hadroNIO outperforms existing solutions, while being transparent for applications and developers.

***Index Terms*—High-speed Networks, Cloud Computing, Ethernet, InfiniBand, OpenUCX, Java**

I. Introduction

With increasing CPU core counts and the availability of high-speed networks, distributed applications need to be scalable to take advantage of modern hardware. Java and its library ecosystem provide developers with the tools to write scalable distributed applications. Programmers can choose to write low-level network code using Java NIO (e.g. Apache ZooKeeper [5]), or implement their projects using high-level RPC frameworks, such as *gRPC* [4]. However, most modern big-data Java applications are based on *netty* [30] (e.g. Apache Cassandra [18], Apache Bookkeeper [13]), which offers full control over the data being sent and received, while its event-driven architecture abstracts the complexity of Java NIO. It utilizes the CPU to its full potential by executing multiple event loops, each in its own thread, and distributing connections evenly over them. Scalability with modern processors is achieved, by automatically detecting the amount of available cores and starting an appropriate amount of threads.

Whether developers choose to use Java NIO directly, or base their projects on netty or an even higher level framework, there is one major drawback to these solutions, as they are ultimately based on NIO, which still uses classic sockets for communication. While this suffices to saturate traditional Gigabit Ethernet hardware, fully exploiting modern network equipment requires more sophisticated programming. InfiniBand and high-speed Ethernet NICs can both be accessed using the native *ibverbs* library, which offers full kernel bypass and thus much lower latencies than the traditional socket API, but implements a vastly different programming model and cannot be accessed directly by Java programs.

There have been several attempts at combining the accessibility of the socket API with the speed of ibverbs, by implementing libraries, which transparently offload socket traffic to high-speed networks using the ibverbs API. However, most of these solutions are not maintained anymore.

To this end, we proposed *hadroNIO* in 2021 [34], a Java library, which transparently replaces the default NIO implementation and offloads traffic via the *Unified Communication X* framework. UCX is a native library, providing multiple communication APIs, including streaming, message passing, active messaging and rdma, and automatically detects the fastest network available to send/receive traffic. Developers can take advantage of a unified set of APIs, while UCX takes care of the low-level network implementation, supporting for example InfiniBand, high-speed Ethernet and shared memory. It can also use classic TCP sockets as a fallback, when no high-speed interconnect is available.

UCX provides an official Java binding called *JUCX*, which is based on the *Java Native Interface*. For a long time, JNI was the only way to interface between Java and native code. It allows Java programs to call native functions and provides many ways to interact with a Java program from native code (e.g. object creation and method upcalls). However, it is not possible to call native functions directly, requiring developers to write glue code. Furthermore, interacting with the JVM from native code may cause performance issues. While we have shown, that JNI can be used for fast access to native functionality [15], it is complex to use and holds several pitfalls for developers.

To allow for easier interoperability between Java and native code, the OpenJDK is currently incorporating *Project Panama*, which offers new ways to interface between Java and native functions, and access off-heap memory, aiming to replace the JNI. The project is available as a preview feature in OpenJDK 20 and can therefore be used with official releases.

Based on Project Panama, we proposed *Infinileap* in 2021, a Java library providing access to UCX via the new Foreign Function & Memory API, as an alternative to JUCX [16]. Since then, we incorporated Infinileap into hadroNIO, allowing users to choose between acceleration via JUCX for higher backwards compatibility down to Java 11, or Infinileap for better performance, but requiring execution on a Java 20 JVM.

The contributions of this paper are:

- Evaluations with hundreds of connections, using a netty-based microbenchmark, as well as the industry proven *Yahoo Cloud Serving Benchmark* [2] on real world applications.
- Optimizations in hadroNIO and extensions for using Infinileap
- An overview of existing socket acceleration solutions for Java

The paper is structured as follows: Section II discusses related work by presenting existing acceleration solutions. Section III presents optimizations to hadroNIO for supporting hundreds of connections and providing low latencies. Section IV presents the benchmarks used for performance evaluation, followed by a the results in Section V. Conclusions are presented in Section VI

II. Related work

Modern high-speed NICs from Mellanox can be configured to use either InfiniBand or Ethernet as link layer protocol. Choosing Ethernet makes these cards fully compatible with the standard socket API, while still being programmable via the ibverbs library. Regardless of the link layer protocol, traditional sockets do not suffice to fully exploit such a NIC.

While we are not aware of any alternative NIO implementations, there are several solutions for accelerating traditional sockets, with only few being still actively maintained. Typically, these can come in three different shapes: kernel modules, native libraries and Java libraries. Since the default NIO implementation is based on classic sockets, these solutions can be used to accelerate Java NIO applications. We have already evaluated some of these solutions, using socket-based microbenchmarks [33] and compared them to hadroNIO with another microbenchmark, directly using the NIO API [34].

A. Kernel modules

IP over InfiniBand [14] exposes InfiniBand devices as standard network interfaces, enabling applications to use them by simply binding to an IP address, associated with such a device. This solution does not require any preloading of libraries, making it the easiest to use. However, it relies on the kernel's network stack, thus requiring context switches which impose a large performance overhead, rendering it unattractive for applications requiring low latency.

Fastsocket [23] replaces the Linux kernel's TCP implementation, aiming to provide better scaling with multiple CPU cores. It has been evaluated using up to 24 cores and 10 Gbit/s Ethernet NICs, showing much better scalability than the default TCP implementation. Fastsocket consists of kernel level optimizations, a kernel module and a user space library. It requires a custom kernel, based on Linux 2.6.32 and officially only supports CentOS 6.5, which is outdated by now. While it would be interesting to see how such an integrated solution would perform on modern high-speed Ethernet hardware, it does not seem to be in active development anymore.

B. Native libraries

mTCP [10] is a TCP-stack, running completely in user space. As Fastsocket, it primarily aims at high scalability, which it achieves by being independent from the kernel's network stack, alleviating the need for context switches in network applications. Contrary to the other solutions, it is not transparent and requires rewriting parts of an application's network code. It has no official support for Java, but there is an unofficial binding called JmTCP, based on the Java Native Interface (JNI). However, it does not seem to be actively maintained, probably requiring Java applications to manually access mTCP via JNI or the experimental Foreign Function & Memory API (Project Panama) [9]. Since it is neither transparent, nor officially supports Java, mTCP does not fit our use case of accelerating netty-based applications.

libvma [20] is a library developed in C/C++ by Mellanox, transparently offloading socket traffic to high-speed Ethernet or InfiniBand NICs. It can be preloaded to any socket-based application (using *LD_PRELOAD*), enabling full kernel bypass without the need to modify an application's code. However, libvma requires the *CAP_NET_RAW* capability, which might not be available, depending on the cluster environment.

