Efficient Messaging for Java Applications running in Data Centers

Kevin Beineke, Stefan Nothaas and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

***Abstract*—Big data and large-scale Java applications often aggregate the resources of many servers. Low-latency and high-throughput network communication is important, if the applications have to process many concurrent interactive queries. We designed DXNet to address these challenges providing fast object de-/serialization, automatic connection management and zero-copy messaging. The latter includes sending of asynchronous messages as well as synchronous requests/responses and an event-driven message receiving approach. DXNet is optimized for small messages (< 64 bytes) in order to support highly interactive web applications, e.g., graph-based information retrieval, but works well with larger messages (e.g., 8 MB) as well. DXNet is available as standalone component on Github and its modular design is open for different transports currently supporting Ethernet and InfiniBand. The evaluation with micro benchmarks and YCSB using Ethernet and InfiniBand shows request-response latencies sub 10 µs (round-trip) including object de-/serialization, as well as a maximum throughput of more than 9 GByte/s.**

Keywords*—*Message passing; Ethernet networks; InfiniBand; Java; Data centers; Cloud computing;

I. Introduction

Today, many interactive applications are built upon very large graphs, e.g., social networks [1], comparing molecular structures in bioinformatics [2] or mobile network state management systems [3]. These graphs consist of billions of small data objects which are typically held in memory to provide low latency access. But, as data volumes grow fast it becomes necessary to aggregate many servers or move to expensive super computers. Usually, big data applications are executed in cloud data centers or on high performance clusters which provide fast networking with 10 GBit/s and beyond. Distributed and parallel processing of in memory data based on very fast networks requires the software stack to be designed carefully, especially if latency is important.

Many big data applications are written in Java and benefit from platform independence and a rich selection of libraries supporting the programmer in designing distributed and parallel applications [1], [4]–[8]. This includes many possibilities to exchange data between Java servers, ranging from high-level Remote Method Invocation (RMI) [9] to low-level byte stream processing using Java sockets [10] or the Message Passing Interface (MPI) for HPC applications [11]. DXNet does not aim at replacing any of those solutions but to rather complement the spectrum.

DXNet is a network library for Java-based applications which has originally been designed for DXRAM a distributed in-memory key-value store and DXGraph a graph processing framework built on top of DXRAM. We provide DXNet as a standalone library through GitHub [12] as we think it could be useful for many other Java-based big data applications.

The contributions of this paper are:

- the DXNet architecture (highly concurrent and transport agnostic);
- zero-copy, parallel de-/serialization of Java objects;
- lock-free, event-driven message handling;
- evaluations with 5 GBit/s Ethernet (Microsoft Azure) and 56 GBit/s Infiniband networks.

The evaluation shows that DXNet efficiently handles high loads with dozens of application threads concurrently sending and receiving messages. Synchronous request/response patterns can be processed in sub 10 µs RTT (Round-Trip Time) with Infiniband transport (including object de-/serialization). And, high throughput is achieved even with smaller payloads, e.g., bandwidth saturation with 1-2 KB payload on InfiniBand and 256 byte payload on Ethernet.

The structure of the paper is as follows: after discussing related work, we present an overview of DXNet in Section III. In Section IV, we describe the lock-free Outgoing Ring Buffer followed by the concurrent serialization in Section V. The next section explains the event-driven processing of incoming data. Sections VII and VIII present thread parking strategies and transport implementation aspects. Evaluation results are discussed in Section IX, followed by the conclusions.

II. Related Work

DXNet combines high-level thread and connection management and a concurrent object de-/serialization with lock-free, event-driven message handling and zero-copy data transfer over Ethernet and InfiniBand (extensible). To the best of our knowledge, no other Java-based network library provides these communication semantics. Because of space constraints, we compare DXNet with the most relevant related work, only.

Distributed shared memory (DSM) is re-gaining attraction due to networks supporting **RDMA** but is not an option for most existing Java applications as DSM requires a different application architecture and an integration in the heap management of the Java Virtual Machine (JVM) [13].

Java's RMI [9] provides a high level mechanism to transparently invoke methods of objects on a remote machine, similar to Remote Procedure Calls (RPC). Parameters are automatically de-/serialized and references result in a serialization of the object itself and all reachable objects (transitive closure) which can be costly [14]. Missing classes can be loaded from

remote servers during RMI calls which is very flexible but introduces even more complexity and overhead. The built-in serialization is known to be slow and not very space efficient [14], [15]. Furthermore, method calls are always blocking.

Manta [16] improves runtime costs of RMI by using a native static compiler. **KaRMI** [17], a drop-in replacement for Java RMI, is implemented in Java without any native code supporting standard Ethernet. KaRMI also replaces Java's built-in serialization reducing overhead and improving overall performance. DXNet does not provide transparent remote method calls but an efficient parallel serialization which avoids copying memory. DXNet is primarily designed for parallel applications and high concurrency, RMI for Web applications and services.

MPI is the state-of-the-art message passing standard for parallel high performance computing and provides very efficient message passing for primitive, derived, vector and indexed data types [18]. As MPI's official support is limited to C, C++ and Fortran, Java object serialization is not provided. Nevertheless, MPI is available for Java applications through implementations of the MPI standard in Java [19] or wrappers of a native library [20].

MPI-2 introduced multi-threading for MPI processes [18] enabling well-known advantages of threads. Prior to MPI-2, intra-node parallelization demanded the execution of multiple MPI processes (and the use of more expensive IPC). To enable multi-threading, the process has to call `MPI_init_thread` (instead of `MPI_init`) and to define the level of thread support ranging from single-threaded execution over funneled and serialized multi-threading to complete multi-threaded execution (every thread may call MPI methods at any time). A lot of effort has been put into the last mode to provide a high concurrent performance [21], [22]. Still, the performance is limited compared to a message passing service designed for multi-threading [21].

One of DXNet's main application domains are ongoing applications with dynamic node addition and removal (not limited to), e.g., distributed key-value stores or graph storages. The MPI standard defines the required functionality for adding and removing processes (over Berkeley Sockets with `MPI_Comm_join` or by calling `MPI_Open_port` and `MPI_Comm_accept` on the server and `MPI_Comm_connect` on the client). Unfortunately, most recent MPI implementations are still not supporting these features entirely [23], [24]. Furthermore, job shutdown and crash handling is also limited [24]. MPI is particularly suitable for spawning jobs with finite runtime in a static environment. DXNet, on the other hand, was designed for up- and down-scaling and handling node failures. In [25], DXNet was used in the in-memory key-value store DXRAM to examine crash behavior and scalability.

High level mechanisms for typical **socket-like interfaces** supporting Gigabit Ethernet (and higher) are provided by Java.nio [26], [27], Java Fast Sockets (JFS) [28] or High Performance Java Sockets [29]. DXNet uses Java.nio to implement a transport for commonly used Ethernet networks.

