Inaugural Dissertation

High Performance Networking for Distributed Java Applications

submitted by

Filip Krakowski

from Poznań

for the acquisition of the doctoral degree
of the Faculty of Mathematics and Natural Sciences
of the Heinrich Heine University Düsseldorf

January 2024

from the Institute of Computer Science
at Heinrich Heine University Düsseldorf

Printed with the permission of the
Faculty of Mathematics and Natural Sciences of the
Heinrich Heine University Düsseldorf

Reporters:

1. Prof. Dr. Michael Schöttner

2. Prof. Dr. Stefan Conrad

Day of the oral exam:

Acknowledgments

I would like to take this opportunity to express my sincere thanks to a number of people, as this work would not have been possible in this form without them. First of all, I would like to thank my supervisor *Prof. Dr. Michael Schöttner*. He was always there to help with technical and organizational matters, made it possible for me to meet interesting people and motivated me in my work in the form of many discussions and new ideas. I would also like to thank my colleague *Fabian Ruhland*, with whom I was given the opportunity to work throughout my doctorate. As our research topics had some overlaps, this enabled a valuable exchange of insights, which helped me a lot in some parts of my work. Further thanks go to our administrator *Dr. Michael Braitmeier*. I could always rely on him in the event of technical problems or questions relating to our infrastructure, so that my work could always be continued without interruption. Furthermore, I would like to thank my mentor *Prof. Dr. Stefan Conrad*, who was there to answer questions and help me with his knowledge. Special thanks go to our former secretary *Angela Rennwanz*, who knew the answer to almost every organizational question, offered her help at any time and thus made my work much easier in many ways. In addition to the people in our department, I would also like to thank the many students who have written their theses with us, as their work has often given me new ideas that have helped my work.

Besides the university, there were also some people who accompanied me on my way and always stood by my side. Here I would first like to thank the person most important to me, my beloved wife *Dr. Martha Krakowski*. She supported me unconditionally in all matters, always had an open ear, gave the necessary motivation and steered me in the right direction more than once. I wouldn't be the person I am today without her. Many thanks also go to my mother *Małgorzata*, my father *Krzysztof*, my stepfather *Darius*, my grandmother *Krystyna* and my grandfather *Gerhard* who always believed in me, supported me and did their best in order to allow me to have a good life. I would also like to thank my siblings *Alex* and *Paula*, who shaped my childhood and whom I can always rely on. In addition, I have to thank my father-in-law *Jan Tatusch* for his motivating discussions and my mother-in-law *Judith Tatusch* for the trust she has placed in me.

Finally, I would like to thank all my friends who gave me the distraction I needed to reorganize my thoughts in times of uncertainty. To everyone else who has accompanied me on my journey, I also wish to express my sincere gratitude.

Abstract

While data processing has become a core component of almost all systems in the digital world, some applications are designed for very large amounts of data or big data and must also be able to produce a result in a short time. Furthermore, big data applications have to be distributed, as a single computer is no longer sufficient due to its limited resources. Due to the ever-increasing complexity of data, they also need a suitable way to process it, which is offered in the form of various big data processing frameworks. Many of these big data frameworks have been developed within the Java ecosystem and the focus has therefore been placed on the Java Virtual Machine. This work pursues the goal of accelerating communication between distributed Java applications using modern technologies. Specifically, an integration of the InfiniBand technology developed by NVIDIA (formerly by Mellanox) within the Java ecosystem is being pursued. For this purpose, a connection to the Verbs API based on the Java Native Interface is first developed, which allows efficient access to InfiniBand hardware within Java applications. The developed solution shows very good results in benchmarks with throughputs of up to 14 million messages per second as well as reaching the maximum bandwidth achievable in practice of a 56 Gbit/s ConnectX-3 controller at 6 GB/s. Based on these findings and a cooperation with Oracle Labs, the focus is being shifted away from the Java Native Interface to the Foreign Function & Memory API recently developed by Oracle. The resulting Infinileap project continues to pursue the goal of connecting InfiniBand hardware in Java applications, but relies on Oracle's Project Panama for native access and on the OpenUCX library, which provides an abstraction layer for high-performance networking hardware. It also aims at providing RDMA functions for Java applications. The benchmarks carried out in the context of the Infinileap project show that using Project Panama with OpenUCX is a good match. Using the Infinileap project and 100Gbit/s ConnectX-5 network controllers, the round-trip latency for send operations and RDMA read and write operations with small amounts of data of less than 256 bytes is around 2 microseconds. Similarly, atomic operations such as Compare & Swap can be executed on remote memory in under 2 microseconds The round-trip latency for smaller messages of 8 bytes is even lower with 1.4 microseconds while the theoretically achievable minimum latency is 1.2 microseconds. The bandwidth of the network card is also reached at 100 Gbit/s when using larger messages starting at around 4 kilobytes. Finally, an integration into Java's NIO framework is developed, which allows existing network applications to use transparent InfiniBand controllers on the network level without major changes to the code. The benchmarks developed in this context show that the Infinileap project offers significant added value compared to other existing solutions such as JUCX and enables more stable operation overall.

Contents

Chapter 1

Introduction

In today's digital world, data is generated almost everywhere. From a simple household thermometer to complex scientific calculations. At the same time, the volume of data generated has grown steadily in the past and continues to do so today due to the increasing number of virtual and physical sensors and the associated new possibilities for data collection. In a blog post published at the end of 2014, for example, Facebook revealed that it generates around 4 petabytes of new data every day, which is stored in a huge storage system called "Hive" and then queried using around 600,000 queries and processed by around 1 million map-reduce jobs[1]. Since this statement was made around 10 years ago, the number of monthly active users on Facebook has more than doubled[2] and many new features have been added, it can be assumed that the company now collects, processes and stores a much larger amount of data. It is also clear today that workloads of this magnitude must be carried out in a distributed environment in which each participating system takes on part of the task, as a single computer would be more than inadequate. Since the basis for new findings lies within the evaluation of collected data, it is important to establish ways to keep the processing of the data as simple as possible, as otherwise a lot of time is lost. Nowadays, this basis is provided by big data frameworks, which allow developers or data scientists to model and execute complex calculations or sequences of operations using a simple programming interface.

Widely known systems belonging to this group are, for example, Apache Spark™[3], Apache Hadoop[4], Apache Storm[5] and Apache Flink®[6]. Since learning the different APIs takes some time, the Apache Beam[7] project was also established, which offers a unified API that is able to map operations to different backends. One characteristic that all these systems have in common is the fact that they use the Java Virtual Machine as their foundation.

"To a large extent Big Data is Java. Hadoop and a large percentage of the Hadoop ecosystem are written in Java. The native MapReduce interface for Hadoop is Java. So you can easily move into big data simply by building Java solutions that run on top of Hadoop. There's also Java libraries like Cascading which make the job easier. Java is also really useful for debugging things even if you use something like Hive.

Beyond Hadoop, Storm is written in Java and Spark (ie: arguably the future of hadoop computing) is in Scala (which runs on the JVM and Spark has a Java interface). So Java covers a massive percentage of the Big Data space. [..]"

- Marcin Mejran, 2014 [8]

The reason for the focus on the Java Virtual Machine is well summarized in an answer published on Quora by Marcin Mejran. At the time of the widespread introduction of big data, existing systems and tools were already designed for the Java Virtual Machine and therefore the entrance to the topic was linked to this technology. While Hadoop played a major role in the initial phase, the aforementioned frameworks were developed over time with numerous new possibilities for data processing. Over time, a distinction was also made within the type of data processing between the previously known batch processing and the new stream processing, in which data is processed on-the-fly or in real time[9]. However, with this new type of processing came further new challenges. Processing real-time data, for example, requires low latencies, as the results to be analyzed may only be valid for a limited time. A good example of this is the processing of aircraft data in real time. If the processing were to take a long time, the reaction could be too late. Low latencies are absolutely essential in such cases.

Storing the data on a persistent storage medium, such as a hard disk, as is the case with conventional batch processing, would greatly increase the latencies here, as write accesses to disks are known to be relatively slow. For this reason, in-memory processing was introduced, in which all data to be processed is stored in the main memory[10]. Since the main memory - apart from the CPU cache - is the fastest storage medium within a computer, the latency with regard to local operations could be minimized as far as possible and at the same time the calculation could be significantly accelerated. The local processing of data could thus be significantly optimized. However, since a big data system should be distributed by design, another bottleneck arose.

Whenever additional data is required within an operation on a computer for the calculation, this must first be retrieved from another computer involved in the system via the network. Modern big data systems address this problem by distributing the data in advance in such a way that a minimal number of external accesses are required to perform the calculations[11]. While this type of data distribution initially represents a major optimization, it is of course not one hundred percent effective, which is why external access must still take place. The next bottleneck that occurs here is the network that connects all the systems involved, as well as the technologies used within it.

Within the Java ecosystem, the Netty project[12] has established itself as the de facto solution for network applications, which is why it is also used in the aforementioned big data systems. This is based on an asynchronous programming model that uses Java NIO in the background and thus standard Ethernet-based network sockets. However, such a programming model is associated with some performance drawbacks, such as the context switches caused by system calls or the need to copy data between kernel and user space. These factors understandably influence the latency of the entire system, as any time spent waiting within the processing of data leads to a delay. An alternative to this is Remote Direct Memory Access (*RDMA*) technology. Here, special network controllers are used which have implemented the protocol stack within the hardware and therefore only require minimal communication with the application for sending messages. Another advantage is the possibility to control the hardware directly by bypassing the system kernel, so that system calls are no longer necessary. Finally, such hardware offers the possibility of accessing remote memory without utilizing the target's processor, which in turn leads to extremely low latencies of less than one microsecond for sending messages when used correctly.

While these capabilities represent a clear advantage over conventional socket programming, there is currently no way to control RDMA hardware within the Java Development Kit. While such support was pursued[13], the plans ultimately came to a standstill and were discontinued. In addition to this, there is related work that has been done at the Chair of Operating Systems at Heinrich Heine University in form of the Ibdxnet project, which pursued similar goals, but developed a large part of the functionality within the C programming language[14]. While the project offers very low-level control of the hardware, it uses Java's Unsafe API in many places, which can lead to unpreventable program crashes if used improperly. An additional disadvantage is the maintenance effort associated with the librarie's core written within the native C programming language. If changes need to be made within the logic, for example, the native part must be recompiled. The native part must also be compiled for each platform on which it is to be used. From the point of view of the Java Virtual Machine, which is designed to enable program code to be executed on any platform without changes, this represents an obstacle.

1.1 Project Hermes

This work aims to make RDMA hardware just as accessible and safe as socket programming within Java by means of a simple abstraction layer. For this purpose, Project Hermes has been started in collaboration with Oracle Labs, which specifies the necessary steps towards a functioning integration and provides a scope for the implementation. Initially, it addresses the outlined goal using the Java Native Interface (JNI) by developing a solution for connecting Java classes with native structs and an efficient way to call native functions from Java space. This is realized within the Neutrino project (see 3.2). Unlike previous solutions, the focus here is not only on performance but also on the straightforward usability of the developed API. The backend used for controlling InfiniBand hardware is the native ibverbs API. Additionaly, the Neutrino project offers various abstractions that bundle the components of the ibverbs API and thus make them more accessible. While it shows very good results, it is based on Java technologies, which are intended for internal use and should therefore not be used within production applications. An alternative to this is a new development by Oracle with the name of Project Panama[15], which is intended to significantly improve access to native functions and native memory within the Java Development Kit (*JDK*). The integration of RDMA hardware within the Java ecosystem is therefore based on this new development from here on. In the beginning, Project Hermes continues using the native ibverbs API, but accesses it using Project Panama. This approach turns out to be disadvantageous in the further course of the work, as the maintenance of such a complex API takes consumes a lot of time and leaves little room for active development. As a result, the backend is replaced by a production-ready framework for controlling high-speed network controllers - namely OpenUCX[16]. While the native ibverbs API provides many configuration options for fine-tuning the respective transport method used, the OpenUCX project simplifies the developer's work in that it automatically determines the optimal configuration parameters so that the hardware can be optimally controlled. Building on this foundation, the Infinileap project (see 4.3) is started with the aim of providing the abstraction layers of the OpenUCX framework directly in Java and thus making RDMA programming accessible in a simple form. A major advantage of this project is that the entire logic is implemented within the Java programming language. Since the underlying OpenUCX library is available on a variety of platforms, Infinileap can therefore also be used on those platforms to access InfiniBand hardware without having to change or recompile the project's code. Using a suitable package manager, in the simplest case it is only necessary to specify a dependency on the Infinileap project in order to access its functions or classes.

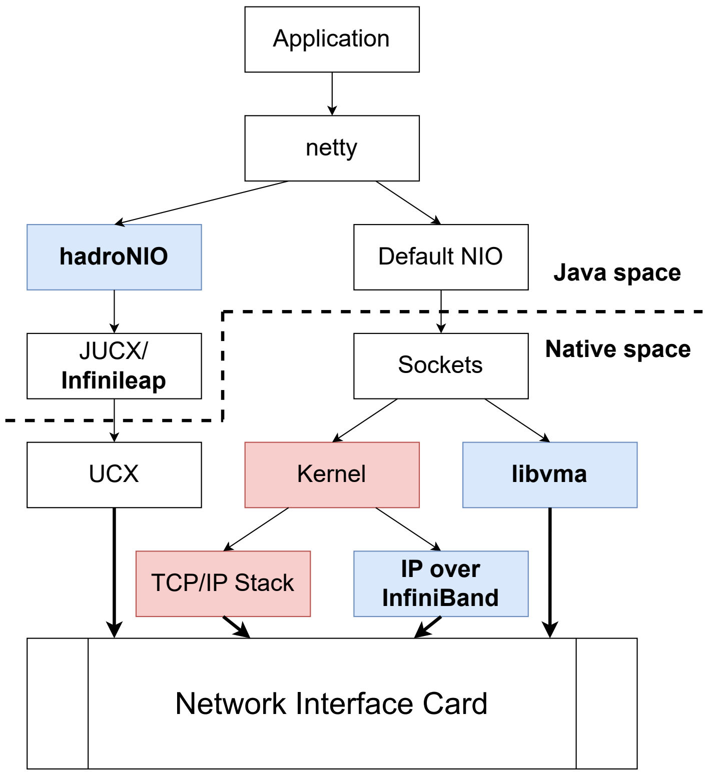


Figure 1.1: Project Hermes Overview. (see 4.4)

Figure 1.1 shows the components and the big picture of Project Hermes. In this context, the author focuses on the basic integration of RDMA hardware in Java within the Infinileap project. He also familiarizes himself with the OpenUCX project, which offers an abstraction layer for various high-performance network controllers. This is used within Infinileap as a backend for controlling RDMA-capable hardware. The developed solution offers an easy entry into RDMA programming using Java, while the code is published on GitHub[17] in the form of an open source project with many examples and comprehensive instructions.

Chapter 2

The Java Ecosystem and its Unsafe Mechanisms

The foundation of the Java programming language is the Java Virtual Machine (*JVM*). Since it operates like a virtual machine and provides a platform-independent instruction set[18], it can be used to implement and compile programs that can be executed in a platform-independent manner. In addition, the Java Development Kit (*JDK*) and the abstractions and functionalities it contains provide an easy entry into programming[19]. This chapter takes a closer look at specific components of the Java ecosystem in order to create a basis for the work that follows.

2.1 Garbage Collection

Java belongs to the group of memory-safe programming languages. Unlike programming languages such as *C++*, it is not possible (without bypasses) to perform illegal memory operations, which can cause the JVM to crash. This security is achieved by transferring all memory allocations and operations from the responsibility of the programmer to the responsibility of the JVM. The memory areas allocated by the JVM are also returned to the programmer not as pure pointers but as typed references to objects. Therefore, a kind of abstraction layer is built to prevent the programmer from accessing raw memory addresses and manipulating the underlying memory at will. To prevent memory leaks, the reserved memory must be released again. This responsibility also falls within the scope of the JVM and thus forms a relief for programmers, since reserved memory does not have to be released manually, as is common in other programming languages like C/C++. This is accomplished by the JVM maintaining an overview of all the object references used at all times and comparing them with the current state of the program at runtime. Within the JVM, this concept is called *Garbage Collection*. From a general point of view, the JVM uses this concept to free the underlying memory of objects to which references no longer exist, so that it can be used for new objects. In detail, however, this approach requires complex mechanisms as well as a good assessment of the runtime environment. An overview of the involved steps is provided in the *Java Garbage Collection Basics*[20] and can be described as follows.

Marking Phase – The JVM maintains a list of all allocated object references. Periodically, it is accessed and run through for garbage collection. While other programming languages use the concept of reference counting (cycles cannot be detected) to automatically free memory areas, Java's garbage collection mechanism is based on reachability analysis[21]. This means that for each object in the maintained list, the JVM periodically checks whether it can be reached by the currently executed program code in form of references. This process starts at object references that the JVM considers guaranteed to exist – the so-called Gargabe Collector Roots (*GC Roots*). These references include, for example, threads that are currently executing or classes that are currently loaded. Starting from each GC Root, the Garbage Collector traverses the object graph and checks whether the object currently under consideration can be found in it. If this is not the case, there is no longer a reference to the object and consequently it can no longer be accessed. Such objects are marked for release by the garbage collector within the marking phase.

Deletion Phase – The list of all allocated objects is used here, too. One difference now is that the previous phase marked all objects to be released using Reachability Analysis. This information is now used and all objects or memory areas that have such a mark are released. After releasing the marked areas, they can be used for new objects, provided that the created gap is sufficiently large. However, in order to efficiently utilize the available memory and prevent fragmentation, the heap is additionally compacted. This means that the free gaps created by releasing marked memory areas are closed by moving the objects that are still in use together within the memory. After the deletion phase is complete, the memory areas of all objects that are no longer accessible or to which no more references exist are released and can be used again.

Under normal use of the Java programming language, many small objects are created on the heap, which are needed only for a short amount of time[22], [23]. A good example of this is the class `java.lang.String`, which is used to store simple strings. Every time a string is needed, whether to read configuration values or to parse a query, an instance of this class must be created. Since operations involving strings can occur very frequently, it is not uncommon to have millions of string instances on the heap from time to time. If this occurs, the garbage collector must step in and ensure that the used memory is released. However, simply traversing millions of objects allocated on the heap would lead to considerable performance losses. Since pointers may also need to be moved during a garbage collection cycle, the garbage collector can trigger a so-called stop-the-world (*STW*).[24] This causes the application's program code to stop all threads. In the case of a web server, for example, this event results in requests from clients not being answered for a short or longer amount of time, thus significantly increasing latency.

To keep the times required for a garbage collection cycle as short as possible, the JVM uses generational garbage collection.[25] This technique is based on an empirical analysis by the JVM developers, which showed that object instances are predominantly used only for a short period of time and can therefore be cleaned up relatively soon after their creation.[20] Based on this knowledge the heap is divided into regions which are called *generations*. Each of these generations stores object instances, depending on their lifetime. In a simple configuration, the heap consists of a total of three generations, the *Young Generation*, the *Old Generation* and the *Permanent Generation*. Each of the mentioned generations is associated with the following characteristics.

- **Young Generation** – After allocating an object, it is first placed in this generation. If the memory consumption within the Young Generation exceeds a certain threshold, a *Minor Garbage Collection* is triggered. This type of collection only considers the address space of the Young Generation and thus only processes objects that do not yet have a long lifetime. In addition, within each minor garbage collection, the number of times an object survived the collection within the young generation is counted, i.e., it was still accessible by reference in the program code. If the value of the counter for an object exceeds a configured threshold, the object is moved to Old Generation.

- **Old Generation** – Objects that are stored within this generation have a longer lifetime. This means that they can be assumed to be used in different places in the program code over a longer period of time. Object instances within the Old Generation are cleaned up again from the heap using a *Major Garbage Collection*. However, this is executed much less frequently compared to the Minor Garbage Collection of the Young Generation, since it has a comparatively much higher runtime and can thus strongly influence the execution of the program. This is due to the fact that with a major garbage collection the entire heap must be traversed, including all allocated object instances.

- **Permanent Generation** – This area is used to store metadata, which usually has a very long lifetime. These include, for example, method definitions that are likely to be needed repeatedly in different parts of the program. Likewise, information about classes is contained here, such as the defined fields or instance variables. Objects of this generation are usually rarely cleaned up, which is why it is only included in a Major Garbage Collection.

Using these sub areas reduces the number of object to analyze during gargabe collection. A characteristic of the Young Generation area is, that it is additionally divided into three sub-areas. These areas are on the one hand the so-called *Eden Space*, in which all newly allocated objects are created, and on the other hand two *Survivor Spaces S0* and *S1*, into which objects are moved that have survived a minor garbage collection in Eden space.

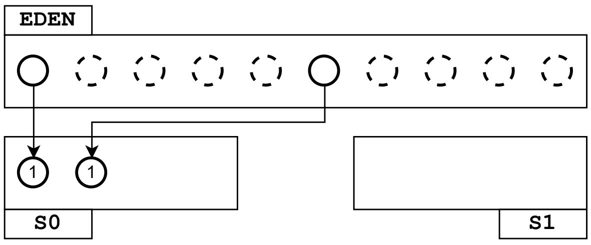


Figure 2.1: Surviving objects get moved into the `S0` survivor space.

During the first Minor Garbage Collection, the objects that are still accessible by the program code are first copied to Survivor Space S0. At the same time, they are assigned a counter value here, which is incremented after each garbage collection, in case of survival. Figure 2.1 shows the described mechanism. Two of the total 10 contained objects in Eden Space were found to remain active using a reachability analysis. Conversely, 8 objects were found to no longer be reachable. Based on this information, the two objects that can still be reached are copied into the Survivor Space S0. Here they are simultaneously assigned a counter with the initial value 1, which indicates that both have survived one garbage collecton cycle.

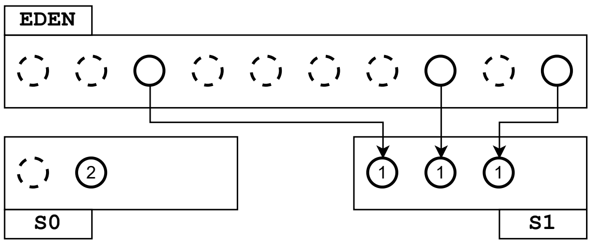


Figure 2.2: The Garbage Collector switches Survivor Spaces.

As soon as the first run of the garbage collection is completed, the Garbage Collector switches the Survivor Space. As can be seen in Figure 2.2, the objects of the next garbage collection cycle that are still accessible are not moved to Survivor Space S0 but to S1, whereupon they are also assigned the initial counter value 1. Since a Minor Garbage Collection includes the entire Young Generation, the Survivor Space S0 is also processed. Here it is determined that the first object is no longer accessible and must be removed accordingly. In the case of the second object, it is determined that it can still be reached, whereupon the counter value is incremented. This sequence of operations is repeated until the counter value of one or more objects of a Survivor Space exceeds a configured threshold. After this, the object(s) will be moved to the Old Generation, as it can be assumed that they are long-lived.

2.2 Java Unsafe API

Instances of objects are created using the `new` keyword in the Java programming language. This has the consequence that the respective constructor (as well as constructors belonging to super classes) of the class to be instantiated is called. Depending on the complexity, an object instantiation can therefore take different amounts of time and, in the worst case, slow down the program's execution[26]. In simple cases where the application creates a small number of objects, this mechanism is desirable because in the constructor the parameters used to initialize the object can be validated and thus errors can be caught early. However, in other cases, such as processing a network stream that provides a large number of values at short intervals, instantiation can have extremely bad performance implications.

> **Example**
> An application processes a large amount of sensor data and stores it on a persistent storage medium such as a hard disk for archiving. While collecting this data, objects of a class describing the data are created. During each instantiation, the constructor associated with the class is called, in which the passed data is validated. Since these created objects have a very short lifetime - namely the time between instantiation and persistence on the storage medium - the garbage collector must become active very frequently at short intervals to clean up the objects that are no longer needed and free up the associated memory. In the overall picture, due to the pauses caused by the garbage collector, this leads to highly fluctuating latencies during the processing of the data.

Many big data frameworks address this problem by using a JDK-internal programming interface which allows to bypass some security precautions of the JVM. This programming interface is called the Java Unsafe API.[27] It is located inside an internal package called `sun.misc` which cannot be used directly in Java application. However, since Java offers the possibility to access private fields of a class without the respective authorization by means of the Reflection API[28], it is possible to use the Unsafe API as follows.

UnsafeProvider.java Java

```java
1  public class UnsafeProvider {
2
3    public static sun.misc.Unsafe get() {
4      try {
5          Field field = sun.misc.Unsafe.class.getDeclaredField("theUnsafe");
6          field.setAccessible(true);
7          return (sun.misc.Unsafe) field.get(null);
8      } catch (NoSuchFieldException | IllegalAccessException e) {
9          throw new RuntimeException(e);
10     }
11   }
12 }
```

Figure 2.3: Accessing the JDKs Unsafe API through the use of Reflection.