While it is highly configurable by exposing many parameters, allowing users to tune the library to the needs of a specific applications, the resulting performance can actually be worse compared to using the traditional socket implementation, as we have shown in previous experiments [35] and it may even not work at all for some distributed scenarios (see V). Additionally, the default configuration is only suited to basic use cases (e.g. single threaded applications), requiring some time being spent on finding the right configuration for complex applications, using multiple threads and connections.

SocksDirect [19] is a closed source library from Microsoft, written in C/C++. Like libvma, it works by preloading it to socket-based applications, redirecting socket traffic via a custom protocol based on RDMA. It also supports acceleration of intra-host communication via shared memory. It achieves low latencies and a high throughput by removing large parts of the synchronization and buffer management involved in traditional socket communication, while being fully compatible with linux sockets, even when process forking is involved.

We were able to get access to the source code from the authors and have successfully tested it with native applications, but so far we could not get the library working with Java applications. Additionally, SocksDirect uses the experimental verbs API, only available in the Mellanox OFED up to version 4.9 [29].

C. Java libraries

The **Sockets Direct Protocol) SDP** [28] provided transparent offloading of socket traffic via RDMA, fully bypassing the kernel's network stack. It was part of the OFED package and introduced into the JDK starting with Java 7. However, support has officially ended and it has been removed from the OFED in version 3.5 [27]

Java Sockets over RDMA (JSOR) [6] has been developed by IBM with the goal to offload all socket traffic of Java applications to high-speed NICs using RDMA. It is included in the IBM SDK up to version 8, requiring their proprietary J9 JVM. JSOR is not available in newer SDK versions and while the old SDK still receives security updates, applications using features not available in Java 8 cannot be used with JSOR.

While it has shown promising results in our benchmarks, there are known problems with connections getting stuck [7] and exceptions [8]. Additionally, we were not able to evaluate JSOR using a bidirectional connection with separate threads for sending and receiving. These problems and its reliance on proprietary technology limit its usability, especially for modern applications.

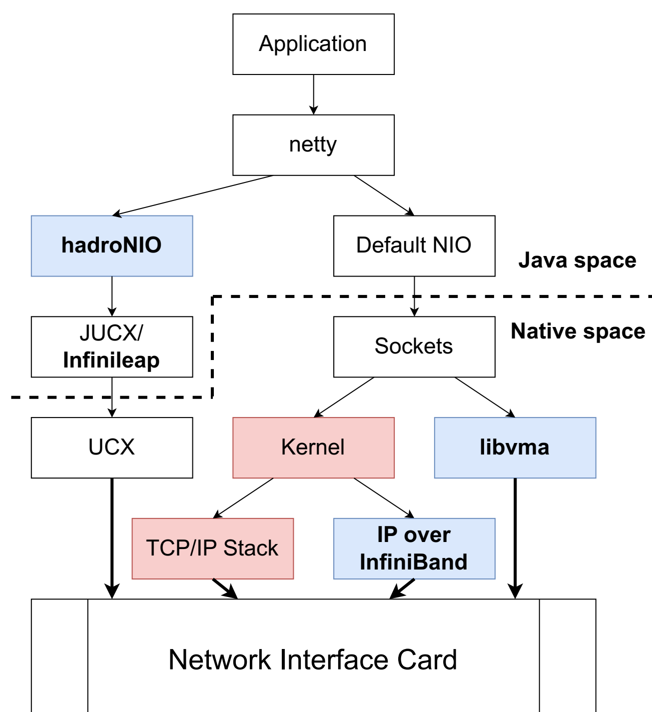

Fig. 1. Application stack overview for different acceleration solutions

D. Application-specific solutions

Other approaches aim at accelerating network performance of a specific application or framework. In 2014, a successful attempt at redesigning Spark's shuffle engine for RDMA usage has been made [25] and refined in 2016 [26]. Similar solutions have been implemented for Apache Storm: In 2019, RJ-Netty has been proposed as a replacement for netty in Apache Storm [38], while in 2021 another approach at integrating RDMA into Storm, based on DiSNi [36] (formerly jVerbs [37]) has been implemented [39].

While these solutions show, that the performance benefit for using high-speed networking hardware can be huge, they are specific to a single framework only and can not be used for general purpose network programming, like transparent acceleration libraries.

III. HADRONIO OPTIMIZATIONS

In our past benchmark results, we saw that hadroNIO provides a substantial acceleration for netty-based applications regarding throughput and is able to saturate high-speed NICs. While it also yields very low round-trip times of around 5 µs when only a single connections is used, latencies rise fast with an increasing amount of connections, making libvma the better solution for applications, that rely on low latency transfers of small messages [35]. Since then, we focused on decreasing round-trip times and provide much better scalability.

A. Faster UCX access via Project Panama

UCX provides an official Java binding called JUCX, which is based on JNI. While JUCX provides full access to the native API, it does not call native methods in an optimized fashion. JNI requires creating a native wrapper library, which can be called from Java code and handles the interaction with the desired native functions. The wrapper library also has the ability to interact with the JVM, for example by creating and manipulating Java objects. However, fast access to native functionality is best achieved by keeping the wrapper code short and performing as little upcalls to Java space as possible [17] [3]. Unfortunately, the native part of JUCX performs a lot of interactions with the JVM, such as object manipulation, throwing exceptions, as well as creation and deletion of global references, slowing down general JUCX performance.

Project Panama avoids such pitfalls, by omitting the need for a wrapper library. Instead it provides a *Foreign Function Interface*, enabling Java programs to directly call functions from native libraries, such as UCX. Additionally, the *Foreign Memory Interface* allows to manipulate off-heap memory from Java space. This way, Java programs can access native structures and process return values coming from native functions.

In 2021, we proposed *Infinileap* an alternative Java binding for UCX, based on Project Panama, providing ultra-low round trip times of less than 2µs and offering great scalability with multiple connections [16]. It successfully utilizes the Foreign Function Interface to efficiently call native UCX functions and makes use of the Foreign Memory Interface to interact with native off-heap structures, returned by these functions.

