RZ 3845 (#ZUR1607-030) 07/13/2016
Computer Sciences 11 pages

Research Report

jVerbs: RDMA support for Java®

Patrick Stuedi, Bernhard Metzler, Animesh Kumar Trivedi

IBM Research – Zurich
8803 Rüschlikon
Switzerland

Research
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

jVerbs: RDMA support for Java

Patrick Stuedi
IBM Research, Zurich
stu@zurich.ibm.com

Bernard Metzler
IBM Research, Zurich
bmt@zurich.ibm.com

Animesh Trivedi
IBM Research, Zurich
atr@zurich.ibm.com

Abstract

We present jVerbs, an API and library offering full RDMA semantics in Java. jVerbs achieves bare-metal latencies in the order of single digit microseconds for reading small byte buffers from a remote host's memory. In the paper, we discuss the design and implementation of jVerbs and demonstrate its advantages with several applications.

1 Introduction

There has been an increasing interest in low latency for data center applications. Remote Direct Memory Access (RDMA) is a standard for efficient and low latency memory to memory data transfer between two networked hosts. RDMA provides zero copy, kernel bypass as well as enhanced networking semantics such as one-sided operations and clean separation of paths for data and control operations. Traditionally, RDMA techniques have been employed in high-performance computing [3]. Consequently, the interfaces to the RDMA network stack are in native languages like C. In contrast, today many data center applications and even large parts of the software infrastructure for data centers are written in Java. Therefore, proper RDMA support in Java will be necessary to avoid giving away the latency advantages of modern network interconnects.

One obstacle towards achieving this goal is that RDMA requires direct access to the networking hardware from user space. Java proposes the Java Native Interface (JNI) to provide Java applications with access to low level tasks that are best implemented in C. Crossing the Java-native boundaries, however, is known to be costly. Even though JNI performance has improved, the performance penalties are still significant compared to the latencies of modern high-speed interconnects.

Luckily, Java as a language has evolved offering more options for implementing efficient RDMA support in Java. We present jVerbs, an API and library for Java offering full RDMA semantics and bare-metal latencies (up to 4-7 microseconds) for reading small byte buffers from a remote host's memory. jVerbs operates in concert with the rest of Linux RDMA ecosystem and works with existing RDMA devices (both software and hardware based). In the paper we discuss the design and implementation of jVerbs and demonstrate how it can be used to boost the performance of existing data center applications such as a distributed cache and web services.

2 The Java Latency Gap

Network latencies seen by Java applications on modern high speed interconnects are lagging far behind the lowest possible latencies that can be achieved with applications written in C. This is partially due to the additional overhead imposed by Java but also due to the limited capabilities in Java to leverage the hardware features of modern interconnects. To illustrate this situation we performed a set of experiments comparing the latencies seen by applications written in Java and C, respectively.

In the first set of experiments we measure the round-trip latency of a 64 byte message exchanged between two hosts connected by a 10 Gbit/s Ethernet network. In the experiment we use Chelsio T4 NICs which can be used both in full RDMA and Ethernet only mode. As can be seen from Figure 1, when implemented in Java using regular TCP-based sockets this benchmark achieves a latency of $47\mu s$. The same benchmark implemented in C achieves a latency of $35\mu s$. The performance difference can mainly be attributed to the longer code path of Java.

We set these results in relation to the latency achievable if RDMA is deployed in the same network setup. Here, we further distinguish between (a) using the RDMA `send/recv` model which resembles the socket behavior and (b) taking full advantage of RDMA semantics through one-sided `read` operations (see Sec-

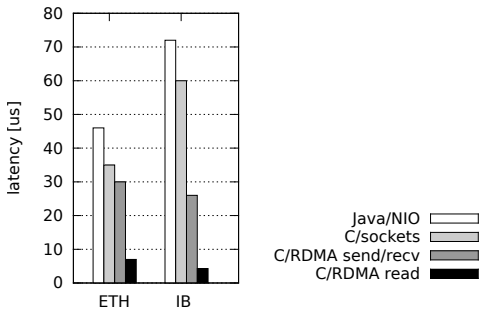

Figure 1: Request/response latencies on modern interconnects. There exists a significant latency gap between Java and other implementations.

tion 3 for details). By applying the `send/recv` model, RDMA protocol offloading and zero copy data movement already speeds up the latency to $30\mu s$. Using RDMA `read` operations the round trip latency further decreases to only $7\mu s$.

In a second set of experiments we exchanged Ethernet with another RDMA interconnect, namely InfiniBand. Infiniband uses a different network link technology and is mainly deployed in high-performance computing. The measurements show an even more dramatic latency gap between the C application running over InfiniBand and using Java on this technology. This increased performance discrepancy is mainly caused by the InfiniBand stack being highly optimized for direct RDMA operations but not for being accessed via sockets/TCP. With that, the plain Java-based implementation ends up lagging behind by up to $70\mu s$ when compared to an RDMA-based implementation written in C.

Those measurements yield two main results. First, better RDMA support in Java is necessary if we want Java applications to benefit from the latency advantages of modern interconnects. Second, only making the whole set of RDMA semantics directly available within Java has the potential to unleash the full RDMA performance to its applications. jVerbs accommodates both targets. To show the motivation behind its design, in the next two sections we describe the core elements of the RDMA technology and discuss the challenges in providing RDMA in Java.