III. Overview

DXNet relieves programmers from connection management, provides transferring Java objects (beyond plain Java.nio stream sockets) and allows the integration of different underlying network transports, currently supporting reliable verbs over InfiniBand and TCP/IP over Ethernet. In this section, we give a brief overview of the interfaces and functionality of DXNet (see Fig. 1). Further details can be found in the GitHub repository [12].

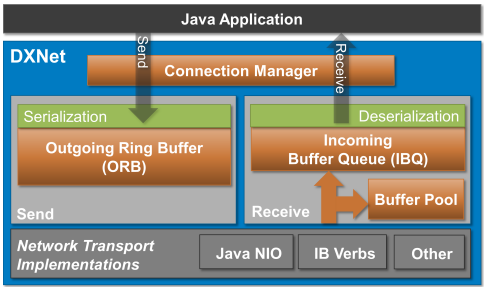

Figure 1. Simplified DXNet Architecture

A. Basic Functionality

Automatic connection management. DXNet abstracts physical network addresses, e.g., IP/Port for Ethernet or GUID for InfiniBand, by using nodeIDs. The aforementioned node address mappings are registered in the library and are mutable for server up- and downscaling. A new connection is opened automatically when a message needs to be sent to another server which is not connected thus far. In case of errors, the library will throw exceptions to be handled by the application. Connections are closed based on a recently used strategy, if the configurable connection limit is exceeded, or in case of network errors which may be reported by the transport layer or detected using timeouts, e.g., absent responses.

Sending messages. DXNet sends messages asynchronously to one or multiple receivers but also provides blocking requests (to one receiver) which return when the corresponding response is received (association of response and requests is transparently managed by DXNet). **Messages are Java objects and serialized** by using DXNet's fast and concurrent serialization (providing default implementations for most commonly used objects, see Section V). The serialization writes directly into the Outgoing Ring Buffer (ORB) which aggregates messages for high throughput (see Section IV) and is allocated outside of the Java heap. Sending data is performed by a decoupled transport thread based on event signaling. DXNet also includes a flow control mechanism, which is not further described here.

Receiving messages. When incoming data is detected by the network transport, it requests a pooled native memory buffer (avoiding to burden the Java garbage collector) and copies the data into the buffer (see Section VI and Fig. 1). The buffer containing the received data is then pushed to the Incoming Buffer Queue (IBQ), a ring buffer storing references on buffers which are ready to be deserialized (see Section VI). The buffer pool and the IBQ are shared among all connections. The buffers of the IBQ are pulled and processed asynchronously by dedicated threads. Message processing includes parsing message headers, creating the message objects and deserializing the payload data. Finally, the **received message is passed back to the application (as a Java object)** using a pre-registered callback method.

A brief overview of DXNet's API is shown in Table I.

TABLE I. DXNET'S APPLICATION INTERFACE

`new DXNet(config,nodeMap)`	initialize/configure (max. connections, server address mappings etc.)

`MyMessage extends Message/Request/Response`
`exportObject(exporter)`
`importObject(importer)`
`sizeOfObject()`
	define message (serializable Java object) by implementing three methods
	serialize message with predefined methods from exporter
	deserialize message with predefined methods from importer
	return payload length

`sendMessage(message)`	send message asynchronously (receivers defined in message instance)
`sendSync(request,timeout)`	send request/response synchronously
`MyReceiver implements MessageReceiver` `onIncomingMessage(message)`	receive messages/requests as Java objects pre-registered callback handler function

B. High Throughput and Low Latency

A key objective of DXNet is to provide high throughput and low latency messaging even for small messages found in many graph applications, for instance. We achieve this with a thread-based and event-driven architecture using lock-free synchronization, zero-copy, and zero-allocation.

Multithreading. All processing steps like serialization, deserialization, message transfer and processing are handled by multiple threads which are decoupled through events allowing high parallelism.

Lock-free event signaling. Dispatching processing events between threads is implemented using lock-free synchronization allowing low-latency signaling. CPU load is managed without impairing latency by parking currently idling threads.

Fast serialization. DXNet implements fast serialization of complex data structures and writes data directly into an ORB. The ORB can be accessed by many threads in parallel and ORBs are not shared between different connections increasing concurrency even more. The processing of incoming messages is also highly scalable because of the event-driven architecture.

Zero copy. DXNet does not copy data for messaging (except de-/serialization). For TCP/IP, we rely on Java's DirectByteBuffers and for InfiniBand on verbs pinning the buffers used by DXNet.

Zero allocation. DXNet uses object pooling wherever possible avoiding time-consuming instance creation and, even more important, not burdening the Java garbage collector which may block an application in case of low memory for multiple seconds.

C. Network Transport Interface

DXNet supports different underlying reliable network transports. The integration of a new transport protocol requires implementing just five methods:

- signal data availability on connection (callback);
- pull data from ORB and send it;
- push received data to IBQ;
- setup a connection;
- close a connection.

IV. LOCK-FREE OUTGOING RING BUFFER

The Outgoing Ring Buffer (ORB) is a key component for outgoing messages and essential for providing high throughput and low latency. The latter is achieved by a highly concurrent approach based on lock-free synchronization.

Each connection has one dedicated ORB allowing concurrent processing of different connections. The ORB itself allows many application threads serializing their outgoing objects concurrently and directly into the ORB. The ORBs are allocated outside of the Java heap in native memory allowing zero-copy sending by the network transport. Directly serializing Java objects into the ORB is more efficient than serializing each object in a separate buffer and combining them later by copying these buffers. The ORB preserves message ordering as given by the application threads and aggregates smaller messages in order to achieve high throughput. We decided to use lock-free synchronization for concurrency control which is more complex but more efficient with respect to latency compared to locks.

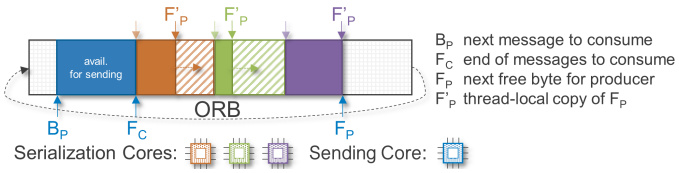

Figure 2. ORB for parallel serialization and aggregating outgoing messages.

A. Basic Lock-Free Approach

The ORB has a configurable but fixed size and is accessed concurrently by several producers (application threads) and one consumer (dedicated transport thread for sending messages). The configurable buffer size limits the maximum number of messages/bytes to be aggregated. For our experiments (see Section IX), we used 1 MB and 4 MB ORBs.

Fig. 2 shows the ORB with three application threads producing data (serialization cores). All pointers move forward from left to right with a wrap around at the end. The white area between F_P and B_P is free memory.

Messages available for sending (fully serialized) are found by the consumer (sending core) between B_P and F_C. The consumer sends aggregated messages and moves B_P forward accordingly but not beyond F_C. All messages between F_C and F_P are not yet ready for sending as parallel serialization is still in progress.