Furthermore, Infinileap avoids some design decisions, made by JUCX, that lead to a lower performance. For example, send and receive requests are generally processed in an asynchronous manner by UCX. The programmer can either manually check if a request is finished, or register a callback to be notified about a completion. UCX may however decide to directly process small requests to achieve lower latencies. In these cases, it will ignore the callback and instead signal the completion via the return value. However, in case of such an immediate request completion, JUCX will manually call the registered Java callback, thus performing an unnecessary upcall. This is done to simplify the API, but comes at the cost of increased latencies, limiting performance. Infinileap on the other hand, provides a true representation of the native UCX API, allowing programmers to leverage the full potential of the UCX framework.

B. Integrating epoll support

Asynchronous network requests in UCX are handled by *workers*. Each worker can have multiple connections associated with it and in order to be notified about completed requests, the programmer must call `progress()`. This method will gather all finished requests on a specific worker and call the associated callback of each request. It can be called in a blocking or non-blocking way, where blocking lets the calling thread sleep until a request is finished (or aborted) and non-blocking returns directly, regardless of request states. In UCX, connections are represented by endpoints. Each endpoint must be associated with a worker at the time of its creation, and cannot be transferred to another worker at a later time. Because of this, hadroNIO uses one worker per connection, instead of one worker per selector. This forces us to use the non-blocking version of `progress()`, because by calling the blocking version on one worker, we might miss events on other workers, which would lead to stuck connections. This *busy polling* implementation works best, when the amount of network threads does not exceed the amount of CPU cores. While it provides very low latencies, we encountered problems when opening hundreds of connections between two nodes, with connection setup times taking over one second per connection. Furthermore, this approach wastes CPU resources, since the selector is working without interruption, even when there is no event to be polled from a worker.

To mitigate these effects, we implemented *epoll*-based polling. By using epoll, one can monitor multiple file descriptors at once (including event file descriptors). The calling thread sleeps until there is an update on at least one of the descriptors and receives a list of descriptors ready for I/O, once it is woken. UCX workers use event file descriptors internally for their blocking `progress()` implementation and also expose them via a getter-function. This feature was only available in the native UCX library, but we wrapped it in Infinileap and also ported this functionality to JUCX [12]. However, epoll is not available with the standard Java tools. In order to use it, we leveraged the open source library *linux-epoll.java*, which exposes native epoll functionality via the *Java Native Access* library [24] [11].

While epoll might help saving resources, letting a thread sleep and wakeup costs time and affects latency negatively, especially with only a few active connections. Once an event has been processed, it is advisable to keep the thread active for a short amount of time, so that following events can be processed faster. Our epoll-based selector implementation respects that, by using busy polling first, and leveraging epoll after no event has ben processed for a configurable amount of time (default: 20 µs).

IV. Benchmarks

We evaluated hadroNIO in three different scenarions using a netty-based microbenchmark, a distributed key-value store build on top of gRPC [4] and ZooKeeper [5]. This chapter elaborates on the different applications and benchmarks used for these experiments.

A. Netty Microbenchmark

Our microbenchmark measures round-trip times using netty. It runs on two nodes (server/client) and supports an arbitrary number of connections. All communication is done directly in the netty channel handlers. Once a handler reads an incoming message from a channel, it directly sends an answer. This way, no additional threads are needed and the benchmark solely uses the netty worker threads, allowing us to saturate the CPU with a number of threads matching the number of logical cores, but not overwhelming it with too many threads.

B. gRPC Key-Value Store

gRPC is a framework to perform remote procedure calls in distributed systems [4]. It uses HTTP/2 as its transport protoctol and supports multiple programming languages. Its Java implementation is based on netty, rendering it a candidate for acceleration via hadroNIO. However, by default gRPC uses netty's epoll-based channel implementation, instead of the NIO-based one. This can easily be changed, but needs updates in a few lines of the application's setup code. Receiving requests is handled by a netty worker thread pool, while a separate executor thread pool performs the requested method calls. Both are configurable by the programmer.

To evaluate gRPC performance we implemented a distributed key-value store, originally based on the example code by Carl Mastrangelo [1]. We implemented safe access from multiple clients at once by using a `ConcurrentHashMap` and enhanced the store with support for multiple servers, by implementing client-side static hashing and distributing keys over servers according to their hash values.

For benchmarking, we decided to use the *Yahoo! Cloud Serving Benchmark* (YCSB) [2], an industry approved benchmark for evaluating (distributed) cloud databases. It expects data to be stored in records, which have unique keys assigned with them and each record containing an arbitrary number of fields. During the benchmark, records are read (or updated) and the time for each operation is measured. Results can be given as a histogram or time series.

C. Apache ZooKeeper

Apache ZooKeper is a highly reliable hierarchical key-value store, used for coordination of distributed cloud services [5]. Data is stored on disk and can be replicated on multiple servers to increase reliability. The values are organized in nodes and the key naming scheme follows a hierarchical structure, resembling a filesystem. ZooKeeper uses the Java NIO API directly, instead of being built on top of netty. Like gRPC, it uses one thread pool for network communication, called the *selector thread pool*, and one worker thread pool for performing I/O. It may however be configured to do all work inside the selector threads and omit second thread pool.

The YCSB repository contains an official binding for ZooKeeper, making it the obvious choice for our evaluation.

V. Evaluation

This sections presents the evaluation results, comparing hadroNIO based on Infinileap and JUCX with libvma and classic sockets using 100 GBit/s high-speed NICs.

A. Evaluation setup

We used our netty microbenchmark, as well as the YCSB (described in chapter IV) to evaluate application performance. We look at two types of figures: For scalabilty evaluation, we use graphs with an increasing amount of connections on the x-axis, and either the round-trip time in microseconds or the operation throughput on the y-axis. In such cases, all benchmark runs were executed 5 times and the graphs depict average values with the error bars showing the standard deviation. To get a better idea of the latency variation, we also executed time series benchmarks using the YCSB with a granularity of 1 ms and a fixed connection count. We depcit the results as scatter plots with the elapsed time in seconds on the x-axis and the request time in microseconds on the y-axis. We cut off the first 30 seconds as warmup time.

All experiments were performed on identical nodes, provided by the Oracle Cloud Infrastructure, using the *HPC Cluster* Terraform stack [31].