The code example in figure 2.3 shows how to access the Unsafe API using the Reflection API. Within the example, the following operations are performed in the respective associated lines.

⑤ The `Unsafe` class stores an instance of itself inside a private static field named `theUnsafe`. This field is first retrieved using the Reflection API and stored inside the `field` variable.

⑥ The instance method `setAccessible` of the class `Field` allows to override the access rights of individual fields using the Reflection API. This is done at this point to gain access to the `theUnsafe` field.

⑦ With the help of the instance method `get` of the class `Field` it is possible to access the stored value or in case of an object the stored reference which is held in the field. The parameter of the function defines from which instance the respective field is to be extracted. In this case the field to be accessed is statically defined, so no instance is needed and for this reason `null` can be passed as a parameter.

In summary, calling the `UnsafeProvider#get` class method allows to get an instance of the `Unsafe` class. This instance can then be used to call the instance methods of the class and thus execute internal functionalities at the user level. The functions of this internal API can be roughly divided into a few categories

2.2.1 Object Instantiation

Besides the usual way of creating objects using the `new` keyword with the associated constructor call, the Unsafe API provides a function that can instantiate arbitrary classes without calling the respective constructor. This function is called `allocateInstance`. Unlike normal object instantiation, it merely reserves the memory needed for the instance and then returns an object reference to that reserved area.

AllocationDemo.java Java

```java
public class AllocationDemo {

  public static void main(String[] args) {
    var unsafe = UnsafeProvider.get();
    var internalData = unsafe.allocateInstance(InternalData.class);
  }
}
```

Figure 2.4: Allocating an instance using the Unsafe API.

Figure 2.4 shows an example instantiation of a class named `InternalData` using the Unsafe API. The class `InternalData` does not need to have a public or accessible constructor. So even if the only constructor of the class is declared with the `private` keyword, an instance of the respective class can be created and used this way. This is a great advantage especially when deserializing objects. For example, if a class triggers a side effect within its constructor, it would occur every time the class is instantiated. With a large number of objects, such behavior can lead to performance degradation. To work around this, the object can first be allocated using the `allocateInstance` function and the fields inside it can then be set manually without executing the constructor and its side effects. A deserialization of objects, which is realized in this way, entails a very low overhead.

2.2.2 Synchronization

Another category of available functions is the synchronization of the executed code at runtime. Here the Unsafe API offers functions for synchronization on a very low level. Such functions are mainly known from programming languages such as C or C++, for example.

First, it is possible to exclude individual threads from scheduling and include them again. The Unsafe API functions used for this are `park` and `unpark`. They can be used to precisely control the execution of concurrent tasks. One application, for example, is the implementation of an custom scheduler if special requirements or mechanisms need to be met. A thread which calls the `park` method is stopped immediately and must then be restarted by another thread using its reference and the `unpark` method. An exception is the call of the `park` method with parameters which express that the thread should only be stopped for a certain amount of time. For this, the method receives a `boolean` parameter indicating that stopping the thread is not absolute. In addition, the second parameter of type `long` specifies for how many nanoseconds the thread should be stopped. After the specified time has elapsed, the thread is automatically woken up again and thus does not need to be brought back into operation using the `unpark` method.

Another group of functions within the synchronization area is the so-called memory fencing[29]. Under normal circumstances, the CPU uses many optimizations during the execution of a program to achieve good performance. One of these optimizations is memory reordering. Here the CPU can rearrange memory accesses - i.e. read and write operations - in the execution sequence, so that operations which, according to the code, should take place after other operations in terms of time, are executed before them. For example, if the CPU determines that several pending read operations can be combined without affecting the overall result of the remaining operations, it performs this optimization step. Especially with contiguous memory areas, this type of optimization can greatly increase performance, since the CPU cache can be utilized better. However, there are also cases where the reordering of operations by the CPU is not desired. This may be the case, for example, if synchronization mechanisms are to be implemented which are based on the sequence of read and write operations performed. For such cases, the Unsafe API offers various memory fences.

- `loadFence`
  This function ensures that all **read** operations defined before its call are not reordered with either read or write operations after its call.

- `storeFence`
  This function ensures that all **write** operations defined before its call are not reordered with either read or write operations after its call.

- `fullFence`
  This function ensures that all **read and write** operations defined before its call are not reordered with either read or write operations after its call.

Within the x86 architecture, the above functions map respectively to the `LFENCE`, `SFENCE`, and `MFENCE` instructions[30]. This fact already shows the low level at which the operations are carried out.

Atomic memory accesses represent another group of functions within the Unsafe API's synchronization operations. Using the Java programming language, atomic operations are performed by means of already existing classes such as `AtomicInteger`, `AtomicLong` and `AtomicRefernce`. However, in some cases, access to atomic operations at a low level is needed. Here, the Unsafe API provides access to functions that directly operate on the underlying memory while using the corresponding processor instructions. The most commonly used of these operations is the Compare-And-Swap (*CAS*) operation, which is implemented by the x86 processor instruction `CMPXCHG`.

CompareAndSwapDemo.java Java

```java
 1 public class CompareAndSwapDemo {
 2
 3   public static void main(String[] args) {
 4     var unsafe = UnsafeProdiver.get();
 5     var internalData = unsafe.compareAndSwapLong(
 6       null,
 7       0x4000,
 8       0x42,
 9       0x24
10     );
11   }
12 }
```

Figure 2.5: Executing a Compare-And-Swap Operation using the Unsafe API.

Figure 2.5 shows an exemplary CAS operation which is executed on a value of type `long` (64 bit). The parameter `null` in line 6 specifies that the operation is not to be executed in the context of an object, but directly on the memory address specified in line 7. The semantics of a CAS operation dictate that a value at a given memory address is compared to a given value and, if equal, replaced by another given value. The value being compared with is passed in line 8 in the preceding example, while the value being used for the replacement operation is defined in the following line 9. In summary, the operation compares the 64-bit value stored at the virtual memory address `0x4000` with the numerical value `0x42` and exchanges it with the numerical value `0x24` if it is equal. The use of this programming interface is particularly useful when the allocation of objects such as the `AtomicLong` class must be avoided due to performance requirements. In such cases a permanent memory area can be created, in which all synchronization variables are stored. Consequently, this has the advantage that the garbage collector does not have to manage any references.

2.2.3 Memory Manipulation and Management

All objects allocated using the `new` keywords are created within a memory area or heap managed by the JVM. This memory area is the one that is constantly checked and cleaned up by the garbage collector. As mentioned in 2.1, a garbage collector cycle can lead to the shifting of memory addresses or pointers due to the compaction of the heap. In some scenarios, this behavior is highly undesirable. If, for example, a memory address is transferred to a hardware component which then accesses the transferred pointer by means of Direct Memory Access (*DMA*)[31], the address must not change between transfer and access. If this case does occur, the hardware component accesses a memory area which no longer contains the expected data.

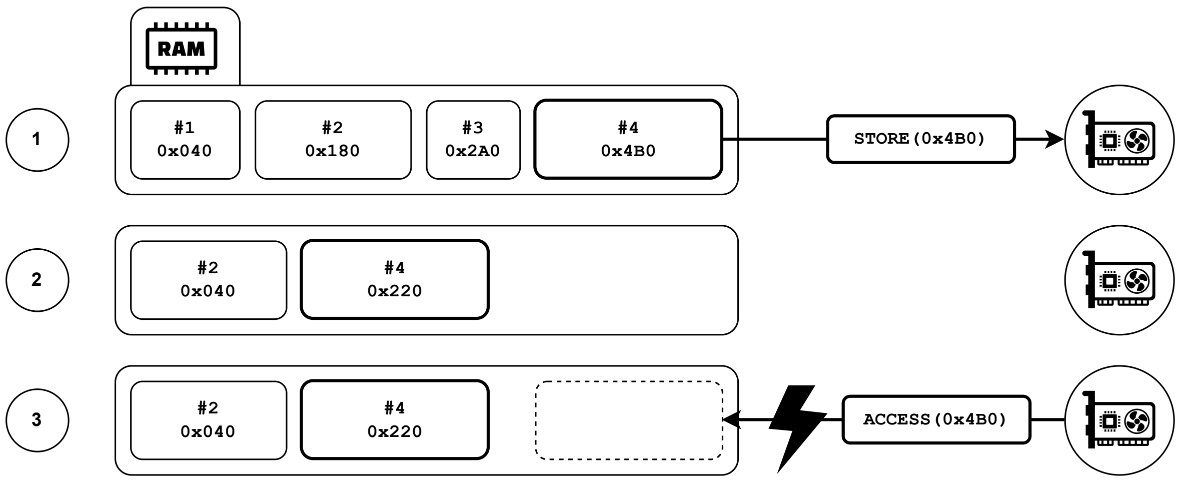


Figure 2.6: A failing direct memory access after heap compaction.

Such a scenario is shown in Figure 2.6. In step ① the program first transfers the memory address of the object `#4` to the hardware controller. It then stores the address within its memory for later use. After this step, the garbage collector starts a cycle and determines that the objects `#1` and `#3` can be cleaned up. This operation is performed in step ②, whereupon objects `#2` and `#4` are each given new memory addresses by compacting the heap and thus recopying the memory. In step ③ the hardware controller now tries to access the memory address `0x4B0` stored in its memory using DMA. Since the expected data has already been copied, the controller erroneously accesses data that no longer has any relation to the intended operation. Such an error can initially remain undetected in the case of calculations, as the resulting end result can still appear plausible. In the case of network applications, however, it would quickly become apparent, as network packets containing invalid values would be sent.

This problem is addressed by the JDK providing the `ByteBuffer` class[32], which uses the Unsafe API internally. With its help it is possible to allocate memory areas which are located outside the managed heap area of the JVM. This unmanaged heap area is called *Off-Heap Memory* within the Java ecosystem, and does not get compacted automatically. From the point of view of the garbage collector, this memory area does not exist, which is why it must be managed manually. For this purpose, the Unsafe API offers the two methods `allocateMemory` and `freeMemory`, analogous to the functions `malloc` and `free` known from C. The `ByteBuffer` class reserves a memory area within the off-heap memory during the allocation of an object using the Unsafe API's methods. It also releases the reserved memory again as soon as the garbage collector attempts to clean up the associated object. This way, it is possible to use memory areas outside the managed heap and continue to rely on the safeguards of the JVM. However, in cases where explicit management of memory is necessary, it is better to use the Unsafe API directly. For example, if certain characteristics of the CPU, such as the cache, are to be utilized for optimization purposes. Here, the memory can be created and used in such a way that the CPU cache is optimally utilized by aligning the allocated data. It may also be necessary to carry out read or write operations on a specific memory address, which the `ByteBuffer` class does not permit. An example of this is a memory address which is mapped into the JVM process via a memory mapping.

In addition to simple allocation and deallocation of off-heap memory, the Unsafe API also provides methods for manipulating memory. These can either be used to read and write fields of individual objects directly without any access control, or to manipulate off-heap data.

PrivateValue.java Java

```java
1 public class PrivateValue {
2   private long value;
3 }
```

UnsafeWriteDemo.java Java

```java
1 public class UnsafeWriteDemo {
2
3   public static void main(String[] args) {
4     var unsafe = UnsafeProdiver.get();
5     var value = unsafe.allocateInstance(PrivateValue.class);
6     unsafe.putLong(value, 0x0, 0x42);
7   }
8 }
```

Figure 2.7: Direct write access to an object's private field using the Unsafe API.

The example in Figure 2.7 shows how a field that should not be accessible under normal circumstances can be manipulated using the Unsafe API. This is accomplished by using the `putLong` method of the Unsafe API within the `UnsafeWriteDemo` class in line 6. The first parameter specifies the object on which the write operation is to be carried out. In this case, it is the previously allocated object `value` of type `PrivateValue`. The second parameter specifies the offset at which the operation is to be executed within the object. This is the offset in bytes from the start of the reserved memory area of the object's fields. As the associated class only has one field, the offset here is `0x0`. The third parameter specifies which value is to be written at the corresponding position in the memory. Here it is the value `0x42`. After the operation has been executed, the `value` field contains the value `0x42`, although external access to this field has been restricted or prohibited using the `private` keyword. This example shows one of the most powerful operations of the Unsafe API, namely the manipulation of arbitrary objects and memory areas without any security controls.

Since these operations skip almost all safety checks and map directly to the respective underlying processor instructions, they can be of great advantage in performance-critical applications. For example, the deserialization and serialization of data for sending within a network application or for storage on a persistent storage medium can be greatly accelerated in this way, as objects can be copied directly out of the memory using the Unsafe API for the purpose of serialization by means of read operations and copied back in again by means of write operations (see 2.3)

2.3 High-Performance Object Serialization based on Ahead-of-Time Schema Generation

Filip Krakowski, Fabian Ruhland and Michael Schöttner. High-Performance Object Serialization based on Ahead-of-Time Schema Generation. In 2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), TrustCom 2023, Exeter, United Kingdom, November 01-03, 2023.

Contributions:
The mechanisms presented in this paper were initially started in the context of the DXRAM project[33]. The idea was to replace the serialization mechanism within the application in order to achieve data transmission with even lower latency. The author implemented the Skema project for this purpose, which can serialize data located in the heap or off-heap to any location in the main memory using the Java Unsafe API.

As part of the implementation, the author compared different frameworks (Kryo and FST), which pursue similar goals but solve them in a different way. Based on this, the author first developed a recursive mechanism that can run through the fields of any class in a deterministic order. This mechanism is one of the main contributions of this work. The author then adapted the developed mechanism so that it could be stopped and resumed at any byte boundary. For this purpose, the author first designed a way to save the recursively traversed path during serialization. The implementation was then also carried out by the author in the form of a stack, which stores the indices of the traversed fields and thus enables the path to be traced. This solution finally led to the next main contribution of this work, which is partial serialization and was also implemented by the main author.

Finally, to evaluate the performance of the solution, the author implemented a series of benchmarks using the Java Microbenchmark Harness framework and visualized these in a suitable form using the Pandas library in Python.

While the author wrote the textual part of this work, Michael Schöttner and Fabian Ruhland contributed valuable input in the form of many discussions regarding the implementation and results of the developed benchmarks.

Status: published

High-Performance Object Serialization based on Ahead-of-Time Schema Generation

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*—Many of today's Big Data systems are developed in the Java programming language and require fast object serialization and deserialization for efficient and high-speed message exchange. We address this by introducing Skema, a library providing high-performance serialization of Java objects. Skema works based on Ahead-of-Time schema generation, so that only a minimal number of operations need to be performed at runtime to serialize an object. This approach allows Skema to also serialize individual objects only partially which is a great advantage for network communications with fixed buffer sizes. Furthermore, targeted optimizations, such as caching the size of the serialized form of objects that have a fixed size so that they do not have to be recomputed for each operation, are presented. In addition to a more in-depth explanation of the implemented procedures, this paper also presents benchmark results comparing the Skema library against Kryo, FST and the native Java serialization function. The results of these experiments achieve very good values in comparison with the remaining libraries. For example, one experiment shows that there can be nearly a hundredfold increase in the speed of deserializing objects when compared to the native Java serialization mechanism by using Skema. We also show that deserializing objects using the native Java mechanism requires a large amount of additional memory per operation, which puts a burden on the garbage collector afterwards. Finally, to investigate the scalability of the implemented solution, the benchmarks are also performed using different numbers of threads and the results are presented graphically.**

***Index Terms*—Java, Serialization, Native, High-Performance**

I. Introduction

The serialization of objects represents one of the most fundamental mechanisms in the software engineering environment. In addition to the many possibilities for its use, it makes it much easier for developers to save as well as load complex data or even application state from persistent storage devices. Likewise, it allows language-specific objects to be sent between processes in a distributed system over the network without much effort and thus enables, for example, the continuation of a calculation with its associated state in another environment[1], [2]. In this area, new solutions are also constantly being developed, such as zero-copy serialization, in which no copies of the data to be serialized are created [3], [4]. With the introduction of persistent random access memory modules (NVDIMM), it is now even possible to continue or resume a process terminated by an error without having to recompute the state that existed at the time of the error [5], [6]. In such a case, the programmer must of course still check whether the persisted state still has its validity and has not been corrupted, otherwise the application would operate using incorrect data. Another important application area is the scaling of distributed applications. Here, individual instances can be quickly booted up and shut down by keeping the application state constantly in persistent memory [7], [8]. Pausing and resuming the application is thus possible and fast at any time. A high degree of serialization also exists within Java-based big data systems, which must first serialize the data to be processed and then distribute it to additional instances of the system for the actual computation. Since the distribution of data is a core task in such systems, it must be carried out quickly. Serialization is therefore a function that should be optimized as much as possible.

Serialization in the context of Java can be distinguished between two types. On the one hand, there are libraries where the user must first create definitions for the data to be serialized in the form of schemas, i.e. write them manually. Well-known examples of this are Google's Protobuf [9] as well as Flatbuffers [10] libraries and the Cap'n'Proto library [11]. The second option is to use Java's reflection mechanism, which allows objects to be inspected for their structure at runtime, but produces a non-negligible overhead in terms of performance [12].In addition to these two types of serialization, there are also works based on modifying the Java Virtual Machine (JVM) so that it can directly access the memory managed by the JVM and then exchange it between instances of a distributed system using Remote Direct Memory Access (RDMA) without having to process the data first [13], [14]. Since these solutions require a modified version of the JVM, they will not be discussed in detail in the remainder of this paper. The solution presented in this paper focuses on combining the mentioned methods by generating schemas ahead-of-time once at program startup using Java Reflection and then reusing them. The resulting contributions of this paper are a recursive method for generating schemas of arbitrary Java classes, the possibility to serialize and deserialize object instances only partially, a method for detecting classes with static size to avoid recalculating the object's size, and an easy-to-use interface for integration into existing software systems.

II. Java Object Serialization

Since serialization of data is a very frequently used operation within the Java environment, the development kit natively supports the ability to convert objects into byte streams. The required classes and interfaces are available in the standard `java.io` package. A core feature of the provided serialization functionality is that it can be used independently of the deployed Java Virtual Machine. This means that an object can be serialized on an Oracle HotSpot JVM, for example, then sent over the network and finally deserialized on an Eclipse OpenJ9 JVM so that both instances are identical from the JVM's point of view.

Any class that will later be used in the context of Java's built-in serialization feature must implement the marker interface `java.io.Serializable` (an empty interface with no methods or fields). The presence of this interface tells the JVM that the implementing class is eligible for serialization. Another mechanism, which enables a more fine-grained control of the byte representation of the objects to be serialized, is the `java.io.Externalizable` interface. Classes implementing this interface must define the two methods `readExternal()` as well as `writeExternal()`, in which the data stored within the instance must be manually transformed into a byte representation. It should be noted that - using Java's automatic serialization feature via the `Serializable` marker interface - only fields are serialized which are not defined with either the `transient` or the `static` keyword. To transform instances of the class to be serialized into byte streams, the class `java.io.ObjectOutputStream` is used. On an instance of this class, an object that needs to be serialized can be passed to the `writeObject()` method to write it to the underlying data stream. The target of this write operation can be chosen freely, which makes it possible to write to persistent storage, such as solid-state drives, or directly to existing network streams, such as Java sockets. With the introduction of lambda functions in Java 8, it is also possible to serialize functions in addition to simple objects, for example, to execute them on remote computers. This type of serialization is primarily found in distributed compute platforms, where functions are distributed to worker instances and then executed on them. Another building block in which the native Java serialisation mechanism is anchored is Java's Remote Method Invocation (RMI). Here, functions are exported to external consumers by means of interfaces that inherit from a parent interface `java.rmi.Remote`, so that they can access them via a network. Building on this function, it is possible to develop larger distributed systems that communicate using RMI and encode and decode the actual data using Java's serialisation function.

While the JDK's natively supported serialization features bring many advantages, one characteristic stands out as a significant negative: performance and storage overhead. Java's built-in serialisation mechanism extracts the values of an object's fields using reflection, which is known to be much slower than ordinary field access. Major factors that come into play here are the validation of access operations by means of reflection. For example, for each field access, it must first be checked whether the field to be accessed also belongs to the provided instance of an object or its class. It must also be checked whether the caller has the necessary rights to access the respective field or not. An example of this are package-private fields, which are to be accessed from outside the declaring package via reflection. In such a case, an error must be thrown at runtime, as access would not be possible under normal circumstances and an error would be thrown at compile time.

III. Framework Design

This chapter presents the fundamental mechanisms of the Skema framework as well as some optimizations to speed up the time it takes to perform a serialization operation.

A. Class layout

Within the Java programming language, data structures are modeled in a object-oriented fashion using classes. In a general sense, classes can be regarded as providing a blueprint for the creation of the actual objects or instances of the respective class. For this purpose, each class defines the fields it contains, such as primitive data types (`int`, `long`, `float`, ...). Besides these simple field types, complex data types can also be used. These include references to further instances of classes (i.e. objects) or arrays of elements.

Tree.java Java
```java
public class Tree {
  private final Node root;

  ...
}
```

Node.java Java
```java
public class Node {
  private final int value;
  private final List<Node> children;

  ...
}
```

Fig. 1. A simple tree data structure storing integer values defined using two Java classes.

Using these properties, complex data types, such as the tree structure shown in Figure 1, can be modeled. Besides the fields of a class, functions can also be defined, which are executed in the context of the created instance. However, since the functions belonging to a class are stored statically and are internally passed an instance as a parameter to operate on the respective object, the individual functions are not serialized due to their existence. If an entire class, which was previously unknown, is to be loaded, the .class file would have to be sent to the target, which would then have to load the received class definition via its class loader. Since these are mechanisms outside the context of serialization, they are not discussed in detail further.

In general, the statement can be made that every data structure defined in Java must work with primitive data types at its deepest level, since all complex data types consist of just those. The resulting conclusion is that the data to be serialized are exclusively primitive data types, that is 1-, 2-, 4- or 8-byte values or contiguous memory regions (arrays) consisting of them. Furthermore, any field that has a primitive data type forms a direct reference into the computer's random access memory. In the example from Figure 1, the `value` field of the `Node` class would thus point directly to an address in the program's virtual address space, and any write access to this field would manipulate the underlying memory. Knowing that instances of a Java class are merely a structured collection of primitive data types in the program's virtual address space, a serialization mechanism can be developed that efficiently exploits these properties. To do this, it is first necessary to determine how the fields of a class are arranged in memory. The Java Development Kit provides the Unsafe API [15] with its `jdk.internal.misc.Unsafe` class for these purposes, which, as the package name suggests, is used for internal purposes. Even though it is not well regarded to use this class, it is very widely used in many projects and offers many advantages in terms of performance, such as:

- **Object instantiation**
  Using the `allocateInstance` method, it is possible to instantiate a class without calling its constructor first. During deserialization, it is assumed that the data or fields read are correct and represent an exact copy of a previously instantiated class for which the constructor was called. Here it is therefore not necessary to call the constructor again, which in turn saves many CPU cycles.
- **Manual memory management**
  Using the `allocateMemory` and `freeMemory` methods, it is possible to manage memory manually similar to `malloc` and `free` in the C programming language. Since these memory areas are outside the garbage collector's management area, the garbage collector does not have to check accessibility and is thus relieved at the expense of increased programming effort.
- **Class introspection**
  The `Unsafe` class offers with its `objectFieldOffset` method the possibility to determine the memory offset of a field within a class. In addition, the class defines several getter and setter functions which can be used to read and write primitive data types at freely chosen memory offsets within an instance of a class. These two functionalities together allow to manipulate any fields of an object regardless of the access rights.

The Skema framework takes advantage of these and several other features to generate a schema for each class that is to participate in the serialization process.

B. Automatic schema generation

Within the framework, a class schema describes how and at which memory address the contained fields must be read or written. Since this information is not provided in ready-made form by the JDK, it must first be generated using suitable methods. Consequently, before an instance of a class can be serialized, the corresponding schema must be generated. The only way to find out which fields are defined within a class at runtime is to use Java's Reflection functionality. For example, it is possible to retrieve the fields of a class using the `getDeclaredFields` method defined on the `Class` class. The information obtained in this way includes, among other things, the names of the fields and their type, which can be a primitive or complex data type. Using the `objectFieldOffset` function mentioned in chapter III-A, all memory offsets belonging to the queried fields can then be obtained.

SensorReading.java Java

```java
public class SensorReading {
  private final long timestamp;
  private final double   value;

  ...
}
```

Fig. 2. A simple data structure describing a sensor reading.

Looking at the preceding example in Figure 2, it is programmatically possible to determine that the `SensorReading` class has the two fields `timestamp` of the primitive data type `long` and `value` of the primitive data type `double` by means of reflection at runtime of the program. With this information and the mentioned `objectFieldOffset` function it is then possible to determine at which memory offsets these two fields are located within the object. This process, i.e. the extraction of the information belonging to all fields, is performed recursively until the entire class graph has been traversed and thus all fields are covered. The information collected in this way is finally bundled and stored in a schema, which is accessed during serialization.