F_P is moved forward concurrently (if the buffer has enough space left) by the producers using a Compare-and-Set (CAS) operation, available in Java through `Unsafe` (see Section IV-C). Therewith, each producer can concurrently and safely store the position of F_P in a local variable F'_P and adjust F_P by its message size. All F'_P pointers (thread-local variables) are used by the associated producer for writing its serialization data concurrently at the correct position in the ORB. The light-colored arrows in Fig. 2 show the starting point of each serialization core (producer) whereas the solid-colored

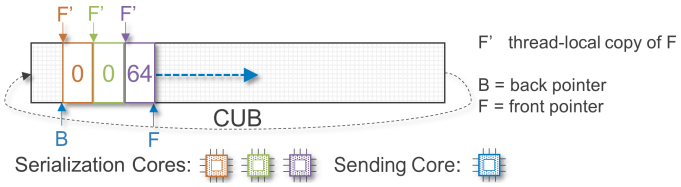

Figure 3. Catch-Up Buffer (CUB). Allowing faster producers returning early and not wasting CPU cycles for waiting.

ones show the current position. In the example, the purple producer finished its serialization first and the green and orange producers are still working.

F_C is moved forward by producers when messages are fully serialized. In Fig. 2, the purple producer finishes before the orange and green ones but cannot set F_C to F_P because the two preceding messages (from the other producers) have not been completely serialized yet. Each producer can easily detect unfinished preceding messages by comparing its starting point (light-colored arrow) with F_C. Obviously, the purple starting point is not equal to F_C. A naive solution lets fast producers wait for slower ones. As we do not want to impact latency we cannot use locks/conditions here. An alternative solution is to busy-poll until all preceding messages have been serialized. Finally, F_C can be set forward and the thread can return.

B. Optimized Lock-Free Solution

The basic solution already avoids the overhead of locks, but with increasing number of parallel serializations the probability of threads having to wait for slower ones increases. The busy-polling can easily overload the CPU. Reducing the polling frequency of producers by sleeping (> 1 ms) or parking (> 10 µs) increases latency too much. Instead, we propose a solution which avoids having fast producers waiting for slower ones by leaving a notice and returning early to the application. This notice includes the message size so that slower producers can move F_P forward for the faster ones. But, message ordering must be preserved.

Our solution is based on another configurable fixed-size ring buffer called Catch-Up Buffer (CUB). As mentioned before, we allocate one ORB for each connection which is now complemented by one associated CUB (e.g., with 1000 entries) for every ORB. The CUB is implemented using an integer array, each entry for one potential left-back notice from faster producers. An entry will be 0 if there is no notice or > 0 representing the message size if a producer finished faster than its predecessors. In the latter case, a slower producer will move forward F_P by the left-back message size.

Fig. 3 shows a CUB corresponding to the ORB shown in Fig. 2. The front pointer F is moved concurrently using a CAS operation (similar to F_P in the ORB). The colored F' are the thread-local copies needed by the producers to leave back a potential notice at the correct position in the CUB. The 64 is a notice from the purple producer (its message size, filled purple box in Fig. 2.) who finished fastest and returned already to the application. The green and orange producers are still working (0 = no notice). If the green producer would now finish before the orange one it would also fill in its message size and return immediately.

If the orange producer finishes next, it moves forward F_C in the ORB as well as B in the CUB (leaving no notice). The green one will do the same, but twice as it will detect the notice (64) after committing its serialization and thus move forward F_C in the ORB by 64 bytes and also B (now pointing to F in the CUB, indicating we are done).

It is important that the order of entries in the ORB and the CUB is consistent, meaning, we need to move forward F and F_P, as well as B and F_C synchronously. We do this, by storing each of those two indexes in one 64-bit long variable in Java and, as the CAS operation is working atomically on 64-bit longs, we can avoid locks.

Two more challenges remain, namely large messages which cannot be serialized at once and a potential ORB overflow during the serialization (both discussed in Section V).

C. Native Memory

The ORB is allocated in native memory (off Java heap) allowing the underlying network transports to send messages without copying them. The class `Unsafe` provides basic methods for memory allocation, memory copy and reading/writing primitives from/into native memory. Furthermore, `Unsafe` is very fast because of extensive optimizations and is widely used in third-party libraries [30].

We favor Unsafe over DirectByteBuffers [27] for two reasons. First, access is faster (e.g., missing boundary checks we already handle on higher level). Second, Unsafe is more versatile because it allows accessing memory which was allocated in C/C++ code (e.g., used for InfiniBand).

V. Serialization

DXNet is designed to send and receive Java objects which need to be de-/serialized from/into a byte stream of messages. The built-in serialization of Java (interface `Serializable`) as well as file-based solutions are too slow and have a large memory footprint [31] (because of automatic un-/marshaling and the use of separators). Other binary serializer like Kyro [32], for instance, either do not support writing directly into native memory or interruptible processing which is needed by DXNet (see Section V-A and V-B). We propose a new serializer addressing all these limitations while still being intuitive to use. The programmer has to implement two interfaces `Importable` and `Exportable`. The former requires implementing the method `importObject()`, the latter `exportObject()` and both `sizeOfObject()`.

A. Export

Exporter. The serialization (or export) of Java objects requires an `exporter` which is passed to `exportObject()`. The exporter class provides default method implementations for the serialization of all primitives, *compact numbers* and Strings and can be extended for supporting custom types (all types can also be arranged in arrays). Compact numbers are coded integers using a variable number of bytes as needed to reduce space overhead.

The exporter writes directly into the ORB by using Unsafe (see Section IV). It stores the start position within the ORB, the size of the ORB and the current position within the message.

Exporting an object involves two steps: exporting the message header (see Fig. 4) which has a fixed size and exporting the variable-sized payload by calling `exportObject()`.

DXNet uses its `default exporter` for serialization which is optimized for performance. It is complemented by

Figure 4. Message header. Cat.: message, request or response; X: exclusive or not (ordering).

two other exporters (described below) for handling messages which do not fit in the ORB without copying buffers.

Buffer overflow. If the end of the ORB will be reached during the serialization of an object, DXNet switches to the `overflow exporter`. The overflow exporter performs a boundary check for each data item of an object and writes bytes with a wrap-around to the beginning of the ORB, if necessary. The resulting message is sent as two pieces over the network stream avoiding copying data.

Large messages. Serialized objects resulting in messages larger than the ORB must be written iteratively. First, the entire unused section of the ORB (see Fig. 2) is reserved and filled with the first part of the message. If the back pointer is reached, the export is interrupted and its current state is stored in an `unfinished operation` instance to allow resuming serialization as soon as there is free space in the ORB again.

Unfinished operation. The instance stores the interrupt position within the message and the rest of the current operation. Depending on the operation, the rest is either a part of a primitive which can be stored in a `long` within the unfinished operation or an object with partly uninitialized fields whose reference can be stored.