CPU	2x Intel(R) Xeon(R) Gold 6154 CPU (18 Cores/36 Threads @3.00 GHz)
RAM	384 GB DDR4 @2933 MHz
NIC	Mellanox Technologies MT28800 Family [ConnectX-5] (100 GBit/s) Ethernet
Storage	Oracle 6.4 TB NVMe SSD v2
OS	Oracle Linux 8.7 with Linux kernel 4.18.0-425
OFED	MLNX 5.4-3.6.8.1
Java	OpenJDK 20.0.1
UCX	1.14.1
libvma	9.8.30

Fig. 2. Hardware specification of the OCI systems.

Each of the OCI nodes disposes of two CPUs in distinct sockets, which can hurt performance, if applications are not aware of that. To avoid such problems, we used the tool *numactl* to pin our benchmark processes to the CPU, which the network card is connected to. For our gRPC scenario, we started two servers on one node, with each server instance being pinned to an individual CPU.

Regarding libvma, some setup is needed for it to work properly. To accommodate that, we set the amount of hugepages to 16384, as recommended by the libvma readme file [22] (shmmax was already pre-configured with a high enough value). Furthermore, we followed the instructions in the official libvma wiki and set the environment variables VMA_RING_ALLOCATION_RX and VMA_RING_ALLOCATION_TX to 20, while also increasing the amount of receive buffers to 2000000 to improve performance with multiple network threads [21]. For the netty micro-benchmark, we set VMA_SPEC to latency, while in the other scenarios, libvma performed better without it. Lastly, we ran our benchmarks with root privileges, when using libvma, because just granting CAP_NET_RAW did not work as described.

For hadroNIO, we used the default configuration with 8 MiB large send and receive buffers and a buffer slice length of 64 KiB.

B. Netty microbenchmark results

We used our netty microbenchmark to measure round-trip times with 16 byte messages and up to 512 connections in increments of 8. However, for libvma we do not have results for more than 96 connections, because we faced problems with hanging connections, causing the benchmark to not finish. This also occurred with less connections, forcing us to restart the benchmark multiple times, but when using more than 100 connections, it happened so often, that it was not practical to get more libvma results. Furthermore, we aborted the benchmark with hadroNIO based on JUCX at 128 connections, because it became so slow, that letting it run further would haven taken too long. To achieve such a high connection count with hadroNIO, we used our epoll-based selector implementation. We configured netty to use 36 worker threads, matching the amount of logical CPU cores available.

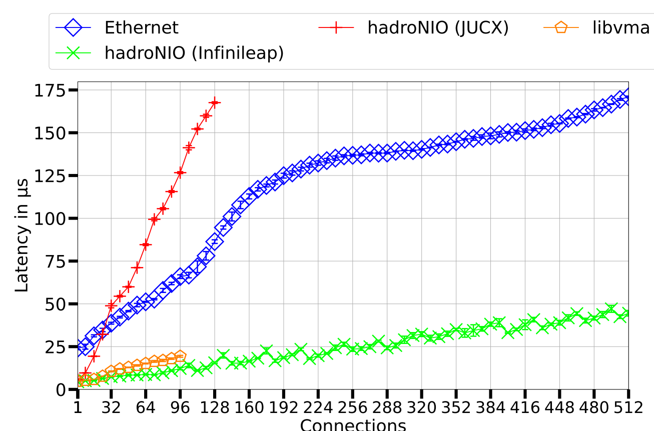

Fig. 3. Netty microbenchmark round-trip times with 16 byte messages

Depicted by Fig. 3, we see hadroNIO and libvma starting very close to each other, with hadroNIO based on Infinileap having a marginal advantage over libvma (4.5 µs vs. 4.7 µs) and our JUCX-based solution yielding slightly higher latencies of around 5.6 µs. However, all perform better than classic Ethernet which needs almost 25 µs on average per iteration.

Going further, latencies rise fast using JUCX and getting even slower than Ethernet from 32 connections onward. With Infinileap, round-trip times climb slowly, staying under 10 µs up to 80 connections, with the gap between hadroNIO (Infinileap) and libvma also growing slowly. libvma breaks 10 µs at 32 connections and the last result we got for it is around 19 µs with 96 connections. At that point, the Infinileap-based solution still yields latencies of 11-12 µs, while Ethernet measures 65 µs and JUCX is by far the slowest with 126 µs.

Going over 100 connections, the values rise moderately for hadroNIO, but never exceed 45 µs, even with more than 500 connections. We can see a slight sawtooth pattern coming from the use epoll. Each time a multiple of 36, which matches the amount of active worker threads, is reached, it performs best and latencies rise up to the next multiple of 36, where a slight drop, of at maximum 5 µs, can be observed.

Overall, hadroNIO based on Infinileap performs by far the best, offering a 5x performance improvement over Ethernet for up to 256 connections, and still a 3.5x improvement with 512 connections. The official Java binding for UCX, (JUCX) seems overwhelmed by this synthetic scenario, while libvma offers good performance but cannot handle over 100 connections.

C. gRPC key-value store benchmark results

For our gRPC evaluation, we started two key-value store servers on one node, each pinned to an individual CPU and three clients on different nodes requesting data from the servers. To evaluate performance with small values, we took the YCSB workload configuration *workload C* and altered it to use records containing only a single 16 byte field, with 1000 records being distributed evenly over the two servers. Each client started off with a single benchmark thread performing 1 million get requests, and added one thread and another 1 million requests with each iteration. At the end, each client had 36 active connections to each server, equalling a total amount of 216 connections being managed by the server node's HCA. On the server side, we used 18 netty worker threads and 18 executor threads to process the method calls, while each client also started 18 netty worker threads and one YCSB benchmark thread per connection. We found that in this scenario, hadroNIO performs best with a busy polling selector, so we used that instead of the epoll-based selector used in the netty microbenchmark. Unfortunately, we experienced problems with hanging connections when using libvma during some benchmark runs and with more than 5 benchmark threads per client, exceptions in netty's HTTP/2 module occurred. Due to this, we decided to exclude libvma from the scalability test. However, we can get an idea of libvma's performance with gRPC from Fig. 5.

