DXNet: Scalable Messaging for Multi-Threaded Java-Applications Processing Big Data in Clouds

Kevin Beineke, Stefan Nothaas and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: [Kevin.Beineke,Stefan.Nothaas,Michael.Schoettner]@hhu.de

***Abstract*—Many big data and large-scale cloud applications are written in Java or are built using Java-based frameworks. Typically, application instances are running in a data center on many virtual machines which requires scalable and efficient network communication. In this paper, we present the practical experience of designing an extensible Java network subsystem, called DXNet, providing fast object de-/serialization, automatic connection management and zero-copy messaging. Additionally, we present a Java.nio-based transport for DXNet, called EthDXNet, to efficiently utilize Ethernet networks. The proposed design uses a zero-copy send and receive approach for asynchronous messages and requests/responses. DXNet is optimized for small messages (< 100 bytes) in order to support graph-based applications, but also works well with larger messages (e.g., 8 MB). DXNet is available on GitHub, and its modular design is open for different transport implementations currently supporting Ethernet and InfiniBand. EthDXNet is based on Java.nio socket channels complemented by application-level flow control to achieve low latency and high throughput for 10 GBit/s and faster Ethernet networks. Furthermore, a scalable automatic connection management and a low-overhead interest handling provide efficient network communication for dozens of servers, even for small messages (< 100 bytes) and an all-to-all communication pattern. The evaluation with micro-benchmarks and the Yahoo! Cloud Serving Benchmark (YCSB) shows the efficiency and scalability with up to 64 virtual machines in the Microsoft Azure cloud. Furthermore, DXNet achieves request-response latencies sub 10 µs (round trip) including object de-/serialization, as well as a maximum throughput of more than 9 GByte/s on a private cluster (using InfiniBand).**

Keywords–Message passing; Ethernet networks; InfiniBand; Java; Cloud computing.

I. INTRODUCTION

This paper is an extended version of the conference paper "Scalable Messaging for Java-based Cloud Applications" published at ICNS 2018 [1].

Big data processing is emerging in many application domains of which many are developed in Java or are based on Java frameworks [2][3][4]. Typically, these big data applications aggregate the resources of many virtual machines in cloud data centers (on demand). For data exchange and coordination of application instances, an efficient network transport is essential. Fortunately, public cloud data centers already provide 10 GBit/s Ethernet, 56 GBit/s InfiniBand and faster.

Java applications have different options for exchanging data between Java servers, ranging from high-level Remote Method Invocation (RMI) [5] to low-level byte streams using Java sockets [6] or the Message Passing Interface (MPI) [7]. However, none of the mentioned possibilities offer high performance messaging, elastic automatic connection management, advanced multi-threaded message handling and object serialization all together.

In this paper, we propose DXNet, a network messaging system which addresses all of these requirements. DXNet is a network library for Java-based applications which has originally been designed for DXRAM [8] a distributed in-memory key-value store and DXGraph [9] a graph processing framework built on top of DXRAM. We provide DXNet as a standalone library through GitHub [10] as we think it is useful for many other Java-based big data applications. DXNet is extensible by transport implementations to support different network interconnects. In this paper, we also present the Ethernet transport implementation for DXNet, called EthDXNet. The Ethernet transport is based on Java.nio and provides high throughput and low latency networking over Ethernet connections.

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
- scalable automatic connection management
- zero-copy approach for sending and receiving data over socket channels
- efficient socket channel interest handling
- evaluations with 5 GBit/s Ethernet (with up to 64 VMs in the Microsoft Azure cloud) 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 Round-Trip Time (RTT) with InfiniBand transport (including object de-/serialization). Also, high throughput is achieved even with smaller payloads, i.e., bandwidth saturation with 1-2 KB payload on InfiniBand and 256-byte payload on Ethernet. Furthermore, the evaluation shows that EthDXNet scales well while per-node message

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

throughput and request-response latency is constant from 2 to 64 nodes, even in an high-load all-to-all scenario (worst case).

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. Section VII presents thread parking strategies. In Section VIII, we describe the sending and receiving procedure of EthDXNet, followed by a presentation of the connection management in Section IX. Section X focuses on the flow control implementation and Section XI on the interest handling. Transport implementations for InfiniBand and Loopback are described in Section XII. Evaluation results are discussed in Section XIII, followed by the conclusion.

II. Related Work

In this section, we discuss related work for this paper. 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 this communication semantics. We compare DXNet with the most relevant related work, only.

A. DSM

Distributed Shared Memory (DSM) is re-gaining attraction due to fast networks supporting **RDMA** but is not an option for most existing Java applications because it requires many modifications within the Java Virtual Machine (JVM) and its memory management [11]. Furthermore, despite all advantages modern networks provide, DSM systems have limited scalability because of their transparent implicit communication [12].

B. Java RMI

Java's RMI [5] 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 [13]. 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 [13][14]. Furthermore, method calls are always blocking.

Manta [15] improves runtime costs of RMI by using a native static compiler. **KaRMI** [16], 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.

C. MPI

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 [17]. As MPI's official support is limited to C, C++ and Fortran, Java object serialization is not considered by the standard. Nevertheless, MPI is available for Java applications through implementations of the MPI standard in Java [18] or wrappers of a native library [19].

MPI-2 introduced multi-threading for MPI processes [17] 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 latter to provide a high concurrent performance [20][21]. Still, the performance is limited compared to a message passing service designed for multi-threading [20].

One of DXNet's main application domains are long running 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 fully supporting these features [22][23]. Furthermore, job shut down and crash handling is also limited [23]. 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 [24], DXNet was used in the in-memory key-value store DXRAM to examine crash behavior and scalability.

D. Sockets

High level mechanisms for typical **socket-like interfaces** supporting Gigabit Ethernet (and higher) are provided by Java.nio [25][26], Java Fast Sockets (JFS) [27] or High Performance Java Sockets [28]. DXNet uses Java.nio to implement a transport for commonly used Ethernet networks.

1) Java.nio: The `java.io` and `java.net` libraries provide basic implementations for exchanging data via TCP/IP and UDP sockets over Input- and OutputStreams [25][6]. To create a TCP/IP connection between two servers, a new `Socket` is created and connection established to a remote IP and port. On the other end, a `ServerSocket` must be listening on given IP-port tuple creating a new socket when accepting an incoming connection-creation request. The connection creation must be acknowledged from both sides and can be used to exchange byte arrays by reading/writing from/to the socket hereafter. While this is sufficient for small applications with a few connections, this basic approach lacks several performance-critical optimizations [29] introduced with `Java.nio` [25][26]. (1) Instead of byte arrays, the read/write

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

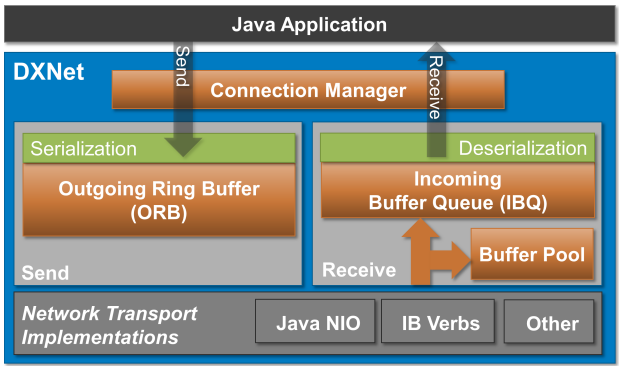

Figure 1. Simplified DXNet Architecture.

methods of Java.nio use `ByteBuffers`, which provide efficient conversion methods for all primitive data types. (2) ByteBuffers can be allocated outside of the Java heap allowing system-level I/O operations on the data without copying as the ByteBuffer is not subject to the garbage collection outside of the Java heap. This relieves the garbage collector as well as lowering the overhead with many buffers. (3) `SocketChannels` and `Selectors` enable asynchronous, non-blocking operations on stream-based sockets. With simple Java sockets, user-level threads have to poll (a blocking operation) in order to read data from a socket. Furthermore, when writing to a socket the thread blocks until the write operation is finished, even if the socket is not ready. With Java.nio, operation interests (like READ or WRITE) are registered on a selector which selects operations when they are ready to be executed. This enables efficient handling of many connections with a single thread. The dedicated thread is required to call the `select` method of the selector which is blocking if no socket channel is ready or returns with the number of executable operations. All available operations (e.g., sending/receiving data) can be executed by the dedicated thread, afterward.