Resume serialization after an interrupt. To continue the serialization, the method `exportObject()` is called again (threads return after being interrupted during serialization) and all previously successfully executed export operations are automatically skipped until the position stored in the unfinished operation is reached. The rest of the object is serialized from there (might be interrupted, again). For exporting large messages, the `large message exporter` is used, which extends the overflow exporter.

B. Import

All incoming messages are written into native memory buffers taken from the incoming buffer pool and are pushed to the IBQ (see Section VI). Each buffer contains received bytes (one or several messages) from the connection stream. The underlying network independently splits and aggregates packets resulting in a buffer beginning and ending at any byte within a message. DXNet is able to serialize split messages without copying buffers.

The import works analogously to the export. Messages are deserialized directly from native memory by using Unsafe (message header and payload). The fast `default importer` is complemented by three other importers (described below) for handling split messages. This requires to handle three situations: buffer overflow (tail of message/header missing), buffer underflow (head of a message/header is missing) and both combined.

Buffer overflow. When the buffer's end will be reached before the message is complete, we switch to the `overflow importer`. It does boundary checks and uses the unfinished operation (see Section V-A) when necessary. Furthermore, the serialization is aborted with an `IndexOutOfBoundsException` handled by DXNet

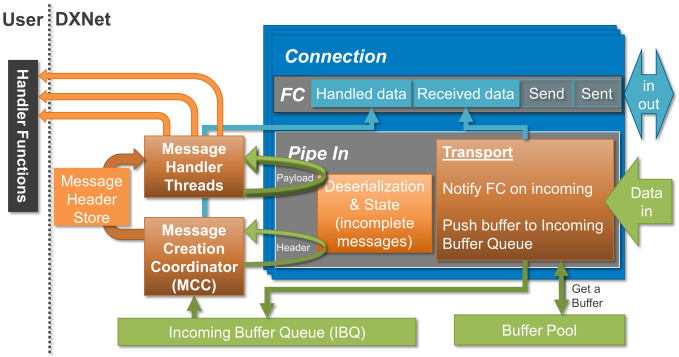

Figure 5. Receiving and processing messages. Green: Native memory access.

avoiding returning invalid values for succeeding operations.

Buffer underflow. This situation occurs after a buffer overflow (on the same stream). It is known apriori and handled by the `underflow importer`, which uses the unfinished operation instance (passed from the overflow importer) containing all information necessary to continue deserialization.

Buffer under- and overflow. When a message's head and tail is missing (likely for large messages), the message is handled by the `underoverflow importer`.

C. Resumable Import and Export Methods

Messages may be split caused by DXNet's buffering or the underlying network. In order to avoid copying buffers, we require both import and export methods to be interruptible and idempotent as they may be called multiple times for one object (to avoid blocking threads, see Sections V-A and V-B). DXNet's importer and exporter methods are sufficient for most object types, but custom object structures must be aware of this and avoid functions causing side effects (e.g., I/O access).

VI. Event-Driven Processing of Incoming Data

Fig. 5 gives an overview of the parallel event-driven processing of incoming data. Like for the ORB, we use multi-threading, lock-free synchronization, zero-copy and zero-allocation to provide high throughput and low latency.

Receiving process. The network transport pulls a buffer from the incoming buffer pool when new data can be received and fills it accordingly. The buffer is then pushed to the IBQ and processed by the **Message Creation Coordinator** thread (MCC) by deserializing the message headers. The message headers are pushed to the **message header store** afterwards. Multiple message handler threads concurrently create the message objects, deserialize the messages' payloads and pass the received Java objects to the application using its registered callback methods. When all data of a buffer has been processed, it is released and pushed back into the incoming buffer pool.

Incoming buffer pool. The buffer pool provides buffers, allocated in native memory, in different configurable sizes (e.g., 8×256 KB, 256×128 KB and 4096×16 KB). The transport pulls buffers using a worst-fit strategy as the amount of bytes ready to be received on the stream is unknown. It can also scale-up dynamically, if necessary.

The buffer pool management consists of three lock-free ring buffers optimized for access of one consumer and N producers (similar to the ORB but without the CUB, see Section IV).

A. Parallel Message Deserialization

Filled buffers are pushed by the transport thread into the IBQ. The IBQ is a basic ring buffer for one consumer and one producer and is synchronized using memory fences. The IBQ may be full and require the transport thread to park for a short moment and retry (see Section VII).

High throughput requires a parallel deserialization. As the received messages of the incoming stream can be split over several incoming buffers (see Section V-B), the buffer processing must be in-order and we need a two-staged approach to enable concurrency. The MCC thread pulls the buffer entries from the IBQ, deserializes all containing message headers (using relevant state information stored in the corresponding connection object) and pushes them into the message header store. Message payload deserialization based on the message headers can then be done in parallel by the message handler threads. This approach is efficient as the time-consuming payload deserialization and message object creation is parallelized.

The deserialization of split messages' payload (last message in buffer, which is not complete) must be in-order as well because all preceding parts of a message must be available to continue the deserialization of a split message. We address this situation by the MCC detecting and deserializing not only the header but the payload fraction within the current buffer, as well, for the split message. The rest of the message in the next buffer can be read by a message handler, again.

Split message headers are not a problem as deserialization of message headers is always done by the MCC which can store incomplete message headers within the connection object and continue with the next buffer.

Message header store. As mentioned before, the MCC pushes complete message headers to the message header store. The latter is implemented as a lock-free ring buffer for *N* consumers and one producer. Synchronization overhead is reduced by the MCC buffering the small message headers and pushing them in batches into the message header store. The batch size is limited but configurable, e.g., 25 headers.

Message header pool. Message headers are pooled, as well, in another single consumer, multiple producers lock-free ring buffer. Furthermore, message headers are pushed and pulled in batches. To reduce the probability of multiple message handler threads returning message headers at the same time, the batch sizes differ for every message handler.

Returning of buffers. A pooled buffer must not be returned before all its messages haven been deserialized. Because of the concurrent deserialization and split messages, we use the MCC incrementing an atomic counter for every message header pushed to the message header store (more precisely, the counter is increased once for every batch of message headers). Accordingly, the message handlers decrement the counter for every deserialized message. When all messages have been deserialized, the buffer can be safely returned to the pool.

We could run out of buffers during high throughput, if the MCC deserializes headers faster than the message handler threads can handle. Although we can scale up the number of incoming buffers, we prefer to throttle the MCC when a predefined number of used buffers is exceeded to reduce the memory consumption. Another benefit of limiting the amount of incoming buffers is that all buffer states like the message counters, the buffers' addresses or the unfinished operations which are filled for incomplete messages can be allocated once and reused for every incoming buffer to be processed.