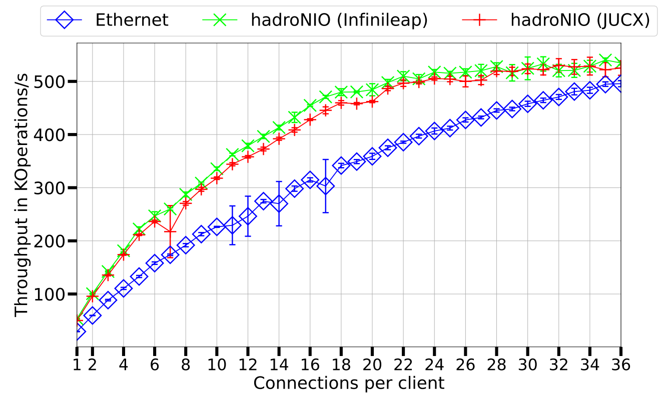

Fig. 4. gRPC key-value store request times 3 clients and 1x16 byte records

Starting with 1 connection per client (6 connections in total), we see a speedup of around 70% for hadroNIO based on JUCX compared to classic sockets (∼50 KOp/s vs. ∼29 KOp/s, see Fig 4). Using Infinileap yields another 3.5 KOp/s over JUCX, resulting in a total performance gain of more than 80%. With a rising amount of connections, the absolute gap between hadroNIO and Ethernet grows further, reaching around 100 KOp/s with 10 connections per client, amounting to a 50% improvement. It reaches its maximum around 16 connections with a difference of more than 150 KOp/s.

Interestingly, the difference between JUCX and Infinileap ist much smaller, compared to the netty microbenchmark. JUCX is still generally slower than Infinileap throughout the benchmark (e.g. ∼426 KOp/s vs. ∼454 KOp/s with 16 connections per client), but in terms of scalability, both solutions perform similarly. Round-trip times in gRPC are much higher than in our netty microbenchmark, starting at around 50µs. This results in less pressure on the JVM, caused by object allocations and upcalls from the native part of JUCX. We think, that for this reason the performance difference between JUCX and Infinileap is much less drastic here.

With 28 or more connections per client, there is virtually no difference between using hadroNIO with JUCX or Infinileap. From there on, we can see no more throughput growth, as it seems like a point of saturation has been reached. When increasing the connection count further, hadroNIO manages to maintain a stable rate of 500-530 KOp/s. With 36 connections, Ethernet ist still slower than hadroNIO with around 495 KOp/s, but the difference is smaller than before.

Fig. 5 presents an in-depth look at gRPC request latencies using the same setup as before, with 3 connections per client (18 connections in total). Ethernet not only yields the slowest performance, but also has more spread out values than the acceleration libraries, with request times generally lying between 90 and 110 µs and a considerable amount of requests even reaching up to 130 µs. As the slowest acceleration solution, libvma still outperforms classic sockets, with most requests being served in less than 90 µs and some reaching answer times of less than 75 µs. However, the best performance is

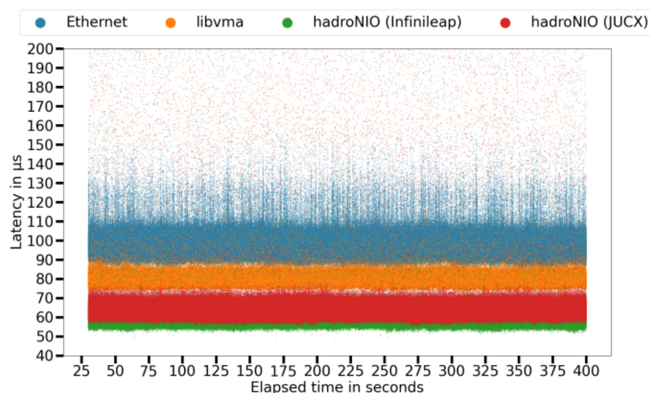

Fig. 5. gRPC key-value store request times with 3 clients ()each using 3 threads) and 1x16 byte records

achieved by hadroNIO, serving the majority of requests in less than 70 µs and with Infinileap request latencies can be as low as 55 µs and are generally lower than with JUCX.

To conclude the gRPC benchmarks, we increased the record size by setting the field length to 1024 byte and storing 16 fields in each record, amounting to 16 KiB of data per request. The graphs look similar, compared to the 16 byte results. We can see that Ethernet profits from the larger amounts of data being sent per request. This was expected, since transferring small amounts of data, with each message causing a context switch into kernel space, is much more inefficient than sending large payloads and thus causing less context switches.

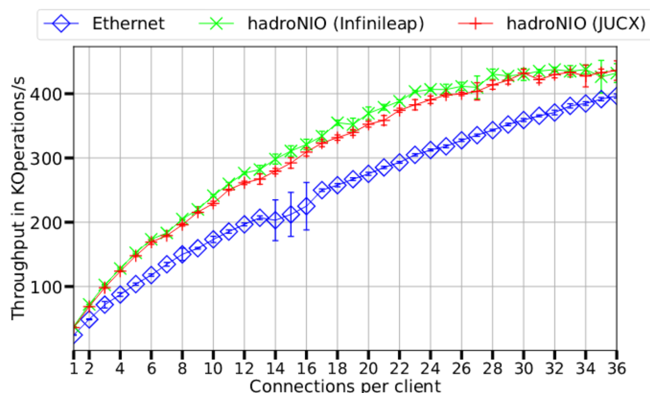

Fig. 6. gRPC key-value store request times with 3 clients and 16x1024 byte records

However, hadroNIO still provides performance improvements of 30-40% with up to 24 connections per client and manages to outperform classic sockets by 20% with 30 connections per client. As before, a point of saturation is reached at around 28 connections with hadroNIO yielding ~430 KOp/s, compared to Ethernet with ~343 KOp/s.

D. ZooKeeper benchmark results

As our last benchmark scenario, we tested Apache ZooKeeper with one server and three clients, each running three benchmark threads. We did not start two ZooKeeper instances on our server node, as we did with gRPC, because ZooKeeper does not distribute values over servers, but rather replicates them, so that each additional server makes the system more reliable. However, running two servers on one node does not increase reliability, since a hardware failure would kill both instances, making this a very untypical scenario. We loaded 1000 records, each with a size of 16 byte, and configured the ZooKeeper server to store data on an NVME SSD, not used by other processes, for fast access. Furthermore, we configured ZooKeeper to not use any worker threads, but perform all work directly in the selector threads, to achieve lower latencies.