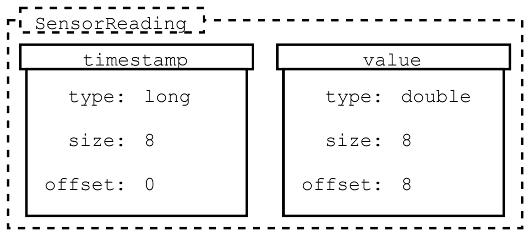


Fig. 3. An exemplary abstract schema of the SensorReading class.

The diagram shown in Figure 3 describes the structure of the SensorReading class by means of field types or field sizes and the associated memory offsets within the instance of the class. Using this information and the provided functions of the Unsafe class to manipulate fields without respecting access rights by specifying memory offsets, it is finally possible to create a Java object by means of write operations directly in memory without making object-oriented access to the fields. Likewise, it is possible to extract or serialize an exact replica of a Java object from the main memory and then restore or deserialize it from the data generated in this way. An important point to note here is that unlike the usual Java serialization mechanism, the functions belonging to Reflection are only called once - just to create the schema of a class - and thus do not cause any performance overhead during runtime. This is possible because the schema describing a class contains all the information that the functions of the `Unsafe` class need to read data from instances of the class or write data to the individual fields.

C. Optimizations

In order to keep the performance stable and predictable during runtime, some optimizations are used to make efficient use of the available information. The Java programming language allows elements to be stored within arrays, so that individual elements are accessed by reference to the array and an index or offset. The length of these array instances can be freely chosen at runtime, so that an array can take up any amount of space in the system's main memory. In order to be able to determine whether the memory provided is sufficient to hold the serialized form of the object to be serialized during the serialization process, its size must first be determined. The same applies in the case where the programmer does not provide a memory buffer, but lets the framework allocate the memory into which the object is to be serialized. The determination of the size of an object turns out to be non-trivial, because a class, which contains at least one array, can form many different instances with different sizes. While the first instance of a class contains an array of length 5, another instance can contain an array of length 3, which makes the two instances different in size. Thus, in cases where fields of dynamic size (i.e., arrays) are used, the framework must first determine the size of these fields for each serialization operation.

The situation is different for fields with primitive data types. Here the sizes in bytes are fixed, which is why an optimization of the size calculation of an object is possible in the following way.

1) Start by generating a schema for a given class.
2) For each visited field, examine whether it has a fixed size and add this size to a counter associated with the schema object.
3) If all fields of a class have a fixed size, the entire class is considered to be of fixed size.

With the help of the size information determined in this way, the calculation of the size for all fields with a fixed size can be skipped later during serialization and the sizes determined in advance can be used. The size of the serialized form of an object is thus determined by the sum of the stored counter of the schema object and all length fields of the arrays contained (directly or transitive) in the object.

In addition to the optimization regarding the size calculation of an object, the serialization functions are implemented in an entirely stateless manner except in the case of partial serialization as referenced in the following Chapter III-D. This means that the provided functions can be used from any thread without having to pay attention to thread safety. Likewise, no new objects are created during serialization, so consequently there is no activity in the garbage collector and the program can execute without interruption.

D. Partial Serialization

A unique feature of the Skema serialization library is the possibility to serialize objects only partially. In detail, this means that serialization can be interrupted at any point and resumed at a later time without having to serialize data that has already been written again. This allows, for example, writing to a bounded network stream buffer before enough memory is available to send the object to be serialized in its entirety. Furthermore, serialization can be stopped not only at field boundaries, but also within fields at any byte. This ensures that the available memory can be used as efficiently as possible. However, it is important to note that the objects to be serialized should not be changed during the pauses, otherwise inconsistent states may occur in the system.

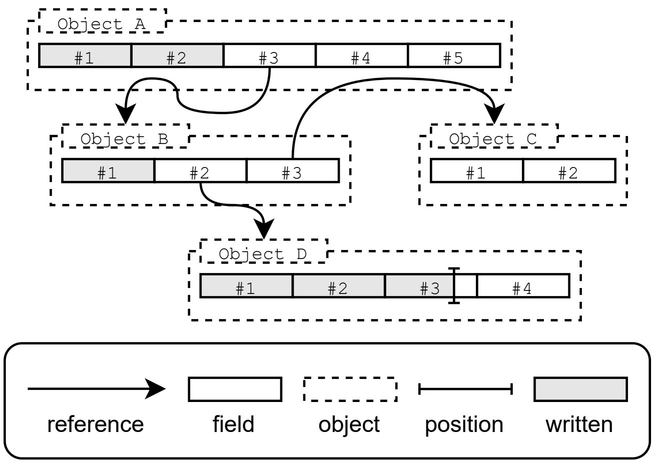


Fig. 4. Skema's partial serialization feature.

For partial serialization, a stateful object representing the current serialization operation is required in addition. This object contains information about the progress of the operation, so that when the serialization function is called again, it can start at the position where it was last stopped. The most important information contained herein is the number of processed bytes, as well as the path within the object graph to the field at which serialization was last stopped. The first information is needed to resume the operation with byte precision, while the second information is used to get to the corresponding field in the object graph. The mentioned path is stored in form of a stack of indices and is traversed when serialization is resumed. Each stored index represents the position of a field within a class. Once the last index has been removed from the stack and the target field has been reached, the serialization operation continues until it again determines that there is not enough memory in the target buffer, or all data has been successfully written.

An exemplary structure of an object graph and the corresponding partial serialization operation can be seen in Figure 4. Here, a total of four objects (Object A through Object D) are serialized with their associated fields. A field can be a reference - i.e. it can refer to another object - or it can store a value and thus not be a reference. If a reference is detected during serialization, it is automatically followed and the index of the corresponding field is stored on a stack. Likewise, the value is taken from the stack as soon as the object being referenced has been fully serialized. As soon as the end of the memory to be written to is reached, the operation is simply aborted and the metadata collected up to that point - that is, the path in the object graph in the form of a stack and the current write position within the last accessed field - is saved. On resumption of the serialization operation, the framework can use the saved stack (in our example `[3,1,3]`) to get to the current field directly and process it further using the saved write position.

IV. Evaluation

In this chapter, the functions and features of the Skema framework are compared and evaluated against other serialization libraries[16], [17]. The following benchmarks are executed on a machine consisting of the hardware shown in Figure 5 and using the software specified in Figure 6.

CPU	1x Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz (22 MB Cache)
RAM	4x Micron Technology 36ASF2G72PZ-2G6E1 16GB

Fig. 5. System specifications of the hardware used in all experiments.

OS	CentOS Linux release 8.1.1911 (Core)
JDK	Eclipse Temurin™ JDK 20.0.1+9
FST	Maven: de.ruedigermoeller:fst:3.0.1
Kryo	Maven: com.esotericsoftware:kryo:5.5.0

Fig. 6. Operating system and software versions used in all experiments.

The benchmarks are written and executed using the Java Microbenchmark Harness (JMH) [18] and can therefore be easily reproduced.

A. Operation Throughput

In the first experiment, the throughput, i.e. the number of serialization as well as deserialization operations per second, is measured. For this purpose, a Java object is created with randomly filled fields based on a specified seed. This object contains one field for each primitive data type and additionally one field for each primitive array type. All stateful objects of the respective frameworks are additionally cached (where possible) so that the garbage collector is not burdened by additionally instantiated objects. This also allows a better comparison of the results, since the respective benchmarks can be run under the same conditions. The operations to be executed are distinguished between serialization and deserialization as well as on-heap memory and off-heap memory. In case of the Kryo library, the classes `UnsafeInput` / `UnsafeOutput` (on-heap) as well as `UnsafeByteBufferInput` / `UnsafeByteBufferOutput` (off-heap) are used, since these also use Java's Unsafe API and thus ensure a better comparability in regard to Skema. Additionally, all off-heap memory areas are allocated page-aligned to better utilize caches and create a more equal baseline for all benchmarks.

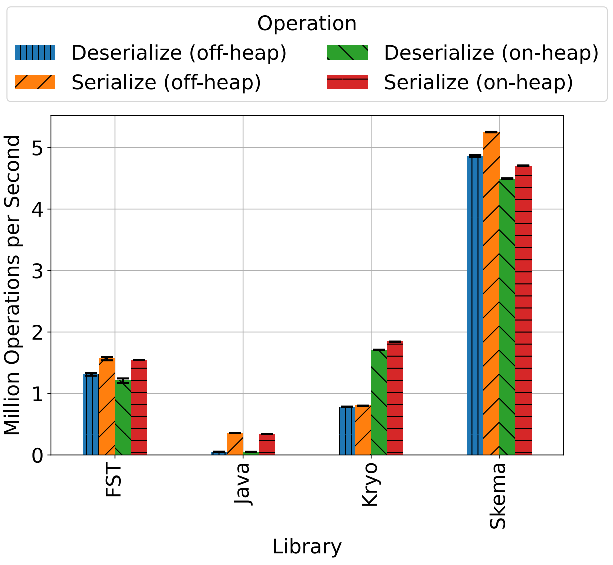


Fig. 7. Average operation throughput for serialization and deserialization in million operations per second.

Figure 7 illustrates the collected results of the throughput benchmark. Each measurement represents an average value, which is formed from five runs of five seconds each, in which operations are performed continuously. One observation that immediately stands out is the virtually non-existent throughput of the native Java deserialization function when using on-heap as well as off-heap memory. This contrasts with the serialization function, which is several times faster in both cases. This observation cannot be made with the other libraries, since the performance of the serialization and deserialization operations is always close to each other in these cases. Another noticeable detail is the large difference in the use of on-heap versus off-heap memory within the Kryo library. Here, the use of on-heap memory, that is, the use of `Unsafe{Input|Output}` classes instead of `UnsafeByteBuffer{Input|Output}`, leads to a doubling of performance. With the FST Library, on the other hand, no major noticeable differences can be found, since the results here are very similar in the case of on-heap as well as off-heap memory. The Skema library performs very well in terms of performance, offering, for example, slightly more than twice the throughput of Kryo in the case of on-heap memory and slightly more than five times the throughput in the case of off-heap memory.

	Serialize		Deserialize	
Library	on-heap	off-heap	on-heap	off-heap
FST	4.55	4.38	24.21	26.16
Kryo	5.42	2.23	34.22	15.64
Skema	13.84	14.68	89.90	96.99

Fig. 8. Comparison of average operation throughput against Java's native serialization mechanism as a baseline.

Figure 8 breaks down the speedups of each library with respect to Java's native serialization feature. Each value represents the factor by which the performance, i.e. the average number of operations per second, increases in comparison. It immediately becomes clear that the use of third-party libraries is strongly advised in case of many deserialization operations within a performance-critical application. While a speedup by a factor of 14.68 is possible when serializing objects using Skema and off-heap memory, almost 100 times as many (96.99) objects can be deserialized within the same time when deserializing compared to Java's native deserialization function.

B. Allocation Rate

During serialization and deserialization, internal functions are called within the respective libraries, which can generate additional temporary data structures. Since these data structures or objects are only required for a short time, the garbage collector must release the memory associated with them again afterwards. Since memory allocations generally have a comparatively high overhead, it is essential to keep memory allocations at a low level in order not to stress the garbage collector and thus avoid pauses during program execution.

The JMH framework supports the ability to add various profilers during the execution of benchmarks. One of these profilers is the gc profiler, which is able to determine the number of bytes allocated per operation or benchmark method invocation. Figure 9 shows the values determined in this way for all libraries examined, distinguishing between serialization

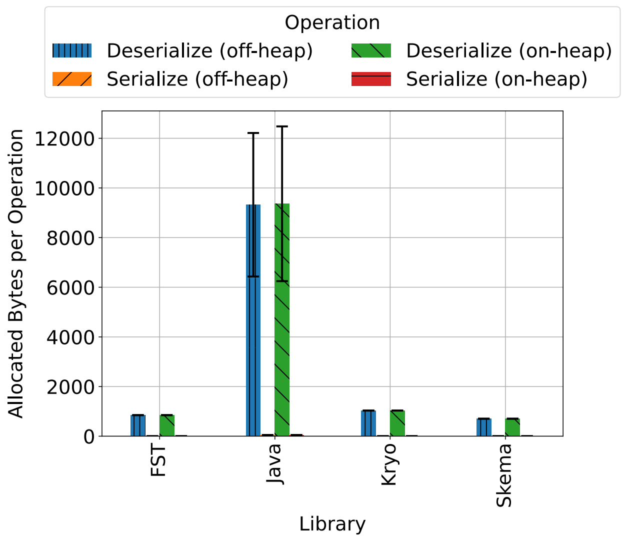


Fig. 9. Average allocation rate for serialization and deserialization in bytes per operation.

and deserialization operations as well as off-heap and on-heap memory.

A noticeable characteristic is the comparatively much higher number of allocated bytes when using the native Java deserialization function, as well as the strong fluctuations of the measured values contained therein. The values determined here may provide an explanation for the poor native deserialization performance observed in Figure 7. On the one hand, the high overhead associated with the allocations can lead to a slowdown during the deserialization operation and, on the other hand, the garbage collector must release the memory occupied during this process once the objects associated with it are no longer accessible. During this release process, there can also be a pause in the execution of the program code, which reduces performance accordingly. In the case of native Java deserialization, the large deviations within the number of bytes allocated per operation can only be explained by the fact that certain data structures are cached and only reallocated when necessary.

As can be seen from the measurements, the other libraries allocate relatively little memory during the deserialization process and also have almost no fluctuations. This is explained by the fact that they do not need any or almost no helper structures and only allocate memory for the fields belonging to the deserializing object. In the case of serializing objects, not a single byte of memory is allocated, so the garbage collector is not required to do any work here. This is possible because the object to be serialized is already allocated and therefore only the contained fields have to be read and then written into a pre-allocated buffer. Unfortunately, this does not apply in the case of native Java serialization, where an average of 50 bytes of memory are allocated per operation.

	Deserialize	
Library	on-heap	off-heap
FST	0.09	0.09
Kryo	0.11	0.11
Skema	0.08	0.08

Fig. 10. Comparison of average allocation rate against Java's native deserialization mechanism as a baseline.

As can be seen in Figure 10, the three libraries FST, Kryo and Skema require only a fraction of memory per deserialization operation compared to Java's native mechanism. Skema performs best with a factor of 0.08 and thus provides the lowest memory overhead. Since the number of bytes allocated per serialization operation is zero for all three libraries, they are not listed within 10.

C. Scalability

The Skema library is implemented stateless for ordinary serialization as well as deserialization operations and also does not allocate any additional memory during serialization. Because of these two properties, no shared structures are accessed during the execution of the operations and no synchronization, which would lead to a degradation of the performance, has to take place. Since the threads can work completely independently in such an environment, parallelism is optimal from an application point of view.

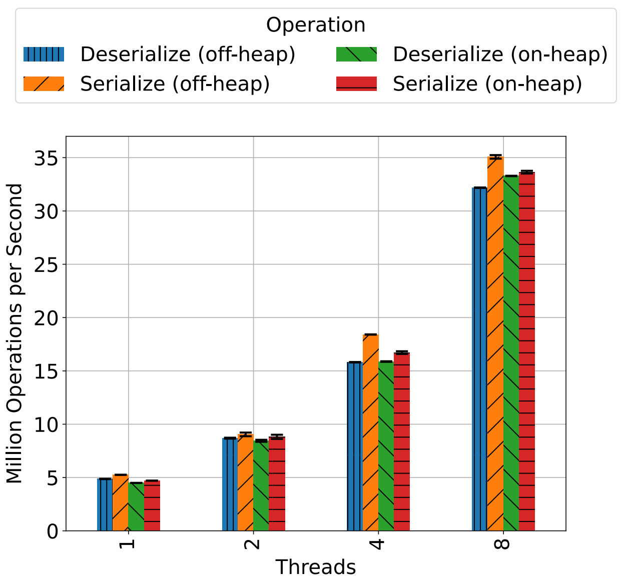


Fig. 11. Skema's average operation throughput for serialization and deserialization in million operations per second and different thread counts.

Figure 11 shows the average number of operations per second using different numbers of threads within the same environment that is used in chapter IV-A. The benchmarks were run in such a way that each thread gets its own buffers and objects. This is necessary to ensure that each thread truly operates independently of the remaining threads. From the measurements, it is easy to conclude that the addition of threads leads to a strong increase in overall performance. While one thread can serialize about 5 million objects per second, this value increases to about 35 million objects per second when 8 threads are used, which corresponds to an increase of about 700%.

	Serialize		Deserialize	
Threads	on-heap	off-heap	on-heap	off-heap
1	1.00	1.00	1.00	1.00
2	1.87	1.72	1.88	1.78
4	3.55	3.50	3.53	3.25
8	7.14	6.67	7.40	6.61

Fig. 12. Comparison of Skema's average operation throughput for serialization and deserialization in million operations per second and different thread counts using 1 thread as a baseline.

Figure 12 shows the speedups achieved by the increase of the number of threads during the execution of the operations. A single thread represents the baseline with a factor of 1.00. The Skema library provides the greatest speedup during deserialization of on-heap data, that is, from a byte array. Here the speedup is 7.40, which means that the use of 8 threads compared to the use of only a single thread leads to an increase of the number of deserialized objects per second by a factor of 7.40.

V. Conclusion & Future Work

In summary, the implemented solution for serialization of Java objects based on ahead-of-time schema generation has a very good performance and in certain cases offers a considerable advantage over native Java serialization functions. Unlike other libraries, it requires no configuration and allows the user to serialize an object with just one line of code. Additionally, the experiments conducted show that multicore systems can be efficiently utilized, as the introduction of additional threads results in a large increase in performance. Based on this work, we plan to integrate and evaluate the developed library in various big data frameworks, such as Apache Spark[19]. Another aspect we would like to investigate is the evaluation of NVRAM-based Java systems that manage their state in the form of an object and store it in persistent memory by means of checkpointing. Here we plan to use our implemented solution for transparent backup as well as recovery of state data and based on the results achieved in this work we expect a good final result.

Acknowledgements

Computational infrastructure and support were provided by the Centre for Information and Media Technology at Heinrich Heine University Düsseldorf.

References

[1] 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 05/29/2023).

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

[3] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang, "Breakfast of champions: Towards zero-copy serialization with nic scatter-gather," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21, Ann Arbor, Michigan: Association for Computing Machinery, Jun. 3, 2021, pp. 199–205, ISBN: 9781450384384. DOI: 10 . 1145 / 3458336 . 3465287. [Online]. Available: https://doi.org/10.1145/3458336.3465287.

[4] A. Wolnikowski, S. Ibanez, J. Stone, C. Kim, R. Manohar, and R. Soulé, "Zerializer: Towards zero-copy serialization," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21, Ann Arbor, Michigan: Association for Computing Machinery, Jun. 3, 2021, pp. 206–212, ISBN: 9781450384384. DOI: 10.1145/3458336.3465283. [Online]. Available: https://doi.org/10.1145/3458336.3465283.

[5] F. A. Aouda, K. Marquet, and G. Salagnac, "Incremental checkpointing of program state to NVRAM for transiently-powered systems," in *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip, ReCoSoC 2014, Montpellier, France, May 26-28, 2014*, IEEE, 2014, pp. 1–4. DOI: 10.1109/ReCoSoC.2014.6861359.

[6] W. Zhang, S. Shenker, and I. Zhang, "Persistent state machines for recoverable in-memory storage systems with nvram," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, USENIX Association, 2020, pp. 1029–1046. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhang-wen.

[7] V. A. Sartakov and R. Kapitza, "Nv-hypervisor: Hypervisor-based persistence for virtual machines," Atlanta, GA, USA: IEEE, 2014, pp. 654–659, ISBN: 978-1-4799-2233-8. DOI: 10.1109/DSN.2014.64.

[8] S. Jaffer, M. Chitnis, and A. Usgaonkar, "Providing high availability in cloud storage by decreasing virtual machine reboot time," in *10th Workshop on Hot Topics in System Dependability, HotDep '14, Broomfield, CO, USA, October 5, 2014*, F. Junqueira and K. Marzullo, Eds., USENIX Association, 2014. [Online]. Available: https : / / www . usenix . org / conference / hotdep14 / workshop-program/presentation/jaffer.

[9] Google, *Protocol Buffers - Google's data interchange format*. [Online]. Available: https : / / github . com / protocolbuffers/protobuf (visited on 05/30/2023).

[10] Google, *Flatbuffers*. [Online]. Available: https://github.com/google/flatbuffers (visited on 05/31/2023).

[11] K. Varda, *Cap'n Proto serialization/RPC system*. [Online]. Available: https://github.com/capnproto/capnproto (visited on 05/31/2023).

[12] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, 7:1–7:50, 2019. DOI: 10.1145/3295739.

[13] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu, "Skyway: Connecting managed heaps in distributed big data systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18, Williamsburg, VA, USA: Association for Computing Machinery, Mar. 19, 2018, pp. 56–69, ISBN: 9781450349116. DOI: 10 . 1145 / 3173162.3173200. [Online]. Available: https://doi.org/10.1145/3173162.3173200.

[14] K. Taranov, R. Bruno, G. Alonso, and T. Hoefler, "Naos: Serialization-free RDMA networking in java," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, I. Calciu and G. Kuenning, Eds., USENIX Association, 2021, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/taranov.

[15] 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.

[16] Esoteric Software, *Kryo Github Repository*. [Online]. Available: https://github.com/EsotericSoftware/kryo (visited on 06/07/2023).

[17] R. Moeller, *FST Github Repository*. [Online]. Available: https://github.com/RuedigerMoeller/fast-serialization (visited on 06/07/2023).

[18] Oracle, *Java Microbenchmark Harness (JMH) GitHub Repository*. [Online]. Available: https://github.com/openjdk/jmh (visited on 06/01/2023).

[19] M. Zaharia, R. S. Xin, P. Wendell, *et al.*, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016. DOI: 10.1145/2934664.

Chapter 3

InfiniBand in the Context of Java

3.1 High-Performance Networking

Nowadays, both large and small applications are increasingly operated within cloud environments or data centers[34]. These include platforms such as Amazon Web Services (AWS), Google Cloud Platform (GCP) and Oracle Cloud Infrastructure (OCI). All these platforms have one thing in common. They operate thousands of servers running virtualized systems, which need to communicate with each other. The speed of this network communication has a strong influence on the scalability of applications running within a data center[35]. The use of conventional network controllers, as known from the consumer sector, would have a strong negative impact on latency and throughput in such a scenario.

Example
Hundreds of servers are operated in a data center, each running an average of 50 applications. Each of these servers is equipped with a Gigabit Ethernet controller, which is shared between the applications. This results in an average available data rate of 20 Mbit/s (2.5 MB/s) for each application. If one of the applications wants to load larger data records from the Internet, long waiting times occur. Similarly, a network controller from the consumer segment is not designed for such intensive parallel use, which is why the latency inevitably also increases and fast reactions to events are no longer possible.

For the reasons outlined in the previous example, data centers use special hardware that is tailored to the respective workload[36]. While fast Ethernet controllers also exist, they are still connected to the protocol overhead. This is the reason why special network protocols are used within data centers, which are designed for high-performance operations. One of these technologies is *InfiniBand* developed by Mellanox (acquired by NVIDIA in 2020).

3.1.1 InfiniBand Network Transport

Most computer networks at the current time are based on Ethernet technology. Applications that are based on this usually use the Sockets API[37] for communication with other applications. For each message exchange, system calls must first be made, which switch from user space to kernel space and then pass through layers of the Open Systems Interconnection (OSI) model[38]. The protocol stack of the Ethernet protocol is also executed here. Switching between user and kernel space also results in latencies due to context switches - the state or registers of the currently running thread must be saved for later recovery or return from kernel space - which can lead to significant delays in applications based on high-frequency data exchange[39]. Within time-critical applications in which the sending and receiving of messages must not exceed certain latency limits, for example because sensor data is only valid for a certain period of time, this could also lead to data that can no longer be evaluated.

For these reasons, Mellanox began developing high-performance network controllers in the early 2000s, which are used for large workloads in data centers and high-performance computing clusters. This family of network controllers was given the name *InfiniBand*. While bandwidths of around 10 Gbit/s (1.25 GB/s) could be provided at the time of introduction, the latest generations of network controllers, such as the ConnectX-7 from NVIDIA[40], offer bandwidths of up to 400 Gbit/s (50 GB/s). Accordingly, a data transfer comprising one terabyte would theoretically be possible with such a controller within 20 seconds. Another feature of InfiniBand network controllers is the extremely low latency when exchanging small messages of less than one kilobyte. Under normal circumstances, such small messages take less than one microsecond from triggering the send operation to triggering the receive operation on the receiver side.

