jVerbs: Ultra-Low Latency for Data Center Applications

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

Network latency has become increasingly important for data center applications. Accordingly, several efforts at both hardware and software level have been made to reduce the latency in data centers. Limited attention, however, has been paid to network latencies of distributed systems running inside an application container such as the Java Virtual Machine (JVM) or the .NET runtime.

In this paper, we first highlight the latency overheads observed in several well-known Java-based distributed systems. We then present jVerbs, a networking framework for the JVM which achieves bare-metal latencies in the order of single digit microseconds using methods of Remote Direct Memory Access (RDMA). With jVerbs, applications are mapping the network device directly into the JVM, cutting through both the application virtual machine and the operating system. In the paper, we discuss the design and implementation of jVerbs and demonstrate how it can be used to improve latencies in some of the popular distributed systems running in data centers.

Categories and Subject Descriptors

D.4.4 **[Operating Systems]:** Communications Management – Network Communication; D.4.7 **[Operating Systems]:** Organization and Design – Distributed Systems

General Terms

Design, Performance, Measurements

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1. http://dx.doi.org/**10.1145/2523616.2523631**

1 Introduction

There has been an increasing interest in low latency for data center applications. One important class of applications driving this trend are real-time analytics such as Shark [16], Dremel [21] or Cloudera's Impala [2]. These systems are often compositions of independent services (e.g, configuration service, locking service, caching service, storage service, etc.) operating on hundreds or thousands of servers. Moreover, processing queries on these systems may require sequential access to these services. In such scenarios, keeping individual access latencies as low as possible is key to reducing the response time experienced by the user.

Several attempts to reduce the latency of individual network requests have been made at both the hardware and the software level.

For instance, Alizadeh et al. show how to reduce network latency by capping link utilization [8]. Vattikonda et al. propose a TDMA-based Ethernet protocol which due to reserved time slots helps to reduce RPC latencies in data centers [26]. An application-centric perspective is taken in RamCloud [23], where the entire data set of an application is held in memory with the goal of reducing data access latencies for applications.

One aspect being ignored by all these works is that today many distributed systems deployed in data centers are actually written in managed languages and run in application-level virtual machines such as the Java Virtual Machine (JVM) or the .NET runtime. Among the examples of such systems are many highly popular applications like Hadoop, HDFS, Zookeepr, DryadLINQ, etc. The latency penalty imposed by the network stack of the application virtual machine is significant. Consequently, the latencies experienced by applications running in such a managed runtime are typically much higher than the latencies experienced by corresponding applications running natively on the operation system.

In this paper, we present jVerbs, a networking API and library for the JVM providing up to a factor of ten lower latencies than the default network stack. The performance advantages are based on two aspects. First – and in contrast to standard sockets – jVerbs offers Remote

Direct Memory Access (RDMA) semantics and exports the RDMA verbs interface. Second, jVerbs maps the networking hardware resources directly into the JVM, cutting through both the JVM and the operating system. Together, these properties allow applications to transfer memory between two JVMs with no extra copy and without operating system involvement.

To achieve the lowest possible latencies, jVerbs requires RDMA-capable networking hardware. Historically, RDMA support has been available only in high-performance interconnects such as Infiniband. As of today, many of the Ethernet-based network cards come with RDMA support.

In the absence of hardware support, jVerbs can be used in concert with software-based RDMA on top of standard Ethernet cards. We show that such a configuration still achieves more than a factor of two better latencies than standard JVM networking.

A key benefit of jVerbs comes from its RDMA interface. Distributed systems can take advantage of the RDMA semantics offered by jVerbs to implement their network operations more efficiently than what is possible today with sockets. For instance, we show how jVerbs can be used to implement fast lookup of key/-value pairs from Memcached without even invoking the server process. Also, we show how jVerbs helps to improve the RPC latencies in Zookeeper and HDFS.

One main challenge in designing jVerbs is to cope with all the latency overheads of penetrating the JVM during network access. At the scale of microseconds, even serialization of function parameters can be costly. To overcome these obstacles, jVerbs employs a technique called *stateful verbs calls* (SVCs) which allows applications to cache and re-use any serialization state occurring during network operations.

In summary, this paper's contributions include (1) an analysis of latency overheads in distributed systems running inside a JVM runtime, (2) the design of jVerbs, in particular the novel approach using stateful verb calls (SVCs) and direct mapping of the network device, and (3) the demonstration of how to use RDMA semantics in jVerbs to lower latencies in several cloud workloads.