Message Ordering. DXNet allows applications to mark messages and thus ensure message ordering on a stream/connection. All marked messages are guaranteed to be processed by the same message handler. All other steps preserve message ordering by default. For achieving maximum throughput, marking all messages is not advisable.

VII. Thread Parking Strategies

Lock-free programming allows low-latency synchronization but can easily overload a CPU by uncontrolled polling using CAS operations. DXNet implements a multi-level flow control with explicit message flow regulation and implicit throttling if memory pools drain and queues fill-up. We address three thread situations: blocked (the thread waits for another thread/server finishing its work because a pool is empty or queue full), colliding (failing CAS operation because another thread entered a critical section faster) and idling (the thread has nothing to do and waits for another thread/server committing new work).

Blocked thread. When blocked, the thread can park to reduce the CPU load because it is too fast compared to other threads/servers. However, the thread should not park for a long period to avoid restraining other threads/servers. Experiments showed that a sane park period is between 10 and 100 µs. Java allows minimum parking times of around 10 to 30 µs for a thread with `LockSupport.parkNanos()` for Linux servers with x86 CPUs.

Colliding thread. When colliding, the thread will repeat the CAS operation with updated values until successful because the thread is about to commit something and this should be done as fast as possible. However, reducing the collision probability (e.g., the ORB optimization described in IV-B) reliefs the CPU significantly.

Idling thread. This situation occurs, if a thread has nothing to do at the moment, e.g., a transport thread polls an empty ORB, the MCC polls an empty IBQ or a message handler polls an empty message header store. However, new work events can arrive within nanoseconds. Latency is minimized when threads do not park or yield, but only as long as the CPU is not overloaded. In case of CPU overload situations, parking threads can reduce latency.

We address this with an overprovisioning detection combined with an adaptive parking approach (10 to 30 µs), if the number of active threads (application threads and network threads) reaches a threshold, e.g., four times the number of cores, see also Section IX-A for the evaluation.

Idling for longer periods, e.g., applications not exchanging messages for a longer period of time, must be addressed, too. DXNet detects this, e.g., a network thread idling for one second (configurable time), and starts parking threads, if idling, reducing CPU load to a minimum.

VIII. Transport Implementations

DXNet has an open architecture supporting different network transport technologies. Currently, we have transport implementations for TCP/IP over Ethernet (using Java.nio), reliable verbs over Infiniband (based on JNI), and Loopback (for evaluation). Because of space constraints, we will only sketch some important aspects of these transports.

The `Ethernet` transport (EthDXNet) implementation is based on Java.nio and maps DirectByteBuffers to the ORB allowing to send data without copying it in user-space. Furthermore, two channels are opened for every connection to avoid channel duplication and for providing a side-band flow control channel for each connection. Channel duplication may occur when two servers create connections to each other simultaneously and must be avoided. The second channel allows exchanging flow control messages necessary to maximize throughput on a connection by using the back-channel.

The `InfiniBand` transport accesses the IBDXNet library (C++) using JNI. IBDXNet utilizes ibverbs to implement direct communication using the InfiniBand HCA. IBDXNet uses one dedicated send and one dedicated receive thread, both processing outgoing/incoming data in native memory. Context switching from C++ to Java was designed carefully and is highly optimized to avoid latency.

The `Loopback` transport is used for the experiments in this paper allowing to study the performance of DXNet without any bottlenecks from a real network. Data is not sent over a network device nor the operating system's loopback device (latency would be considerably high) but is directly copied from the ORB to a pooled incoming buffer. Furthermore, the Loopback transport simulates a server sending and receiving messages at highest possible throughput allowing to evaluate DXNet's performance.

IX. Evaluation

We evaluate the proposed concepts using Loopback and three different networks: 1 GBit/s Ethernet, 5 GBit/s Ethernet and 56 GBit/s InfiniBand. The Loopback is used to evaluate DXNet's concepts without any limitations of an underlying network.

Loopback and 5 GBit/s Ethernet tests were run in Microsoft's Azure cloud in Germany Central with up to 18 virtual machines from the type Standard_DS13_v2 which are memory optimized servers with 8 cores (Intel Xeon E5-2673), 56 GB RAM and shared 10 GBit/s Ethernet connectivity (two instances per connect). We deployed a custom Ubuntu 14.04 image with 4.4.0-59 kernel and Java 8. The tests with 1 GBit/s Ethernet and InfiniBand were executed on our private cluster servers with 64 GB RAM, Intel Xeon E5-1650 CPU and Ubuntu 16.04 with kernel 4.4.0-64.

We use a set of micro benchmarks for the evaluations in Sections IX-A and IX-B which send messages or requests of variable size with a configurable number of application threads. All throughput measurements refer to the payload size which is considerably smaller than the full message size, e.g., a 64-byte payload results in 115 bytes to be sent on IP layer when using Ethernet. Additionally, all runs with DXNet's benchmarks are **full-duplex** showing the aggregated performance for concurrently sending and receiving messages/requests.

A. Loopback

As mentioned before, we want to evaluate the efficiency of DXNet's concepts without any network limitations. Fig. 6 shows message processing times and throughputs for different message sizes when using the Loopback transport on a typical cloud server (Standard_DS13_v2). Messages up to 2 KB can be processed in around 500 ns. Larger messages require

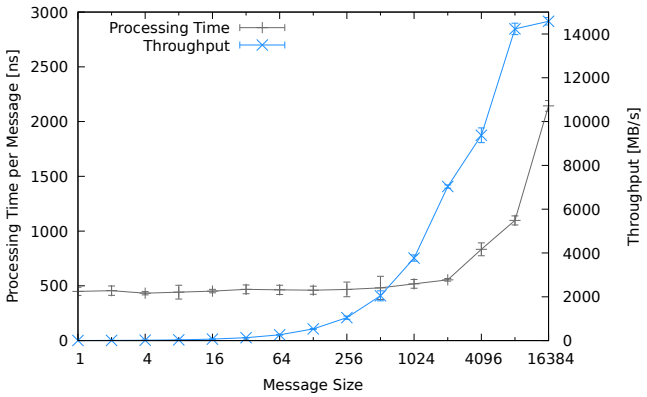

Figure 6. 10^7 Messages, 1 App. Thread, 4 Message Handlers.

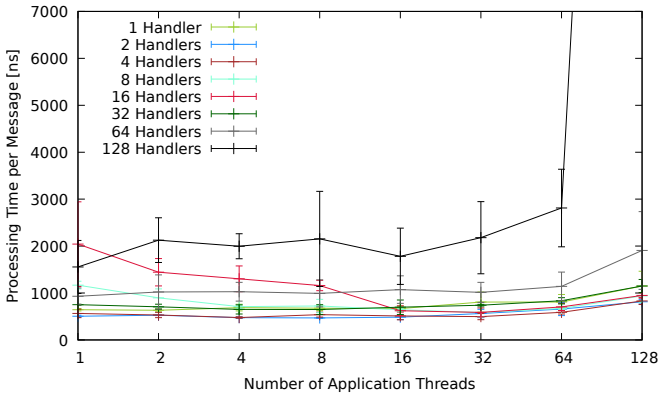

Figure 7. 10^7 Messages, 1024 Bytes Payload.

increasing processing times, as expected. The throughput increases linearly with the message size up to 8 KB messages and is capped at around 14 GByte/s aggregated throughput for sending and receiving of larger messages. The Linux tool `mbw` determined a memory bandwidth of 7.19 GByte/s for a 16 GB array and 16 KB block for the used servers which explains the maximum throughput (saturation of the available memory bandwidth).