3 Background

We briefly provide the necessary background information about RDMA, its API, and implementation in Linux. A more detailed introduction into RDMA can be found in [7].

RDMA Semantics: RDMA provides both send/receive type communication and RDMA operations. With RMDA `send/recv` – also known as **two-sided operations** – the sender sends a message, while the receiver pre-posts an application buffer, indicating where it wants to receive data. This is similar to the traditional socket based communication semantics. RDMA operations comprise `read`, `write`, and `atomics`, commonly referred to as **one-sided operations**. These operations require only one peer to actively read, write, or atomically manipulate remote application buffers.

In contrast to the socket model, RDMA fully separates data transfer operations from control operations. This facilitates pre-allocation of communication resources e.g application buffer registration, connection endpoint and work queue allocations etc. Separating a *fast data path* from a *control path* enables implementing very fast dispatching of data transfer operations without operating system involvement and is key for achieving ultra-low latencies.

API: Applications interact with the RDMA subsystem through a "verbs" interface, a loose definition of API calls providing aforementioned RDMA semantics [8]. By avoiding a concrete syntax, the verbs definition allows for different platform specific implementations.

To exemplify a control type verb, the `create_qp()` creates a queue pair of send and receive queues for holding application requests for data transfer. Data operations such as `post_send()` or `post_recv()` allow applications to asynchronously post data transfer requests into the send/receive queues. Completed requests are placed into an associated completion queue, allocated via the `create_cq()` verb. The completion queue can be queried by applications either in polling mode or in blocking mode.

The verbs API also defines **scatter/gather** data access to minimize the number of interactions between applications and RDMA devices. Furthermore, multiple requests can be submitted by a single `post_send()` or `post_recv()` call.

Implementation: Today concrete RDMA implementations are available for multiple interconnect technologies, such as InfiniBand, iWARP, or RoCE. OpenFabrics is a widely accepted effort to integrate all these technologies from different vendors and to provide a common RDMA application programming interface implementing the RDMA verbs.

As a software stack, the OpenFabrics Enterprise Distribution (OFED) spans both the operating system kernel and user space. At kernel level, OFED provides an umbrella framework for hardware specific RDMA drivers. These drivers can be software emulations of RDMA devices, or regular drivers managing RDMA network interfaces such as Chelsio T4. At user level, OFED provides an application library implementing the *verbs* interface.

	RTT	JNI costs	Overhead
2000/VIA	70μs	20μs	29%
2012/RDMA	3-7μs	2-4μs	28-133%

Table 1: Network latencies and accumulated JNI costs per data transfer in 2000 and 2012. Assuming four VIA operations with base-type parameter per network operation [11], and two RDMA operations with complex parameters per network operation (`post_send()`/`poll_cq()`).

Control operations involve both the OFED kernel and userspace verbs library. In contrast, operations for sending and receiving data are implemented by directly accessing the networking hardware from user space.

4 Challenges

Java proposes the Java Native Interface (JNI) to give applications access to low-level functionality that is best implemented in C. Unfortunately, the overhead of crossing the boundary between Java and C can be quite high, especially for function calls with complex parameters. This is bad news since the key data operations in RDMA like `post_send()`, `post_recv()` and `poll_cq()` all take arrays of complex data types as parameters.

Direct userland network access for Java has been investigated in the late 90s in the context of the virtual interface architecture (VIA). Jaguar [11] is the most sophisticated approach of that time and also closest to jVerbs. Inevitably, Jaguar discusses the performance overhead of using JNI to access networking hardware from Java. The numbers reported are in the order of $1\mu s$ for a simple JNI call without parameters, and 10s of μs for more complex JNI calls. Copying data across the JNI boundary though, is the most expensive operation of all with 270 μs per kB of data. These numbers are best viewed in relation to the round-trip latency of $\sim 70\mu s$ measured natively on VIA. Today, we know that ping/pong latencies of single digit microseconds are possible with modern RDMA interconnects. Thus, we made several experiments to see how a potential JNI solution would fit into the RDMA latency landscape. We found improved JNI overheads of about 100*ns* for simple JNI calls without parameters but $1-2$ μs for more complex JNI calls similar to the `post_send()` verb call.

Considering that multiple verb calls are necessary for a single message round-trip, these numbers add a significant fraction to the RDMA latency. This is an optimistic calculation, looking at RDMA operations referencing a single buffer only. The JNI overhead for scatter/gather operations can easily reach multiples of microseconds.

	jVerb call	description
Control	`jv_create_qp()`	create send/recv queue
	`jv_create_cq()`	create completion queue
	`jv_reg_mr()`	register memory with device
	`jv_connect()`	setup an RDMA connection
Data	`jv_post_send()`	post operation(s) to send queue
	`jv_post_recv()`	post operation(s) to recv queue
	`jv_poll_cq()`	poll completion queue
	`jv_get_cq_evt()`	wait for completion event

Table 2: Most prominent API calls in jVerbs.

The key take away here is that the JNI overhead for complex function calls (including arrays and complex data types) is inacceptable when implementing a low latency network stack. The overhead of simple function calls (with just base-type parameters), however, does not add significant costs to the overall latency.

5 Design of jVerbs

In this section we describe some of the design decisions we made in jVerbs .