2 Motivation

In Figure 1, we analyze latencies of client/server operations available in several popular cloud-based applications. The intent of this experiment is to illustrate (a) how much those latencies lag behind what is potentially possible considering the nature of these operations, and (b) how RDMA can help to reduce the latencies of these cloud applications. All the experiments are executed in a client/server manner between two machines equipped

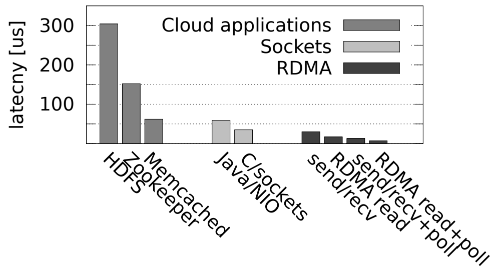

Figure 1: Latencies of networked operations in applications, sockets and RDMA.

with an 8-core Intel Xeon E5-2690 CPU and a Chelsio T4 10 Gbit/s adapter. The cards can be used both in full RDMA and Ethernet only mode. All numbers are averages over 10^6 runs.

The first three bars of Figure 1 show the latencies of operations in the Hadoop Distributed File System (HDFS), Zookeeper and Memcached. For HDFS we measure the latency of a `getBlockLocation()` metadata access, a standard API call exported to applications via the HDFS API. It is used by the Hadoop scheduler to optimize for compute/storage locality, but also as part of regular HDFS read/write operations. A number of real-time applications use HDFS as their storage layer [12, 16]. With fast emerging non-volatile storage devices, metadata operations will become a critical performance factor in these environments. For Zookeeper we measure the latency of an `exist()` operation. This operation checks whether a certain node in the Zookeper tree is valid and is often used by applications to implement locks. In the case of Memcached, the latency refers to the time of accessing a small 32-byte object cached at the server. Here, the Memcached server runs natively, and access is from a JVM-based client using Spymemcached, a popular Java client library for Memcached.

As can be obvserved from Figure 1, the latencies of these three cloud applications range between $150 - 300\ \mu s$. Considering that all of the operations involve exactly one ping-pong message exchange and very little computation on each side, we compare those numbers with the raw roundtrip latency of a ping-pong message exchange implemented using Java sockets. The figure shows that such a message exchange takes $72\ \mu s$. In the case of an HDFS `getBlockLocation()` operation that means about 75% of the time is spent inside the application and about 25% of the time is spent inside the networking stack including the JVM socket implementation. For Zookeeper, the time is equally split between application code and networking code. And for Memcached, almost all of the time is spent for networking.

Bottom line: there are two opportunities to reduce the latency of those applications, (a) by reducing the code path inside the application, and (b) by improving the JVM network latencies itself.

In this paper we show that having RDMA semantics available inside the JVM will help with both problems. First, RDMA is a networking technology providing the lowest networking latencies for native applications today. Depending on which RDMA operation is used (e.g., send/recv, RDMA read, polling), latencies of single-digit microseconds can be achieved (see last four bars in Figure 1). With jVerbs we offer an RDMA interface for the JVM, allowing Java applications to benefit from those ultra-low latencies. Second, the rich semantics of RDMA allow a better integratation of application functionality and network functionality. We will showcase the advantages regarding application integration in Section 7 of this paper.

3 Background

In this section we provide the necessary background information about RDMA, its API, and implementation in Linux. A more detailed introduction to RDMA can be found elsewhere [17].

RDMA Semantics: RDMA provides both send/receive type communication and RDMA operations. With RDMA `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., pinning memory for DMA) and enables data transfers without operating system involvement, which 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 the aforementioned RDMA semantics [19]. 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.

	RTT	JNI costs	Overhead
2000/VIA	$70\mu s$	$20\mu s$	29%
2013/RDMA	$3\text{-}7\mu s$	$2\text{-}4\mu s$	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 [28], and two RDMA operations with complex parameters per network operation (`post_send()/poll_cq()`).

The verbs API also defines **scatter/gather** data access to minimize the number of interactions between applications and RDMA devices.

Security/fairness: Security and performance isolation in RDMA are of particular importance with regard to using RDMA in a shared environment. In terms of security, several options exist. First, memory areas can be grouped into different isolated protection domains. Second, access permissions (e.g., read/write) can be set on a per memory area basis. And last, remote memory access can be allowed/prevented by using standard firewall techniques. In terms of fairness and isolation RDMA does not implement any particular features but when used together with SR-IOV hypervisors can configure NICs to enforce rate limits for each virtual NIC.

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. Control operations involve both the OFED kernel and the 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. The JNI interface, however, has well known weaknesses. First, it is inherently unsafe to execute C code in the context of a Java application. Errors in the C library can crash the entire JVM. Second, the overhead of crossing the boundary between Java and C can be quite high.

The performance problems of the JNI interface have been subject of discussions for many years. In the late 90s as part of the development of the Virtual Interface Architecture (VIA) – a project to allow direct access to the networking hardware from userland – there had been interest in providing networking access for Java through JNI [28]. The numbers reported are in the order of 1 μs for a simple JNI call without parameters, and 10s of μs for more complex JNI calls. Those high performance overheads are caused by the fact that parameters and return values of JNI functions have to be serialized and de-serialized before and after each call.