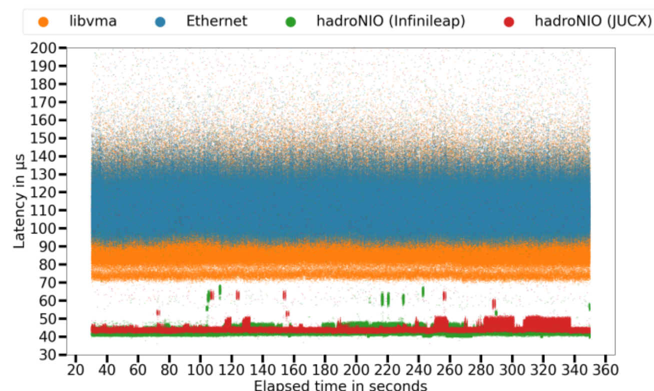

Fig. 7. ZooKeeper request times with 3 clients (each using 3 threads) and 1x16 byte records

As with gRPC, classic Ethernet causes a wide spread of request times, with the majority lying between 95 µs and 130 µs, and some reaching more than 150 µs. While libvma manages to accelerate a good portion of requests to less than 90 µs and a considerable amount is as fast as 75 µs, values are spread equally high as with Ethernet, and a major fraction of latencies are higher than 100 µs. This is hard to see from Fig. 7, because most of the libvma data is covered by Ethernet values. But, looking at the area between 130 µs and 150 µs, libvma values can be seen amongst the Ethernet values.

In this scenario, hadroNIO provides a much better acceleration, with almost all requests being answered in less than 50 µs. Curiously, we can see short latency bursts, with the highest reaching almost 70 µs. However, even during these short bursts, request latencies are still lower, compared to Ethernet and libvma. Overall, Infinileap has a slight advantage over JUCX, but both perform similarly.

VI. Conclusions & Future Work

In this paper, we presented the latest extensions to hadroNIO, including support for accessing the native high-performance networking framework UCX via the new Foreign Function and Foreign Memory Interfaces, included in Java 20. We compared hadroNIO to the native socket acceleration library libvma in a synthetic workload, using a netty microbenchmark, as well as distributed real-world scenarios

based on gRPC and Apache ZooKeeper using the YCSB. Our results show, that hadroNIO is able to outperform libvma in each of these scenarios. Furthermore, libvma has shown problems with high connection counts, not being able to finish all of our benchmarks, while also requiring root privileges.

Depending on the workload, hadroNIO yields an increase in performance of roughly 50% over classic Ethernet sockets and can scale with hundreds of connections in gRPC. For Apache ZooKeeper we saw a 2-3x speedup. Looking at the netty microbenchmark results, one can see that there is much acceleration potential left for real-world applications, with hadroNIO being able to achieve average round-trip times of 10 µs with around 100 connections working concurrently.

Our benchmark results show, that Infinileap, based on Project Panama, performs better than the JNI-based JUCX. In our netty microbenchmark, JUCX has shown unusable performance, even being slower than classic Ethernet with more than 32 connections. However, in our gRPC and ZooKeeper tests, JUCX performs admirably, albeit still slower than Infinileap.

Future plans include a more in-depth evaluation of ZooKeeper performance in different scenarios, as well as performing benchmarks with other real-world applications. Successful tests with Apache Ratis [32] and Apache BookKeeper [13] have already been executed.

VII. Acknowledgment

We thank Oracle for their sponsorship in the context of this work.

This work was supported in part by Oracle Cloud credits and related resources provided by the Oracle for Research program.

References

[1] Carl Mastrangelo. gRPC Key Value store. https://github.com/carl-mastrangelo/kvstore.
[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
[3] M. Dawson, G. Johnson, and A. Low. Best practices for using the java native interface.
[4] gRPC. https://grpc.io/.
[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA, 2010. USENIX Association.
[6] Java Sockets over Remote Direct Memory Access (JSOR). https://www.ibm.com/docs/en/sdk-java-technology/7?topic=networking-java-sockets-over-remote-direct-memory-access-jsorl.
[7] IBM. RDMA communication appears to hang. https://www.ibm.com/docs/en/sdk-java-technology/7?topic=problems-rdma-communication-appears-hang.
[8] IBM. RDMA connection reset exceptions. https://www.ibm.com/docs/en/sdk-java-technology/7?topic=problems-rdma-connection-reset-exceptions.
[9] Project Panama. https://openjdk.java.net/projects/panama/.
[10] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, Apr. 2014. USENIX Association.
[11] Java Native Access. https://github.com/java-native-access/jna#readme.
[12] UCX Pull Request 8453. https://github.com/openucx/ucx/pull/8453.
[13] F. P. Junqueira, I. Kelly, and B. Reed. Durability with bookkeeper. *SIGOPS Oper. Syst. Rev.*, 47(1):9–15, jan 2013.
[14] V. Kashyap. IP over InfiniBand (IPoIB) Architecture. https://www.ietf.org/rfc/rfc4392.txt, April 2006.
[15] F. Krakowski, F. Ruhland, and M. Schöttner. Neutrino: Efficient infiniband access for java applications. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 12–19, 2020.
[16] F. Krakowski, F. Ruhland, and M. Schöttner. Infinileap: Modern high-performance networking for distributed java applications based on rdma. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 652–659, 2021.
[17] D. Kurzyniec and V. Sunderam. Efficient cooperation between java and native codes – jni performance benchmark. In *In The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.
[18] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.
[19] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *ACM SIGCOMM Conference (SIGCOMM)*, August 2019.
[20] libvma GitHub. https://github.com/Mellanox/libvma/.
[21] VMA Parameters. https://github.com/Mellanox/libvma/wiki/VMA-Parameters.
[22] libvma README. https://github.com/Mellanox/libvma/blob/master/README.
[23] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable kernel tcp design and implementation for short-lived connections. *SIGPLAN Not.*, 51(4):339–352, mar 2016.
[24] linux-epoll.java. https://github.com/helins/linux-epoll.java.
[25] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda. Accelerating spark with rdma for big data processing: Early experiences. In *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, pages 9–16, 2014.
[26] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. High-performance design of apache spark with rdma and its benefits on various workloads. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 253–262, 2016.
[27] OFED 3.5 release notes. https://downloads.openfabrics.org/OFED/release_notes/OFED_3.5_release_notes.
[28] Sockets Direct Protocol. https://docs.oracle.com/javase/tutorial/sdp/sockets/index.html.
[29] Statement on support of experimental verbs. https://forums.developer.nvidia.com/t/verbs-exp-h-no-such-file-or-directory/206300/2.
[30] Netty. https://netty.io/index.html.
[31] Oracle Marketplace: HPC Cluster Terraform Stack. https://cloudmarketplace..com/marketplace/en_US/listing/67628143.
[32] Apache Ratis. https://ratis.apache.org/.
[33] F. Ruhland, F. Krakowski, and M. Schöttner. Performance analysis and evaluation of Java-based InfiniBand Solutions. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 20–28, 2020.
[34] F. Ruhland, F. Krakowski, and M. Schöttner. hadronio: Accelerating java nio via ucx. In *2021 20th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 25–32, 2021.
[35] F. Ruhland, F. Krakowski, and M. Schöttner. Accelerating netty-based applications through transparent infiniband support, 2022.
[36] P. Stuedi. Direct storage and networking interface (disni). https://developer.ibm.com/open/projects/direct-storage-and-networking-interface-disni/, 2018.
[37] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14. ACM, 2013.
[38] S. Yang, S. Son, M.-J. Choi, and Y.-S. Moon. Performance improvement of apache storm using infiniband rdma. *The Journal of Supercomputing*, 75:6804–6830, 2019.
[39] Z. Zhang, Z. Liu, Q. Jiang, J. Chen, and H. An. Rdma-based apache storm for high-performance stream data processing. *International Journal of Parallel Programming*, 49:671–684, 2021.