5.1 jVerbs API

The verbs interface is not the only RDMA API, but it represents the "native" API to interact with RDMA devices. Other APIs like uDAPL, Rsockets, SDP or OpenMPI/RDMA are built on top of the verbs, and typically offer higher levels of abstractions at the penalty of restricted semantics and lower performance. With jVerbs as a native RDMA API we decided not to compromise on available communication semantics nor minimum possible networking latency. jVerbs provides access to all the exclusive RDMA features such as one-sided operations and separation of data and control, while maintaining a completely asynchronous, event driven interface. Other higher-level abstractions may be built on top of jVerbs at a later stage if the need arises. Table 2 lists some of the most prominent verb calls available in jVerbs.

5.2 Zero-copy Data Movement

Regular Java heap memory used for storing Java objects cannot be used as a source or sink in a RDMA operation since this would interfere with the activities of the garbage collector. Fortunately, support for off-heap memory has been added to Java since version 1.4.2 of the Java Development Kit (JDK). Off-heap memory is allocated in a separate region outside the control of the

garbage collector, yet it can be accessed through the regular Java memory API (ByteBuffer). jVerbs enforces the use of off-heap memory in all of its data operations. Any data to be transmitted or buffer space for receiving must reside in off-heap memory. In contrast to regular heap memory, off-heap memory can be accessed cleanly via DMA. As a result, jVerbs enables true zero-copy data transmission and reception for all application data stored in off-heap memory. This eliminates the need for data copying across a JNI interface which we know can easily cost multiple 10s of microseconds. In practice though, it looks like at least one copy will be necessary to move data between its on-heap location and a network buffer residing off-heap. In many situations, however, this copy can be avoided by either making sure that network data resides off-heap from the beginning, or by merging the copying with object serialization. As an example of the first, one can imagine a key/value store with the data store being held in off-heap memory. A good example of the second is an RPC call, where the parameters and result values are marshalled into off-heap memory instead of marshalling them into regular heap memory. Both cases will be described in more detail in Section 7.

5.3 Stateful Verb Calls

At its core, any RDMA user library will have to translate data structures exported in the API into a memory layout for the device. For instance, the post_send verb call requires work descriptors provided by the application to be written into a mapped device queue in a device specific format. Such a serialization when done in Java can easily reach several microseconds; too much given the single-digit network latencies of modern interconnects. To mitigate this problem, jVerbs employs a mechanism called stateful verb calls (SVCs). With SVCs, any state that is created as part of a jVerbs API call is passed back to the application and can be reused on subsequent calls. This mechanism manifests itself directly at the API level: instead of returning a return value, the verb calls return a stateful object that represents a verb call for a given set of parameter values. Each SVC object has the following four operations associated with it:

- `exec()`: executes the corresponding jVerbs API call for the given parameter values;
- `result()`: returns the result of the last jVerbs call;
- `valid()`: is *true* to indicate that the object can be executed, *false* otherwise;
- `free()`: releases any state associated with this SVC object;

One key advantage of SVCs is that they can be cached and re-executed as long as they remain valid. Semantically, each execution of an SVC object is identical to a jVerbs call evaluated against the current parameter state of the SVC object. Any serialization state that is necessary while executing the SVC object, however, will only have to be created when executing the object for the first time. Subsequent calls use the already established serialization state, and will therefore execute much faster.

Some SVC objects allow for changing the parameter state after object creation. For instance, the addresses and offsets of SVC objects returned by `jv_post_send()` and `jv_post_recv()` can be changed by the application if needed. Internally, those objects update their serialization state incrementally. Modifications to SVC objects are only permitted as long as they are not extending the serialization state. Consequently, adding new work requests or scatter/gather elements to a SVC `jv_post_send()` object is not allowed.

Stateful verb calls give applications a handle to mitigate the serialization cost. In many situations, applications may only have to create a small number of SVC objects matching the different types of verb calls they intend to use. Re-using those objects effectively reduces the serialization cost to almost zero as we will show in Section 8. Figure 2 illustrates programming with jVerbs and SVCs and compares it to native RDMA programming in C.

6 Implementation

The implementation of jVerbs follows the *abstract factory* pattern. The API is defined as an abstract class and applications use a factory to create a specific instance of the jVerbs interface. Currently there exist two separate implementations of the jVerbs interfaces, jVerbs/mem which is implemented entirely in Java and jVerbs/nat which makes use of JNI in a way that avoids JNI performance overheads almost completely. From a performance standpoint jVerbs/mem has a slight edge over jVerbs/mem. On the other hand jVerbs/nat can run with any RDMA device currently supported by OFED which is not true for jVerbs/mem.

Which implementation of the jVerbs library is instantiated is decided by a system property that can be set by the application. Other than that, the implementation specific details are completely shielded from the application. In the following we discuss each of the two implementations of jVerbs in more detail.

```
/* assumptions: send queue (sq),
   completion queue (cq),
   work requests (wrlist),
   output parameter with
   polling events (plist) */

/* post the work requests */
post_send(sq, wrlist);
/* check if operation has completed */
while(poll_cq(cq, plist) == 0);
```

(a)

```
RdmaVerbs v = RdmaVerbs.open();
/* post the work requests */
v.jv_post_send(sq, wrlist).exec().free();
/* check if operation has completed */
while(v.jv_poll_cq(cq, plist)
    .exec().result() == 0);
```

(b)

```
RdmaVerbs v = RdmaVerbs.open();
/* create SVCs */
RdmaSVC post = v.jv_post_send(sq, wrlist)
RdmaSVC poll = v.jv_poll_cq(cq, plist);
post.exec();
while(poll.exec().result() == 0);
/* modify the work requests */
post.getWR(0).getSG(0).setOffset(32);
/* post again */
post.exec();
while(poll.exec().result() == 0);
```

(c)

Figure 2: RDMA programming (a) using native C verbs, (b) using jVerbs, (c) using jVerbs with SVC.