In the past years, the JNI performance has improved – mostly due more efficient code execution on modern CPUs. In our experiments we found JNI overheads of about 100 *ns* for simple JNI calls without parameters and $1-2$ μs for more complex JNI calls similar to the `post_send()` verb call.

To assess the performance overhead of using JNI in the context of low latency networking operations, one has to set those numbers in relation with network round trip times. The network roundtrip times reported for the VIA were in the order of 70 μs. Today, using modern RDMA capable interconnects roundtrip times of single digit microseconds are possible. We calculate the JNI overhead per roundtrip assuming four JNI/RDMA operations (send/recv on each side) to be required per ping-pong message exchange. This results in a overhead of 29% in the case of the VIA, and 28% – 133% for a configuration using a modern RDMA interconnect. The exact overhead depends on the complexity of the operation. For instance, RDMA scatter/gather operations as offered by `post_send()` and `post_recv()` require more serialization efforts and lead to higher JNI overheads.

There are two main takeaways. First, while JNI latencies have improved in absolute numbers over the past years, the overhead relative to the network latencies has not (see Table 1). This rules out the use of JNI to integrate Java with native RDMA networking. Second, the overhead of serializing complex function parameters needs to be avoided if support for scatter/gather operations is desired. Together, these insights have inspired the design of memory mapped device access and stateful verb calls in jVerbs.

5 Design of jVerbs

The goal of this work is to design a networking framework for the JVM which can provide ultra-low network latencies to cloud-based applications running in a data center. In jVerbs, we are able to achieve this goal by making the following three design decisions: (1) abandon the socket interface in favor of an RDMA verbs interface, (2) directly map the networking hardware resources into the JVM to avoid JNI overhead, and (3) use stateful verb calls to avoid the overhead of repeated serialization RDMA work requests. In the following we describe each of those points in more detail.

5.1 Full RDMA Semantics

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, have been 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 to compromise neither on available communication semantics nor on minimum possible networking latency. jVerbs provides access to all the exclusive RDMA features such as one-sided operations and separation of paths for 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. In Section 7 we illustrate that the raw RDMA semantics in jVerbs is crucial for achieving the lowest latencies in several of the cloud applications discussed. Table 2 lists some of the most prominent verb calls available in jVerbs. The API function are grouped into connection management (CM) operations, verbs operations for data transfer, and operations dealing with state caching (SVC, see Section 5.3).

5.2 Memory-mapped Hardware Access

To achieve the lowest latencies in jVerbs, we employ the same technique as the native C user library when it comes to accessing the networking hardware.

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 makes use of Java's off-heap memory to access these device queues directly from within the JVM. Off-heap memory is allocated in a separate region outside the control of the garbage collector, yet it can be accessed

class	jVerbs API call	description
CM	`createQP()`	returns a new queue pair (QP) containing a send and a recv queue
	`createCQ()`	returns a new completion queue (CQ)
	`regMR()`	registers a memory buffer with the network device
	`connect()`	sets up an RDMA connection
Verbs	`postSend()`	prepares the posting of send work requests (SendWR[]) to a QP, returns an SVC object
	`postRecv()`	prepares the posting of receive work requests (RecvWR[]) to a QP, return an SVC object
	`pollCQ()`	prepares a polling request on a CQ, returns an SVC object
	`getCQEvent()`	waits for completion event on a CQ
SVC	`valid()`	returns true if this SVC object can be executed, false otherwise
	`execute()`	executes the verbs call associated with this serialized SVC object
	`result()`	returns the result of the most recent execution of this SVC object
	`free()`	frees all the resources associated with this SVC object

Table 2: Most prominent API calls in jVerbs (function parameters omitted for brevity).

through the regular Java memory API (ByteBuffer). In jVerbs, we map device hardware resources into off-heap memory using standard memory-mapped I/O. Fast path operations like `postSend()` or `postRecv()` are implemented by directly serializing work requests into the mapped queues. All operations are implemented entirely in Java, avoiding expensive JNI calls or modifications to the JVM.

5.3 Stateful Verb Calls

Even though jVerbs avoids the overhead of JNI by accessing device hardware directly, one remaining source of overhead comes from serializing work requests into the mapped queues. The cost of serialization 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 executing a verb call, jVerbs returns a stateful object that represents a verb call for a given set of parameter values. An application uses `exec()` to execute the SVC object and `result()` to retrieve the result of the call.

One key advantage of SVCs is that they can be cached and re-executed many times. Before executing, however, the application has to verify the SVC object is still in a valid state using the `valid()` function. 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. Once the SVC object no longer needed, resources can be freed using the `free()` API call.

Some SVC objects allow of parameter state to be changed after object creation. For instance, the addresses and offsets of SVC objects returned by `postSend()` and `postRecv()` 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 do not extend the serialization state. Consequently, adding new work requests or scatter/gather elements to a SVC `postSend()` 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 with native RDMA programming in C.