2) Java Fast Sockets: JFS is an efficient Java communication middleware for high-performance clusters [27]. It provides the widely used socket API for a broad range of target applications and is compatible with standard Java compilers and VMs. JFS avoids primitive data type array serialization (JFS does not include a serializer), reduces buffering and unnecessary copies in the protocol and provides shared memory communication with an optimized transport protocol for Ethernet. DXNet provides a highly concurrent serialization for complex Java objects and primitive data types which avoids copying/buffering.

III. DXNET

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 Figure 1). Further implementation details can be found in the GitHub repository [10].

A. Basic Functionality

Automatic connection management. DXNet abstracts physical network addresses, e.g., IP/Port for Ethernet or GUID for InfiniBand, by using **node IDs**. 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 (DXNet transparently manages the association of responses and requests). **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 described in Section X.

Receiving messages. When incoming data is detected by the network transport, it requests a pooled *native* memory buffer and copies the data into the buffer (see Section VI and Figure 1). By using a native memory buffer, we avoid burdening the Java garbage collector. The term *native* is described in Section IV-C. 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.

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 providing low-latency signaling. CPU load is managed without impairing latency by parking currently idling threads (described in Section VII).

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE I. DXNET'S APPLICATION INTERFACE

Method	Description
`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

Fast serialization. DXNet implements fast serialization of complex data structures and writes data directly into an ORB. Many threads can access the ORB 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 up to 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

Sections VIII to XI present an transport for Ethernet networks.

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 messages 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 order as given by the application threads and aggregates implicitly multiple smaller messages in order to achieve high throughput. We decided to use lock-free synchronization for concurrency control which is more complex but highly efficient concerning latency compared to locks.

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 XIII), we used 1 MB and 4 MB ORBs.

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

Messages available for sending (fully serialized) are detected 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 the `Unsafe` class (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 the size of the message to be sent. 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 Figure 2 show the starting point of each serialization core (producer) whereas the solid-colored ones show the current position. In the example, the purple producer finished its serialization first, and the green and orange producers are still serializing.

F_C is moved forward by producers when messages are fully serialized. In Figure 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. A naive solution lets fast producers wait for slower ones (e.g., with wait-notify semantics). As we do not want to impact latency, we cannot

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

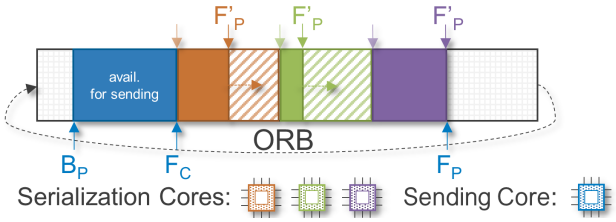

Figure 2. ORB for parallel serialization and aggregating outgoing messages. B_P: next message to consume. F_C: end of messages to consume. F_P: next free byte to produce. F'_P: thread-local copy F_P.

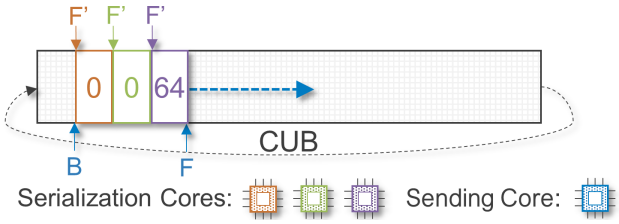

Figure 3. Catch-Up Buffer (CUB). Allowing faster producers returning early and not wasting CPU cycles for waiting. B: back pointer. F: front pointer. F': thread-local copy of F.

use locks/conditions here. An alternative solution is to busy-poll until all preceding messages have been serialized which stresses the CPU. A more sophisticated solution is presented in the next section.

B. Optimized Lock-Free Solution

The basic solution already avoids the overhead of locks, but with an increasing number of parallel serializations the probability of threads having to wait for slower ones increases (a thread might be slower because of less CPU time, for instance, or simply because the message to serialize is larger). 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 *note* and returning early to the application. This note includes the message size so that slower producers can move F_P forward for the faster ones that already left. 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 note from faster producers. An entry will be 0 if there is no note 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 message size read from the CUB.

Figure 3 shows a CUB corresponding to the ORB shown in Figure 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 note at the correct position in the CUB. The 64 is a note from the purple producer (message size of the filled purple box in Figure 2) who finished fastest and returned already to the application. The green and orange producers are still working (0 = no note). 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 note). The green one will do the same, but twice as it will detect the note (64) after committing its serialization and, thus, move forward F_C in the ORB by 64 bytes and also B by one slot (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 (e.g., F and F_P are stored together in one long) and, as the CAS operation works 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, amongst other data structures, is allocated in native memory, i.e., it is located outside of the Java heap in the virtual address space of the Java process. Therefore, data structures stored in native memory are not managed by the JVM (e.g., no garbage collection or type safety). But, this does not pose a security risk for cloud applications as memory access is still managed by the operating system. On the other hand, using native memory allows the underlying network transports to send messages without copying them, for instance. 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 [26] for two reasons. First, access is faster (e.g., missing boundary checks are already handled on a 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 Sections 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`.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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

A. Export

Exporter. The serialization (or export) of Java objects requires an `exporter` which is passed to the `exportObject` method. 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 Figure 4) which has a fixed size and exporting the variable-sized payload by calling the `exportObject` method.

DXNet uses its `default exporter` for serialization which is optimized for performance. It is complemented by 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 Figure 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 `exportObject` method is called again (threads return after being interrupted during serialization) and all previously successfully executed export operations are automatically skipped up to the position stored in the unfinished operation. 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 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 are 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 and only custom object structures must be aware of these requirements and avoid functions causing side effects (e.g., I/O access).

VI. Event-Driven Processing of Incoming Data

Figure 5 gives an overview of the parallel event-driven processing of incoming data. Like for the ORB, we use multithreading, 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** afterward. Multiple message handler threads concurrently create the message objects, deserialize the messages' payloads

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

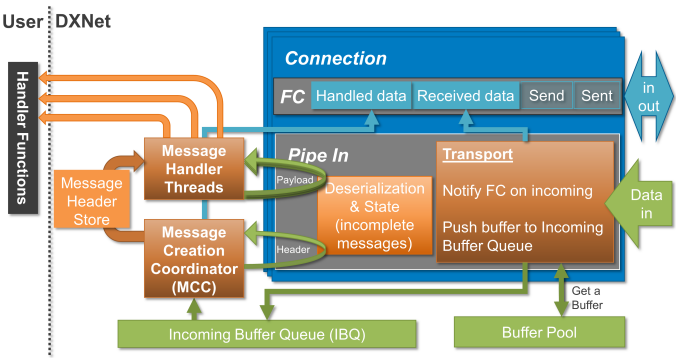

Figure 5. Receiving and processing messages. Green: Native memory access.

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 number of bytes ready to be received on the stream is unknown. The pool can also scale-up dynamically when running out of buffers.

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

The transport thread pushes filled buffers 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 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 the 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 an issue 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, which increases latency because of collisions, the batch sizes differ slightly for every message handler.

Returning of buffers. A pooled buffer must not be returned before all its messages have 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 number 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 (deserialized and executed) by the same message handler. All other steps preserve message ordering by default. For achieving maximum throughput, marking of messages should be used if necessary, only.