6.1 jVerbs/Mem

jVerbs/mem is an implementation of the jVerbs API entirely written in Java. It is based on two key aspects: memory mapped hardware access and direct kernel interaction. Figure 3 serves as a reference throughout the section, illustrating the various aspects of the jVerbs/mem implementation architecture.

6.1.1 Memory-mapped Hardware Access

As mentioned earlier, for all performance-critical operations the native C verbs library interacts with RDMA network devices via three queues: a send queue, a receive queue and a completion queue. Those queues represent hardware resources but are mapped into user space to avoid kernel involvement when accessing them. jVerbs/mem leverages Java's off-heap memory

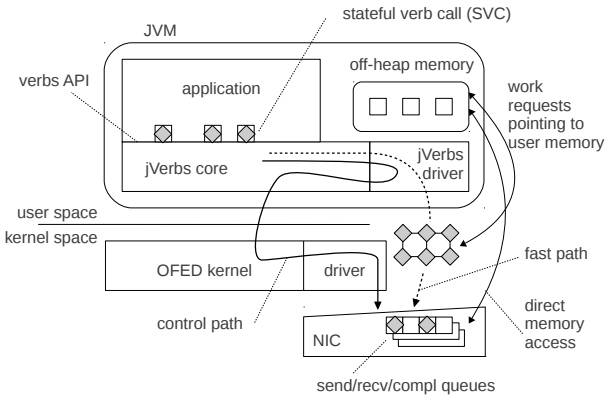

Figure 3: Architecture of the jVerbs/mem implementation.

to map these hardware resources into the Java address space. Fast path operations like `jv_post_send()` or `jv_post_recv()` are implemented by directly serializing work requests into the mapped queues. All the operations are implemented entirely in Java, avoiding expensive JNI calls or modifications to the JVM.

Access to hardware resources is device specific. Hence, the native C verbs user library contains hooks for a device specific driver that implements hardware access from user space. In jVerbs/mem we follow the same model, encapsulating the device specific internals into a separate module (user driver) which interacts with the core jVerbs library through a well defined *user driver interface*. A concrete implementation of this interface knows the layout of the hardware queues and makes sure work requests or polling requests are serialized into the right format. As of now, we have implemented user drivers for Chelsio T4, Mellanox ConnectX-2 and SoftiWARP [10].

User drivers typically make use of certain low-level operations when interacting with hardware devices. For instance, efficient locking is required to protect hardware queues from concurrent access. Other parts require atomic access to device memory or guaranteed ordering of instructions. While such low-level language features are available in C they have not been widely supported in Java for years. With the rise of multicore, however, Java has extended its concurrency API to include many of these features. For instance, Java has recently added support for atomic operations and fine grain locking. Based on those language features it was possible to implement RDMA user drivers entirely in Java using regular Java APIs in most of the cases. The only place where Java `reflection` was required was to obtain an off-heap mapping of device memory. This is due to the failure of Java `mmap()` to work properly with device files. Look-

ing at the Java road map it shows that more low-level features will be added in the coming releases, making the development of jVerbs user drivers even more convenient.

6.1.2 Direct Kernel Interaction

The native C verbs library implements control operations in concert with the RDMA kernel modules. Verbs library and kernel communicate over a shared file descriptor using a well defined binary protocol. Control operations are often referred to as the *slow path* since they typically do not implement performance critical operations. This is, however, only partially true. Certain operations like memory registration may very well be in the critical path of applications. To make sure no additional overhead is imposed for control operations, jVerbs/mem directly communicates with the RDMA kernel via the same file-based binary protocol. Again, one alternative would have been to implement the binary protocol in a C library and interface with it through JNI. But this comes at a loss of performance which is unacceptable even in the control path.

Some RDMA devices require extra parameters to be passed to the kernel driver during control operations. To accommodate those needs, jVerbs/mem invokes the *user driver interface* before interacting with the kernel.

6.1.3 Stateful Verb Calls in jVerbs/mem

In jVerbs/mem we make use of SVCs to cache all the serialization state that occurr within the user driver as part of writing the memory mapped queues. For instance, work requests are cached as a byte array in a serialized form for the target device. Upon executing the given verb call, the serialized byte sequence is unfolded into the right position within the memory mapped device queue.

6.2 jVerbs/nat

jVerbs/nat is an alternative implementation of the jVerbs interface working directly with the OFED RDMA user libraries. This has the advantage that jVerbs/nat can be used with any RDMA capable network interface supported by OFED. The challenge was to integrate with the native OFED user libraries without paying the JNI latency penalty.