Chapter 5

Conclusions and Outlook

The following chapter concludes this thesis, by summarizing the achievements, that have been accomplished for network performance in Java. Furthermore, an outlook for possible future work with regards to further enhancements to Java NIO is provided.

5.1 Achievements

The thesis provides an overview of existing solutions for using high-speed networks with Java, with a focus on InfiniBand. Both transparent acceleration libraries, as well as direct Java bindings for the native ibverbs libraries are presented and benchmarked against each other. The results show, that each of the transparent solutions comes with its own set of problems, ranging from high latencies (IP over InfiniBand) and limited throughput (libvma) to general problems, causing connections to hang or crash (JSOR). Regarding the verbs-based libraries, a similar impression is gained, with both jVerbs and DiSNI showing massive problems when it comes to messaging. The only library showing good performance across all results is neutrino, where the author of this thesis contributed. Neutrino avoids the problems jVerbs and DiSNI have, while providing a simpler API compared to the Stateful Verbs Methods of IBM's solutions.

However, working with the low-level ibverbs API is challenging and a lot of groundwork is required for building scalable network applications upon it. Therefore, the Unified Communication X (UCX) Framework was chosen as a basis for this thesis' main project. It provides a network agnostic API, supporting multiple high-speed interconnects and is maintained by a large consortium.

The Java ecosystem already provides many networking frameworks and big data applications, including Netty, looking upon years of development and a large community of developers, so it was not deemed feasible by the author to compete against them. The objective was to provide a transparent way of integrating high-speed networks into the Java ecosystem, without the problems that existing solutions have. This has been achieved by developing hadroNIO, replacing the default Java NIO implementation with a new approach, based on UCX.

During this thesis, hadroNIO has been successfully tested with well-known Java frameworks, namely Netty, gRPC and Apache ZooKeeper. Evaluation results with 100 GBit/s

hardware show, that Netty accelerated by hadroNIO can achieve end-to-end round-trip times of less than 5 µs with small messages and a single connection. hadroNIO scales very well, which is proven by the results shown in the evaluation paper (see page 76), where it was able to provide average round-trip times of less than 10 µs with 80 active connections, outperforming existing solutions.
Furthermore, hadroNIO works well with hundreds of connections, where other solutions fail to yield comparable performance or do not work at all.
When used with real-world applications, hadroNIO has proven its capabilities by accelerating gRPC by more than 50% over using traditional sockets and drastically lowered response times of Apache ZooKeeper. Compared to libvma, hadroNIO performs better in all benchmarked scenarios without the need for complex configuration, while not requiring any elevated priviliges and being much more compatible with the tested applications (see pages 77-78). It truly acts as a link between the high-level world of Java big data computing and the low-level environment of high-speed networking in HPC systems, combining ease-of-use and high performance into a single transparent library.

5.2 Future Work

hadroNIO can be evaluated with more real applications. For example, successful tests with the log-structured distributed storage service Apache BookKeeper [7] and the consensus library Apache Ratis [33] have been done, but an in-depth performance analysis is still needed. Apache BookKeeper is an especially interesting showcase, since it relies on Apache ZooKeeper for metadata management. In such a rather complicated setup, hadroNIO has to accelerate two different distributed applications communicating with each other. First tests have shown, that such a setup works, proving hadroNIO's compatibility with the NIO standard.
Since gRPC acceleration is already working, any projects based on gRPC could be accelerated and benchmarked. One such project is Apache Arrow, more specifically its communication framework Arrow Flight RPC. Apache Arrow specificies a standardized columnar data format, tailored towards in-memory analytics and query processing [34]. Arrow Flight RPC provides high-performance remote procedure calls for transferring Arrow data, built on top of gRPC [35]. It would be interesting to see, how hadroNIO can improve performance in such a complex application stack.
Furthermore, newer InfiniBand hardware with up to 400 GBit/s is available by now. This could enable even lower latencies for Java networking and especially large-scale distributed applications, accelerated by hadroNIO, could profit from the increased bandwidth.

Besides these obvious directions, there is also a more interesting route to explore: While hadroNIO enables Java applications to fully exploit messaging performance of high-speed networks, there is still no official support for Remote Direct Memory Access (RDMA) in Java. In Chapter 2, some libraries allowing to access the native ibverbs API, including RDMA functionality, from Java are presented. However, working with the ibverbs interface is complex and none of these solutions are actively maintained anymore. An alternative strategy might be to enhance the NIO API with RDMA semantics, allowing developers to access remote memory without having to learn a new API from scratch. It would be interesting to see, how well RDMA directives can be integrated into the socket interface provided by Java NIO.

Acronyms

ACK Acknowledgement

API Application Programming Interface

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CSV Comma Separated Values

DiSNI Direct Storage and Network Interface

gRPC gRPC Remote Procedure Calls

HCA Host Channel Adapter

HPC High Performance Computing

HTTP Hypertext Transfer Protocol

IB InfiniBand

IP Internet Protocol

IPoIB Internet Protocol over InfiniBand

JDK Java Development Kit

JNI Java Native Interface

JFS Java Fast Sockets

JIB Java InfiniBand Benchmark

JSON JavaScript Object Notation

JSOR Java Sockets over RDMA

JVM Java Virtual Machine

MPI Message Passing Interface

MSG Message

MTU Maximum Transmission Unit

NACK Negative Acknowledgement

NIC Network Interface Card

NIO New Input/Output

OCI Oracle Cloud Infrastructure

OFED OpenFabrics Enterprise Distribution

OP Opertation

OSU Ohio State University

RAM Random Access Memory

RC Reliable Connected

RDMA Remote Direct Memory Access

RNR Receiver Not Ready

RPC Remote Procedure Call

RoCE Remote Direct Memory Access over Converged Ethernet

RTT Round Trip Time

SDK Software Development Kit

SSD Solid State Disk

SDP Sockets Direct Protocol

TCP Transmission Control Protocol

TLS Transport Layer Security

UCX Unified Communication Framework X

UD Unreliable Datagram

UDP Unreliable Datagram Protocol

UI User Interface

YCSB Yahoo! Cloud Serving Benchmark

Bibliography

[1] M. T. Inc., "Introduction to Infiniband", 2003.