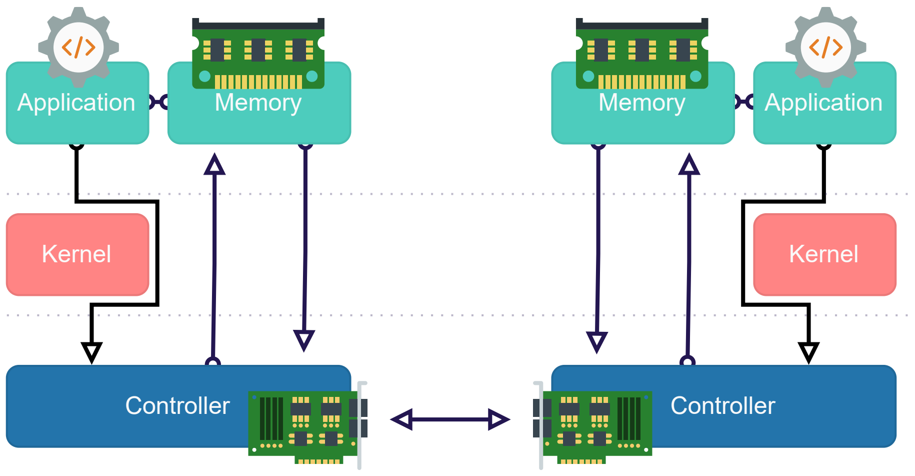


Figure 3.1: Bypassing the kernel in an InfiniBand-enabled application.

These properties are mainly achieved by changing the programming model. Whereas with the Sockets API it is an easy matter to call the `send` function (`<sys/socket.h>`) to send a buffer, InfiniBand offers its own API with far more complex functions. This API is called *Verbs API*[41]. It is implemented in the C programming language and has the biggest difference to the Sockets API in the fact that it can bypass system calls or the kernel and communicates directly with the network controller. This property is illustrated in Figure 3.1, where an application bypasses the kernel to access the controller directly, which in turn is able to access the computer's main memory using DMA. When sending a message using InfiniBand, an application first writes the data to the area of the main memory assigned to it and then informs the network card where this data is located and to which recipient it is to be sent. The InfiniBand network controller then reads the data from the main memory via the Peripheral Component Interconnect Express (*PCIe*)[42] bus and sends it to the network controller of the receiver, whereupon the latter writes the data, also via DMA, to an area in the main memory reserved for this purpose and then informs the application that new data has been received.

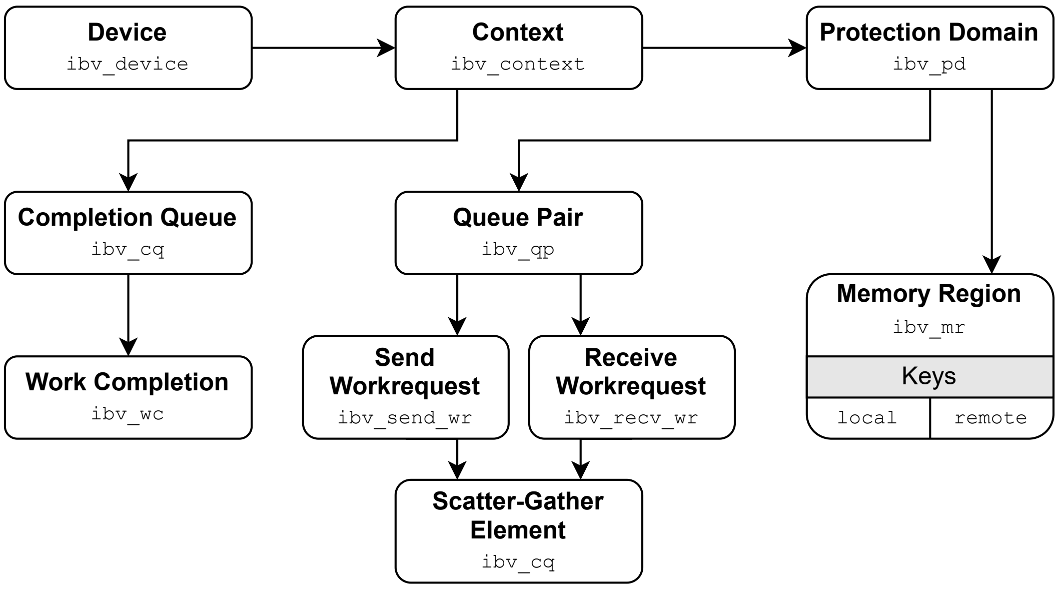


Figure 3.2: Selected components belonging to the Verbs API.

In detail, sending a message using the Verbs API requires interaction with some of its components. An overview of some important components is shown in Figure 3.2. The tasks of the individual components are as follows.

Device - This component bundles the information belonging to the network controller or device and must be determined in the first step using the `ibv_get_device_list` function. In addition to the name of the device, access is also granted here to certain characteristics such as the speeds of the individual network ports.

Context - Before a device's resources can be used, a context must first be created for it using the `ibv_open_device` function. It then bundles all resources and is used to manage them. In abstract terms, this can be compared to a session in which the user must first log in.

Protection Domain - So-called protection domains can be created for each verbs context using the `ibv_alloc_pd`, which are used to prepare memory for use with InfiniBand hardware. Unlike conventional network programming with sockets, memory areas must first be registered with the InfiniBand hardware before they can be used. This is necessary in order to pin the physical memory so that it is not accidentally copied by the operating system between the triggering and execution of an operation.

Memory Region - The memory areas registered with the InfiniBand hardware are called memory regions and can be created using the `ibv_reg_mr` function. In addition to the aforementioned pinning of the memory, two additional keys are created which can be seen as passwords. These keys are the `local key` and the `remote key`, which must be specified for the execution of send, write or read operations.

Queue Pair - While the Ethernet protocol relies on sockets, the Verbs API uses so-called queue pairs. As the name suggests, these queues come in pairs. A queue is created on both the sender and receiver side using the `ibv_create_qp`. Both sides then exchange their connection information - each queue has a `local id`, which is comparable to an IP address - and connect both queues to form a pair.

Send & Receive Workrequest - To instruct the network card to carry out an operation, so-called work requests must be transmitted to it. In addition to the type of operation, these also contain information regarding the virtual memory address and the size of the data to be sent. Work requests are divided into two categories - send and receive - and must also be sent separately to the network card using the functions `ibv_post_send` and `ibv_post_recv`.

Scatter-Gather Element - The Verbs API supports scatter-gather operations. This means that several memory areas that are not necessarily contiguous can be specified within a single workrequest. These are then sent to the recipient as a single unit. All memory areas must be specified within a list of so-called scatter-gather elements and stored within the workrequest. A linked list of the `ibv_sge` struct must be created for this purpose.

Completion Queue - All network-related operations of the Verbs API are executed non-blocking. As soon as a work request has been passed through to the hardware, the corresponding function returns immediately. To find out whether an operation was successful or failed, a so-called completion queue must be created using the `ibv_create_cq` function. Within this queue, the network controller stores elements that describe the status of previously initiated operations.

Work Completion - The elements within a completion queue are called work completions. They indicate whether an operation has been executed successfully and can be retrieved by polling the `ibv_poll_cq` function. After sending a work request to the hardware, the caller receives an identifier, which is also available within a work completion for the purpose of association.

Due to the many components and orchestration required to connect two InfiniBand network controllers, simple programs that could be implemented in just a few lines using the Socket API can take several hundred lines using the Verbs API. Within applications that require the much higher performance characteristics of an InfiniBand network, however, this property is of secondary importance, as the Verbs API has far more powerful functions in addition to simple message sending, which cannot be found within the Sockets API.

3.1.2 Remote Direct Memory Access

Since the InfiniBand protocol stack is implemented directly within the hardware, unlike Ethernet controllers, and therefore no preparation of the data by the kernel is necessary, the InfiniBand technology offers a special functionality called Remote Direct Memory Access (*RDMA*)[43]. Analogous to local DMA operations, this functionality makes it possible to directly write to or read from the remote random access memory of another computer. Newer developments, such as GPUDirect[44], also make it possible to use not only the main memory as a source or target, but also the integrated memory of a graphics card, which benefits distributed machine learning applications in particular. As with simple send operations, the creation of a work request is necessary for the execution of a write or read operation within the Verbs API. An important difference, however, is the use of the previously mentioned keys belonging to memory regions. As direct operations on remote memory entail a certain security risk - for example, confidential data could be read out - access to the memory must be regulated. The `remote key` is used for this purpose. The Verbs API assigns an individual key to each memory region that has been registered with the network controller so that access can be controlled at a fine granular level. Whenever a remote memory region is to be accessed, the accessing party must know the key of the respective region and specify it in the work request so that it can be sent to the target controller together with the operation.

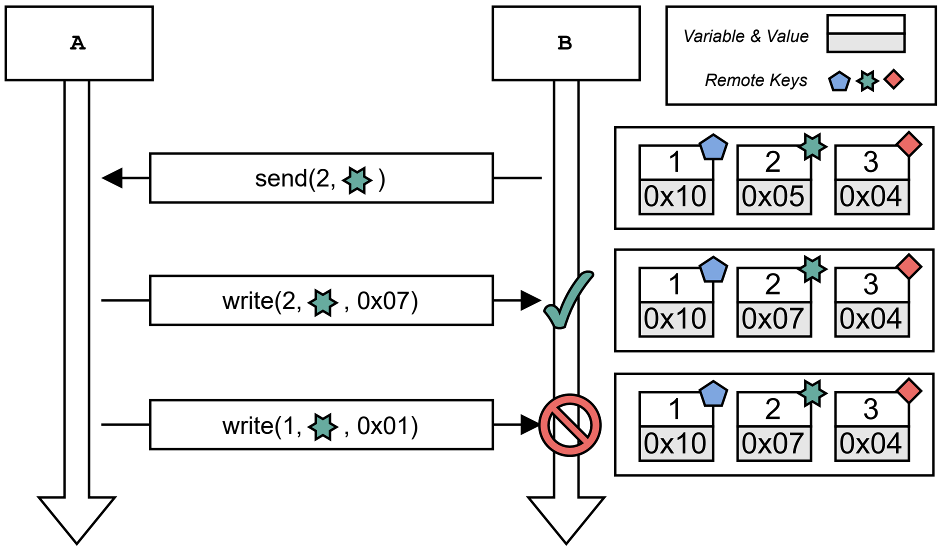


Figure 3.3: Accessing memory regions using a remote key.

As the `remote key` is only known to the program that creates the associated memory region, it must first be transmitted to every network participant who is to be given access to it. The knowledge of this key can be compared to that of a symmetric cryptographic key. As soon as a participant in the network knows it, any arbitrary operation can be carried out on the memory associated with it. An example of an exchange involving two network parties `A` and `B` with a subsequent operation is shown in Figure 3.3. First, `B` sends information about its memory region `2` together with the associated key (shown here as a geometric object) to the network participant `A`. In the next step, `A` performs a write operation on the remote memory region `2` to write the value `0x07`. For the purpose of authenticating the write access, `A` also sends the previously communicated key. The network controller of the party `B` receives the operation together with the key and checks whether the key is valid for the specified memory region. In this case, the keys match, whereupon the write operation is executed and the value `0x07` is now stored within the memory region `2`. In the next step, `A` performs another write operation, but this time the memory region `1` is to be accessed. The request also contains the key of the memory region `2`. When the operation arrives at the network controller of party `B`, it detects that an incorrect key has been specified and cancels the corresponding operation. This ensures that only those applications that have previously been granted permission can access certain memory areas.

Since the CPU of the computer is not involved during the execution of the Verbs operations, the only limitation in terms of performance lies within the characteristics of the network controller, the main memory and the PCIe bus. RDMA operations are therefore particularly suitable for transferring large volumes of data. For example, huge contiguous memory areas in the gigabyte range can be transferred at almost full bandwidth - using a 400 Gbit/s Infiniband controller, this is around 50 GB/s. Another advantage is the predictable performance of the network card. As it works independently of the CPU and is only responsible for sending and receiving messages or operations, a certain performance can be predicted here, depending on the workload in terms of operations, as the controller, unlike the CPU, can work without interruption. In summary, it can be said that the use of RDMA in areas with time-critical or data-heavy applications is a good candidate for network communication.

3.1.3 Java Native Interface

Functionalities at a lower level, such as the targeted invocation of system calls or functions of integrated libraries, cannot be easily accessed in Java. This is due to the safety of the language, as access to such functionality can lead to a program crash if used incorrectly, which cannot be prevented by the JVM. One cause of this can be, for example, access to a memory area that is not assigned to the program or does not exist in its virtual address space. In the event of such access, mechanisms of the operating system take effect and a segmentation fault is triggered. Nevertheless, it is possible, at the expense of safety, to write Java program code that can interact with native compiled programs. The Java Native Interface (*JNI*)[45] integrated in the JVM forms the building block required for this. It allows special methods to be defined on the Java side, which are forwarded to an associated native function when called.

Native.java Java

```java
package de.hhu.bsinfo;

public class Native {
  public static native void hello();
}
```

native.c C

```c
JNIEXPORT void JNICALL Java_de_hhu_bsinfo_Native_hello(JNIEnv* env) {
  printf("Hello World\n");
}
```

Figure 3.4: Interconnecting Java code with native functionalities.

An example of this is shown in Figure 3.4. Here, a class `Native` with an associated native method `hello` is defined within the Java package `de.hhu.bsinfo`. By specifying the `native` keyword, the JVM is instructed to redirect this native method to the function defined in the C source code file `native.c`. A special characteristic here is the structure of the name of the native function. Since a Java function can only be identified by means of its name, package and class, the names of the native functions that are to be called must follow a specific pattern. First of all, each function must begin with the prefix `Java_` to ensure a distinction between functions that are used exclusively in the native part and functions that are called from Java. This is followed by the name of the package, whereby the dots in the package name must be replaced by underscores, as the C programming language does not allow dots within variable and function names. Finally, the class name and the function name follow in order to clearly identify the associated Java function. Within the parameter list of the native function, a variable `env` of the type `JNIEnv` is always transferred as the first parameter. This parameter can be used to access JVM functionalities from the native part of the code. This includes the following functions, for example.

- `AllocObject(JNIEnv *env, jclass clazz)`
  This function can be used to allocate a Java object from the native part of the code. A special detail here is that, as with the Unsafe API, the constructor is not called. This means that the object created is initially uninitialized.

- `NewDirectByteBuffer(JNIEnv* env, void* address, jlong capacity)`
  In contrast to the Java side, the JNI allows instances of the `ByteBuffer` class with a specified virtual memory address to be created in the native part of the code. All that is required is the virtual address and the size of the respective buffer. The resulting `ByteBuffer` instance can then be used within Java.

- `GetDirectBufferAddress(JNIEnv* env, jobject buf)`
  Similarly, it is possible on the native side to query the virtual memory address for a `ByteBuffer` object allocated on the Java side. This is particularly useful if the underlying memory is to be used with the Unsafe API, but the `ByteBuffer` is created by a third-party library.

- `NewGlobalRef(JNIEnv *env, jobject obj)`
  The garbage collector of the JVM has no knowledge of the references used in the native part. If an object reference is saved in the native part for later use, for example within a global variable, the garbage collector would clean up the associated object as soon as it is no longer accessible on the Java side. The native code would therefore have an invalid reference in such a case. To prevent this case, the `NewGlobalRef` function can be used, which creates a long-lived reference

within the native code and thus prevents the garbage collector from cleaning it up until it has been released again using the `DeleteGlobalRef` function.

- `Throw(JNIEnv *env, jthrowable obj)`
  As the programming language does not support the concept of exceptions, the JNI provides a function to propagate exceptions from the native part of the code to the Java part. However, it should be noted that the method signature of the native method should be adapted to the possible throwing of an exception, as otherwise exceptions may not be handled by the Java code and ultimately lead to the program crashing.

Performance Pitfalls

Just like with the Unsafe API functions, care must be taken when using the JNI to ensure that the operations to be executed are called with valid parameters, otherwise there is a risk of the program crashing. In addition, there are some implementation details that are not immediately apparent at first glance, but which can lead to sudden drops in performance[46]. For example, there are functions for accessing primitive arrays, which originate from the Java Space, like `GetIntArrayElements` for retrieving a pointer directed at the raw memory belonging to an integer array. A special aspect here, however, is the type of access. In some cases where the JVM cannot guarantee that the corresponding array within the heap will not be moved by compaction, a copy of the array is created and passed on to the native space. If a method that accesses an array in this way is called very often at short intervals, this results in many copy operations in a very short time, which can take a non-negligible amount of execution time. To avoid such scenarios, the JNI offers functions for direct access to arrays. If a pointer to an array is to be retrieved without triggering a copy operation, the `GetPrimitiveArrayCritical` function can be used. Operations that are executed on the returned pointer should also be completed in a relatively short time, as some mechanisms of the JVM, such as garbage collection, can be stopped until the complementary `ReleasePrimitiveArrayCritical` function is called in order to prevent the memory area within the heap from being moved.

As the aforementioned Verbs API is implemented in the C programming language, the JNI is a good candidate for the integration of InfiniBand functionalities within the Java programming language. Considering the special characteristics associated with the JNI, high-performance networking can thus be provided within the Java ecosystem. This objective is being pursued jointly with a subsequent evaluation in the following two works.

3.2 Neutrino: Efficient InfiniBand Access for Java Applications

---

Filip Krakowski, Fabian Ruhland and Michael Schöttner. Neutrino: Efficient InfiniBand Access for Java Applications. In 19th International Symposium on Parallel and Distributed Computing, ISPDC 2020, Warsaw, Poland, July 5-8, 2020.

Contributions:
As the main developer of the Neutrino project, the author pursued the goal of providing access to InfiniBand hardware within Java applications. First, the author looked at existing solutions, such as jVerbs or DiSNI, and came to the conclusion that each of the alternatives considered has certain disadvantages. Based on these findings, the author developed an alternative that is easy to use and also causes little overhead in terms of performance.

The basis and one of the main contributions of the project is the possibility to connect Java classes with native structs within C code. This function was largely implemented by the author and then refined in collaboration with Fabian Ruhland. The required manual mapping between Java classes and native structs was carried out in collaboration with Fabian Ruhland, whereby the author later developed a tool that automates this step. The framework's architecture and its main components were developed by the author. This includes a further main contribution in the form of the processing of requests. Here the author developed a highly efficient mechanism based on epoll, which is able to saturate the maximum throughput of the network card using small messages.

Finally, the author developed distributed benchmarks to evaluate the performance of the developed solution. Michael Schöttner and Fabian Ruhland were involved in this process by evaluating the results and providing suggestions regarding the cause of certain behaviors. The textual part of this work was written by the author, while Michael Schöttner and Fabian Ruhland were involved in the form of proofreading and various discussions.

Status: published

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.

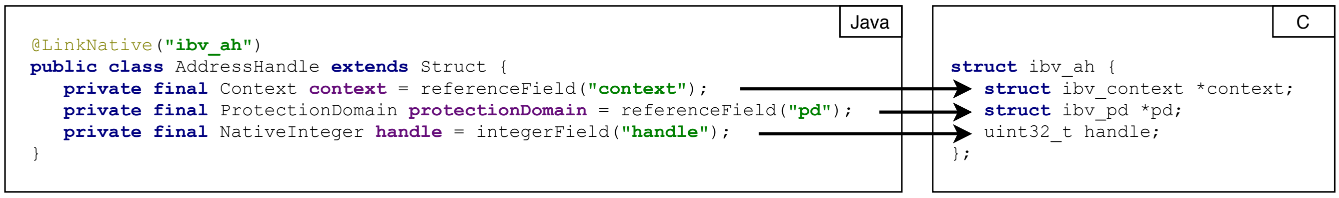
Ibdxnet is an InfiniBand message passing transport we developed in the past for *DXNet* [8] both for distributed and parallel Java applications. While our previous efforts are based on transparent serialization of messaging objects we are now developing the successor *neutrino* aiming at providing RDMA 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



```java
@LinkNative("ibv_ah")
public class AddressHandle extends Struct {
   private final Context context = referenceField("context");
   private final ProtectionDomain protectionDomain = referenceField("pd");
   private final NativeInteger handle = integerField("handle");
}
```

```c
struct ibv_ah {
   struct ibv_context *context;
   struct ibv_pd *pd;
   uint32_t handle;
};
```

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.

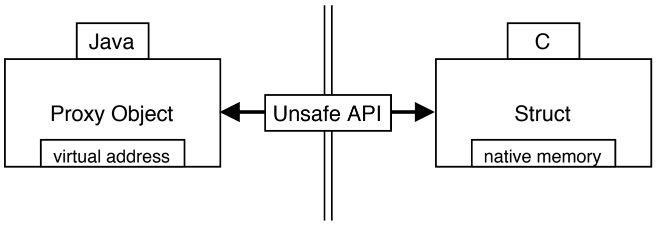
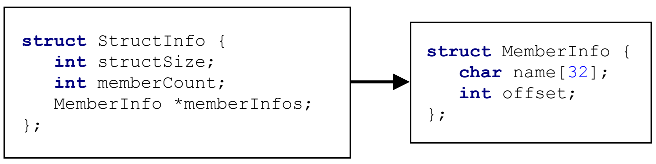


Fig. 2. Components of the structured native memory access.

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.

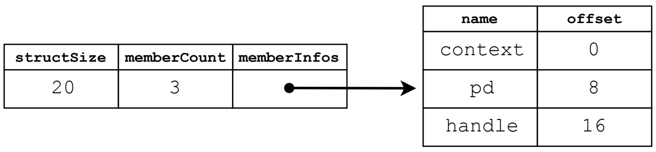


```c
struct StructInfo {
   int structSize;
   int memberCount;
   MemberInfo *memberInfos;
};
```

```c
struct MemberInfo {
   char name[32];
   int offset;
};
```

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.

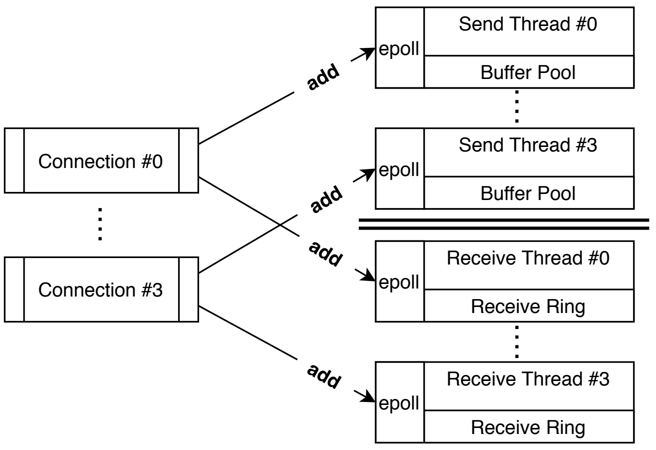


Fig. 5. Adding individual connections to sender and receiver threads.

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.

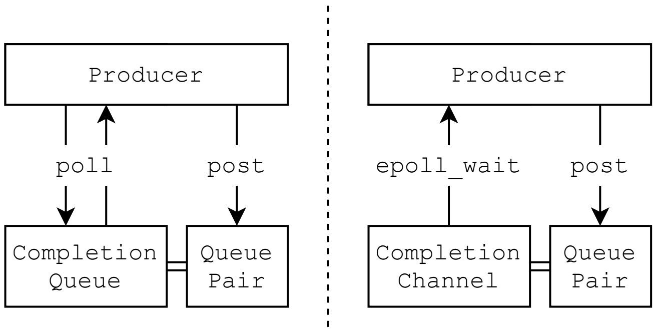


Fig. 6. Continuous polling of the completion queue (left) and notification based waiting on completions (right).

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.

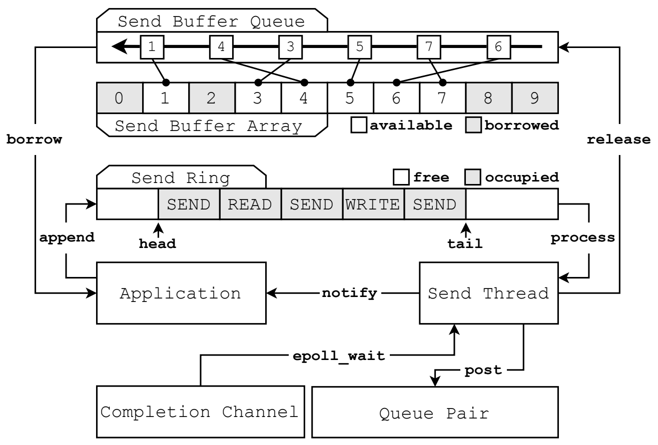


Fig. 7. Processing of outgoing operations using a ring buffer for requests and a queue of buffers for writing messages.

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.