6 Implementation

jVerbs is implemented entirely in Java using 17K LOC. jVerbs is packaged is as a standalone Java library and we have tested it successfully with both the IBM and the Oracle JVM. In this section we highlight several aspects of the implementation of jVerbs in more detail. Figure 3 serves as a reference throughout the section, illustrating the various aspects of the jVerbs software architecture.

```
/* 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)

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

(b)

```
Verbs v = Verbs.open();
/* create SVCs */
RdmaSVC post = v.postSend(sq, wrlist)
RdmaSVC poll = v.pollCQ(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 with SVC objects executed immediately, (c) using jVerbs with SVC objects cached and re-executed.

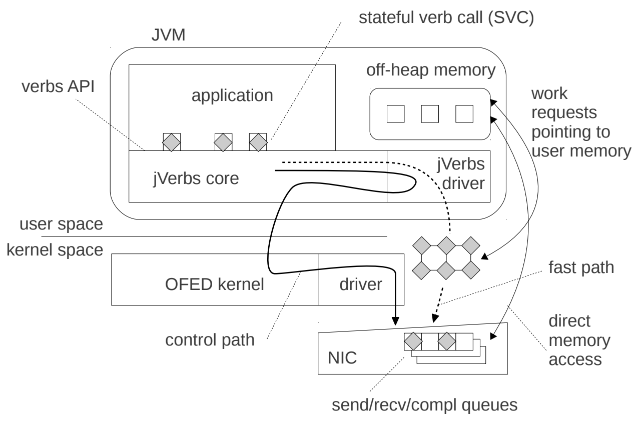

Figure 3: jVerbs user space libraries (core and driver) interacting with the application and the RDMA ecosystem (OFED kernel, driver and NIC).

6.1 Zero-copy Data Movement

Regular Java heap memory used for storing Java objects cannot be used as a source or sink in an RDMA operation as this would interfere with the activities of the garbage collector. Therefore, 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 making sure that network data resides off-heap from the beginning. This is the case in the Memcached implementation we discuss in Section 7, and in several existing applications like Apache DirectMemory [4], Cloudera's MemStore [1], or in the Netty networking framework [3]. If the data cannot be stored off-heap, one might still be able to use off-heap memory during the data serialization process. A good example for this is the RPC framework we built using jVerbs, where the parameters and result values are marshalled into off-heap memory rather than marshalling them into regular heap memory.

6.2 Direct Kernel Interaction

Control operations in RDMA – such as connection establishment or the creation of queue pairs – require kernel involvement. These 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 ensure that no additional overhead is imposed for control operations, jVerbs directly interfaces with the RDMA kernel using standard file I/O. Specifically, jVerbs opens the RDMA device file and communicates with the kernel via the standard RDMA application binary interface (ABI), a binary protocol specified as part of OFED. 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.

6.3 User Drivers

Access to hardware resources in the data path is device specific. Hence, in jVerbs we encapsulate the device specific internals into a separate module 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 that work requests or polling requests are serialized into the right format. Currently, we have implemented user drivers for Chelsio T4, Mellanox ConnectX-2 and SoftiWARP [25].

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. Although such low-level language features are available in C, they were not widely supported in Java for quite some time. 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` is required was for obtaining an off-heap mapping of device memory. This is because Java `mmap()` does not work properly with device files. The Java road map shows that more low-level features will be added in the coming releases, making the development of jVerbs user drivers even more convenient.

7 Applying jVerbs in the Cloud

In the following, we describe the implementation of a latency-optimized communication subsystem for the three cloud applications we used in the motivating examples (Section 2). Both Zookeeper and HDFS employ an RPC type of communication. To reduce the latencies of these applications, we first present the implementation of jvRPC, a low-latency RPC system built using jVerbs. The communication system of Memcached also resembles an RPC architecture and could be accelerated using jvRPC as well. However, given the communication pattern of Memcached, an even more radical approach can be implemented by directly using the one-sided RDMA semantics available in jVerbs.

7.1 Low-latency RPC

Remote Procedure Call (RPC) is a popular mechanism to invoke a function call in a remote address space [10].

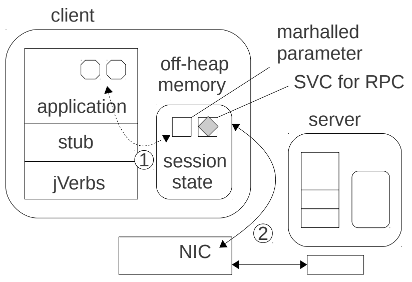

Figure 4: Zero-copy RPC using jVerbs (omitting details at server): 1) marshalling parameters into off-heap memory 2) zero-copy transmission of parameters and RPC header using scatter/gather RDMA `send`.