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 was too fast executing its work compared to other threads/servers. However, the thread

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

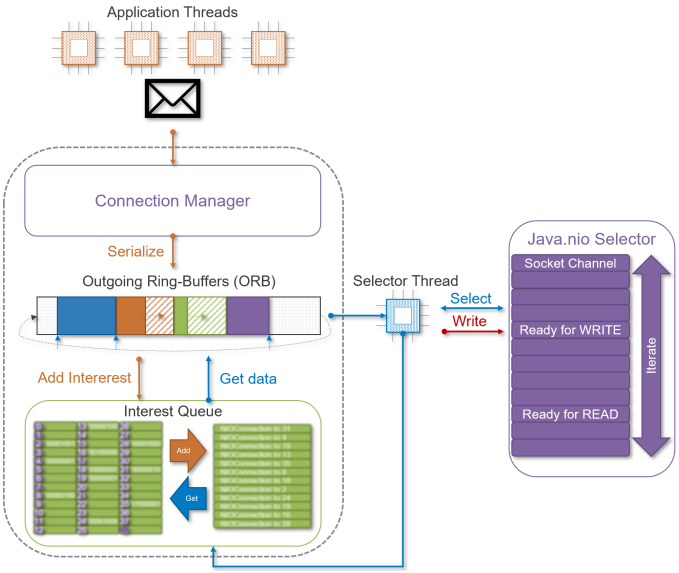

Figure 6. Data structures and Threads. Details of the Interest Queue can be found in Figure 8.

should not park for a long period to avoid restraining other threads/servers. Experiments showed that a reasonable 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 Section 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 the 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 XIII-A for the evaluation.

Idling for longer periods, e.g., applications not exchanging messages for a longer period, 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. EthDXNet - Sending and Receiving

In the following sections, we describe the Ethernet transport of DXNet, called EthDXNet. An overview of the most important data structures and threads of EthDXNet are depicted in Figure 6.

A. Sending of Data

To send messages, the DXNet API methods `sendMessage` or `sendSync` are called by the application threads (or message handler threads). In DXNet, messages are always sent asynchronously, i.e., application threads might return before the message is transferred. It is possible, though, to wait for a response before returning to the application (sendSync). After getting the `ConnectionObject` (a Java object) from the **Connection Manager**, the message is serialized into the ORB associated with the connection. For performance reasons, many application threads can serialize into the same or different ORBs in parallel (more in Section IV). The actual message transfer is executed by the **SelectorThread**, a dedicated daemon thread driving the Java.nio back-end. Thus, after serializing the message into the ORB, the application thread must signal data availability for the corresponding connection. This is done by registering a `WRITE` interest (see Table II) for given connection in the **Interest Queue** (see Section XI). When ready, Java.nio's **Selector** wakes-up the SelectorThread (which is blocked in the `select` method of the Selector) to execute the operation and thus to transfer the message.

After returning from the select method, a **SelectionKey** is available in the ready-set of the Selector. It contains the operation interest WRITE, the socket channel and attachment (the associated ConnectionObject). This SelectionKey is dispatched based on the operation. In order to send the message over the network, the SelectorThread pulls the **data block** from the ORB of the corresponding connection and calls the `write` method of the socket channel. From this point, we cannot distinguish single messages anymore because messages are naturally aggregated to data blocks in the ORBs, which is a performance critical aspect. The write method is repeatedly called until all bytes have been transferred or the method returned with return value 0. The second case indicates congestion on the network or the receiver and is best handled by stopping the transfer and continue it later. After sending, the back position (B_p, see Figure 2) of the ORB is moved by the number of bytes transferred to free space for new messages to send. Additionally, if the transfer was successful and the ORB is empty afterward, the SelectionKey's operation is set to `READ` which is the preset operation and enables receiving incoming data blocks. If the transfer failed, the connection is closed (see Section IX). If the transfer was incomplete or new data is available in the ORB, the SelectionKey is set to `READ | WRITE` (combination of READ and WRITE by using the bitwise or-operator) which triggers a new WRITE operation when calling select the next time but also allows receiving incoming messages. It is important to change the SelectionKey to this state as keeping only the WRITE operation could result in a deadlock situation in which both ends try to transfer data, but none of them can receive data on the same connection. This causes the **kernel socket receive buffers** to fill up on both sides preventing further data transfer.

The ORB is a ring buffer allocated in native memory (outside of the Java heap). In order to pass a ByteBuffer to the socket channel, which is required for data transfer, we wrap a **DirectByteBuffer** onto the ORB and set the ByteBuffer's position to the front position in the ORB and the limit to the back position. A DirectByteBuffer

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

is a ByteBuffer whose underlying byte array is stored in native memory and is not subject to garbage collection. This enables native operations of the operating system without copying the data first. The socket channel's send and receive operations are examples for those native operations, thus, benefiting from the DirectByteBuffer. Java does not support dynamically changing the address of a ByteBuffer. Therefore, on initialization of the ORB, we allocate a new DirectByteBuffer by calling `allocateDirect` of the Java object ByteBuffer and use the underlying byte array as the ORB. To do so, we need to determine the memory address of the byte array, which can be obtained with `Buffer.class.getDeclaredField("address")`. That is, during serialization the ORB is accessed with `Java.unsafe` by reading/writing from/to the actual address outside of the Java heap, but the socket channel accesses the data by using the DirectByteBuffer's reference (with adjusted position and limit). We do not access the ORB by using the DirectByteBuffer during serialization because of performance and compatibility reasons.

Although this approach prevents copying the data to be sent on user-level, the data is still copied from the ORB to the **kernel socket send buffer** which is a necessity of the stream-based socket approach. Therefore, correctly configuring the kernel socket buffer sizes (one for sending and one for receiving) has a significant impact on performance. We empirically determined setting both buffer sizes to the ORBs' size offers good performance without increasing the memory consumption too much (typically the ORBs are between 1 and 4 MB depending on the application use case).

B. Receiving of Data

Receiving messages is always initiated by Java.nio's **Selector** which detects incoming data availability on socket channels. When a socket channel is ready to be read from, the SelectorThread selects the SelectionKey and dispatches the READ operation. Next, the SelectorThread reads repeatedly by calling the `read` operation on the socket channel until there is nothing more to read or the buffer is full. If reading from the socket channel failed, the socket channel is closed. Otherwise, the ByteBuffer with the received data is flipped (limit = position, position = 0) and pushed to the IBQ (see Figure 1). The buffer processing is explained in Section VI.

In order to read from a socket channel, a ByteBuffer is required to write the incoming data into. Continually allocating new ByteBuffers would decrease the performance drastically. Therefore, we implemented a **buffer pool**. The buffer pool provides DirectByteBuffers in different configurable sizes (e.g., 8×256 KB, 256×128 KB and 4096×16 KB). The SelectorThread pulls them using a worst-fit strategy as the number 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, see Section VI.

The pooled DirectByteBuffers are wrapped to provide the ByteBuffer's reference as well as the ByteBuffer's address. The reference is used for reading from the socket channel and the address is necessary to deserialize the messages from the ByteBuffer.

IX. Automatic Connection Management

For sending and receiving messages, DXNet has to manage all open connections and create/close connections on demand. A connection is represented by an object (ConnectionObject), containing a node ID to identify the connection based on the destination, a **PipeIn** and a **PipeOut**. The PipeOut consists mostly of an ORB, a socket channel and flow control for outgoing data. The PipeIn contains a socket channel, flow control for incoming data, has access to the buffer pool (shared among all connections) and more data structures important to buffer processing, which are not further discussed in this paper.

1) Connection Establishment: Connections are created in two ways: (1) actively by creating a new connection to a remote node or (2) passively by accepting a remote node's connection request. In both cases, the connection manager must be updated to administrate the new connection. Figure 7 shows the procedure of creating a new connection (active on the left side and passive on the right). The core part is the TCP handshake, which can be seen in the middle.

Active connection creation: A connection is created actively if an application thread wants to send a message to a not yet connected node. To establish the connection, the application thread creates a new ConnectionObject (including PipeIn and PipeOut and all its components), opens a new socket channel and connects the socket channel to the remote node's IP and port. Afterwards, the application thread registers a `CONNECT` operation, creates a `ReentrantLock` and `Condition` and waits until the Condition is signaled or the connection creation was aborted (using a lock here is not a performance issue as connections are held open). To correctly identify the corresponding ConnectionObject to a socket channel, the ConnectionObject is attached to the SelectionKey when registering the CONNECT interest and all following interests.

The SelectorThread continues the connection establishment by applying the CONNECT interest and selecting the socket channel when the remote node accepted the connection or the connection establishment failed. After selecting the SelectionKey, the socket channel's status is checked. If it is pending, the connection creation was successful so far, and the socket channel can be completed by calling `finishConnect`. If the connection establishment was aborted, the application thread is informed by setting a flag (which is checked periodically by the application thread).