In Fig. 6, we studied messages with up to 16 KB payload size as DXNet is primarily designed to perform well with small messages. We also tested larger messages (larger than the ORB, configured with 4 MB here) and measured a message throughput of around 5.4 GByte/s with 8 MB messages. The throughput is lower as application threads and transport thread work sequentially for larger messages (see Section V-A). However, if the application needs to often handle large messages, throughput can easily be improved by using a larger ORB.

DXNet is designed to efficiently support concurrent application threads sending and receiving messages in parallel. Fig. 7 shows that the processing time for 1 KB messages is stable from one to 64 and only slightly increases with 128 application threads. Additionally, Fig. 7 shows the performance with a varying number of message handlers peaking with two to four. Obviously, 128 application threads and 128 message handlers overstress the CPU (8 cores) significantly. The results for all other constellations are as expected showing DXNet's capability of efficiently handling hundreds of concurrent threads.

We also evaluated request-response latency by measuring the **RTT**, which includes sending a request, receiving the

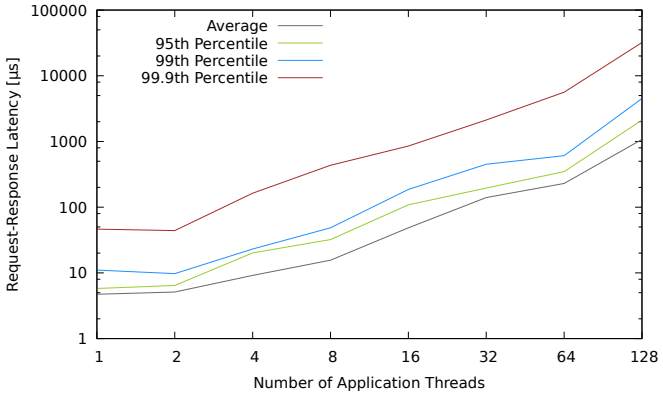

Figure 8. 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

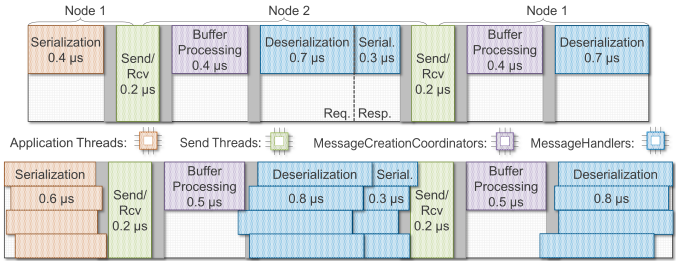

Figure 9. Breakdown of Request-Response Latency for 1024-byte Requests. One application thread (on top) and four (at the bottom). Grey bars indicate inter-thread communication.

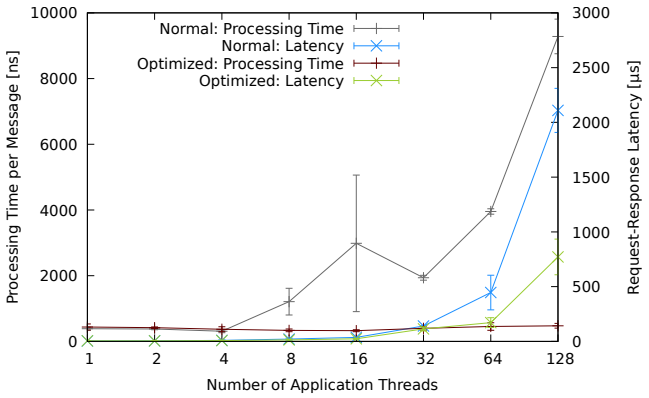

Figure 10. 10^7 Message or 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

request, sending the corresponding response and receiving the response. Fig. 8 shows the latency for small requests with increasing number of application threads. The average RTT with one and two application threads is under 5 µs. With up to eight threads the RTT increases slower than the number of threads because requests can be aggregated for sending. With more threads the increase rate is higher.

Fig. 9 shows the breakdown of request-response latency for one and four application threads and 1024-byte requests. This is a best-effort approximation as time measurement is costly and influences the processing. As expected de-/serialization accounts for the majority of the RTT and deserialization is slower than serialization because of the message object allocation and creation. With more application threads or asynchronous messages, all depicted steps are executed in parallel.

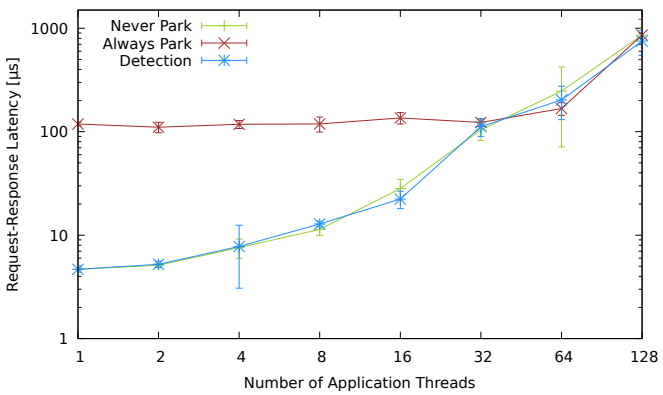

Figure 11. 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

Optimized Outgoing Ring Buffer. The benefits of the CUB, discussed in Section IV, can be seen in Fig. 10. Without the optimization the message processing time increases significantly with more than four application threads sending messages (with 128 threads nearly 20 times higher). Furthermore, the RTT diverges considerably with more than 32 application threads as well.

Overprovisioning Detection. Fig. 11 shows the importance of the thread parking strategy (see Section VII). The RTT is 25 times larger when using one application thread and always parking network threads. All three strategies match with 32 threads and diverge a little with more threads. The never park strategy is at disadvantage with many threads (128) and the RTT is around 100 µs larger than with the adaptive approach.

The evaluation with Loopback transport shows the high throughput and low latency of DXNet. Furthermore, DXNet offers a high stability when used with many threads sending and receiving messages in parallel.

B. *Comparing Network Transports*

Fig. 12 shows the message processing time and throughput for all three network transports (Ethernet and Loopback on cluster and cloud instances) with varying payload size. As expected, InfiniBand has the lowest processing overhead and highest throughput of all physical devices.