Zookeeper uses RPC type of communication between clients and the server. In HDFS, an RPC-like communication is used between clients and the namenode node holding the metadata of the system. To reduce the latency in each of those systems, we have developed jvRPC, a simple prototype of an RDMA-based RPC system based on jVerbs. jvRPC makes use of RDMA `send/recv` operations and scatter/gather support. The steps involved in an RPC call are illustrated in Figure 4.

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 ensure that 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 `jpostSend()` 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. As the SVC object is cached, subsequent RPC calls of the same session only require marshalling of the RPC parameters but not the re-creation of the serialization state of the verb call.

The synchronous nature of RPC calls allows jvRPC to optimize for latency using RDMA polling on the client side. Polling is CPU expensive, but it leads to significant latency improvements and is recommended for low-latency environments. 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-

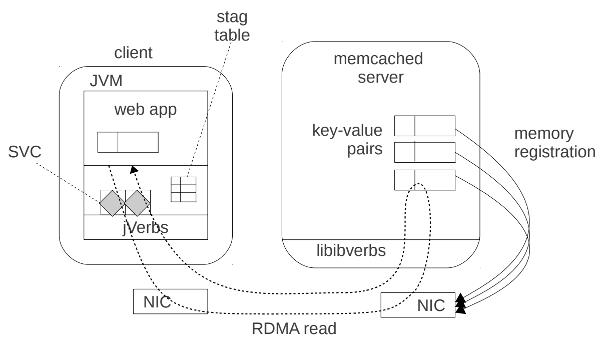

Figure 5: Low-latency memcached access for Java clients using jVerbs.

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 send operation.

RPC based on user-level networking is not a new idea, similar approaches have been proposed [9, 14]. jvRPC, however, is specifically designed to leverage the semantical advantages of RDMA and jVerbs (e.g., scatter/-gather, polling, SVCs). We have integrated jvRPC into a prototype of Zookeeper and HDFS. In Section 8, we show that by using jvRPC, latencies of these cloud services can be reduced to nearly match 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 [22, 24]. We have used jVerbs to implement a client accessing Memcached through an RDMA-based protocol in [24].

Basic idea: The interaction between the Java client and the memcached server is captured in Figure 5. 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 terminology also called `stags`) of the keys when accessing a key for the first time. 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 to access key/value pairs reduces the load at the server because of 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 uses 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 can also fetch multiple key/value pairs at once via the `multiget` API call. The call semantics of `multiget` match 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. Updates for a given key are always stored in a new memory block at the server which avoids conflicts in case of concurrent reads. After an update, the server either switches the stags for the old and the new value, or informs the clients about the new key-to-stag binding (see [22, 24]). From a performance standpoint, objects that are part of an update operation 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 evaluate the latency performance of jVerbs in detail. We first discuss the latencies of basic RDMA operations and later demonstrate the advantages of using jVerbs in the context of existing cloud applications. Before presenting our results, it is important to describe the hardware setup used for the evaluation.

8.1 Test Equipment

Experiments are executed on two sets of machines. The first set comprises two machines connected directly to each other. Both are equipped with a 8 core Intel Xeon E5-2690 CPU and a Chelsio T4 10 Gbit/s adapter with

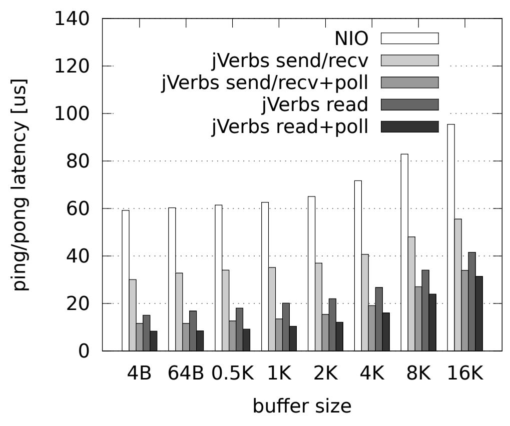

Figure 6: Comparison of round-trip latencies of basic jVerbs operations with traditional socket networking in Java.

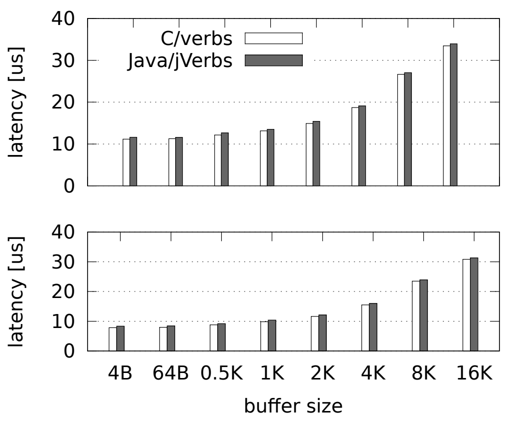

Figure 7: Comparing latencies using the native C verbs interface with jVerbs, top: `send/recv` operation with polling, bottom: `read` operation with polling. jVerbs adds negligible performance overhead.