The remote node has to identify the new node currently creating a connection. Thus, the node ID is sent to the remote node using the newly created channel. Furthermore, the SelectorThread marks the PipeOut as connected and signals the condition so the application thread can continue. It adds the connection to the connection manager, increments the connection counter and starts sending data, afterward.

Passive connection creation: For accepting and creating an incoming connection, the Selector implicitly selects a SelectionKey with `ACCEPT` operation interest which is processed by the SelectorThread by calling `accept` on the socket channel. This creates a new socket channel and acknowledges the connection. Afterward, the interest `READ` is registered in order to receive the node ID of the remote node. After selecting and

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

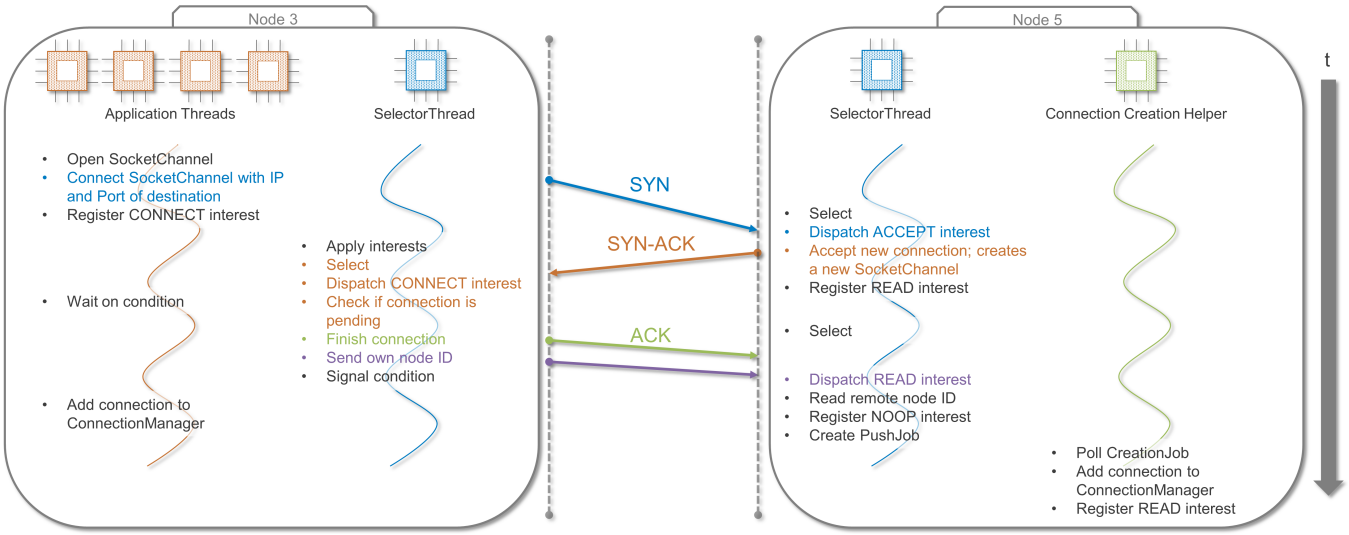

Figure 7. Connection Creation

dispatching the interest, the node ID is read by using the socket channel's read method.

At this point, the socket channel is ready for sending and receiving data, but the connection object has yet to be created and pushed to the connection manager. This process is rather time-consuming and might be blocked if an application thread creates a connection to the same node at the same time (connection duplication is discussed in Section IX-2). Therefore, the SelectorThread creates a job for creating the connection and forwards it to the **ConnectionCreationHelper** thread. Additionally, the interest is set to `NO-OP` (0) to avoid receiving data before the connection setup is finished and the connection is attached to the SelectionKey.

The ConnectionCreationHelper polls the job queue periodically. There are two types of jobs: (1) a connection creation job and (2) a connection shutdown job. The latter is explained in Section IX-3. When pulling a connection creation job, the ConnectionCreationHelper creates a new ConnectionObject (including the pipes, ORB, FC, etc.) and registers a READ interest with the new ConnectionObject attached. Furthermore, the PipeIn is marked as connected.

To be able to accept incoming connection requests, every node must open a `ServerSocketChannel`, bind it to a well-known port and register the `ACCEPT` interest. Furthermore, for selecting socket channels, a `Selector` has to be created and opened.

2) Connection Duplication: It is crucial to avoid connection duplication which occurs if two nodes create a connection to each other simultaneously. In this case, the nodes might use different connections to send and receive data which corrupts the message ordering and flow control. There are two approaches for resolving this problem: (1) detecting connection duplication during/after the connection establishment and (2) avoiding connection duplication by using two separate socket channels for sending and receiving.

Solution 1: **Detect and resolve connection duplication** by keeping one connection open and closing the other one. Apparently, the other node must decide consistently which can be done by considering the node IDs (e.g., always keep the connection created by the node with higher node ID). One downside of this approach is the complex connection shutdown. It must ensure that all data appended to the ORB of the closing connection, to be sent over the closing connection, has already been sent and received. Furthermore, message ordering cannot be guaranteed until the connection duplication situation is resolved.

Solution 2: **Avoid connection duplication** by using two socket channels per connection: one for sending and one for receiving (implemented in EthDXNet). Thus, simultaneous connection creation leads to **one** ConnectionObject with opened PipeIn and PipeOut (one socket channel, each) whereas a single connection creation opens either the PipeOut (active) or PipeIn (passive). This approach requires additional memory for the second socket channel, Java.nio's Selector has more socket channels to manage and connection setup is required from both ends. The additional memory required for the second socket channel is negligible as the kernel socket buffers are configured to use a little socket receive buffer for the outgoing socket channel and a little socket send buffer for the incoming socket channel. The second TCP handshake (for connection creation, both sides need to open and connect a socket channel) is also not a problem as both socket channels can be created simultaneously and for a long-running big data application connections among application instances are typically kept over the entire runtime. Finally, the overhead for Java.nio's Selector is difficult to measure but is certainly not the bottleneck taking into account the limitations of the underlying network latency and throughput. **Sending out-of-band (OOB) data** is possible by utilizing **the unused back-channel of every socket channel**. We use this for sending flow control data in EthDXNet (see Section X).

3) Connection Shutdown: Connections are closed on three occasions: (1) if a write or read access to a socket channel failed, (2) if a new connection is to be created, but the configurable connection limit is reached or (3) on node shutdown. In the first case, the SelectorThread directly shuts down the connection. In the second case, the application thread registers a `CLOSE` interest to let the SelectorThread close the connection

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

asynchronously. On application shutdown, all connections are closed by one **Shutdown Hook** thread.

To shut down a connection, first, the outgoing and incoming socket channels are removed from the Selector by canceling the SelectionKeys representing a socket channel's registration. Then, the socket channels are closed by calling the socket channels' `close` method. At last, the connection is removed from the connection manager by creating a shutdown job handled by the ConnectionCreationHelper (case (1)) or directly removing it when returning to the connection management (cases (2) and (3)). The ConnectionCreationHelper also triggers a `ConnectionLostEvent`, which is dispatched to the application for further handling (e.g., node recovery).

When dismissing a connection (case (2)), directly shutting down a connection might lead to data loss. Therefore, the connection is closed gracefully by waiting for all outstanding data (in the connection's ORB) to be sent. Priorly, the connection is removed from connection management to prevent further filling of the ORB. Afterward, a CLOSE interest is registered to close the socket channels asynchronously. The SelectorThread does not shut down the socket channels on the first opportunity but postpones shutdown for at least two RTT timeouts to ensure all responses are received for still outstanding requests.

X. Flow Control

DXNet implements a **flow control (FC)** protocol to avoid flooding a remote node with messages. This would result in an increased overall latency and lower throughput if the remote node cannot keep up with processing incoming messages. When sending messages, the per-connection dedicated FC checks if a configurable threshold is exceeded. This threshold describes the **number of bytes sent by the current node but not fully processed by the receiving node**. The receiving node counts the number of bytes received and sends a confirmation back to the source node in regular intervals. Once the sender receives this confirmation, the number of bytes sent but not processed is reduced. If an application send thread was previously blocked due to exceeding this threshold, it can now continue with processing the message.