6.2.1 Native Integration

The key insight from section 4 was that the cost of calling a JNI function is significant when passing complex parameter types, but neglectable when passing base type parameters. In jVerbs/nat (see Figure 4) we use JNI to bridge (jverbs.so) between Java and the native libraries

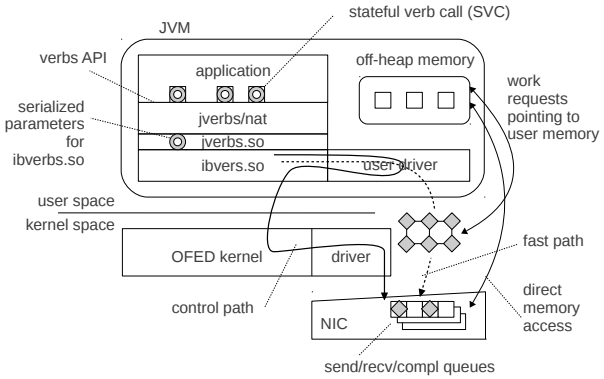

Figure 4: Architecture of the jVerbs/nat implementation.

(ibverbs.so), but we make sure all the JNI function calls only take a small number of base type parameters as input.

To illustrate the approach let us walk through the call stack of jVerbs/nat for the case of the `jv_post_send` verb call. An application calls `jv_post_send` with a list of work request referencing memory regions to be transmitted to a remote host. In jVerbs/nat we want to call the native `post_send` API function available in the OFED user library. Calling `post_send` via a regular JNI interface would, however, impose a significant latency overhead due to marshalling and de-marshalling of the work requests at the JNI border. Instead, in jVerbs/nat we serialize the work request directly into off-heap memory using the native C memory layout. This memory layout exactly matches the memory layout of the C parameters that are used for calling the native `post_send` API function call. With the work requests properly serialized in off-heap memory, passing them to the native OFED function (`post_send`) simply involves a JNI call containing a base type parameter holding the address to the off-heap representation of the parameter. The advantage is that the Java parameters only have to be serialized once, avoiding the intermediate JNI serialization.

6.2.2 Stateful Verb Calls in jVerbs/nat

The concept of stateful verb calls is used in jVerbs/nat to cache the aforementioned serialized parameters prepared and ready to use in off-heap memory. That way, repeating jVerbs API calls will not have to undergo the parameter translation process (see Figure 4).

7 Applications

RDMA programming is different from socket programming and often requires major re-thinking of application structures. One key characteristic of RDMA is that it

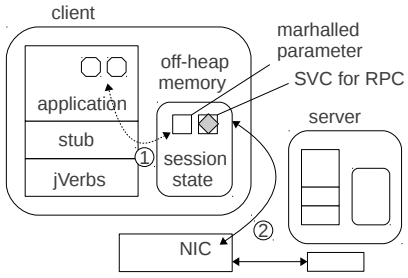

Figure 5: Zero-copy RPC using jVerbs (skipping details at server): 1) marshalling parameters into off-heap memory 2) zero-copy transmission of parameters and RPC header using scatter/gather RDMA `send`.

cleanly separates data from control operations. This is done for good reason, namely because it allows applications to push most of their control operations out of the critical path. With the control state being set up properly, applications can benefit from the latency advantages of fast RDMA data operations. In the following, we describe the implementation of two systems explicitly designed to achieve low-latencies using jVerbs.

7.1 Low-latency RPC

Remote Procedure Call (RPC) is a popular mechanism to invoke a function call in a remote address space [5]. Data center software infrastructure and applications heavily rely on RPC as a key mechanism for accessing their services. Remote Method Invocation (RMI) is Java's API and toolset for performing the object oriented equivalent of RPC. RMI is based on TCP/IP sockets and is therefore not optimized for high-performance interconnects with user-level networking interfaces. We have developed jvRPC, a simple prototype of an RDMA-based RPC system for Java. jvRPC makes use of RDMA `send/recv` operations and scatter/gather support. The steps involved in a RPC call are illustrated in Figure 5.

Initialization: First, both client and server set up a session for saving the RPC state across multiple calls of the same method. The session state is held in off-heap memory to make sure it can be used with jVerbs operations.

Client: During an RPC call, the client stub first marshalls parameter objects into the session memory. It further creates an SVC object for the given RPC call using jVerbs and caches it as part of the session state. The SVC object represents a `jv_post_send()` call of type `send` with two scatter/gather elements. The first element points to a memory area of the session state that holds the RPC header for this call. The second element points to the marshalled RPC parameters. Executing the RPC call then comes down to executing the SVC object. Since the SVC object is cached, subsequent RPC calls of the same session only require marshalling of the parameters but not the re-creation of the serialization state of the verb call.

The synchronous nature of RPC calls allow jvRPC to optimize for latency using RDMA polling on the client side. Polling is CPU expensive though it leads to significant latency improvements. jvRPC uses polling for lightweight RPC calls and falls back to blocking mode for compute-heavy function calls.

Server: At the server side, incoming header and RPC parameters are placed into off-heap memory by the RDMA NIC from where they get de-marshalled into on-heap objects. The actual RPC call may produce return values residing on the Java heap. These objects together with the RPC header are again marshalled into off-heap session memory provided by the RPC stub. Further, an SVC object is created and cached representing a send operation back to the client. The send operation transmits both header and return values in a zero-copy fashion. Again, RDMA's scatter/gather support allows the transmission of header and data with one single RDMA operation.

RPC based on user-level networking is not a new idea, similar approaches have been proposed in [4, 6]. jvRPC, however, is specifically designed to leverage the semantical advantages of RDMA and jVerbs (e.g., scatter/-gather, polling, SVCs). In Section 8 we show that jvRPC achieves latencies that are very close to the raw network latencies of RDMA interconnects.