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 the one discussed in Section 8.5. 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 repeated several experiments using an unmodified Oracle JVM version 1.7, and did not see any major performance differences.

8.2 Basic Operations

We start off by comparing the latencies of different RDMA operations with the latencies of regular socket-based networking.

The measured latency numbers discussed in this section are captured in Figure 6. The benchmark is measuring the roundtrip latency for messages of varying sizes. Five bars representing five different experiments are shown for each of the measured data sizes. 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 negligible, thus, we decided to omit 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 of the five bars shown per data size in Figure 6. We measured socket latencies of 59 μs for 4-byte data buffers, and 95 μs for buffers of size 16K. Those numbers allow us to put the jVerbs latencies 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 6, two-sided operations in jVerbs achieve a roundtrip latency of 30 μs for 4-byte buffers and 55 μs for buffers of size 16K. 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 reduce the latencies by an additional 65% (see third bar in Figure 6). In low latency environments, polling is better than interrupt-based event processing which typically requires costly process context switching.

One-sided operations: One-sided RDMA operations provide a semantic advantage over traditional rendezvous-based socket operations. This performance advantage is demonstrated by the last two bars shown in Figure 6. 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 μs can be achieved for small data buffers, whereas latencies of 31 μs 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.

Operation	no SVC	with SVC	Speedup
send/recv	35μs	30μs	1.16
send/recv+poll	18μs	10μs	1.8
read	17.5μs	15μs	1.17
read+poll	11.2μs	7μs	1.6
10-SG read	28.8μs	18μs	1.6

Table 3: Latency implications of using SVCs for different jVerbs operations. SVCs reduce latencies throughout the band, but are most useful for reads in polling mode and scatter/gather operations.

RPC call	Default	jvRPC	Gains
Zookeeper/exist()	152μs	21.6μs	7.0x
Zookeeper/getData(1K)	156μs	24.3μs	6.4x
Zookeeper/getData(16K)	245μs	50.3μs	4.8x
HDFS/exist()	329μs	28.9μs	11.3x
HDFS/getBlockLocation()	304μs	29.5μs	10.3x

Table 4: RPC latencies in Zookeeper and HDFS, once unmodified and once using jvRCP

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. In those experiments, the performance difference never exceeded 5%. In Figure 7 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 the performance of jVerbs is at par with the performance of native verbs validates the design decisions made in jVerbs.

8.4 Benefits of Stateful Verb Calls

Stateful verb calls (SVCs) are offered by jVerbs as a mechanism for applications to cope with the serialization costs occurring as part of RDMA operations. We have tested the latency implications of using SVCs versus not using the feature for different RDMA operations, and the numbers are reported in Table 3. There are multiple observations that can be drawn from the table. First, SVCs consistently help to reduce latency. Second, not all RDMA operations benefit equally from SVCs. For instance, operations in polling mode benefit more than operations in blocking mode. Operations in blocking mode are generally slower, making the relative overhead of not using SVCs appear smaller. Third, the benefits of SVCs come into play in particular when using scatter/gather operations. Scatter/gather operations require a substantial serialization effort. Using SVCs, this serialization cost can be saved, which results in significant latency gains – up to 37% for a scatter/gather `read` with 10 elements per `postSend()`.

8.5 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 stack, also known as iWARP. In Figure 8, 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-byte buffers outperform the socket latencies by far. This discrepancy is because Infiniband is optimized for RDMA but not at all for sockets. The gap increases further for the larger buffer size of 16K. This is because RDMA operations in jVerbs are able to make use of the 40 Gbit/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 provide 40 Gbit/s bandwidth and latencies around 3 μs.

The right-hand side of Figure 8 shows the latencies of using jVerbs on SoftiWARP [6] and compares the numbers with those of 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 Chelsio NICs in Ethernet mode. 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 over standard sockets (70% in Figure 8). The performance gain comes from zero-copy transmission, memory mapped QPs (jVerbs maps QPs available in SoftiWARP into the JVM just as it does for regular hardware supported RDMA devices) and fewer context switches. The result showcases the performance advantages of jVerbs/SoftiWARP in a commodity data environment.

8.6 Applications

In this section we show the benefits of using jVerbs in cloud-based applications.