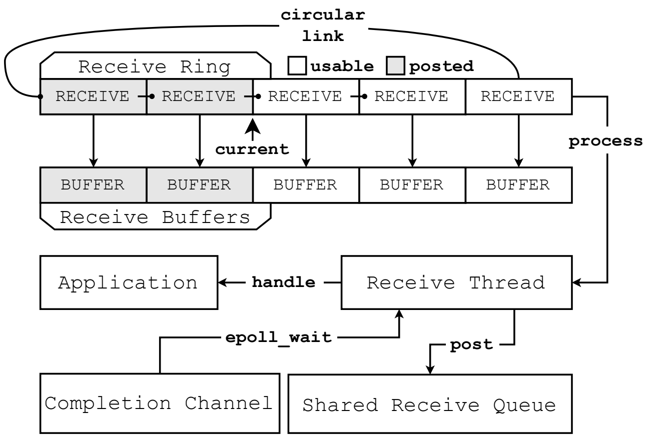


Fig. 8. Processing of incoming messages using a circular linked list of pre-allocated work requests.

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.

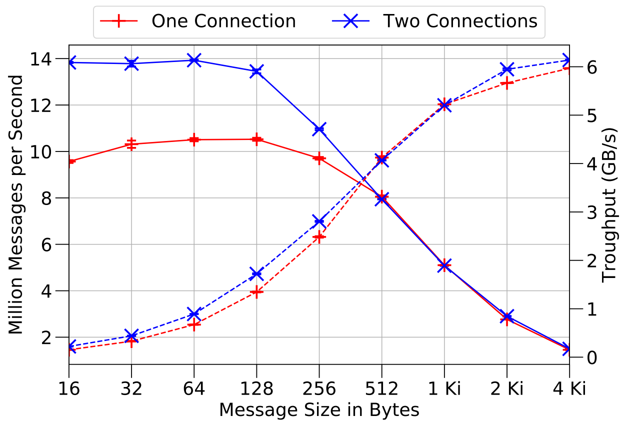


Fig. 10. Average message (solid line) and network (dashed line) throughput by message sizes using one and two connections.

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

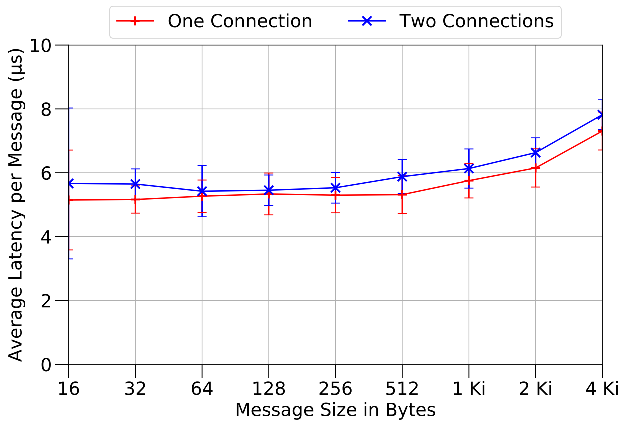


Fig. 11. Average latency by message sizes using one and two connections.

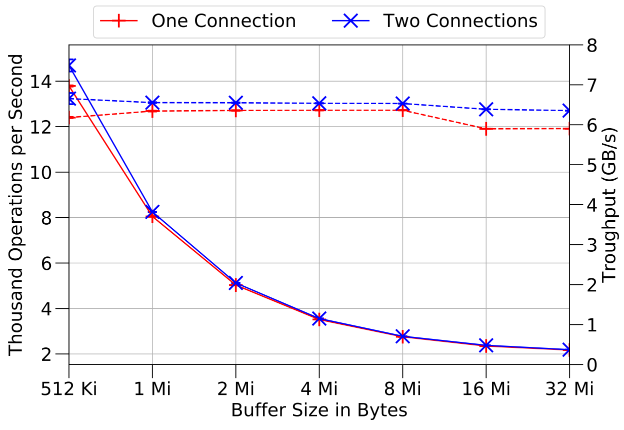


Fig. 12. Average read operation (solid line) and network (dashed line) throughput by buffer sizes using one and two connections.

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.

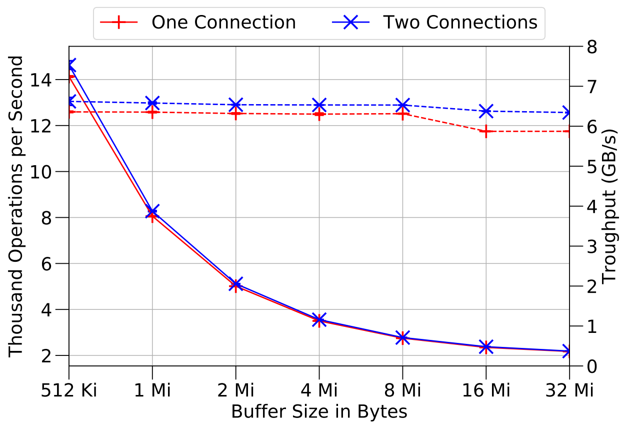


Fig. 13. Average write operation (solid line) and network (dashed line) throughput by buffer sizes using one and two connections.

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. (2018). "Direct Storage and Networking Interface (DiSNI)," [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. (2019). "Introducing apache arrow flight: A framework for fast data transport," [Online]. Available: https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight (visited on 04/19/2020).

3.3 Performance analysis and evaluation of Java-based InfiniBand Solutions

Fabian Ruhland, *Filip Krakowski*, and Michael Schöttner. Performance analysis and evaluation of Java-based InfiniBand Solutions. In 19th International Symposium on Parallel and Distributed Computing, ISPDC 2020, Warsaw, Poland, July 5-8, 2020.

Contributions:
In this work, various InfiniBand solutions - both native and those usable in Java - were compared with each other. For this purpose, Fabian Ruhland developed a benchmark suite called Observatory, which can use various backends by means of interfacing.

One of the author's contributions is the Neutrino backend, which was implemented by the author as the main developer and adapted based on the feedback received within the Observatory project. Fabian Ruhland developed the benchmarking framework, carried out the associated benchmarks and finally analyzed the data. During this phase, the author and Michael Schöttner were involved in discussions regarding the results and any discrepancies within the measurements or their causes. Finally, the author was available to answer questions on the use of the Neutrino project.

The paper was written by Fabian Ruhland, while the author and Michael Schöttner were involved in several discussions regarding the design and implementation of the Observatory benchmark framework and provided valuable input.

Status: published

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.

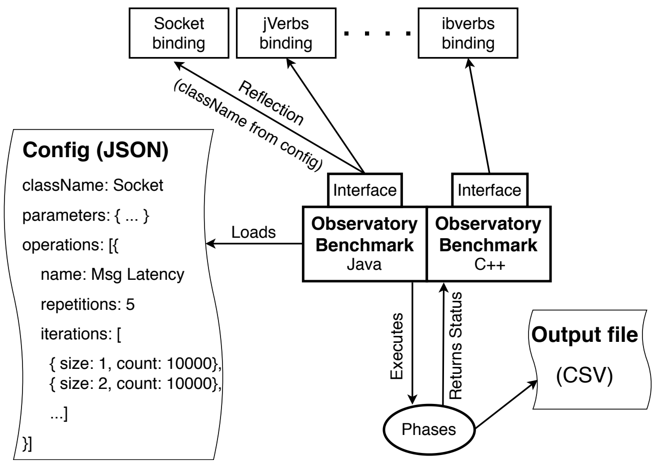


Fig. 1: Observatory architecture design

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.

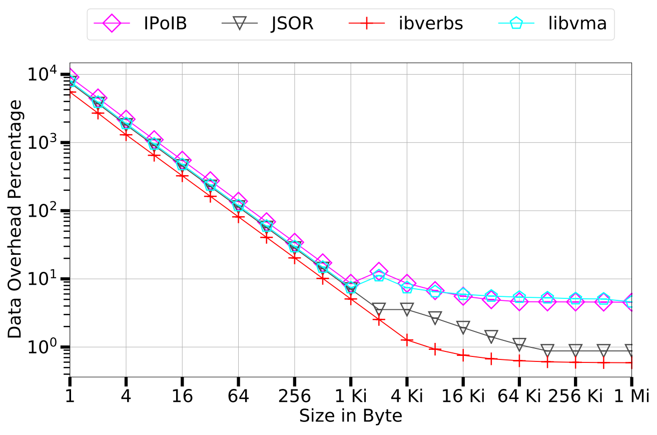


Fig. 2: Avg. overhead (%) in relation to the payload size.

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%.

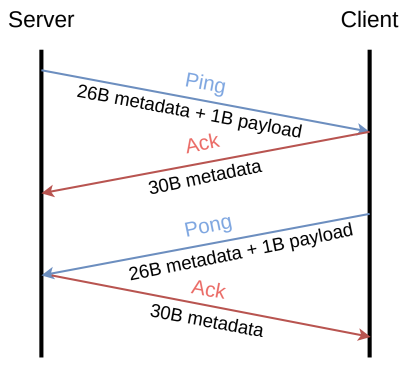


Fig. 3: Verbs-based ping-pong communication pattern.

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-

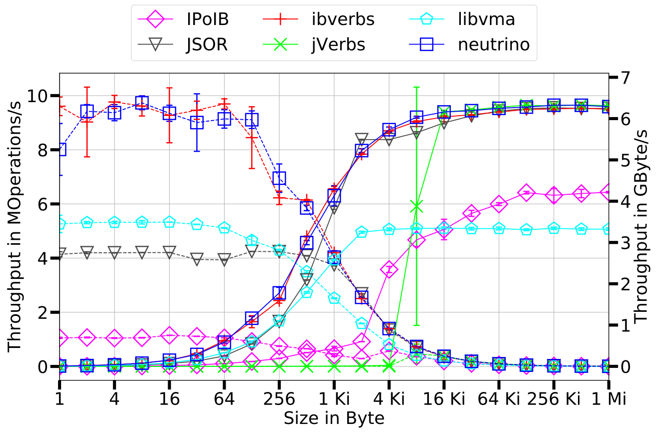


Fig. 4: Unidir. throughput (msg), increasing message size,

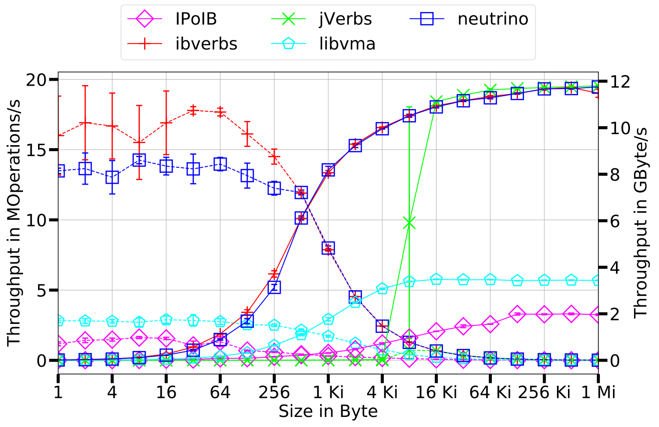


Fig. 6: Bidir. throughput (msg), increasing message size.

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.

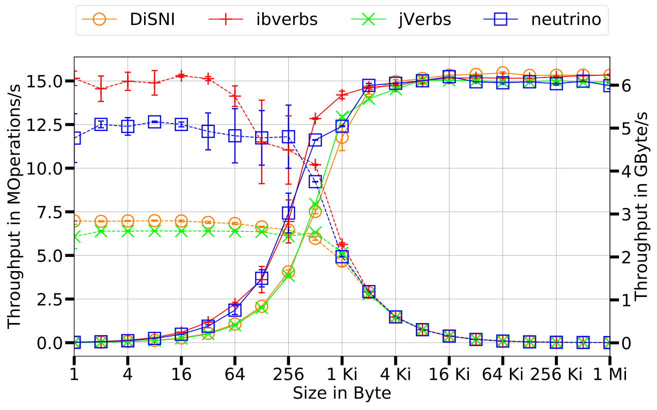


Fig. 5: Unidir. throughput (write), increasing buffer size.

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

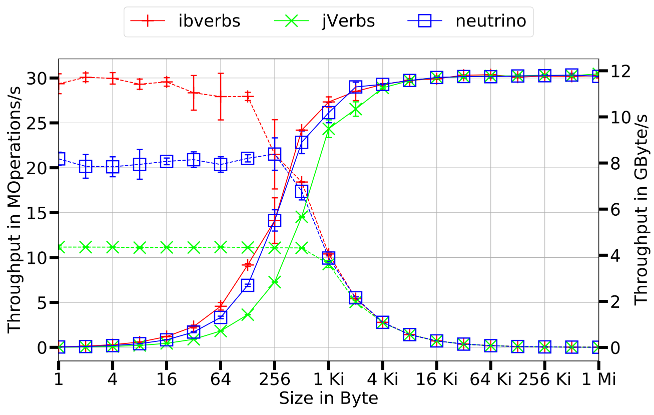


Fig. 7: Bidir. throughput (write), increasing buffer size.

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.

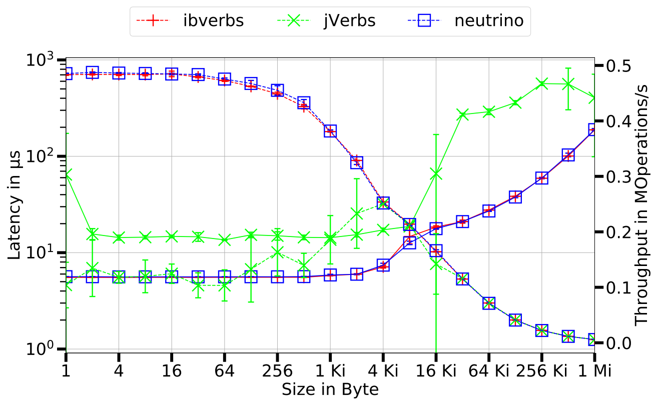


Fig. 8: 99.99% latency (msg), increasing message size.

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.

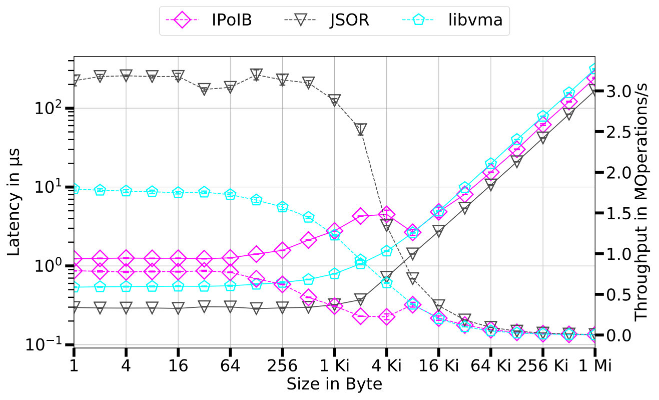


Fig. 9: Avg. latency (msg), increasing message size.

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).

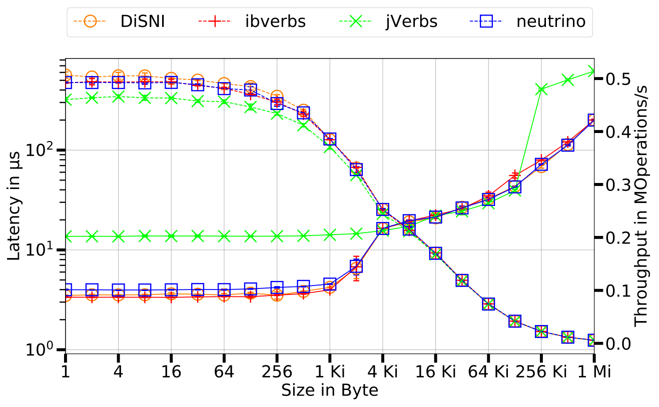


Fig. 10: 99.99% latency (write), increasing message size.

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.

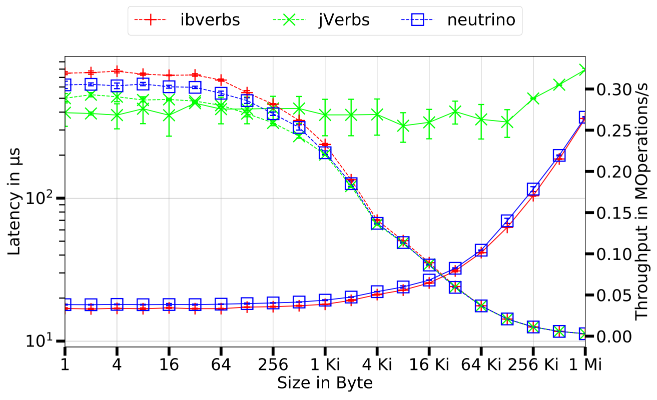


Fig. 11: 99.99% ping-pong latency, increasing message size.

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.

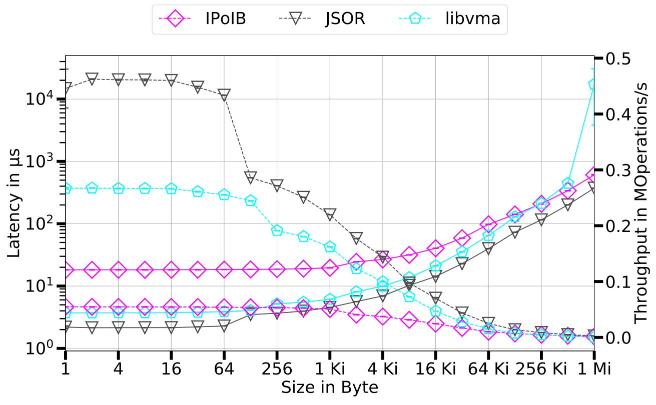


Fig. 12: Avg ping-pong latency, increasing message size.

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 4

Modern Foreign Function & Memory Access in Java

4.1 Foreign Function & Memory API

The use of the Unsafe API is explicitly not recommended by Java developers due to the lack of guarantees regarding compatibility and availability - it is still an internal API, which can change between individual Java versions or even disappear completely. A good example of this is the `defineClass` method, which could be used to load arbitrary classes at runtime using a byte array. This method was initially marked as deprecated within the Unsafe API with Java version 9 and then completely removed with the release of Java version 11[47]. Projects and libraries that relied on this function therefore had to switch to alternative methods of defining classes at runtime. As the JDK developers recognized a greater demand for functionalities that can access native areas of the system, the development of *Project Panama* began in 2014.

> *"We are improving and enriching the connections between the Java virtual machine and well-defined but "foreign" (non-Java) APIs, including many interfaces commonly used by C programmers."*
>
> \- Oracle, 2014 [15]

As explained in the previous quote, this project aims to significantly improve the interaction between Java and native code. One of the main components involved in this is the *Foreign Function & Memory API* first introduced in JDK Enhancement Proposal 412[48]. The goal here is to gradually transfer the functionalities provided by the Unsafe API into more superior abstractions that carry some safety guarantees. Furthermore, another objective is to significantly reduce the effort required to call a native function from the Java space by dropping the requirement for a layer of glue-code. Instead, it relies on tools that can generate the code required to control native functions. One of these tools is *jextract*[49], which accepts header files belonging to C libraries and generates Java classes from them, which map the data structures and functions available within the library. It supports the generation of readable source code as well as precompiled class files. As this tool was increasingly used in this work, a plugin for the Gradle build system was also developed, which integrates jextract into the build process. As this tool was increasingly used for a project developed within this work, a plugin for the Gradle build system was developed - also in the context of this work - which integrates jextract into the build process[50]. This plugin allows the specification of data structures and functions of a header file within the build configuration, whereupon the corresponding classes are automatically available within the project - i.e. without manually calling the jextract tool.

build.gradle   Groovy

```groovy
1 jextract {
2   header("${project.projectDir}/src/main/c/stdio.h") {
3     libraries = [ "stdc++" ]
4     targetPackage = "org.unix"
5     className = "Linux"
6     functions = [ "printf" ]
7   }
8 }
```

Figure 4.1: Using the gradle-jextract plugin to access native functions.

The code snippet in figure 4.1 shows an example configuration of the plugin. The plugin is instructed to make the `printf` function of the standard library available in the Java code using the jextract tool by means of the following steps within the following lines.

② The header file to be evaluated for extraction is specified in the first step using the `header` method.

③ The `libraries` array contains the names of the shared libraries required at runtime. These are loaded automatically before the generated methods are used.

④ The `targetPackage` property specifies the package in which the generated code is to be placed.

⑤ The `className` property specifies the name of the class in which the generated functions are to be defined.

⑥ Finally, the `functions` array is used to specify which functions are to be extracted from the header file. These functions are then available within the specified class.

In addition to individual functions, it is also possible to extract structs, constants, type definitions, unions and individual variables from the header file using the corresponding configuration options of the plugin. However, one restriction should be noted here, as otherwise unexpected errors may occur during program execution. Functions that are defined within the C source code using the `inline` keyword are inserted in their entirety at compile time at the locations where they are called, so that no real function call takes place. In other words, there is no function with a corresponding function name in the compiled code or within the shared library. As the Foreign Function & Memory API is based on a dynamic lookup within the shared library using the function's name at runtime, such a function cannot be found or called. In such a case, the corresponding lookup simply returns `null`, which can lead to confusion during programming[51].

```
Linux.java                                                          Java
 1 static final AddressLayout POINTER = ValueLayout.ADDRESS.withTargetLayout(
 2   MemoryLayout.sequenceLayout(JAVA_BYTE)
 3 );
 4
 5 static final FunctionDescriptor PRINTF_FD = FunctionDescriptor.of(
 6   ValueLayout.JAVA_INT,
 7   POINTER
 8 );
 9
10 static final MethodHandle PRINTF_MH = RuntimeHelper.downcallHandleVariadic(
11   "printf",
12   PRINTF_FD
13 );
14
15 public static int printf(MemorySegment __format, Object... x1) {
16   try {
17     return (int)PRINTF_MH.invokeExact(__format, x1);
18   } catch (Throwable ex) {
19     throw new AssertionError("should not reach here", ex);
20   }
21 }
```

Figure 4.2: Code generated by jextract for calling the native printf function.

A simplified form of the generated code of the jextract tool for the `printf` function is shown within the code snippet in Figure 4.2. To ensure that Java can check at runtime whether the function arguments specified during the call are of the correct type, a `FunctionDescriptor` must first be specified (lines ⑤ to ⑧). Within this, the native function's signature is specified. The first argument specifies the type of the return value. The subsequent arguments then specify the types of the parameters. As the return value of the native `printf` function is an integer value - namely the number of characters written or output - the first argument is set to `JAVA_INT`, i.e. an integer. The following arguments are pointers to arrays of bytes, i.e. strings, which is why the following parameter is set to `POINTER` (a layout previously defined in lines 1 to 3). In order to obtain a `MethodHandle`, a class that can forward calls to native functions, the `downcallHandleVariadic` method of the helper class `RuntimeHelper` generated by jextract must be called (line 10). Here, the name of the function, as defined within the shared library, must be specified as the first parameter. This is used to determine the entry address of the function within the shared library. The second parameter is the previously created `FunctionDescriptor` with which the `MethodHandle` can check the parameter types when called and generate an exception in the event of an error, which can be handled by the program code. Finally, a helper function is generated in lines 15 to 21, with which the `printf` function can be called as if it was called within C source code. A notable advantage here is that not a single line of C code needs to be written; all definitions take place within Java. Accordingly, no additional code needs to be compiled for existing shared libraries, as is the case with JNI.

NativeHelloWorld.java Java

```java
 1 import java.lang.foreign.Arena;
 2 import static org.unix.Linux.printf;
 3
 4 public final class NativeHelloWorld {
 5
 6   public static void main(String... args) {
 7     try (var arena = Arena.openConfined()) {
 8       var format = arena.allocateUtf8String("Hello %s");
 9       var value = arena.allocateUtf8String("World");
10
11       printf(format, value.address());
12     }
13   }
14 }
```

Figure 4.3: Calling the native printf function from Java code using generated bindings.

The generated code can then be called using a simple import within Java. This is demonstrated in an example in Figure 4.3. First, a `Arena` is opened in line 7, which represents an abstraction within the Foreign Function & Memory API for providing off-heap memory and provides some auxiliary functions for interacting with native functions. One of these functions is `allocateUtf8String` (used in lines 8 and 9), which converts a Java string into a memory buffer that can be passed on to native code. This is necessary because the standard `String` class uses UTF-16 encoding for storing its data within the Java programming language[52]. As soon as the parameters for the function call have been prepared, the `printf` function can be called (line 11). Visually, the function call barely differs from the notation that would be used within C code. Code that was previously written in the C programming language can therefore be transferred to Java very easily in this way. Another advantage is the Oracle developers' goal of making the Foreign Function & Memory API superior to JNI in terms of performance and ease of use. Benchmarks already show that the API can work faster than JNI[53]. The aforementioned advantages can be utilized to create an easier to maintain, usable and more efficient connection between InfiniBand hardware and the Java programming language.