7.2 Low-latency Memcached

Memcached is a prominent in-memory key/value store often used by web applications to store results of database calls or page renderings. The memcached access latency directly affects the overall performance of web applications. Memcached supports both TCP and UDP based protocols between client and server. Recently, RDMA-based access to memcached has been proposed [9]. The work focuses on using software-based RDMA to lower the CPU footprint of the memcached server. The same architecture when used with hardware-supported RDMA, however, can be used to reduce the access latency of memcached clients. We therefore used jVerbs to implement a Java client accessing memcached through an RDMA-based protocol as described in [9].

Basic idea: The interaction between the Java client and the memcached server is captured in Figure 6. First, the memcached server makes sure it stores the key/value pairs in RDMA registered memory. Second, clients learn about the remote memory identifiers (in RDMA termi-

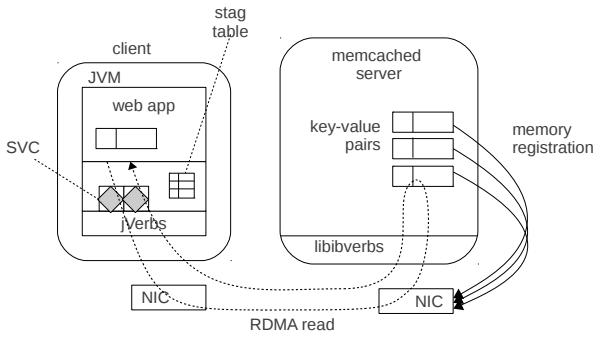

Figure 6: Low-latency memcached access for Java clients using jVerbs.

nology also called `stags`) of the keys when accessing a key for the first time. And third, clients fetch key/-value pairs through RDMA read operations (using the previously learned memory references) on subsequent accesses to the same key.

Get/Multiget: Using RDMA `read` operations for accessing key/value pairs reduces the load at the server due to less memory copying and fewer context switches. More important with regard to this work is that RDMA `read` operations provide clients with ultra-low latency access to key/value pairs. To achieve the lowest possible access latencies, the memcached client library makes use of jVerbs SVC objects. A set of SVC objects representing RDMA read operations are created at loading time. Each time a memcached `GET` operation is called, the client library looks up the `stag` for the corresponding key and modifies the cached SVC object accordingly. Executing the SVC object triggers the fetching of the key/value pair from the memcached server.

Memcached also has the ability to fetch multiple key/-value pairs at once via the `multiget` API call. The call semantics of `multiget` match up well with the scatter/-gather semantics of RDMA, allowing the client library to fetch multiple key/value pairs with one single RDMA call.

Set: Unlike in the GET operation, the server needs to be involved during memcached SET to insert the key/-value pairs properly into the hash table. Consequently, a client cannot use one-sided RDMA `write` operations for adding new elements to the store. Instead, adding new elements is implemented via `send/recv` operations. Nevertheless, objects can be transmitted without intermediate copies at the client side if they are marshalled properly into off-heap memory before.

8 Evaluation

In this section we discuss the performance of jVerbs by first analyzing its performance for basic RDMA operations, and then looking at more complex applications running on top of jVerbs. Before presenting our results, it is important to describe the hardware setup used for evaluation in detail.

8.1 Test Equipment

Experiments are executed on two sets of machines. The first set comprises two machines connected directly to each other. These machines are equipped with a 8 core Intel Xeon E5-2690 CPU and a Chelsio T4 10 Gbit/s adapter with RDMA support. The second set comprises two machines connected through a switched Infiniband network. These machines are equipped with a 4 core Intel Xeon L5609 CPU and a Mellanox ConnectX-2 40 Gbit/s adapter with RDMA support. We have used the Ethernet-based setup for all our experiments except for the one discussed in Section 8.4. We have further used an unmodified IBM JVM version 1.7 to perform all the experiments shown in this paper. As a sanity check, however, we have repeated several experiments using an unmodified Oracle JVM version 1.7 without having seen any major performance differences. We use the jVerbs/mem implementation throughout the evaluation. However, we again repeated most of the experiments also with jVerbs/nat without noticeable performance difference.

8.2 Basic Operations

In this section we are showing the latency performance of the different RDMA operations available in jVerbs and demonstrate the gain they add compared to traditional Java/sockets. The measured latency numbers discussed in this section are captured in Figure 7. The benchmark is measuring the round-trip latency for messages of varying sizes. Five bars are shown for each of the measured data sizes representing five different experiments. Each data point represents the average value over 1 million runs. In all our experiments, the standard deviation across the different runs was small enough to be neglectable, thus, we decided to omit reporting error bars. We used Java/NIO to implement the socket benchmarks. For a fair comparison, we use data buffers residing in off-heap memory for both the jVerbs and the socket benchmarks.

Sockets: The Java socket latencies are shown as the first bar among the five bars shown per data size in Figure 7. We measured socket latencies of $59\mu s$ for 4 byte data buffers, and $95\mu s$ for buffers of size $16K$. Those numbers allow us to put the upcoming jVerbs latencies

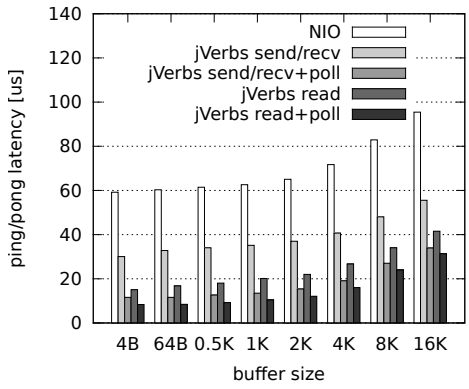

Figure 7: Round-trip latencies of basic jVerbs operations compared to traditional socket networking in Java.