The comparison between the 1 GBit/s Ethernet of the private cluster and 5 GBit/s Ethernet in Azure cloud reveals interesting insights. Obviously, message throughput is higher in the cloud for large messages. But, message throughput is higher and processing time is lower on the cluster for messages smaller than 64 bytes which is most likely caused by the virtualization overhead of cloud servers. Loopback is also considerably faster on cluster instances (< 300 ns processing time and > 16 GByte/s throughput).

Fig. 13 shows the request-response latency and throughput for requests sent by four application threads. Again, 1 GBit/s Ethernet on our cluster performs better for small payloads (< 1024) than 5 GBit/s Ethernet in the cloud. For larger requests the bandwidth becomes more and more important favoring the cloud network. Both Ethernet networks are far off the latencies InfiniBand achieves. For small request (< 512 byte payload) the RTT is consistently under 10 µs and rises to only 16 µs for 16 KB requests. Hence, the throughput is much higher with InfiniBand as well.

The evaluation with three physical transports confirms the

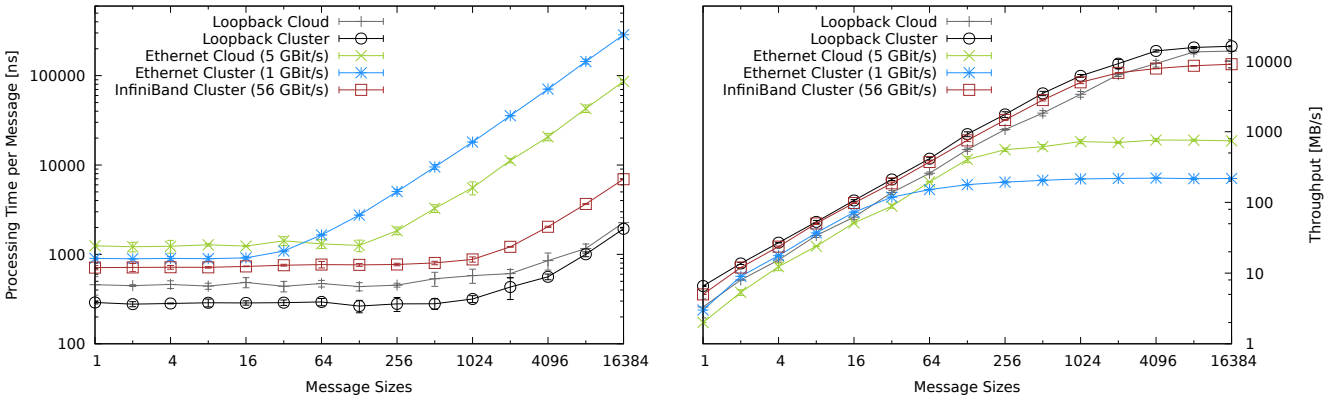

Figure 12. 10^8 Messages, 1 App. Thread, 2 Message Handlers.

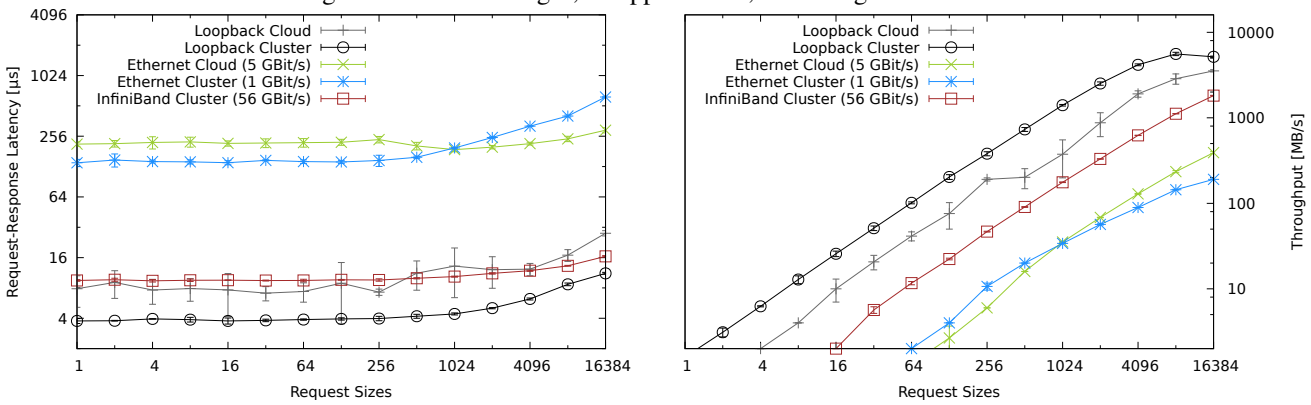

Figure 13. 10^7 Requests, 4 App. Threads, 2 Message Handlers.

results gathered with Loopback and DXNet performs strong especially with InfiniBand (RTT < 10 µs, throughput > 9 GByte/s full-duplex).

C. Yahoo! Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) was designed to quantitatively compare distributed serving storage systems [33]. The benchmark offers a set of simple operations (reads, writes, range scans) and a tabular key-value data model to evaluate online storage systems regarding their elasticity, availability and replication. Furthermore, YCSB is easily extensible for new storage systems and new workloads. For our evaluation, we used the in-memory key-value store DXRAM [34] which utilizes DXNet and created an individual workload: one 64-byte object per key, 10^6 keys, uniform distribution, 90 % read and 10 % write operations, 10^7 operations. The tests were run in the Microsoft Azure cloud with one storage server and an increasing number of client servers (maximum 16) which each hosted up to 80 client threads.

Fig. 14 shows the average operation latency and throughput with 10 to 1280 client threads. The operation latency starts at around 230 µs which is in line with previous latency measurements. The latency grows slowly up to 480 client threads but then exponentially indicating server congestion. The throughput rises up to 640 client threads with more than one million operations per second and remains stable with more client threads.

The evaluation with YCSB shows DXNet's high performance for a client-server scenario (one server can serve more than 1000 clients).

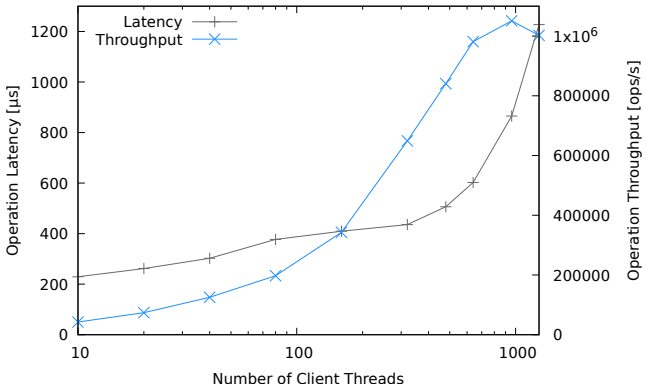

Figure 14. 6 Message Handlers

X. Conclusions

Many big data applications as well as large scale interactive applications are written in Java and aggregate the resources of many servers in a cloud data center, high performance cluster or private cluster. Efficient network communication is very important for these application domains. RMI while being comfortable to use is not fast enough for these applications. Plain sockets are difficult to handle especially if efficiency and scalability need to be addressed. MPI was designed for spawning processes with finite runtime in a static environment. Thus, multi-threading performance and support for adding/removing nodes to an existing environment are limited.