EthDXNet uses the *Transmission Control Protocol* (TCP) which already implements a flow control mechanism on protocol layer. Still, DXNet's flow control is beneficial when using TCP. If the application on the receiver cannot read and process the data fast enough, the sender's TCP flow control window, the maximum amount of data to be sent before data receipt has to be acknowledged by the receiver, is reduced. The decision is based on the utilization of the corresponding kernel socket receive buffer. In DXNet, reading incoming data from kernel socket receive buffers is decoupled from processing the included messages, i.e., many incoming buffers could be stored in the IBQ to be processed by another thread. Thus, the kernel socket receive buffers' utilizations do not necessarily indicate the load on the receiver leading to delayed or imprecise decisions by TCP's flow control.

This section focuses on the implementation of the flow control in EthDXNet. Flow control data has to be sent with high priority to avoid unintentional slow-downs and fluctuations regarding throughput and latency. Sending flow control data in-band, i.e., with a special message appended to the data stream, is not an option because the delay would be too high. TCP offers the possibility to send **urgent data**, which is a single byte inlined in the data stream and sent as soon as possible. Furthermore, urgent data is always sent, even if the kernel socket receive buffer on the receiver is full. To distinguish urgent data from the current stream (urgent data can be at any position within a message as the transfer is not message-aligned), a dedicated flag within the TCP header needs to be checked. This flag indicates if the first byte of the packet is urgent data. Unfortunately, Java.nio does not provide methods for handling incoming TCP urgent data.

We solve this problem **by using both unused back-channels** of every socket channel which are available because of the double-channel connection approach in EthDXNet. Thus, the incoming stream of the outgoing socket channel and the outgoing stream of the incoming socket channel of every connection are used **for sending/receiving flow control data**.

Sending flow control data: When receiving messages, a counter is incremented by the number of received bytes for every incoming buffer. If the counter exceeds a configurable threshold (e.g., 60% of the flow control window), a `WRITE_FC` interest is registered. This interest is applied, selected and dispatched like any other WRITE interest. But, instead of using the socket channel of the PipeOut, **the PipeIn is used to send the flow control data**. The flow control data consists of one byte containing the number of reached thresholds (typically 1). If the threshold is smaller than 50%, for example, 30%, it is possible that between registering the WRITE_FC interest and sending the flow control data, the threshold has been exceeded again. For example, if the current counter is at 70% of the windows size which is more than two thresholds of 30%. In this case 2 * 30% = 60% is confirmed by sending the value 2. After sending flow control data, the SelectionKey is reset to READ to enable receiving messages on this socket channel, again.

Receiving flow control data: To be able to receive flow control data, the socket channel of **the PipeOut must be readable** (register READ). If flow control data is available to be received, the socket channel is selected by the Selector, and the SelectorThread reads the single byte from the socket channel of the PipeOut. When processing serialized messages on the sender, a counter is incremented. Application threads which want to send further messages if the counter reached the limit (i.e., the flow control window is full) are blocked until the receiver acknowledges message receipt. The read flow control value is used to decrement the counter to re-enable sending messages. Usually, the limit is never reached as the flow control data is received before (if the threshold on the receiver is low enough).

In Section VIII-A, we discussed the end-to-end situation of both nodes sending data to each other, but never reading (if the SelectionKey's operation stays at WRITE) causing a deadlock. This situation cannot occur with two socket channels per connection as reading and writing are handled independently. But, a similar situation is possible where two nodes send data to each other, but flow control data is not read for a while. This does not cause a deadlock but decreases performance. By setting the interest to READ | WRITE, flow control data is read from time to time ensuring contiguous high throughput.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

TABLE II. JAVA.NIO INTERESTS

Interest	Description
OP_READ	channel is ready to read incoming data
OP_WRITE	set if data is available to be sent
OP_CONNECT	set to open connection
OP_ACCEPT	a connect request arrived
NO-OP	do nothing

TABLE III. ETHDXNET INTERESTS

Interest	Description (refers to attached connection)
CONNECT	set OP_CONNECT for outgoing channel
READ_FC	set OP_READ for outgoing channel
READ	set OP_READ for incoming channel
WRITE_FC	set OP_WRITE for incoming channel
WRITE	set OP_WRITE for outgoing channel
CLOSE	shutdown both socket channels

XI. EFFICIENT MANAGEMENT OF OPERATION INTERESTS

Operation interests are an important concept in Java.nio and are registered in the Selector to create and accept a new connection, to write data or to enable receiving data. The operation interests are complemented by the ConnectionObject (as an attachment) and the socket channel stored together in a SelectionKey. As soon as the socket channel is ready for any registered operation, the Selector adds the corresponding SelectionKey to a ready-set and wakes-up the SelectorThread waiting in the `select` method. If the SelectorThread is not waiting in the select method, the next select call will return immediately. The SelectorThread can then process all SelectionKeys.

A. Types of Operations Interests

The operation interests can be classified into two categories: **explicit operation interests and implicit operation interests**. Implicit operations are registered as presets after socket channel creation and after executing explicit operations. For example, a READ interest is registered for a socket channel if data is expected to arrive on this socket channel. The operation is then selected implicitly by the Selector whenever data is available to be received. Another example is the ServerSocketChannel which implicitly accepts new incoming connection requests if the ACCEPT interest has been registered before. Explicit operations are single operations which need to be triggered explicitly by the application. For example, when the application wants to send a message, the application thread registers a WRITE interest. When the socket channel is ready, the data is sent and the socket channel is set to the preset (in our case READ). It is not forbidden by Java.nio to keep explicit operations registered. But, as a consequence, the operations are always selected (every time select is called) which increases CPU load and latency. Therefore, in EthDXNet, every explicit operation is finished by registering an implicit operation.

The set of Java.nio operation interests is extended by EthDXNet to support flow control and to enable closing connections asynchronously. Table II shows all interests specified by Java.nio and Table III lists all interests used in EthDXNet. The interests READ, WRITE and CONNECT are directly mapped onto OP_READ, OP_WRITE and OP_CONNECT. OP_ACCEPT is registered and selected by the Selector and must not be registered explicitly. READ_FC and WRITE_FC are used to register OP_READ and OP_WRITE interests for the back-channel used by the flow control. The interest CLOSE does not have a counterpart because the method `close` can be called explicitly on the socket channel.

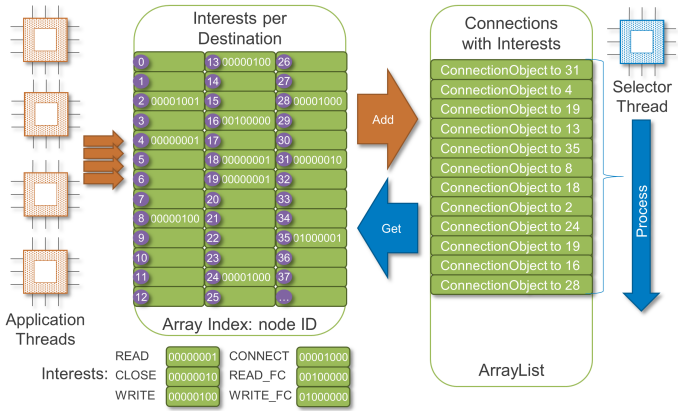

Figure 8. Interest Queue: the application threads add new interests to the Interest Queue. If interest was 0 before, the ConnectionObject is added to an ArrayList.

B. Interest Queue

None of the interests in Table III are registered directly to the Selector because only the SelectorThread is allowed to add and modify SelectionKeys. This is enforced by the Java.nio implementation which blocks all register calls when the SelectorThread is waiting in the `select` method. This obstructs the typical asynchronous application flow and can even result in a deadlock if the Selector does not have implicit operations to select. This problem can be avoided by always waking-up the SelectorThread before registering the operation interest and synchronizing the register and select calls. However, this workaround results in rather high overhead and a complicated workflow. Instead, **we address this problem with an Interest Queue** (see Figure 8) and register all interests in one bulk operation executed by the SelectorThread before calling select. This approach provides several benefits while solving the above problem: first, **the application threads can return quickly** after putting the operation interest into the queue and even faster (without any locking) when the interest was already registered (which is likely under high load). Second, **the operation interests can be combined** and put in a semantic order (e.g., CONNECT before WRITE) before registering (a rather expensive method call). Finally, **the operation interest-set can be easily extended**, e.g., by a CLOSE operation interest to asynchronously shut down socket channels.