4.2 Unified Communication X

The *Neutrino* project developed in 3.2 already provides an abstraction layer for working with InfiniBand hardware in Java, which is based on the JNI and the Unsafe API. It also uses a similar mechanism like the jextract tool of the Foreign Function & Memory API to determine metadata associated with data structures, such as field offsets and sizes. One difference here, however, is that the metadata is statically integrated within the natively written code and any change to the Verbs API therefore requires this metadata to be modified and the JNI glue-layer to be recompiled. In the long term, this project requires a great deal of maintenance, which can be easily circumvented using the Foreign Function & Memory API. As mentioned in subsection 3.1.1, the Verbs API can be used for communication and control of InfiniBand hardware. However, as this API is based on a very low level and requires many configuration steps, it can take some time before a fully functional abstraction for using the hardware is developed and can be used. For this reason, the Unified Communication X (UCX)[16] project was established by the Unified Communication Framework Consortium. It provides a high-level framework for the use of high-speed interconnect hardware and aims to make it easy to use. This framework consists of a total of three different layers, which can be used as needed.

- `ucs` / Unified Communication Services
  This layer contains shared functionalities that can be used by the other layers and also the program layer. These include, for example, functions for managing memory or the use of non-blocking event queues.

- `ucp` / Unified Communication Protocols
  All high-level functions for using high-speed interconnect hardware are provided within this layer. In contrast to the Verbs API, barely any configuration steps are necessary and the functions are immediately ready for use.

- `uct` / Unified Communication Transport
  However, if precise control of the functionality is required, the transport layer

of the framework can be used, which provides low-level functions. These can be compared with the functions of the Verbs API, but still offer a certain degree of abstraction.

The high-level layer of the API, the `ucp` layer, builds on top of the `uct` layer. It bundles several low-level function calls within a single high-level function call in order to minimize the load on the programmer that would otherwise be caused by the configuration or orchestration of an operation.

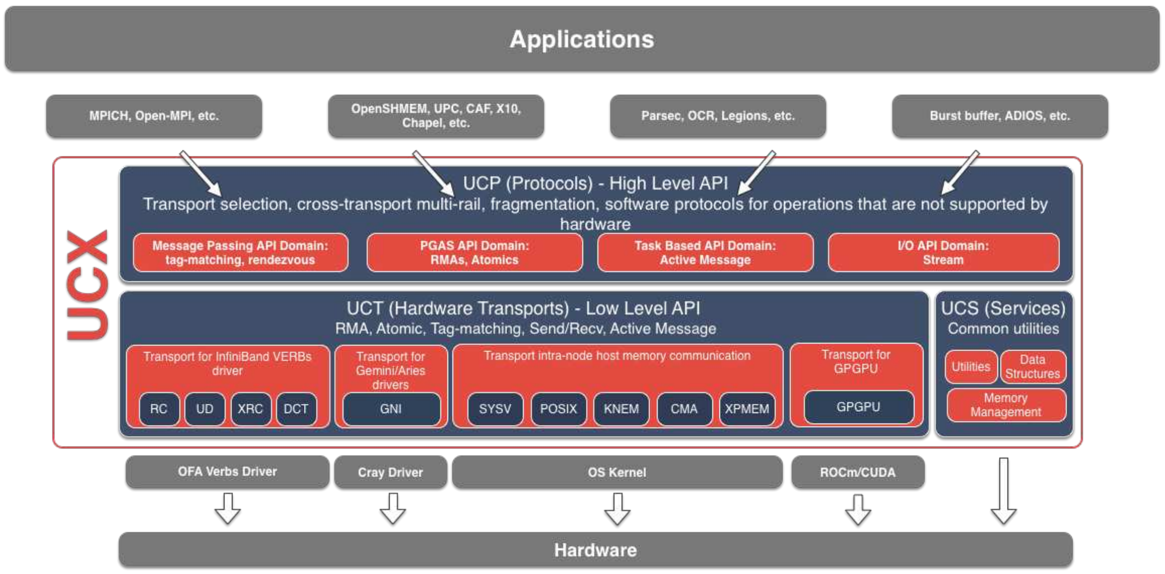

Applications
MPICH, Open-MPI, etc.
OpenSHMEM, UPC, CAF, X10, Chapel, etc.
Parsec, OCR, Legions, etc.
Burst buffer, ADIOS, etc.
UCX
UCP (Protocols) - High Level API
Transport selection, cross-transport multi-rail, fragmentation, software protocols for operations that are not supported by hardware
Message Passing API Domain: tag-matching, rendezvous
PGAS API Domain: RMAs, Atomics
Task Based API Domain: Active Message
I/O API Domain: Stream
UCT (Hardware Transports) - Low Level API
RMA, Atomic, Tag-matching, Send/Recv, Active Message
Transport for InfiniBand VERBs driver
RC
UD
XRC
DCT
Transport for Gemini/Aries drivers
GNI
Transport intra-node host memory communication
SYSV
POSIX
KNEM
CMA
XPMEM
Transport for GPGPU
GPGPU
UCS (Services)
Common utilities
Utilities
Data Structures
Memory Management
OFA Verbs Driver
Cray Driver
OS Kernel
ROCm/CUDA
Hardware

Figure 4.4: The Unified Communication X framework's architecture[54].

The entire architecture of the framework is shown within Figure 4.4. A unique feature of the framework is that it provides support for other high-speed interconnect hardware besides InfiniBand such as Cray's Gemini/Aries network controller. It also offers the option of using the integrated memory within graphics cards for data transfers using the CUDA framework (NVIDIA) or the ROCm framework (AMD). However, one of the most powerful features of the framework is the way in which it is configured. Using the high-level API (`ucp`), the framework scans the hardware of the executing computer and determines optimal parameters for the configuration based on this information. Configuration by the program is therefore not strictly necessary and may be a disadvantage in some cases where the effects of certain configuration options are not fully known. In the case of automatic configuration using multiple Network Interface Controllers (NICs), the framework also supports a feature called *Multi-Rail Support*[55]. In this context, several NICs are used in parallel during a data transmission. For this purpose, the framework uses the information determined belonging to the NICs in order to achieve an optimal distribution of the data to the individual NICs. This means that larger data transmissions can be split over several physical connections and the available bandwidth can be optimally utilized. This mechanism is described in more detail in the following example.

> **Example**
> A computer has three network controllers - a 400 Gbit/s InfiniBand controller and two 10 Gbit/s Ethernet controllers. A data transfer is initiated using the UCX framework. As the framework has previously scanned the configuration of the computer and knows about the available NICs, it can distribute the data according to the speeds of the NICs and send it via each of them. In total, this results in a theoretically usable data transmission bandwidth of 420 Gbit/s instead of the 400 Gbit/s available when only using the InfiniBand card.

The use of several network controllers is particularly advantageous when there is an increased amount of large data transfers, as the bandwidths of the individual physical connections can be optimally utilized and latency can be neglected.

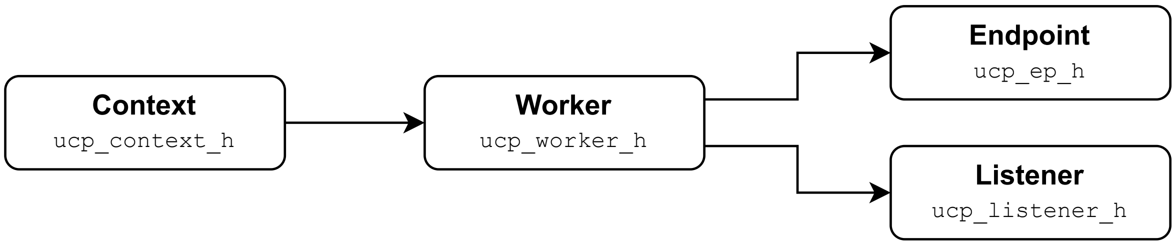


Figure 4.5: Basic UCX components required for establishing a network connection.

Unlike the Verbs API, a small number of components are required for network communication using the UCX framework. These are displayed within Figure 4.5 together with the names of their associated type definitions. The components shown each have the following tasks.

- **Context** - As with the Verbs API, a context must first be created using the UCX framework, which is used to manage the resources created with it. This context is initialized using the `ucp_init` function.
- **Worker** - This abstraction can be compared to an event loop. The worker contains logic for triggering send operations and for querying the statuses associated with the executed operations. It should be noted here that a run of the logic must be triggered manually by a call in the program code, i.e. the worker itself does not provide any threading functionalities and a mechanism for continuous triggering must be developed itself. The worker is created using the `ucp_worker_create` function, which accepts the context as a parameter.

- **Endpoint** - Within the UCX framework, an endpoint can be compared to a socket of the Sockets API. Individual messages, streams or even RDMA operations can be executed using the functions that can be called on it. The multi-rail feature mentioned above is supported within an endpoint, so that an endpoint can also be assigned to several physical connections. An endpoint is created using the `ucp_create_ep` function, specifying the parent worker.

- **Listener** - In order for a server to accept connections, a listener must be created within the UCX framework, which acts similarly to a server socket of the Sockets API. Together with it, a callback is defined, which is called for connection requests and can therefore decide in fine granularity whether a connection should be accepted, established or rejected. A listener is created using the `ucp_listener_create` function and specifying the worker.

Unlike the Verbs API, connections can also be established by specifying an IP address and a port, provided the computers involved are connected within an IP network. For this purpose, the UCX Framework exchanges the connection information of the InfiniBand connection out-of-bounds via an IP connection so that the computers can then connect via InfiniBand. Since the use of a relatively small API layer leads to a significantly reduced effort within the maintenance caused by changes, the UCX framework in combination with Java's Foreign Function & Memory API is ideally suited for the development of an easily maintainable integration of InfiniBand hardware within the Java ecosystem. This objective is pursued in the following project and later compared with other frameworks and libraries in an evaluation.

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

Filip Krakowski, Fabian Ruhland and Michael Schöttner. Infinileap: Modern High-Performance Networking for Distributed Java Applications based on RDMA. In 27th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2021, Beijing, China, December 14-16, 2021.

Contributions:
The Infinileap project is based on the findings of the Neutrino project and relies on new Java technologies that improve access to native functionalities as well as on the OpenUCX Project, which abstracts access to InfiniBand hardware. It was mainly implemented by the author, while Fabian Ruhland contributed some bug fixes.

First, the author implemented the new access layer using the new Foreign Function & Memory API. As the components used were still in incubator status at this stage, there was also regular communication with project leaders at Oracle, which also involved Michael Schöttner and Fabian Ruhland in a number of video conferences. For the same reason, the author had to regularly adapt the implemented solution to the changes in the JDK and made contributions to the OpenJDK project in the form of bug fixes. The author also uncovered a number of issues in the OpenUCX project, which were subsequently discussed with the respective developers. The main contribution of this work is the development of an abstraction layer based on the OpenUCX library and some auxiliary functions in Java, which makes access to InfiniBand hardware significantly easier.

Furthermore, the author has evaluated the implemented solution in specially developed benchmarks, which have extended the JMH framework with a distributed mode. Another main contribution are the insights gained from this. These showed the Oracle developers how the use of the Foreign Function & Memory API in real applications affects performance. The paper was written by the author, while Michael Schöttner contributed some suggestions for improvement and did the proof reading. In addition, Michael Schöttner and Fabian Ruhland provided valuable advice in many discussions.

Status: published

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.

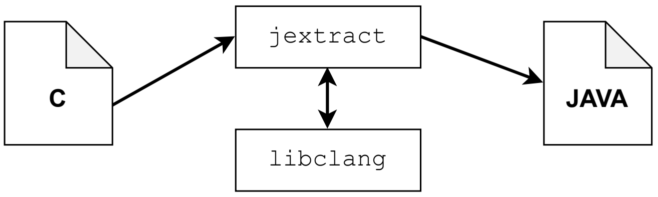


Fig. 1. Jextract's process of generating native bindings.

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.

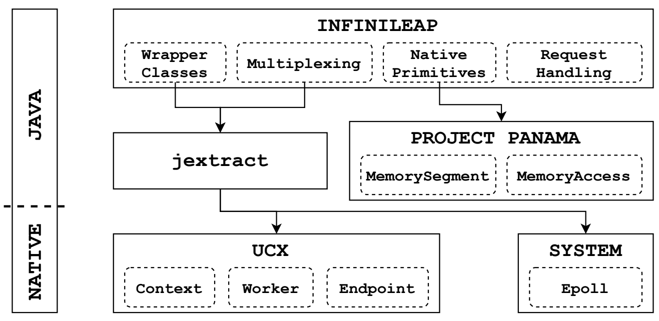


Fig. 2. Infinileap's architecture and dependencies.

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. Buffers are an exception to this rule, since the associated addresses must be freely selectable by the user. RDMA operations can thus be performed either on instances derived from the `NativeObject` class or directly on an instance of Project Panama's `MemorySegment` class.
- **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.

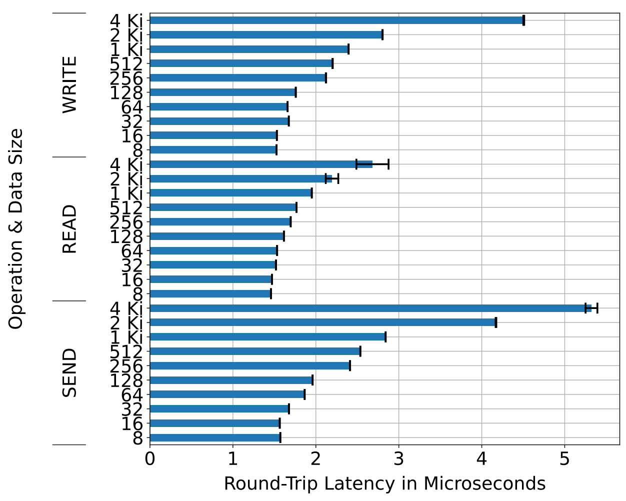


Fig. 5. Average Round-trip latency for RDMA write, RDMA read and send operations measured in microseconds per operation.

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

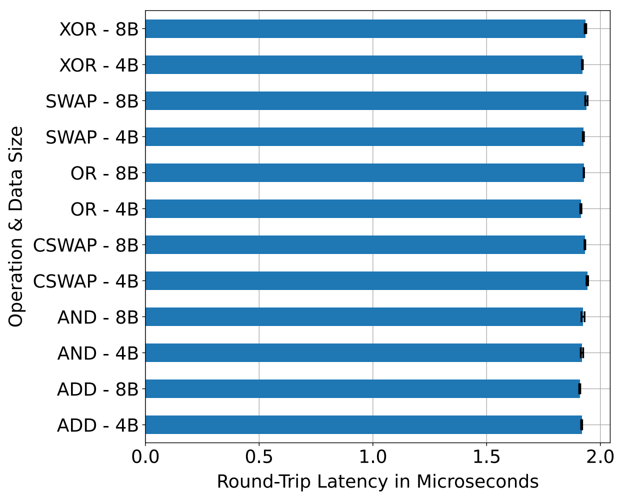


Fig. 6. Average operation latency for all UCX-supported atomic operations and data sizes.

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.

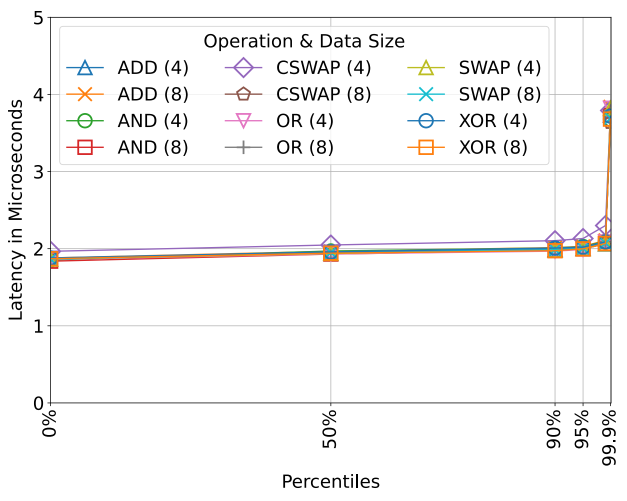


Fig. 7. Operation latency for all supported atomic operations by percentiles.

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.

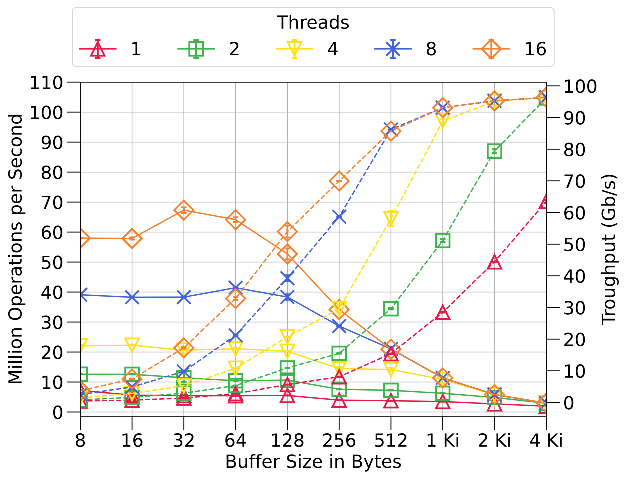


Fig. 8. Average RDMA write operation throughput in million operations (solid line) and gigabit per second (dashed line).

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.

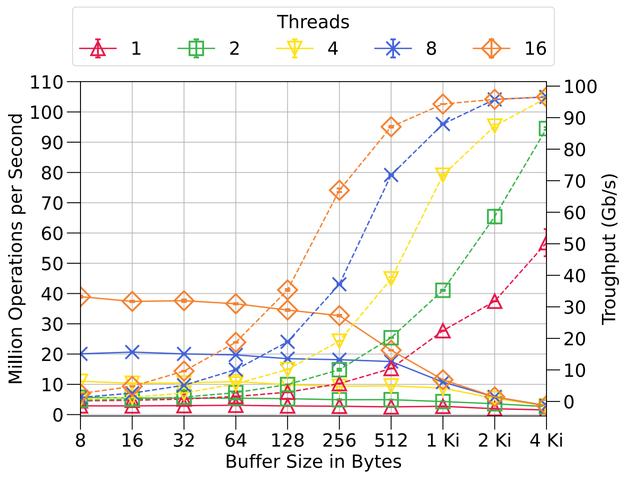


Fig. 9. Average RDMA read operation throughput in million operations (solid line) and gigabit per second (dashed line).

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

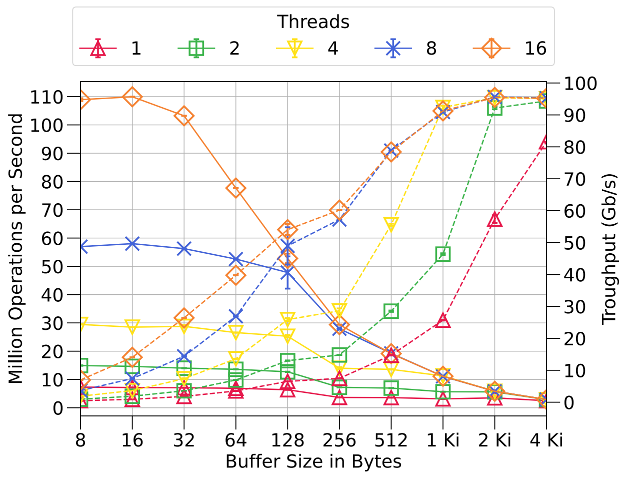


Fig. 10. Average send operation throughput in million messages (solid line) and gigabit per second (dashed line).

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, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "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. (Jul. 7, 2009). "Best practices for using the java native interface," [Online]. Available: https://developer.ibm.com/articles/j-jni.

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

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

[20] D. Smith. (Aug. 13, 2020). "Jep 401: Primitive objects (preview)," [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.

4.4 Transparent network acceleration for big data computing in Java

Fabian Ruhland, *Filip Krakowski* and Michael Schöttner. Transparent network acceleration for big data computing in Java. In 2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), TrustCom 2023, Exeter, United Kingdom, November 01-03, 2023.

Contributions:
This work aims to transparently accelerate Java applications by extending Java's NIO module with InfiniBand functionality. To this end, Fabian Ruhland developed a connection layer called hadroNIO between Java's NIO components and two InfiniBand Java solutions using the Service Provider Interface.

One of the author's contributions is the Infinileap framework, which enables access to InfiniBand hardware within Java applications. Alongside another externally developed framework, this was used by Fabian Ruhland to realize an asynchronous socket channel implementation based on InfiniBand. Another contribution of the author is the adaptation of the implementation of a ring-buffer data structure of the Agrona project. This was modified by the author so that the publication of data within the ring buffer can be carried out in two steps, consisting of a reservation and the subsequent write process. The final benchmarks were implemented and evaluated by Fabian Ruhland, while the author and Michael Schöttner were involved in discussions regarding the results and possible improvements.

The author and Michael Schöttner also provided other valuable suggestions, such as optimizing the epoll-based event loop with regard to a stage-based wake-up mechanism. The paper was written by Fabian Ruhland, whereby the author and Michael Schöttner were involved in several discussions regarding various aspects of the library's functionality.

Status: published

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.

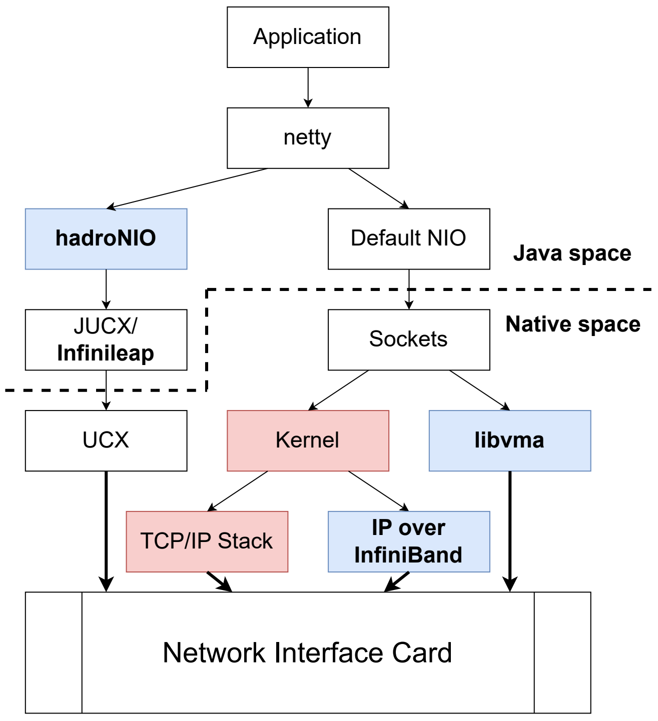


Fig. 1. Application stack overview for different acceleration solutions

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.

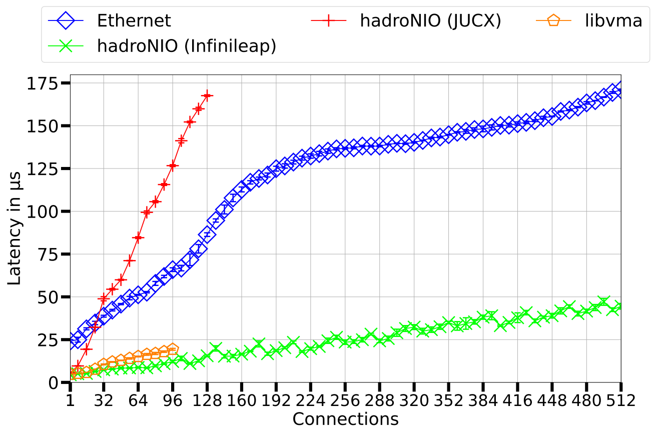


Fig. 3. Netty microbenchmark round-trip times with 16 byte messages

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.

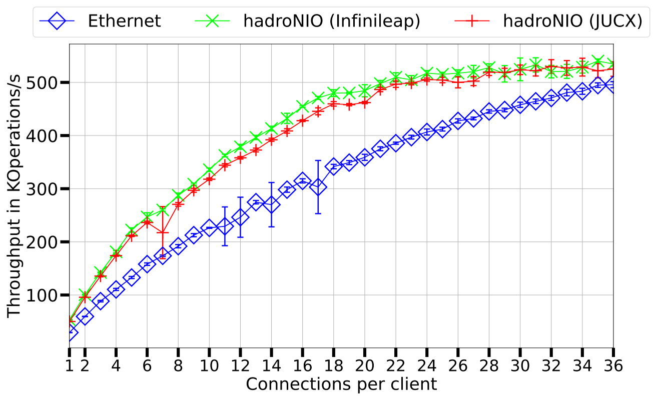


Fig. 4. gRPC key-value store request times 3 clients and 1x16 byte records

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

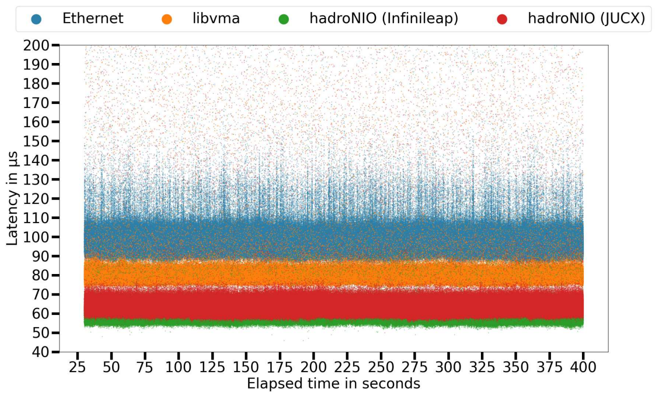


Fig. 5. gRPC key-value store request times with 3 clients ()each using 3 threads) and 1x16 byte records

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.