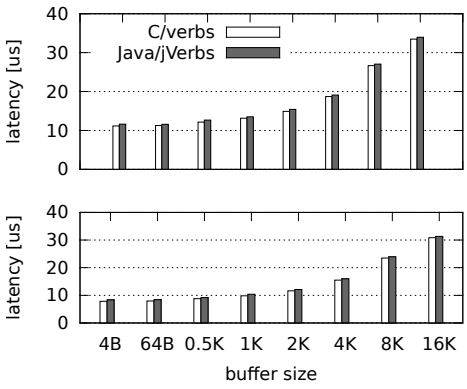

Figure 8: Comparing latencies using the native C verbs interface with jVerbs, top: `send/recv` operation with polling, bottom: `read` operation with polling. jVerbs adds negligible performance overhead.

into perspective. A detailed performance analysis of the Java/NIO stack, however, is outside the scope of this work.

Two-sided operations: Two-sided operations are the RDMA counterpart of traditional socket operations like `send()` and `recv()`. As can be observed from Figure 7, two-sided operations in jVerbs achieve a roundtrip latency of $30\mu s$ for small buffers and $55\mu s$ for larger buffers. This is about 50% faster than Java/sockets. Much of this gain can be attributed to the zero-copy transmission of RDMA `send/recv` operations and its offloaded transport stack.

Polling: One key feature offered by RDMA and jVerbs is the ability to poll a user mapped queue to determine when an operation has completed. By using polling together with two-sided operations we can bring down the latencies by an additional 65% (see third bar in Figure 7).

One-sided operations: One-sided RDMA operations provide a semantic advantage over traditional rendezvous based socket operations. The performance advantage of one-sided operations is demonstrated by the last two bars shown in Figure 7. These bars represent the latency numbers of a one-sided `read` operation, once used in blocking mode and once used in polling mode. With polling, latencies of $7\mu s$ can be achieved for small data buffers, whereas latencies of $31\mu s$ are achieved for buffers of size $16K$. This is another substantial improvement over regular Java sockets. Much of this gain comes from the fact that no server process needs to be scheduled for sending the response message.

8.3 Comparison with Native Verbs

One important question regarding the performance of jVerbs is how it compares with the performance of the native verbs interface in C. For this reason, we compared all the benchmarks of Section 8.2 with their C based counterparts. Throughout those experiments, the performance difference has never exceeded 5%. In Figure 8 we compare the latencies of native C verbs with jVerbs for both `send/recv` and `read` operations. In both experiments polling is used, which puts maximum demands on the verbs interface. The fact that jVerbs performance is at par with the performance of native verbs approves the design decisions made in jVerbs.

8.4 Different Transports

RDMA is a networking principle which is independent of the actual transport that is used for transmitting the bits. Each transport has its own performance characteristics and we have already shown the latency performance of jVerbs on Chelsio T4 NICs. Those cards provide an RDMA interface on top of an offloaded TCP/IP/Ethernet stack, also known as iWARP. In Figure 9, we show latencies of RDMA operations in jVerbs for two alternative transports: Infiniband and SoftiWARP.

In the case of Infiniband, the latency numbers for small 4 bytes buffers outperform the socket latencies by far. This discrepancy, described in Section 2, is due to Infiniband being optimized for RDMA but not at all for sockets. The gap increases further for larger buffer sizes of 16K. This is because RDMA operations in jVerbs are able to make use of the 40Gbit/s Infiniband transport, whereas socket operations need to put up with the Ethernet emulation on top of Infiniband. The latencies we see on Infiniband are a good indication of the performance landscape we are going to see for RDMA/Ethernet in the near future. In fact, the latest iWARP NICs today provide

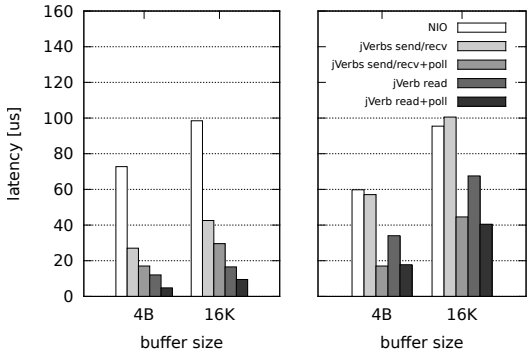

Figure 9: Comparing latency performance of jVerbs on Infiniband (left) and SoftiWARP/Ethernet (right) to Java/NIO operations.

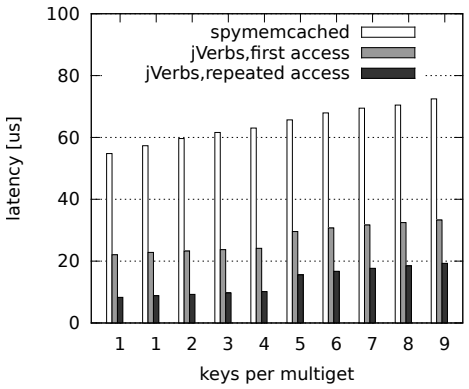

Figure 10: Memcached access latencies for different Java clients.