Figure 8 shows the Interest Queue consisting of a byte array storing the operation interests of all connections (left side in Figure 8) and an ArrayList of ConnectionObjects containing connections with new operation interests sorted by time of occurrence (right side in Figure 8).

The byte array has one entry per node ID providing access

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

time in O(1). In DXNet, the node ID range is limited to 2^{16} (allowing max. 65,536 nodes per application). Thus, the operation interests of all connections can be stored in a 64-KB byte array. An array entry is not zero if at least one operation interest was added for given connection to the associated node ID. Operation interests are combined with the bitwise or-operator to avoid overwriting any interest. By combining operation interests, the ordering of the interests for a single connection is lost. But, this is not a problem because a semantic ordering can be applied later when processing the interests.

The ordering within the interests of one connection can be reconstructed but not the ordering across different connections. Therefore, whenever an interest is added to a non-zero entry of the byte array, the corresponding ConnectionObject is appended to an ArrayList. The order of operation interests is then ensured by processing the interest entries in the ArrayList in ascending order. The ArrayList also allows the SelectorThread to iterate only relevant entries and not all 2^{16}.

Processing operation interests: The processing is initiated either by the Selector implicitly waking up the SelectorThread if data is available to be read or an application thread explicitly waking up the SelectorThread if data is available to be sent. As waking-up the SelectorThread is a rather expensive operation (a synchronized native method call), it is essential to call it if necessary, only. Therefore, the SelectorThread is woken-up after adding the first operation interest to the Interest Queue across all connections (the ArrayList is empty after processing the operation interests), only. If the SelectorThread is currently blocked in the select call, it returns immediately and can process the pending operation interests.

Figure 9 shows the basic processing flow of the SelectorThread. The first step in every iteration is to register all operation interests collected in the ArrayList of the Interest Queue. The SelectorThread gets the destination node ID from the ConnectionObject and the interests from the byte array. Operation interests are registered to the Selector in the following order:

1) CONNECT: register SelectionKey OP_CONNECT with given connection attached to an outgoing channel.
2) READ_FC: register SelectionKey OP_READ with given connection attached to an outgoing channel.
3) READ: register SelectionKey OP_READ with given connection attached to an incoming channel.
4) WRITE_FC: change SelectionKey of an incoming channel to OP_WRITE if it is not OP_READ | OP_WRITE.
5) WRITE: change SelectionKey of an outgoing channel to OP_WRITE if it is not OP_READ | OP_WRITE.
6) CLOSE: keep interest in queue for delay or close connection (see Section IX-3).

The order is based on following rules: (1) a connection must be established before sending/receiving data, (2) setting the preset READ is done after connection creation, only, (3) all READ and WRITE accesses must be finished before shutting down the connection and (4) the flow control operations have a higher priority than normal READ and WRITE operations. Furthermore, re-opening a connection cannot be done before the connection is closed and closing a connection is only possible if the connection has been established before. Therefore it is not possible to register CONNECT and CLOSE together.

```
1  while (!closed) {
2   processInterests();
3
4   if (Selector.select() > 0) {
5    for (SelectionKey key :
         Selector.selectedKeys()) {
6     // Dispatch key
7     if (key.isValid()) {
8      if (key.isAcceptable()) {
9       accept();
10     } else if (key.isConnectable()) {
11      connect();
12     } else if (key.isReadable()) {
13      read();
14     } else if (key.isWritable()) {
15      write();
16     }
17    }
18   }
19  }
20 }
```

Figure 9. Workflow of SelectorThread.

Finally, the processing of registered operation interests includes resetting the operation interest in the byte array and removing the ConnectionObject from the ArrayList.

XII. Other Transport Implementations

DXNet has an open architecture supporting different network transport technologies. Currently, we have transport implementations for TCP/IP over Ethernet (described in Section VIII), reliable verbs over Infiniband (using native verbs over JNI), and Loopback (baseline for evaluation). We only sketch some important aspects of the InfiniBand transport here, as it will be published separately.

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.

XIII. Evaluation

We evaluate the proposed concepts using DXNet's Loopback transport and three different networks: 1 GBit/s Ethernet, 5 GBit/s Ethernet (a shared 10 GBit/s Ethernet connection) and 56 GBit/s InfiniBand. The Loopback is used to evaluate

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

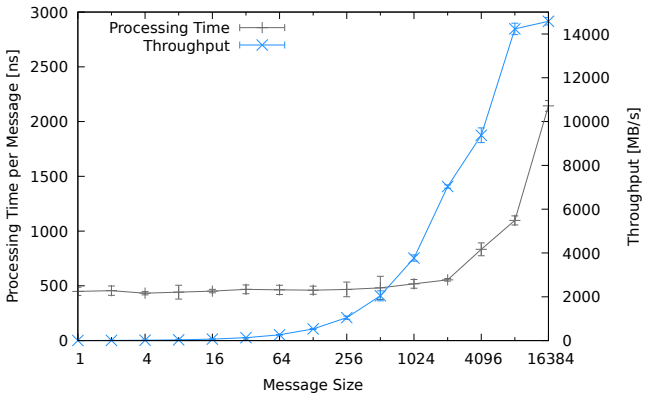

Figure 10. 10^7 Messages, 1 App. Thread, 4 Message Handlers.

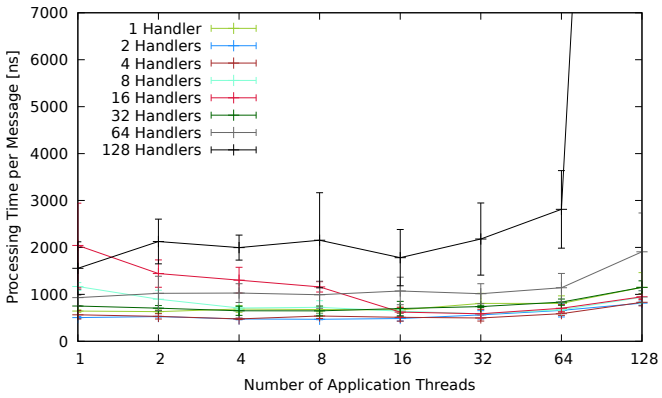

Figure 11. 10^7 Messages, 1024 Bytes Payload.

DXNet's concepts without any limitations of an underlying network. The Ethernet networks are used to evaluate EthDXNet.

Loopback and 5 GBit/s Ethernet tests were run in Microsoft's Azure cloud in Germany Central with up to 65 virtual machines (64 running the benchmark and one for deployment) 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 a 4.4.0-59 kernel and Java 8. In order to manage the servers, we created two identical scale-sets (as one scale-set is limited to 40 VMs). 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 evaluation sending 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 Transport*

As mentioned before, we want to evaluate the efficiency of DXNet's concepts without any network limitations. Figure 10 shows message processing times and throughputs for different

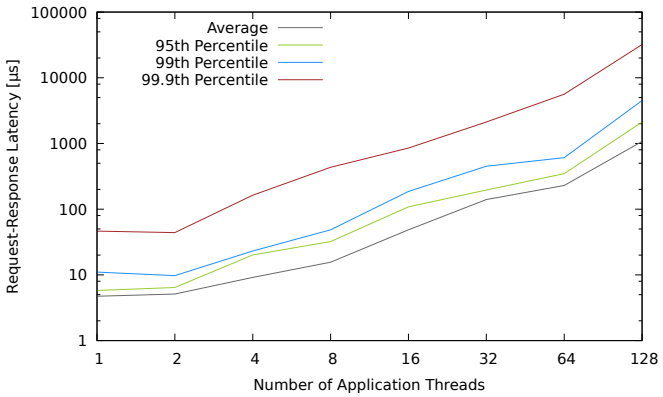

Figure 12. 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

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 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 Figure 10, 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 handle large messages often, 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. Figure 11 shows that the processing time for 1 KB messages is stable from one to 64 and only slightly increases with 128 application threads. Additionally, Figure 11 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 to efficiently handle hundreds of concurrent threads.