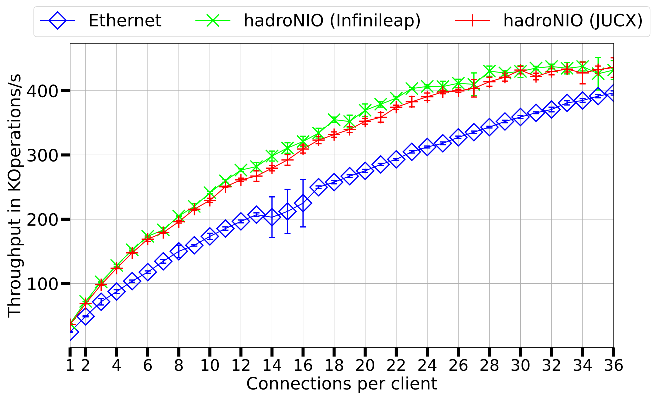


Fig. 6. gRPC key-value store request times with 3 clients and 16x1024 byte records

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.

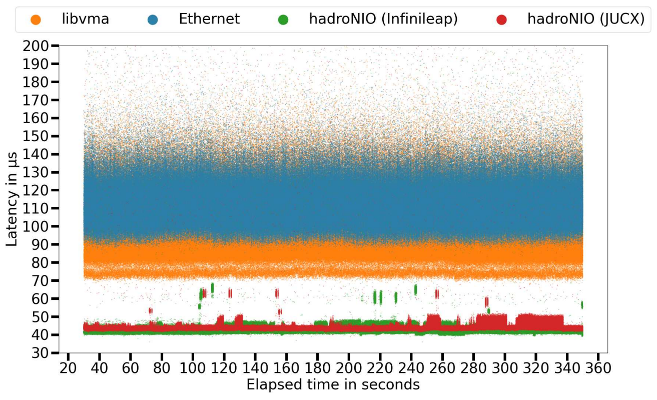


Fig. 7. ZooKeeper request times with 3 clients (each using 3 threads) and 1x16 byte records

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

Application Integration

With the increasing demand for systems to evaluate and analyze big data records, more and more frameworks for this purpose have emerged within the Java ecosystem over time. Examples of this are some projects created under the Apache Software Foundation, such as Apache Spark™[3], Apache Hadoop[4], Apache Storm[5] and Apache Flink®[6]. While some of these projects are implemented in JVM-compatible languages such as Scala, they all use the JVM as their foundation. For this reason, they also make use of the networking functionalities provided by the JDK. The following chapter focuses on the integration of high-speed interconnect hardware within existing frameworks aiming at transparently accelerating applications based on them.

5.1 The Java Development Kit's Networking Options

The Java Development Kit offers two basic building blocks for implementing network applications. These can be divided into the categories "*blocking*" and "*non-blocking*". Both categories have different approaches to the execution of operations. Roughly speaking, the non-blocking components within the JVM were introduced after the realization that the blocking components could no longer deliver the required efficiency within highly parallel systems.

5.1.1 Blocking Network I/O

The first available variant of network operations is blocking I/O implemented within classes located within the `java.net` package. The classes implemented here are based on the assumption that each connection within a network application is handled by a separate thread, so that many threads are busy processing network operations in parallel[56]. However, this type of design has a major disadvantage, which is particularly noticeable with many connections. As soon as a network operation is required, the associated thread must be stopped at the corresponding point due to the blocking nature of the network operation until the result can be retrieved or the function returns.

> **Example**
> An abstract comparison of this is an office (the processor) with several counters (the threads). Each customer to be served (the operation) must first queue at the counter. While one customer is being served at the counter (the operation is being carried out), all customers behind him in the queue must wait. If the customer needs more time, customers behind him may decide to leave the queue (abort due to timeout) in order to visit the office again at a later time.

As described in the previous example, the use of blocking I/O operations within network applications can lead to higher latencies or even timeouts during processing. However, the use of blocking I/O is still possible within smaller applications that are designed for handling only a dozen of connections. One advantage over non-blocking I/O is the straightforward implementation of network applications.

NetworkClient.java — Java

```java
1 public final class NetworkClient {
2
3   public static void main(String... args) {
4     Socket socket = new Socket("localhost", 1234);
5     DataOutputStream out = new DataOutputStream(socket.getOutputStream());
6     out.writeUTF("Hello World");
7     out.flush();
8   }
9 }
```

Figure 5.1: Establishing a blocking socket connection with a remote server.

Figure 5.1 shows a simple four-line program that establishes a connection with a remote server and then sends a message. First, a `Socket` is created in line ④, which is given the host and the port to which it should connect. In this example, the program connects to another process within the same computer, which is why `localhost` is specified here as the host. In the next step, an abstraction of the `java.net` package is used in line ⑤, which wraps the `OutputStream` of the socket to provide functions that can be used to send simple data types. This abstraction is the `DataOutputStream` class. It is used in the following line ⑥ to send a simple `String` using the `writeUTF` instance method. Finally, the instance method `flush` is called in line ⑦ to ensure that the data is sent, as it may still be temporarily held for buffering purposes. Using the steps described above, it is possible to write a simple program that connects to a server and sends it a message. On the server side, receiving messages using blocking I/O operations is also very easy and can be implemented in just a few lines.

NetworkServer.java Java

```java
1 public final class NetworkServer {
2
3   public static void main(String... args) {
4     ServerSocket server = new ServerSocket(1234);
5     Socket client = server.accept();
6     DataInputStream in = new DataInputStream(client.getInputStream());
7     System.out.println(in.readUTF());
8   }
9 }
```

Figure 5.2: Starting a blocking server instance and receiving data.

The creation of a server application using blocking I/O is shown in Figure 5.2. In line 4, an instance of the `ServerSocket` class must first be created. This type of socket listens on a defined port (in this example `1234`) and is able to process incoming connection requests. To accept an incoming connection request, the instance method `accept` is called, which blocks until a client requests a connection. The return value of the method is an object of type `Socket` which can be used to communicate with the client. For this purpose, the `InputStream` of the client is wrapped within a `DataInputStream` (line 6), which is the counterpart to a `DataOutpuStream`. It can be used to convert or deserialize the data sent by the client back into Java objects. This is done within the print statement in line 7 by calling the instance method `readUTF` (the counterpart to `writeUTF`) and outputting the resulting string (in this example `Hello World`) to the console. As in the case of the client side, it is thus also possible to write a functioning program in just a few lines using blocking I/O, which can accept connections via a network and receive messages afterwards.

However, it should be emphasized that the example can only accept one connection due to the use of the main thread (the thread that calls the `main` method) for processing incoming connections and also terminates immediately as soon as a message has been received and output. A server that wants to process several connections would therefore have to use several threads in which the respective `Socket` instances are processed. As each thread can process exactly one connection, the operating system or the underlying hardware quickly reaches its limits. This limitation was already recognized some time ago and addressed within the "C10K Problem"[57]. The core statement here is that operating systems provide various mechanisms (such as Linux's `epoll`) for parallel processing of many individual connections using a small limited number of threads, which should always be preferred within applications that expect a high number of connections. It is precisely for this reason that new network-related components have been developed within the JDK, which make use of these non-blocking mechanisms and make them available to the program for use.

5.1.2 Non-Blocking Network I/O

In contrast to the blocking I/O, there is the non-blocking I/O within the JDK, which bundles its classes within the `java.nio` package. The classes contained herein belong to Java's New I/O (*NIO*) APIs[58]. The programming model used here is based on asynchronous processing of operations. In detail, this means that triggering an operation does not lead to the executing thread being blocked, but the corresponding function places the operation to be executed in a queue for later execution. The program is then notified of the status of the operation by means of a callback, i.e. a function that is called on completion of the operation, and can react accordingly. A program based on non-blocking I/O generally uses a thread to accept connections and then passes the accepted connections to a thread of a thread pool, which manages a fixed number of threads (usually twice the number of processor cores). The threads of the thread pool are in turn responsible for processing incoming and outgoing operations inside an *event loop* (i.e. an endless loop). Since the execution of a continuous loop without intermediate pauses would lead to very high energy consumption due to the constantly working processor, Java's NIO API provides an abstraction for the use of operating system-specific multiplexing mechanisms[59]. This abstraction is the `Selector` class. Together with the `SocketChannel` class - the asynchronous counterpart to the `Socket` class - this abstraction can be used to implement highly parallel network applications using just a few threads. A `SocketChannel` can be assigned to a selector using the instance method `register`. In addition to the `Selector` instance, this method accepts a set of operations to which the selector should react, as well as a freely selectable attachment.

AsyncSelector.java Java

```java
Selector selector = Selector.open();
SocketChannel clientA = ...
SocketChannel clientB = ...
clientA.register(selector, SelectionKey.OP_READ, clientA);
clientB.register(selector, SelectionKey.OP_WRITE, clientB);
while (true) {
  selector.select();
  for (SelectionKey key : selector.selectedKeys()) {
    if (key.isReadable()) {
      // Perform read
    }

    if (key.isWritable()) {
      // Perform write
    }
  }
}
```

Figure 5.3: Asynchronous processing of multiple connections using a selector.

The previous example in Figure 5.3 shows the processing of two connections by a single thread using a `Selector` instance. This is created in line (1) using the class method `open`. Two different `SocketChannel` instances are then registered with the `Selector` in the lines (2) to (5). During registration, the second parameter of the `register` instance method can be used to specify which events should cause the `selector` to select the corresponding channel. The possible options for this are as follows.

- `SelectionKey.OP_READ` - The corresponding channel is selected as soon as incoming data is available for a read operation.
- `SelectionKey.OP_WRITE` - The associated channel is selected as soon as free memory space is available within its internal buffer for an outgoing write operation.
- `SelectionKey.OP_CONNECT` - The corresponding channel is selected as soon as a connection with the other side has been established or an error has occurred during the connection setup.
- `SelectionKey.OP_ACCEPT` - The associated channel is selected as soon as an incoming connection request can be processed. This option is only relevant for server applications.

Within the infinite loop, in line (7) the `select` instance method of the `Selector` is called. This initially causes the executing thread to block at this point and wait for one of the specified events. The method only returns when one of the events (`OP_READ` or `OP_WRITE`) occurs on the associated channels. Then, in line (8), the selected keys - abstractions that indicate which event has occurred and carry the specified attachment - can be iterated over using the `selectedKeys` instance method. Finally, the `isReadable` and `isWritable` instance methods of the `SelectionKey` class are used to check which event has occurred (lines (9) and (13)), whereupon a corresponding action, which is omitted in the previous example for reasons of simplicity, can be executed. Compared to blocking I/O, a special characteristic here is the way in which the individual connections are processed. Since the `select` method can select several connections on which the corresponding registered events have been recognized, these can be processed specifically or isolated from the rest of the connections that do not currently require processing. This means that the processor is only occupied with processing connections that are waiting to be processed and can use the resources thus freed up elsewhere or for other threads. As a result, this leads to dramatically better scaling, as the processor only needs to process when it is needed, thereby greatly optimizing efficiency.

While the blocking I/O API only accepts buffered streams as a source for data, the Java NIO API also offers the advantage of using off-heap memory. For this purpose, the `ByteBuffer` class introduced in 2.2.3 is used. The `SocketChannel` class also provides two instance methods `read` and `write` for read and write operations. Each of these receives a `ByteBuffer` instance as a parameter and then executes the corresponding operation on it.

ReadOperation.java Java

```java
1 SocketChannel channel = ...
2 ByteBuffer buffer = ByteBuffer.allocate(32);
3 int bytes = channel.read(buffer);
```

Figure 5.4: Execution of a read operation on a ByteBuffer instance.

An example of a read operation on a `ByteBuffer` instance is shown in Figure 5.4. First, a `ByteBuffer` instance, which comprises 32 bytes of memory, is allocated in line ②. This instance is then passed to the `read` instance method of the `SocketChannel` class in line ③. This results in any data that is ready to be read within the `SocketChannel` being copied to the reserved memory of the `ByteBuffer` instance. However, it should be noted here that it is not known in advance how many bytes can be read, so the buffer's size should be selected accordingly, as otherwise several read operations may be necessary instead of just one. It is also possible that not all the required data is available for processing because it has not yet been transferred. In such cases, the return value of the `read` instance method must be considered. It indicates how many bytes were copied into the reserved memory area of the `ByteBuffer` instance. If this number is smaller than the expected number of bytes, the read operation must be continued at a later time when the `SocketChannel` is selected again by the `Selector` due to new incoming data.

While this type of programming initially poses some challenges, it is indispensable within high-performing networking systems, which is why many well-known networking frameworks, such as Netty[60], rely on it. However, since such frameworks do not provide support for RDMA-enabled hardware like InfiniBand, the following two papers show how an integration can be transparently achieved by using Java NIO's `SelectorProvider` feature without having to modify existing application code.

5.2 hadroNIO: Accelerating Java NIO via UCX

Fabian Ruhland, *Filip Krakowski* and Michael Schöttner. hadroNIO: Accelerating Java NIO via UCX. In 20th International Symposium on Parallel and Distributed Computing, ISPDC 2021, Cluj-Napoca, Romania, July 28-30, 2021.

Contributions:
This work is the direct predecessor to the work presented in 4.4 and was mainly developed by Fabian Ruhland.

Unlike with the successor version, the Infinileap framework was not yet ready for use, which is why it was not included in this work. For this reason, Fabian Ruhland used the only available Java interface to OpenUCX at the time, which was JUCX. During the implementation, the author helped to track down some bugs within the JUCX framework and find a solution for them. One example of this was the author's discovery of a memory leak within the JUCX project, which was caused by the incorrect handling of global references within the native part of the code. Based on this finding, Fabian Ruhland created a pull request in the JUCX project, whereupon this error was fixed with the next release. As the author was working on the Infinileap project at the same time and was therefore familiar with the OpenUCX framework, there was also an exchange regarding the use of the framework, which cleared up some areas of unclarity.

The paper was written by Fabian Ruhland, whereby the author and Michael Schöttner were involved in some discussions on the conception and implementation of various components of the framework.

Status: published

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

978-1-6654-3281-8/21/$31.00 ©2021 IEEE

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 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`.

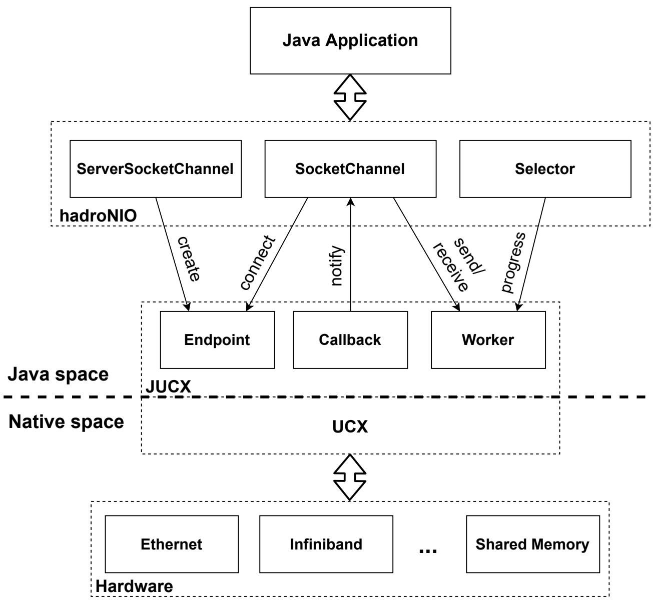


Fig. 1. Architecture overview

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.

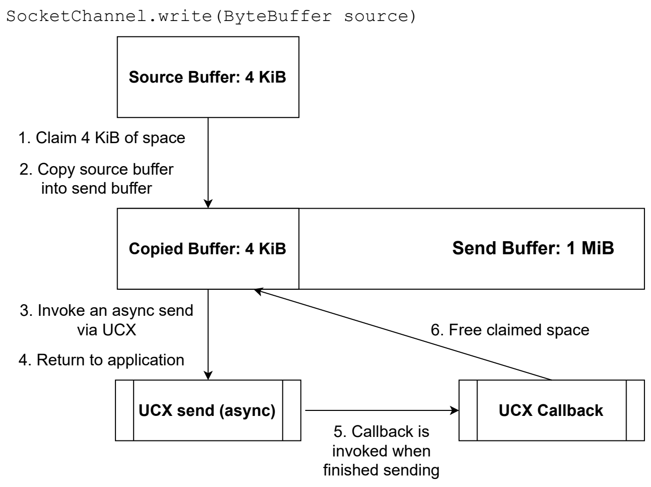


Fig. 2. Write mechanism with a 4 KiB source buffer and a 1 MiB intermediate buffer

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.

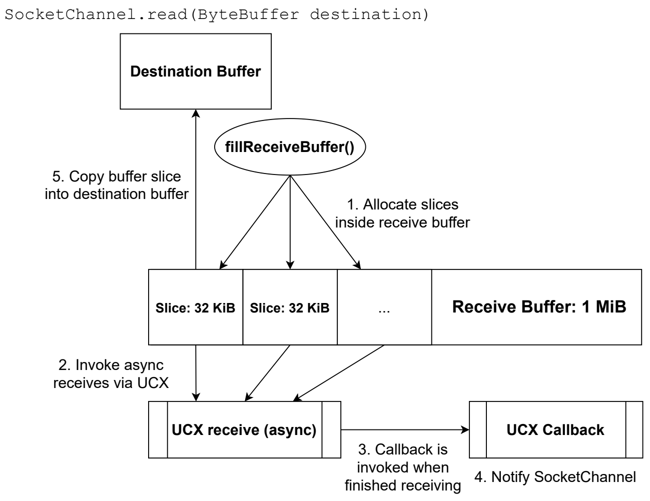

SocketChannel.read(ByteBuffer destination)
Destination Buffer
fillReceiveBuffer()
5. Copy buffer slice into destination buffer
1. Allocate slices inside receive buffer
Slice: 32 KiB
Slice: 32 KiB
...
Receive Buffer: 1 MiB
2. Invoke async receives via UCX
UCX receive (async)
UCX Callback
3. Callback is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

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

We use *Observatory*, a micro benchmark for Java-based InfiniBand solutions, for evaluating hadroNIO. Observatory allows evaluating both messaging and RDMA performance with single point-to-point connections, regarding throughput and latency. We have used it in the past 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 have extended Observatory with a new NIO binding, which can be used for both blocking and non-blocking socket channels. To evaluate the overhead introduced by hadroNIO, compared to directly using JUCX, Observatory includes a binding for JUCX, developed by Mellanox [11] [12]. We also compare hadroNIO to IPoIB, probably the most used acceleration solution, transparently available in many environments. Unfortunately, we were not able to generate results for libvma with our NIO benchmark, because 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 experiments.

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 always fit 128 slices. Furthermore, we used blocking socket channels for evaluating latency and non-blocking socket channels for the throughput benchmarks.

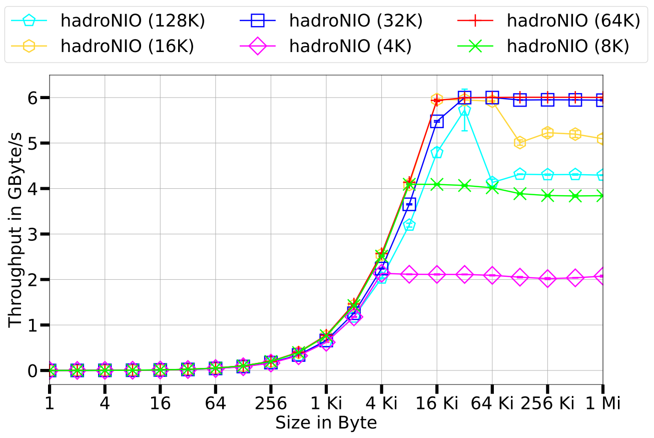


Fig. 5. Throughput results with non-blocking socket channels and different buffer configurations.

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.

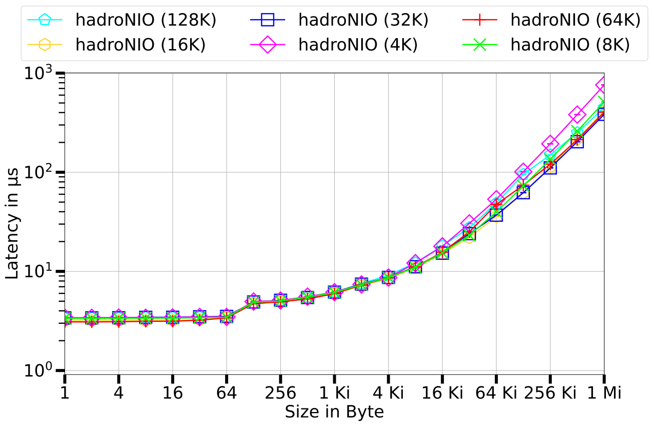


Fig. 6. Latency results with blocking socket channels and different buffer configuration.

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 sent. 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 plan to study the performance drop caused by 128 KiB slices and re-evaluate 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.

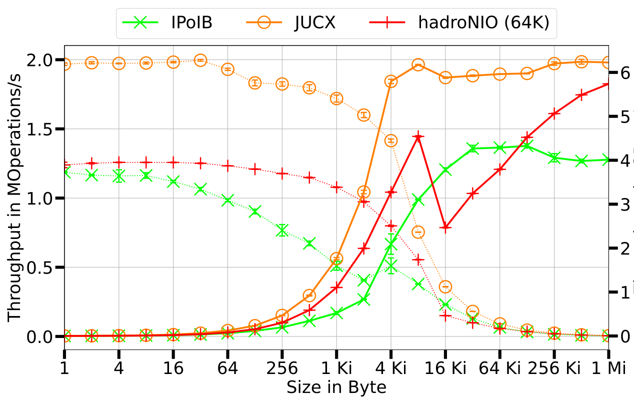


Fig. 7. Throughput results with blocking socket channels.

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

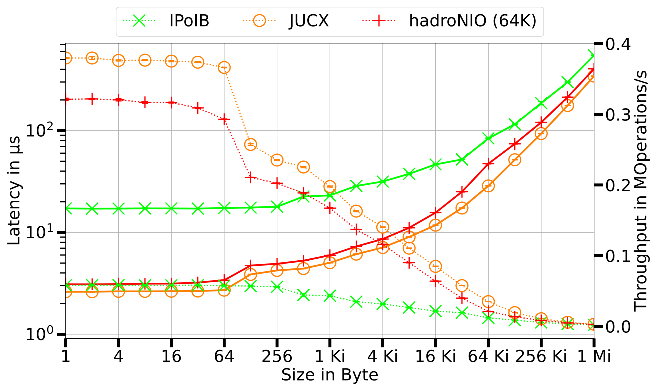


Fig. 8. Average round trip latency with blocking socket channels.

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.

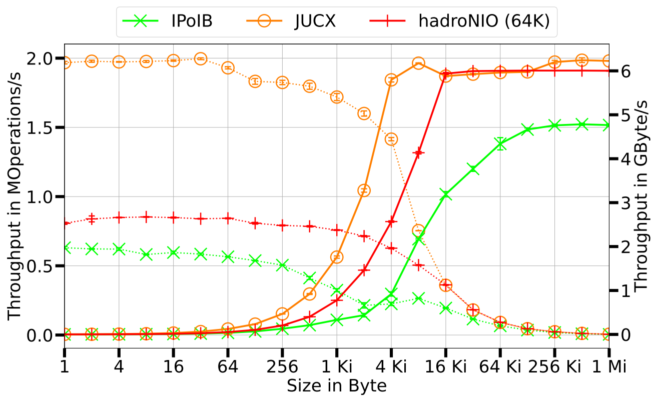


Fig. 9. Throughput results with non-blocking socket channels.

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

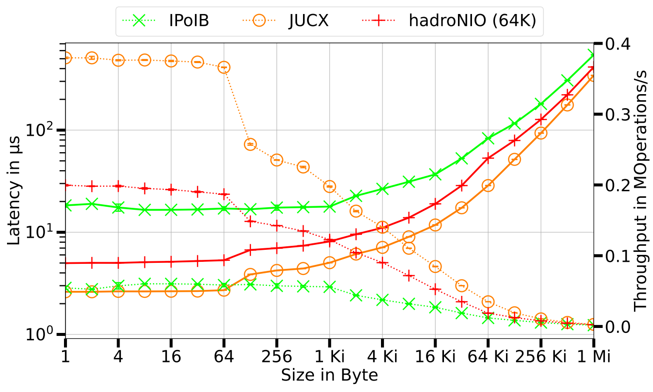


Fig. 10. Average round trip latency with non-blocking socket channels.