In this paper, we proposed DXNet, a Java open-source network library complementing the communication spectrum.

DXNet provides fast parallel serialization for Java objects, automatic connection management, automatic message aggregation and an event-driven message receiving approach including a concurrent deserialization. DXNet offers high-throughput asynchronous messaging as well as synchronous request/response communication with very low latency. Finally, its architecture is open for supporting different transport protocols. It already supports TCP with Java.nio and reliable verbs for Infiniband. DXNet achieves high performance and low latency by using lock-free data structures, zero-copy and zero-allocation. The proposed ring buffer and queue structures are complemented by different thread parking strategies guaranteeing low latency by avoiding CPU overload.

Evaluations on a private cluster and in the Microsoft Azure cloud show message processing times of sub 300 ns resulting in throughputs of up to 16 GByte/s which saturate the memory bandwidth of a typical cloud instance. For the request/response pattern, DXNet is able to provide sub 10 µs RTT latency using the InfiniBand transport (sub 4 µs over Loopback). Finally, DXNet is also able to efficiently handle highly concurrent processing of many small messages resulting in throughput saturations for Ethernet with 256 bytes payload and InfiniBand with 1-2 KB payload.

The InfiniBand transport IBDXNet is work in progress and final results will be published separately (throughput: >10.4 GByte/s). Future work also includes more experiments at larger scales including comparisons with other network middlewares, as well as evaluations using a 100 GBit/s InfiniBand network.

REFERENCES

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, pp. 1804–1815, Aug. 2015.

[2] M. S. Engler, M. El-Kebir, J. Mulder, A. E. Mark, D. P. Geerke, and G. W. Klau, "Enumerating common molecular substructures," *PeerJ Preprints*, vol. 5, p. e3250v1, Sep. 2017.

[3] P. Satapathy, J. Dave, P. Naik, and M. Vutukuru, "Performance comparison of state synchronization techniques in a distributed lte epc," in *IEEE Conf. on Network Function Virtualization and Software Defined Networks*, 2017.

[4] S. Ekanayake, S. Kamburugamuve, and G. C. Fox, "Spidal java: High performance data analytics with java and mpi on large multicore hpc clusters," in *Proceedings of the 24th High Performance Computing Symposium*, 2016, pp. 3:1–3:8.

[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[6] "Cassandra," http://cassandra.apache.org, accessed: 2018-03-14.

[7] "Interactive query with apache hive on apache tez," http://hortonworks.com/hadooptutorial/supercharging-interactivequeries-hive-tez/, accessed: 2018-03-14.

[8] "Impala - cloudera," https://www.cloudera.com/products/open-source/apache-hadoop/impala.html, accessed: 2018-03-14.

[9] S. Microsystems, "Java remote method invocation specification," https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html, accessed: 2018-03-14.

[10] Oracle, "Package java.net," https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html, accessed: 2018-03-14.

[11] S. Mintchev, "Writing programs in javampi," University of Westminster, Tech. Rep. MAN-CSPE-02, Oct. 1997.

[12] K. Beineke, S. Nothaas, and M. Schoettner, "Dxnet project on github," https://github.com/hhu-bsinfo/dxnet, accessed: 2018-03-14.

[13] W. Zhu, C.-L. Wang, and F. C. M. Lau, "Jessica2: a distributed java virtual machine with transparent thread migration support," in *Proceedings. IEEE International Conference on Cluster Computing*, 2002, pp. 381–388.

[14] S. P. Ahuja and R. Quintao, "Performance evaluation of java rmi: A distributed object architecture for internet based applications," in *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '00, 2000, pp. 565–569.

[15] M. Philippsen, B. Haumacher, and C. Nester, "More efficient serialization and rmi for java," *Concurrency: Practice and Experience*, vol. 12, pp. 495–518, 2000.

[16] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient java rmi for parallel programming," *ACM Trans. Program. Lang. Syst.*, vol. 23, pp. 747–775, Nov. 2001.

[17] C. Nester, M. Philippsen, and B. Haumacher, "A more efficient rmi for java," in *Proc. of the ACM 1999 Conf. on Java Grande*, 1999, pp. 152–159.

[18] M. P. I. Forum, Ed., *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center, 2015, 2015. [Online]. Available: https://books.google.de/books?id=Fbv7jwEACAAJ

[19] A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core hpc systems using java," in *Journal of Parallel and Distributed Computing*, 2009, pp. 532–545.

[20] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li, "mpijava: A java interface to mpi," http://www.hpjava.org/mpiJava.html, accessed: 2018-03-14.

[21] G. "Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded mpi communication on multicore petascale systems," in *"Recent Advances in the Message Passing Interface"*, 2010, pp. 11–20.

[22] H. V. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced thread synchronization for multithreaded mpi implementations," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 314–324.

[23] R. Latham, R. Ross, and R. Thakur, "Can mpi be used for persistent parallel services?" in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2006, pp. 275–284.

[24] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using mpi in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13, 2013, pp. 43–48.

[25] K. Beineke, S. Nothaas, and M. Schoettner, "Fast parallel recovery of many small in-memory objects," in *International Conference on Parallel and Distributed Systems (ICPADS)*, vol. 23, in press.

[26] Oracle, "Java i/o, nio, and nio.2," https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html, accessed: 2018-03-14.

[27] R. Hitchens, *Java NIO*. Sebastopol, CA, USA: O'Reilly Media, 2009.

[28] G. L. Taboada, J. Touriño, and R. Doallo, "Java fast sockets: Enabling high-speed java communications on high performance clusters," *Comput. Commun.*, vol. 31, pp. 4049–4059, Nov. 2008.

[29] G. L. Taboada, J. Tourino, and R. Doallo, "High performance java sockets for parallel computing on clusters," in *Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[30] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: The java unsafe api in the wild," *SIGPLAN Not.*, vol. 50, pp. 695–710, Oct. 2015.

[31] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat, "Pickling state in the javatm system," in *Proc. of the 2nd Conf. on USENIX Conf. on Object-Oriented Technologies*, 1996, pp. 19–19.

[32] "Kryo - java serialization and cloning: fast, efficient, automatic." https://github.com/EsotericSoftware/kryo, accessed: 2018-03-14.

[33] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[34] K. Beineke, S. Nothaas, and M. Schoettner, "High throughput log-based replication for many small in-memory objects," in *IEEE 22nd International Conference on Parallel and Distributed Systems*, 2016, pp. 535–544.