We also evaluated request-response latency by measuring the **RTT**, which includes sending a request, receiving the request, sending the corresponding response and receiving the response. Figure 12 shows the latency for small requests with an 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.

Figure 13 shows the breakdown of request-response latency for one and four application threads and 1024-byte requests.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

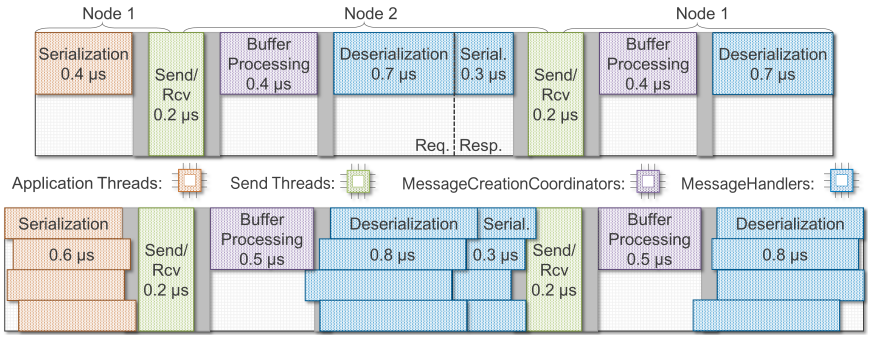

Figure 13. Breakdown of Request-Response Latency for 1024-byte Requests. One application thread (on top) and four (at the bottom). Grey bars indicate inter-thread communication.

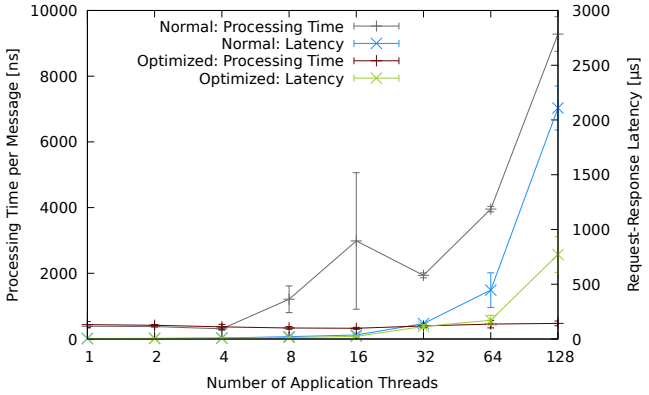

Figure 14. 10^7 Message or 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

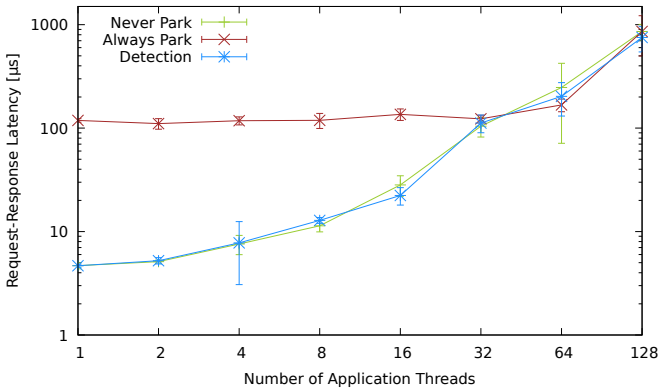

Figure 15. 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

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.

Optimized Outgoing Ring Buffer. The benefits of the Catch-Up Buffer, discussed in Section IV, can be seen in Figure 14. 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. Figure 15 shows the importance of the thread parking strategy (see Section VII). The RTT is 25 times higher 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 a disadvantage with many threads (128) and the RTT is around 100 µs higher than with the adaptive approach.

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

B. *Comparing Network Transports*

Figure 16 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).

Figure 17 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 different physical transports confirms the results gathered with Loopback. DXNet performs strongly especially with InfiniBand (RTT < 10 µs, throughput > 9 GByte/s full-duplex).

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

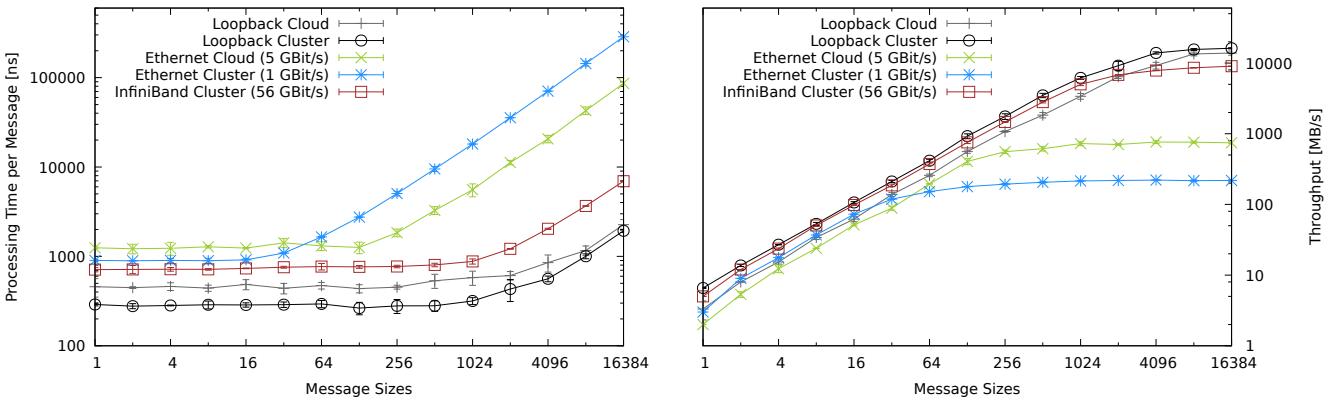

Figure 16. 10^8 Messages, 1 App. Thread, 2 Message Handlers.

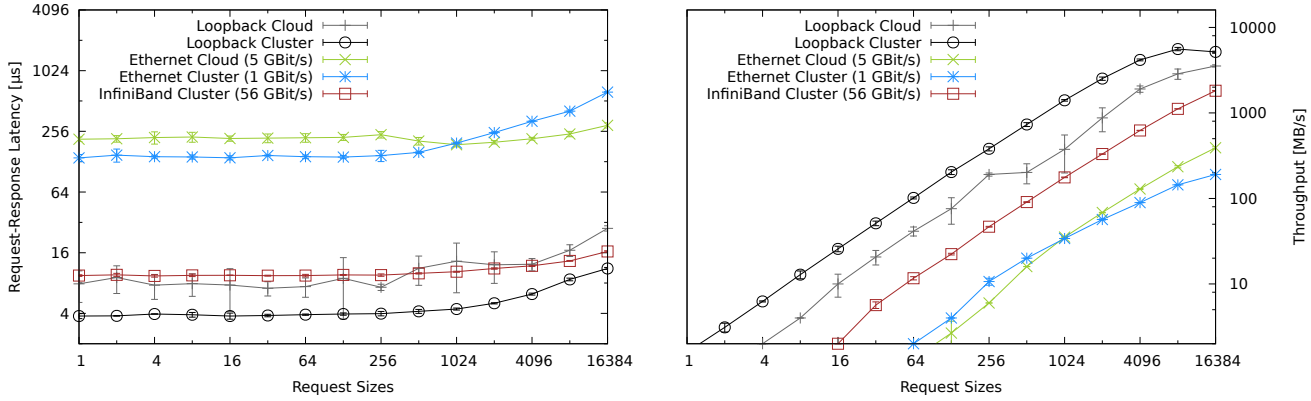

Figure 17. 10^7 Requests, 4 App. Threads, 2 Message Handlers.

TABLE IV. ADDITIONAL PARAMETERS

Parameter	Value
ORB Size	4 MB
Flow Control Windows Size	2 MB
Flow Control Threshold	0.6
net.core.rmem_max	4 MB
net.core.wmem_max	4 MB

C. Scalability of (Eth)DXNet

Message Throughput: First, we measured the asynchronous message throughput with an increasing number of nodes in an all-to-all test with message payloads of 64 and 4096 bytes. For instance, when running the benchmark with 32 nodes, each node sends 25,000,000 64-byte messages to all 31 other nodes and therefore each node has to send and receive 775,000,000 messages in total. Additional network parameters can be found in Table IV.