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.

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.

5.3 Accelerating netty-based applications through transparent InfiniBand support

Fabian Ruhland, *Filip Krakowski* and Michael Schöttner. Accelerating netty-based applications through transparent InfiniBand support. In Networking and Internet Architecture, arXiv, online, Sep 28, 2022.

Contributions:
This work pursues the goal of transparently accelerating the Netty framework using InfiniBand technology and the solution implemented in 5.2. For this purpose, Fabian Ruhland developed a compatibility layer between Netty and hadroNIO, which enabled the use of SocketChannel instances originating from hadroNIO within Netty and provided further functionalities for a successful integration.

Fabian Ruhland carried out the subsequent evaluation within the Oracle Cloud. Here, the author helped to set up the cloud environment using a Terraform project provided by Oracle. The author also provided advice when problems arose within the cloud environment. The implementation and execution of the benchmarks and the subsequent evaluation of the results were carried out by Fabian Ruhland. The author and Michael Schöttner participated in this phase in the form of discussions regarding the measured performance values.

The paper was written by Fabian Ruhland, with the author and Michael Schöttner providing advice and feedback.

Status: published

Accelerating netty-based applications through transparent InfiniBand support

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

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

Michael Schttner
Department Operating Systems
Heinrich Heine University
Dsseldorf, 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.

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

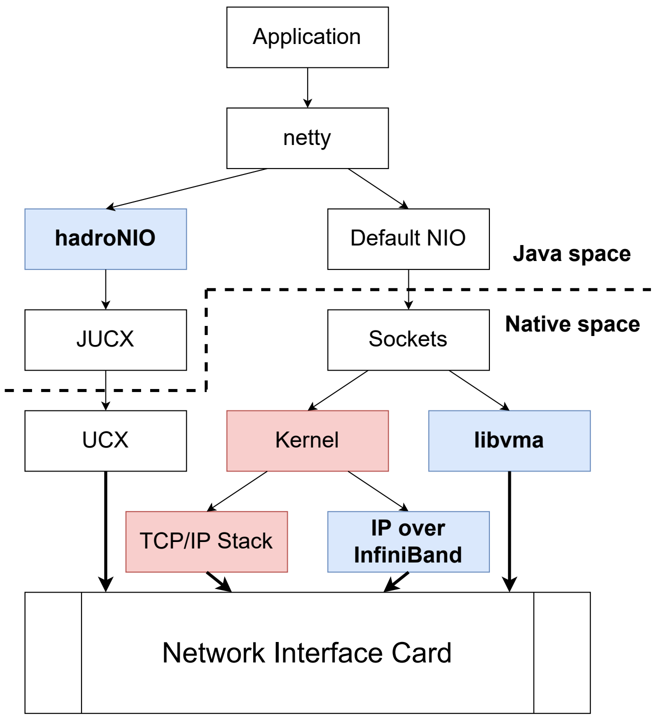


Fig. 1. Application stack overview

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)

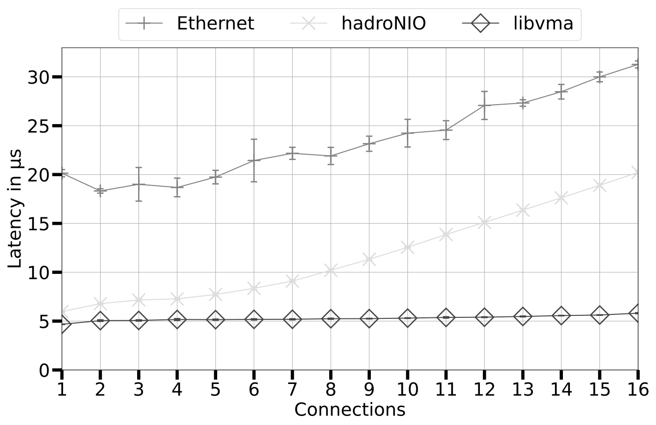


Fig. 3. Average round-trip times with 16-byte messages

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.

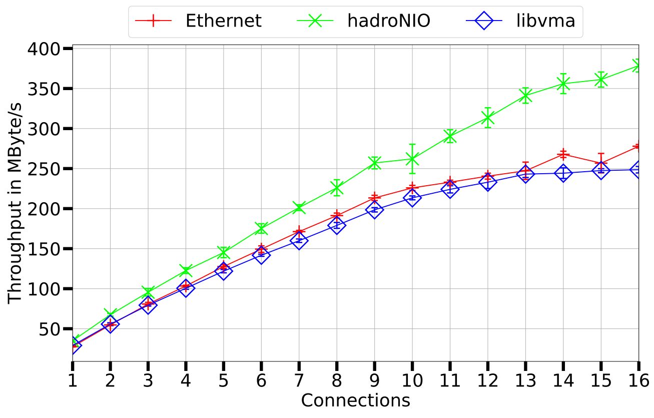


Fig. 4. Average throughput with 16-byte messages

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)

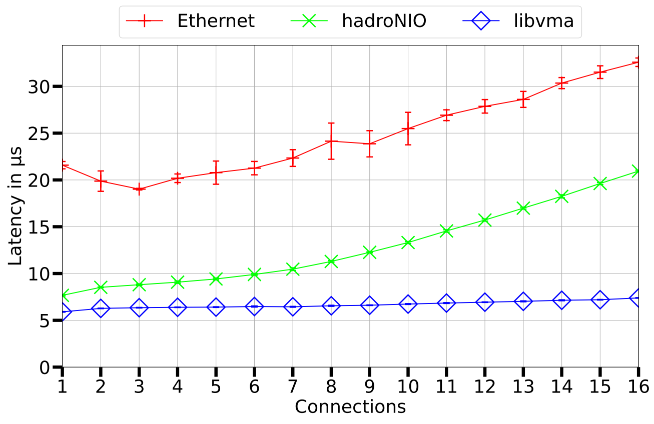


Fig. 5. Average round-trip times with 1 KiB messages

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,

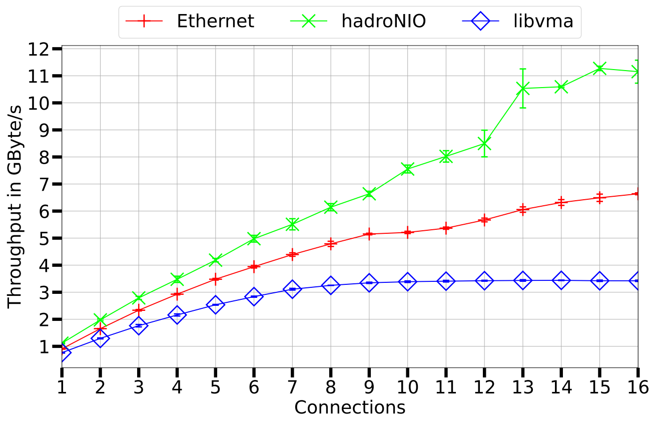


Fig. 6. Average throughput with 1 KiB messages

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)

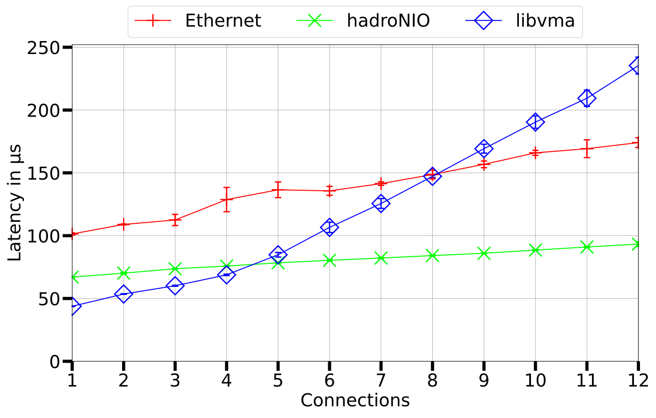


Fig. 7. Average round-trip times with 64 KiB messages

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.

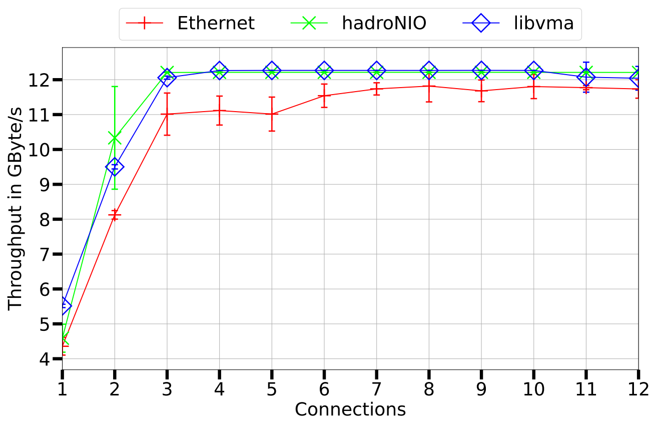


Fig. 8. Average throughput with 64 KiB messages

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):915, 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):3540, 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):339352, 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 305320, 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. Schttner. 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. Schttner. 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 6

Conclusion & Outlook

This thesis shows that the use of the Java programming language in the context of high-performance networking applications is possible and efficient. The work initially began with the Java Native Interface, which at that time was the only option within the Java ecosystem to access native functionalities. This resulted in the *Neutrino* project (3.2), which showed that a high-performance connection to native code is possible if various optimizations and peculiarities of the JNI are strictly taken into account. Using this knowledge, a connection to the native Verbs API was developed, which finally allowed InfiniBand hardware to be used efficiently within Java applications. In the subsequent benchmarks developed for this purpose, it was shown that connecting the hardware within Java led to a network performance that could keep up with existing native-based solutions or even surpassed some of them. However, since the implemented solution was not easy to maintain, another way was chosen to provide InfiniBand hardware support within the JDK. While there were already efforts within the JDK to provide support for RDMA[13], these were no longer maintained after some time, so that in the context of this work a project could be created in cooperation with Oracle Labs[61], which continues these original goals but is based on new technologies such as the Foreign Function & Memory API presented in 4.1. The resulting exchange enabled close cooperation with the JDK core developers in order to quickly find a solution when problems with new technologies arose. It also allowed the direct co-design of new features in the form of discussions within the mailing list or active co-development of the Java Development Kit in the form of bug fixes[62]. The solutions developed in 4.3 showed that the expected added value of the new JDK technologies was achieved, as benchmarks, which were used to evaluate the developed interfaces to native InfiniBand hardware using the Foreign Function & Memory API, achieved very good performance. In this context, it was also shown that the integration of native functionalities can greatly accelerate existing program code. Particularly within applications in which many processes are executed in a distributed manner and which have to exchange large amounts of data with each other, such as distributed big data applications, the use of the developed technologies can lead to significant improvements in performance. To make it easier to start programming with the Foreign Function & Memory API, additional tooling has been developed that makes it straightforward to integrate native libraries in Java (see 4.1).

It is expected that the use of the Foreign Function & Memory API will increase significantly with the upcoming Java version 22[63], which will be released in early 2024, as it will no longer be in preview status and can therefore be used by many developers. In some projects, such as Netty, the integration of the Foreign Function & Memory API is already being tested in the form of incubator projects[64]. In general, it is to be expected that the project landscape of the Java ecosystem will expand considerably with the release of the API, as existing projects written in native programming languages will be easily usable within Java projects from this point onwards.

The use of RDMA technology in connection with big data frameworks in Java also offers great potential for the future. While many aspects such as data distribution, access and the storage media used have already been optimized, the focus has recently been shifted to the data format. Accesses to the main memory can vary greatly in terms of latency depending on the respective access pattern. Ideally, data is aligned and contiguous in the memory so that caches can be optimally utilized. If this idea is applied to the data format used within big data applications, the latency is further reduced due to the shorter access time. The Apache Arrow project[65], which proposes a unified data format for distributed applications, makes use of this fact. Here, data is stored in the form of columns within the main memory, whereby each column is a contiguous memory area and is also aligned. In conjunction with RDMA-capable hardware, this results in highly efficient data transmissions, as the data is available in the main memory in the most suitable form for the network controller. A further advantage is the possibility of zero-copy[66] serialization, in which data can be sent directly in the form in which it is stored in the main memory. This further reduces latency, as it is no longer necessary to copy/transform the data since both sides have agreed on the same in-memory data format and are able to process it.

Based on the Apache Arrow project, there is also the Apache Arrow Flight project[67]. This provides a system for querying in-memory Arrow data. While the data format here is highly optimized, the project uses the gRPC framework for communication between individual nodes, which in turn is based on Netty and therefore uses Ethernet sockets. Integrating the solution developed in Project Hermes would be of great benefit here, as the transmission of data using RDMA offers significantly lower latency and higher throughput. In particular, the Arrow data format, which is optimally adapted to RDMA, offers great potential in conjunction with RDMA programming, as the network card can access the data very efficiently in this form and thus send it quickly. One way of implementing this is to integrate Infinileap within the Apache Arrow Flight project, which would then be able to access data using RDMA operations between the nodes involved. This idea is being pursued in a forthcoming project called Java Direct Flight. Here, the data transfer of larger data is to be offloaded to RDMA operations, while the coordination of the individual nodes continues to take place by means of messages via the gRPC protocol. Finally, such a solution can be integrated into an existing big data framework, which benefits in particular during the shuffle phase due to the significantly accelerated data transfer.

In summary, the use of RDMA technology will be a necessity in the future due to the constantly increasing amount of data, as conventional socket programming will no longer be appropiate for transfering data in a reasonable time due to the associated overhead. The solutions developed in this thesis provide a good base for this and - the author hopes - can be used in existing Java applications and computing frameworks.

Bibliography

[1] J. Wiener and N. Bronson, "Facebook's Top Open Data Problems", Oct. 21, 2014. [Online]. Available: https://research.facebook.com/blog/2014/10/facebook-s-top-open-data-problems.

[2] S. J. Dixon, *Facebook: Quarterly number of mau (monthly active users) worldwide 2008-2023*, Nov. 9, 2023. [Online]. Available: https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/.

[3] Apache Software Foundation, *Apache Spark™*. [Online]. Available: https://spark.apache.org/.

[4] Apache Software Foundation, *Apache Hadoop*. [Online]. Available: https://hadoop.apache.org/.

[5] Apache Software Foundation, *Apache Storm*. [Online]. Available: https://storm.apache.org/.

[6] Apache Software Foundation, *Apache Flink®*. [Online]. Available: https://flink.apache.org/.

[7] Apache Software Foundation, *Apache Beam*. [Online]. Available: https://beam.apache.org.

[8] M. Mejran, *How can I switch careers from Java to big data?*, 2014. [Online]. Available: https://www.quora.com/How-can-I-switch-careers-from-Java-to-big-data.

[9] S. Shahrivari, "Beyond batch processing: Towards real-time and streaming big data", *Computers*, vol. 3, no. 4, pp. 117–129, 2014, ISSN: 2073-431X. DOI: 10.3390/computers3040117. [Online]. Available: https://www.mdpi.com/2073-431X/3/4/117.

[10] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey", *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, pp. 1920–1948, 2015. DOI: 10.1109/TKDE.2015.2427795.

[11] B. Nicolae, C. Costa, C. Misale, K. Katrinis, and Y. Park, "Towards memory-optimized data shuffling patterns for big data analytics", in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 409–412. DOI: `10.1109/CCGrid.2016.85`.

[12] The Netty project, *Netty Homepage.* [Online]. Available: `https://netty.io`.

[13] Y. Lu, *JEP 337: RDMA Network Sockets.* [Online]. Available: `https://openjdk.org/jeps/337`.

[14] S. Nothaas, K. Beineke, and M. Schoettner, "Ibdxnet: Leveraging infiniband in highly concurrent java applications", Dec. 5, 2018. arXiv: `1812.01963 [cs.NI]`.

[15] Oracle, *Project panama: Interconnecting jvm and native code*, 2014. [Online]. Available: `https://openjdk.org/projects/panama/`.

[16] UCF Consortium, *Unified Communication X Homepage.* [Online]. Available: `https://openucx.org/`.

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

[18] Oracle, *The Java Virtual Machine Instruction Set.* [Online]. Available: `https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html` (visited on 10/17/2023).

[19] D. Maier, N. Grcevski, and V. Sundaresan, "An introduction to java development kit 7", in *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '11, Toronto, Ontario, Canada: IBM Corp., 2011, pp. 366–367.

[20] Oracle, *Java Garbage Collection Basics*, Oct. 17, 2023. [Online]. Available: `https://www.oracle.com/webfolder/technetwork/Tutorials/obe/java/gc01/index.html` (visited on 10/17/2023).

[21] u. Nikolić and F. Spoto, "Reachability analysis of program variables", *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 4, Jan. 2014, ISSN: 0164-0925. DOI: `10.1145/2529990`. [Online]. Available: `https://doi.org/10.1145/2529990`.

[22] R. E. Jones and C. Ryder, "A study of java object demographics", in *Proceedings of the 7th International Symposium on Memory Management*, ser. ISMM '08, Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 121–130, ISBN: 9781605581347. DOI: `10.1145/1375634.1375652`. [Online]. Available: `https://doi.org/10.1145/1375634.1375652`.

[23] H. Inoue, D. Stefanovic, and S. Forrest, "On the prediction of java object lifetimes", *IEEE Transactions on Computers*, vol. 55, no. 7, pp. 880–892, 2006. DOI: `10.1109/TC.2006.107`.

[24] M. Beckwith, *Garbage First Garbage Collector Tuning*, Aug. 2023. [Online]. Available: `https://www.oracle.com/technical-resources/articles/java/g1gc.html` (visited on 10/20/2023).

[25] Sun Microsystems, "Memory Management in the JavaHotSpot™ Virtual Machine", *Oracle Whitepapers*, Apr. 2006. [Online]. Available: `https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf` (visited on 10/20/2023).

[26] W. Huang, Y. Qian, W. Srisa-an, and J. Chang, "Object allocation and memory contention study of java multithreaded applications", in *IEEE International Conference on Performance, Computing, and Communications, 2004*, ser. PCCC-04, IEEE. DOI: `10.1109/pccc.2004.1395032`.

[27] 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`.

[28] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection", *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 2, Feb. 2019, ISSN: 1049-331X. DOI: `10.1145/3295739`. [Online]. Available: `https://doi.org/10.1145/3295739`.

[29] P. E. McKenney, "Memory barriers: A hardware view for software hackers", *Linux Technology Center, IBM Beaverton*, 2010.

[30] Intel Corporation, *Intel 64 and ia-32 architectures software developer's manual - volume 4*, Intel Corporation, Sep. 2023.

[31] Wikipedia contributors, *Direct memory access*, [Online; accessed 16-November-2023], 2023. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Direct_memory_access&oldid=1165054892`.

[32] Oracle, *Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification - Class ByteBuffer*. [Online]. Available: `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/ByteBuffer.html`.

[33] F. Klein and M. Schottner, *Dxram: A persistent in-memory storage for billions of small objects*, 2013. DOI: `10.1109/pdcat.2013.23`.

[34] S. Patidar, D. Rane, and P. Jain, "A survey paper on cloud computing", in *2012 Second International Conference on Advanced Computing & Communication Technologies*, 2012, pp. 394–398. DOI: 10.1109/ACCT.2012.15.

[35] J. Moura and D. Hutchison, "Review and analysis of networking challenges in cloud computing", *Journal of Network and Computer Applications*, vol. 60, pp. 113–129, 2016, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2015.11.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S108480451500288X.

[36] T. Chen, X. Gao, and G. Chen, "The features, hardware, and architectures of data center networks: A survey", *Journal of Parallel and Distributed Computing*, vol. 96, pp. 45–74, 2016, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2016.05.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731516300399.

[37] L. Kalita, "Socket programming", *International Journal of Computer Science and Information Technologies*, vol. 5, no. 3, pp. 4802–4807, 2014.

[38] Day, J.D. and Zimmermann, H., "The osi reference model", 12, vol. 71, 1983, pp. 1334–1340. DOI: 10.1109/PROC.1983.12775.

[39] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch", in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07, San Diego, California: Association for Computing Machinery, 2007, 2–es, ISBN: 9781595937513. DOI: 10.1145/1281700.1281702. [Online]. Available: https://doi.org/10.1145/1281700.1281702.

[40] NVIDIA, *ConnectX-7 400G Adapters*. [Online]. Available: https://nvdam.widen.net/s/csf8rmnqwl/infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471.

[41] G. Kerr, "Dissecting a small infiniband application using the verbs API", *CoRR*, vol. abs/1105.1827, 2011. arXiv: 1105.1827. [Online]. Available: http://arxiv.org/abs/1105.1827.

[42] Wikipedia contributors, *Pci express*, [Online; accessed 16-November-2023], 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=PCI_Express&oldid=1183939525.

[43] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A remote direct memory access protocol specification", Tech. Rep., 2007.

[44] G. Shainer, P. Lui, and T. Liu, "The development of mellanox/nvidia gpudirect over infiniband: A new model for gpu to gpu communications", in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, ser. TG '11, Salt Lake City, Utah: Association for Computing Machinery, Jul. 18, 2011, p. 1, ISBN: 9781450308885. DOI: `10.1145/2016741.2016769`. [Online]. Available: `https://doi.org/10.1145/2016741.2016769`.

[45] S. Liang, *Java Native Interface: Programmer's Guide and Specification.* 1999. [Online]. Available: `https://api.semanticscholar.org/CorpusID:58765427`.

[46] D. Kurzyniec and V. Sunderam, "Efficient cooperation between java and native codes–jni performance benchmark", in *The 2001 international conference on parallel and distributed processing techniques and applications*, 2001.

[47] P. Sandoz, *8193033: Remove terminally deprecated sun.misc.unsafe.defineclass*, Mar. 20, 2018. [Online]. Available: `https://github.com/openjdk/jdk/commit/cfb102ab895e503d96ee273eb920f054bb861840`.

[48] Oracle, *Jep 412: Foreign function & memory api (incubator)*, Apr. 10, 2021. [Online]. Available: `https://openjdk.org/jeps/412`.

[49] Oracle, *Calling native functions with jextract.* [Online]. Available: `https://docs.oracle.com/en/java/javase/21/core/call-native-functions-jextract.html`.

[50] F. Krakowski, *Gradle-jextract.* [Online]. Available: `https://github.com/krakowski/gradle-jextract`.

[51] M. Cimadamore and F. Krakowski, *Panama-dev mailing list : [foreign-jextract] jextract-generated methodhandle is null*, Sep. 18, 2020. [Online]. Available: `https://mail.openjdk.org/pipermail/panama-dev/2020-September/010826.html`.

[52] Oracle, *Java® Platform, Standard Edition & Java Development Kit Version 20 API Specification | Class String.* [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`.

[53] A. Zakusylo, *java-native-benchmark.* [Online]. Available: `https://github.com/zakgof/java-native-benchmark/tree/master`.

[54] UCX Developers, *Unified Communication X Architecture.* [Online]. Available: `https://github.com/openucx/ucx#architecture`.

[55] OpenUCX Developers, *Openucx documentation - frequently asked questions.* [Online]. Available: `https://openucx.readthedocs.io/en/master/faq.html?highlight=multi%20rail#what-is-the-default-behavior-in-a-multi-rail-environment`.

[56] K. L. Calvert and M. J. Donahoo, *TCP/IP sockets in Java: practical guide for programmers*. Morgan Kaufmann, 2011.

[57] D. Kegel, *The C10K problem*. [Online]. Available: `http://www.kegel.com/c10k.html`.

[58] Oracle, *More New I/O APIs for the Java Platform*. [Online]. Available: `https://openjdk.org/projects/nio/`.

[59] L. Gammo, T. Brecht, A. Shukla, and D. Pariag, "Comparing and evaluating epoll, select, and poll event mechanisms", in *Linux Symposium*, vol. 1, 2004.

[60] N. Maurer, *Netty in action*, M. Wolfthal, Ed. Shelter Island: Manning, 2016, 279 pp., ISBN: 9781617291470.

[61] Oracle, *Oracle Labs Homepage*. [Online]. Available: `https://labs.oracle.com/pls/apex/r/labs/labs/about`.

[62] F. Krakowski, *8248415: Create VarHandles for pointer fields through the MemoryHandles API*. [Online]. Available: `https://github.com/openjdk/panama-foreign/pull/216`.

[63] M. Cimadamore, *JEP 454: Foreign Function & Memory API*. [Online]. Available: `https://openjdk.org/jeps/454`.

[64] Netty Developers, *New proposed API for buffers in Netty*. [Online]. Available: `https://github.com/netty/netty-incubator-buffer-api`.

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

[66] Wikipedia contributors, *Zero-copy — Wikipedia, the free encyclopedia*, [Online; accessed 7-December-2023], 2023. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Zero-copy&oldid=1168039071`.

[67] 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).

List of Figures

Eidesstattliche Versicherung

nach §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.

Hilden, January 2024

Filip Krakowski