RPC call	RMI	jvRPC	Speedup
`foo()`	$101\mu s$	$16.6\mu s$	6
`foo(class)`	$149\mu s$	$19.3\mu s$	7.7
`foo(byte[1K])`	$193\mu s$	$22.9\mu s$	8.4
`foo(byte[16K])`	$591\mu s$	$61.5\mu s$	9.6

Table 3: RPC latencies for Java/RMI and jvRPC

40 Gbit/s bandwidth and latencies below 3μs.

The right side of Figure 9 compares the latencies of using jVerbs on SoftiWARP [1] and compares the numbers to standard Java sockets. SoftiWARP is a software-based RDMA device implemented on top of kernel TCP/IP. In the experiment we run SoftiWARP on top of a standard Intel 10 Gbit/s NIC. SoftiWARP does not achieve latencies that are as low as those seen by Infiniband. Using jVerbs together with SoftiWARP, however, still provides a significant performance improvement compared to standard sockets (70% in Figure 9). This is appealing since one big advantage of SoftiWARP is that it runs on any commodity NIC.

8.5 Applications

We are testing jVerbs in the context of the two applications described earlier.

jvRPC: We have implemented jvRPC by manually writing RPC stubs for a set of function calls. In Table 4 we compare the latency performance of jvRPC with Java RMI. As can be seen, jvRPC achieves RPC latencies of 16μs for simple void functions, and around 19-61μs for calls taking objects and large byte arrays as parameters. On the other hand, the RMI latencies range from 100 to almost 600μs for the same function calls. Overall, jvRPC performs a factor of 6-10 times faster than RMI. Interestingly, the gap increases with increasing size of the RPC parameters. Here, the data copying inside the RMI stack including the copying inside the NIO subsystem may cause some of these latency overheads. Again, a detailed performance analysis of Java/RMI is outside the scope of this work. One important observation, however, is that the RPC latencies of jvRPC are only marginally higher than the fastest `send/recv` latencies that can be achieved (see Figure 7). The small latency penalty comes from marshalling parameters into off-heap memory and from the fact that polling can be used only at the client side while waiting for the RPC response. Using polling at the server side would decrease the latency further, but is inefficient since the server does not know the time frame during which an RPC call is coming in.

Memcached/jVerbs: Figure 10 shows the latencies of accessing memcached from a Java client. We compare two setups: (a) accessing unmodified memcached using a standard Java client library [2] and (b) accessing a modified memcached via RDMA using the jVerbs-based client library as discsussed in Section 7.2. The RDMA-based setup is further subdivided into two cases, (a) accessing keys for the first time and (b) repeated access of keys. In the benchmark we measure the latency of a `multiget` operation with increasing numbers of keys per operation. As can be observed from the Figure, standard memcached access from Java takes between 55 and 72 μs depending on the number of keys in the `multiget` request. With RDMA-based memcached access this number reduces to 21-32 μs if keys are accessed for the first time, and 7-18 μs for repeated access to keys. These latencies are very close to the raw network latencies of `send/recv` and `read` operations that are used to implement key access in each of the cases.

One important observation from Figure 10 is that the benefits of RDMA increase with the increasing num-

ber of keys per `multiget` operation. This shows that RDMA scatter/gather semantics are a very good fit for implementing the `multiget` operation. Overall, using RDMA reduces Java-based access to memcached substantially – by a factor of 10 compared to a standard client.

9 Conclusion

In this paper we have presented jVerbs, a library framework offering ultra-low latencies for Java applications. Our measurements show that jVerbs provides bare-metal network latencies in the range of single digit microseconds for small messages – a factor of 2-8 better than the traditional Java socket interface on the same hardware.

References

[1] Softiwarp. http://www.gitorious.org/softiwarp.

[2] Spymemcached. http://code.google.com/p/spymemcached/.

[3] ADIGA, N., ET AL. An overview of the bluegene/l supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference* (nov. 2002), p. 60.

[4] BILAS, A., AND FELTEN, E. W. Fast rpc on the shrimp virtual memory mapped network interface. *J. Parallel Distrib. Comput. 40*, 1 (Jan. 1997), 138–146.

[5] BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.

[6] CHANG, C., AND VON EICKEN, T. A software architecture for zero-copy rpc in java. Tech. rep., Ithaca, NY, USA, 1998.

[7] FREY, P. Zero-Copy Network Communication: An Applicability Study of iWARP beyond Micro Benchmarks. In *Dissertation submitted to ETH ZURICH*.

[8] HILLAND, J., CULLEY, P., PINKERTON, J., AND RECIO, R. Rdma protocol verbs specification (version 1.0). http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf.

[9] STUEDI, P., TRIVEDI, A., AND METZLER, B. Wimpy nodes with 10gbe: leveraging one-sided operations in soft-rdma to boost memcached. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 31–31.

[10] TRIVEDI, A., METZLER, B., AND STUEDI, P. A case for rdma in clouds: turning supercomputer networking into commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (New York, NY, USA, 2011), APSys '11, ACM, pp. 17:1–17:5.

[11] WELSH, M., AND CULLER, D. Jaguar: Enabling efficient communication and i/o in java. In *Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications* (1999).