Zookeeper and HDFS: We have implemented jvRPC

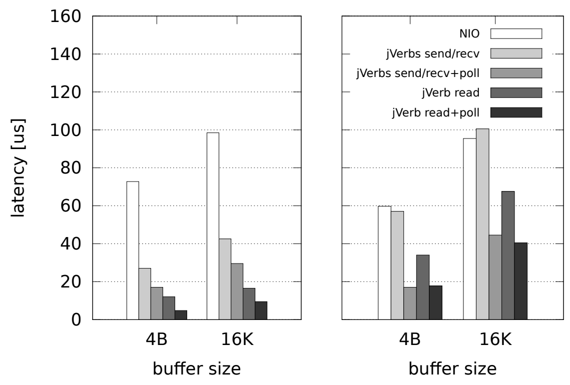

Figure 8: Comparing the latency performance of jVerbs on Infiniband (left) and on SoftiWARP/Ethernet (right) with Java/NIO operations.

RPC call	Insts/ops		IPC
	kernel	user	
Zookeeper/exist()	22K	18K	0.51
Zookeeper/exist()+jvRPC	10K	13K	0.73
HDFS/getBlockLocation	25K	79K	0.38
HDFS/getBlockLocation+jvRPC	11K	45K	0.64

Table 5: Instructions exection profile and instructions/cycle (IPC) for Zookeeper and HDFS, with and without jvRPC.

by manually writing RPC stubs for a set of function calls occurring in Zookeeper and HDFS. As a proof of concept, we have integrated jvRPC into Zookeeper and HDFS as a communication substrate between client and server for those particular function calls.

In Table 4 we compare the latencies of several operations in unmodified Zookeeper and HDFS with the latencies when using jvRPC. As can be seen, using jvRPC the latencies of the listed operations in Zookeeper and HDFS can be reduced substantially. For instance, an `exist()` operation with unmodified Zookeeper takes 152 μs, but takes only 21 μs when jvRPC is used. This corresponds to a factor of seven improvement. Similar speedups are possible for other operations in Zookeeper and HDFS.

One interesting aspect is related to the `getData()` operation in Zookeeper. For better performance we modified Zookeeper to store the data part of each Zookeeper node in off-heap memory. Using jvRPC we can then avoid the extra data copy that takes place in the default implementation of Zookeeper.

Further analysis reveals that jvRPC leads to a shorter code path and a more efficient instruction execution. Table 5 shows the number of instructions executed to handle one RPC operation. The reduced number of instructions (hence shorter code path) in kernel and userspace can be attributed to network offloading and RDMA semantics, respectively. Consequently, higher instructions executed per CPU cycle (IPC) represent better application level performance.

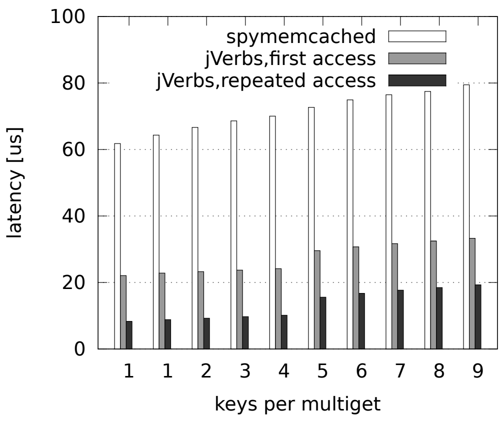

Figure 9: Memcached access latencies for different Java clients.

However, at this point we would like to emphasize that the jvRPC implementation – with its sole focus on the latency – represents an extreme point in the application design spectrum. It demonstrates that by carefully designing the application (e.g. removing slow operations such as memory allocation out of the fast data flow path), it is possible to achieve near raw link latencies for applications running inside a managed runtime such as the JVM.

Memcached/jVerbs: Figure 9 shows the latencies of accessing memcached from a Java client. We compare two setups: (a) accessing unmodified memcached using a standard Java client library [7] 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 number of keys per operation. The figure shows standard memcached access from Java takes between 62 and 79 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 to 7-18 μs for repeated access to keys. These latencies are very close to the raw network latencies of the `send/recv` and `read` operations that are used to implement key access in each of the cases.

One important observation from Figure 9 is that the benefits of RDMA increase with the increasing number 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 ten for repeated single key GET operations.

9 Related Work

The work that is closest to jVerbs is Jaguar [28], which implements user-level networking in Java based on the virtual interface architecture (VIA). This work was done in the late 90s with preconditions (network latencies, CPU speed, Java language features, user-level networking stacks) that were quite different from the technological environment we have today. jVerbs differs from Jaguar in two ways. First, jVerbs is a standalone library that works in concert with any JVM. This is different from Jaguar, which is highly integrated with the JVM and requires modifications to the JIT compiler. Second, jVerbs offers full RDMA semantics that go beyond the feature set of BerkeleyVIA [13] used in Jaguar.

In [15] the authors propose Java-based access to the VIA architecture using either native methods implemented in a C library or, alternatively, through modifications to the Java compiler. Unfortunately, the JNI-based approach imposes a latency penalty and the proposed compiler modifications tie the implementation to a certain JVM.