[2] M. Zaharia, R. S. Xin, P. Wendell, *et al.*, "Apache Spark: A Unified Engine for Big Data Processing", *Commun. ACM*, vol. 59, pp. 56–65, Oct. 2016.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink™: Stream and batch processing in a single engine", *IEEE Data Eng. Bull.*, vol. 38, pp. 28–38, 2015.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system", *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010, ISSN: 0163-5980. DOI: `10.1145/1773912.1773922`. [Online]. Available: `https://doi.org/10.1145/1773912.1773922`.

[5] K. Beineke, S. Nothaas, and M. Schöttner, "Dxnet: Scalable messaging for multi-threaded java-applications processing big data in clouds", 2019. [Online]. Available: `https://api.semanticscholar.org/CorpusID:250921021`.

[6] *Netty*, `https://netty.io/index.html`.

[7] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper", *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 1, pp. 9–15, Jan. 2013, ISSN: 0163-5980. DOI: `10.1145/2433140.2433144`. [Online]. Available: `https://doi.org/10.1145/2433140.2433144`.

[8] *gRPC*, `https://grpc.io/`.

[9] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems", in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10, Boston, MA: USENIX Association, 2010, p. 11.

[10] Fabian Ruhland, *Observatory*, `https://github.com/hhu-bsinfo/observatory`.

[11] P. Shamis, M. G. Venkata, M. G. Lopez, *et al.*, "UCX: an open source framework for HPC network APIs and beyond", in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, IEEE, 2015, pp. 40–43.

[12] Fabian Ruhland, *hadroNIO*, `https://github.com/hhu-bsinfo/hadronio`.

[13] Filip Krakowski, Fabian Ruhland, *neutrino*, `https://github.com/hhu-bsinfo/neutrino`.

[14] Maurizio Cimadamore, Alex Buckley, John Rose, Paul Sandoz, Brian Goetz, *Foreign Function & Memory API (Preview)*, `https://openjdk.org/jeps/424`.

[15] Filip Krakowski, Fabian Ruhland, *infinileap*, `https://github.com/hhu-bsinfo/infinileap`.

[16] Fabian Ruhland, *Detector*, https://github.com/hhu-bsinfo/detector.

[17] Fabian Ruhland, *jDetector*, https://github.com/hhu-bsinfo/jdetector.

[18] Zeyd Ben-Halim, Eric S. Raymond, Jürgen Pfeifer, Thomas E. Dickey, *ncurses*, https://invisible-island.net/ncurses/announce.html.

[19] Fabian Ruhland, *ib-scanner*, https://github.com/hhu-bsinfo/ib-scanner.

[20] S. Nothaas, F. Ruhland, and M. Schöttner, *A benchmark to evaluate infiniband solutions for java applications*, 2019. arXiv: 1910.02245 [cs.NI].

[21] P. Stuedi, B. Metzler, and A. Trivedi, "Jverbs: Ultra-low latency for data center applications", in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, ACM, 2013, 10:1–10:14.

[22] P. Stuedi, *Direct storage and networking interface (disni)*, https://developer.ibm.com/open/projects/direct-storage-and-networking-interface-disni/, 2018.

[23] F. Krakowski, F. Ruhland, and M. Schöttner, "Neutrino: Efficient infiniband access for java applications", in *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2020, pp. 12–19. DOI: 10.1109/ISPDC51135.2020.00012.

[24] F. Ruhland, F. Krakowski, and M. Schöttner, "Performance analysis and evaluation of Java-based InfiniBand Solutions", in *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2020, pp. 20–28. DOI: 10.1109/ISPDC51135.2020.00013.

[25] *Openucx documentation*, https://openucx.org/documentation/.

[26] *Agrona github*, https://github.com/real-logic/Agrona.

[27] E. Stang, *Hadronio pull request by edwin stang*, https://github.com/hhu-bsinfo/hadroNIO/pull/3.

[28] P. Rudenko, *Observatory pull request by peter rudenko*, hhttps://github.com/hhu-bsinfo/observatory/pull/1.

[29] F. Krakowski, F. Ruhland, and M. Schöttner, "Infinileap: Modern high-performance networking for distributed java applications based on rdma", in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, 2021, pp. 652–659. DOI: 10.1109/ICPADS53394.2021.00087.

[30] *UCX Pull Request 8829*, https://github.com/openucx/ucx/pull/8829.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb", in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154, ISBN: 9781450300360. DOI: 10.1145/1807128.1807152. [Online]. Available: https://doi.org/10.1145/1807128.1807152.

[32] Carl Mastrangelo, *gRPC Key Value store*, https://github.com/carl-mastrangelo/kvstore.

[33] *Apache Ratis*, https://ratis.apache.org/.

[34] *Apache arrow*, https://arrow.apache.org/docs/index.html.

[35] *Arrow flight rpc*, https://arrow.apache.org/docs/format/Flight.html.

Eidesstattliche Erklärung
laut §5 der Promotionsordnung vom 15.06.2018

Ich versichere an Eides Statt, dass die Dissertation von mir selbständig und ohne unzulässige fremde Hilfe unter Beachtung der „Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Heinrich-Heine-Universität Düsseldorf“ erstellt worden ist. Die aus fremden Quellen direkt oder indirekt übernommenen Inhalte wurden als solche kenntlich gemacht. Die Dissertation wurde in der vorgelegten oder in ähnlicher Form noch bei keiner anderen Fakultät eingereicht. Ich habe bisher keine erfolglosen Promotionsversuche unternommen. Ich versichere weiterhin, dass alle von mir gemachten Angaben wahrheitsgemäß und vollständig sind.

Düsseldorf, 16.01.2024

Fabian Ruhland