The VIA architecture is one out of several approaches for user-level networking developed in the 90s. Other works include Shrimp [11], or U-net [27]. These systems have influenced and shaped the RDMA technology of today. We believe that the concepts behind jVerbs – although applied to RDMA – are generic and could be used to enable efficient JVM access for other user-level networking systems.

The Socket Direct Protocol (SDP) is a networking protocol implemented on RDMA and enabling zero-copy and kernel-bypass for standard socket applications [5]. Because of the restrictions of the socket-interface, however, SDP cannot provide the ultra-low latencies of raw RDMA verbs. Recently, Open Fabrics has ended their support for SDP in favor of rsockets [18]. But similar to SDP, rsockets do not offer the semantics necessary for ultra-low latencies. Moreover, currently there is no Java support available for rsockets.

There have been early proposals for RPC based on user-level networking including work for Java [9, 14]. These works are not that different from jvRPC described in Section 4. We believe, however, that a full exploitation of the RDMA semantics – such as proposed by jvRPC – is necessary to provide RPC latencies in the $15-20\ \mu$s range. The performance measurements of jvRPC show that such latencies are possible in the context of real applications like Zookeeper and HDFS.

Accelerating memcached access using RDMA has been investigated in [20, 24]. The work in [20] integrates memcached access through some additional transparency layer and adds some latency overhead. In this work we have extended the pure verbs-based approach for accessing memcached from Java [24]. The performance measurements show that such an approach leads to access latencies in the single microseconds range – a factor of 10 faster than standard memcached access in Java.

10 Conclusion

Lately, there has been a lot of important research looking into reducing latencies inside the data center, both at the hardware level (e.g., switches, MAC protocol) and at the application level (e.g., in-memory storage). In this paper, we have presented jVerbs, a library framework offering ultra-low latencies for an important class of applications running inside a Java Virtual Machine. 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 socket interface on the same hardware. It achieves this by integrating RDMA semantics (with kernel bypass and zero-copy) and carefully managing the JVM run-time overheads. With the help of the new expressive jVerbs API, the raw network access latencies were also successfully translated into improved performance in several popular cloud applications.

To the best of our knowledge, jVerbs is the first transport-independent and portable implementation of RDMA specifications and semantics for the JVM. Confident from our experiences with the development of jVerbs for multiple interconnects and applications, we are now looking into exploring its applicability beyond our client-server setup, such as for large-scale data-processing frameworks involving thousands of nodes. jVerbs has the potential to accelerate the performance of these applications.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Remzi Arpaci-Dusseau, for their helpful comments and suggestions.

References

[1] Apache HBase with MemStore-Local Allocation Buffers. http://blog.cloudera.com/blog/.

[2] Distributed Query Execution Engine using Apache HDFS. https://github.com/cloudera/impala.

[3] Netty: Asynchronous Event-Driven Network Application Framework. http://netty.io.

[4] Off-heap Cache for the Java Virtual Machine. http://directmemory.apache.org.

[5] Socket Direct Protocol http://www.infinibandta.org/specs.

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

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

[8] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association.

[9] A. Bilas and E. W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *J. Parallel Distrib. Comput.*, 40(1):138–146, Jan. 1997.

[10] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, Feb. 1984.

[11] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. *SIGARCH Comput. Archit. News*, 22(2):142–153, Apr. 1994.

[12] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[13] P. Buonadonna, A. Geweke, and D. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–15, Washington, DC, USA, 1998. IEEE Computer Society.

[14] C. Chang and T. von Eicken. A Software Architecture for Zero-Copy RPC in Java. Technical report, Ithaca, NY, USA, 1998.

[15] C.-C. Chang and T. von Eicken. Interfacing Java to the Virtual Interface Architecture. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 51–57, New York, NY, USA, 1999. ACM.

[16] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Fast Data Dnalysis using Coarse-grained Distributed Memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 689–692, New York, NY, USA, 2012. ACM.

[17] P. Frey. Zero-Copy Network Communication: An Applicability Study of iWARP beyond Micro Benchmarks, Dissertation. http://e-collection.library.ethz.ch/view/eth:1653.

[18] S. Hefty. Rsockets, 2012 OpenFabris International Workshop, Monterey, CA, USA. 2012.

[19] J. Hilland, P. Culley, J. Pinkerton, and R. Recio. RDMA Protocol Verbs Specification (Version 1.0). http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf.

[20] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 743–752, Washington, DC, USA, 2011. IEEE Computer Society.

[21] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, Sept. 2010.

[22] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC'13, Berkeley, CA, USA, 2013. USENIX Association.

[23] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[24] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-sided Operations in Soft-RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 31–31, Berkeley, CA, USA, 2012. USENIX Association.

[25] A. Trivedi, B. Metzler, and P. Stuedi. A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 17:1–17:5, New York, NY, USA, 2011. ACM.

[26] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical TDMA for Datacenter Ethernet. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[27] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. ACM.

[28] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. In *Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications*, 1999.