Figure 18 shows the average payload throughput for single nodes and Figure 19 the aggregated throughput of all nodes.

For 64-byte messages, the payload throughput is between 200 and 260 MB/s for all node numbers, showing a minimal decrease from 2 to 16 nodes. With 4096-byte messages the throughput improves with up to 8 nodes peaking at 1370 MB/s full-duplex bandwidth (5.5 GBit/s uni-directional). With 64 nodes the throughput is still above 5 GBit/s resulting in an aggregated throughput of 83,376 MB/s. The minor decline in both experiments can be explained by an uneven deployment

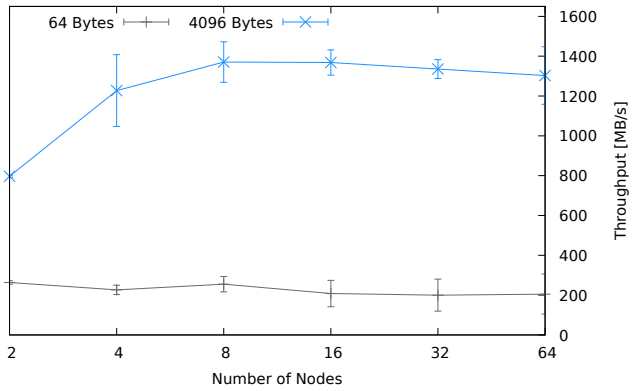

Figure 18. Message Payload Throughput per Node. 1 Application Thread, 2 Message Handler Threads.

of our network benchmark causing the last nodes starting and finishing a few seconds later. The end-to-end throughput between two nodes seems to be bound at around 3.2 GBit/s in the Microsoft Azure cloud as tests with iperf showed, too.

The benchmarks show that DXNet, as well as EthDXNet scale very well for asynchronous messages under high loads.

Request-Response Latency: The next benchmarks are used to evaluate request-response latency by measuring the RTT. Figure 20 shows the RTTs for an all-to-all scenario with 2 to 64 nodes and 1, 16 and 100 application threads. Furthermore, all-to-all tests with `ping` are included to show network latency limitations.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

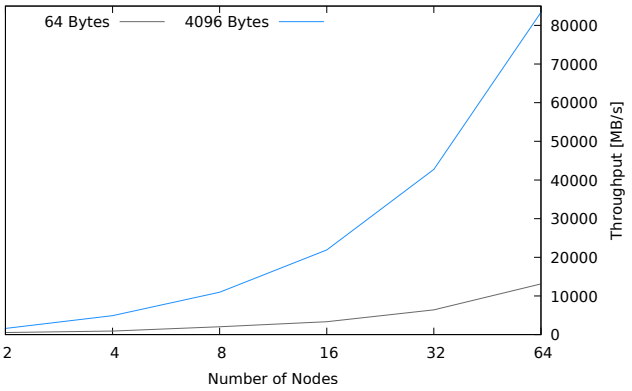

Figure 19. Aggregated Message Payload Throughput. 1 Application Thread, 2 Message Handler Threads.

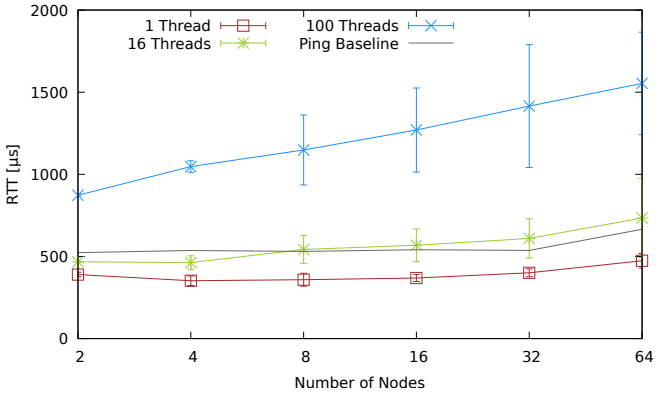

Figure 20. Average Request-Response Latency. 1 to 100 Application Threads, 2 Message Handler Threads.

The latency of the Azure Ethernet network is relatively high with a minimum of 352 µs measured with DXNet and one application thread (Figure 20). A test with up to 4032 ping processes shows that the average latency of the network is even higher (> 500 µs). In DXNet, own requests are combined with responses (and other requests if more than one application thread is used). This reduces the average latency for requests. Additionally, the ping baseline shows an increased latency for more than 32 nodes, by using one scale-set for the first 32 nodes and another one for the last 32 nodes. Different scale-sets are most likely separated by additional switches which increases the latency for communication between scale-sets.

EthDXNet is consistently under the ping baseline demonstrating the low overhead and high scalability of EthDXNet (and DXNet) when using one application thread. With 16 application threads, the latency is slightly higher and on the same level as the baseline, but the throughput is more than 10 times higher as well (in comparison to DXNet with one application thread). Furthermore, both lines have the same bend from 32 to 64 nodes as the baseline.

With 100 application threads per node (up to 6,400 in total), the latency increases noticeably, as expected, because the CPU is highly overprovisioned. In this situation, the latency between writing a message into the ORB and sending it increases dramatically with more open socket channels. Furthermore, requests can be aggregated more efficiently in the ORBs with less open connections masking the overhead with a few nodes.

The latency experiments show that EthDXNet scales up to 64 nodes without impairing latency. With a very high number of application threads (relative to the available cores) the latency increases, as expected, but is still good.

D. Yahoo! Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark 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 [8] which

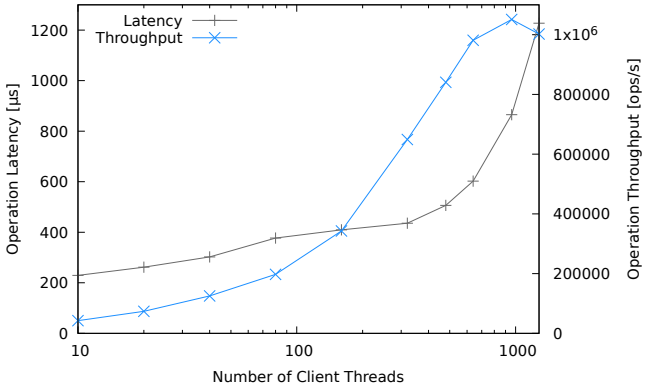

Figure 21. 6 Message Handlers.

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.

Figure 21 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).

XIV. Conclusion and Future Work

Big data applications, as well as large-scale interactive applications, are often implemented in Java and typically executed on many nodes in a cloud data center. Efficient network communication is crucial for these application domains. RMI while being comfortable to use is not fast enough. 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-

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

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 concurrent deserialization. DXNet offers high-throughput asynchronous messaging as well as synchronous request-response communication with very low latency. 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. Finally, its architecture is open for supporting different transport protocols. It already supports TCP with Java.nio and reliable verbs for InfiniBand. We described our practical experiences in designing a transport implementation for Ethernet networks, EthDXNet, integrated into DXNet. EthDXNet provides a double-channel based automatic connection approach using back-channels for sending flow control data and an efficient operation interest handling which is important to achieve low-latency message handling with Java.nio's Selector.

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 evaluation in the Microsoft Azure cloud shows the scalability of EthDXNet (together with DXNet) achieving an aggregated throughput of more than 83 GByte/s with 64 nodes connected with 5 GBit/s Ethernet (10 GBit/s Ethernet limited by SLAs). Request-response latency is almost constant for an increasing number of nodes as long as the CPU is not overloaded. Future work includes experiments on larger scales with application traces.

The InfiniBand transport IBDXNet is work in progress and the 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.

This work has been partially funded by the German DFG.

REFERENCES

[1] K. Beineke, S. Nothaas, and M. Schoettner, "Scalable messaging for java-based cloud applications," *ICNS 2018, The Fourteenth International Conference on Network and Services*, vol. 14, pp. 32–41, May 2018.

[2] 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.

[3] 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.

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

[5] S. Microsystems, "Java remote method invocation specification," https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html, accessed: 2018.11.17.

[6] Oracle, "Package java.net," https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html, accessed: 2018.11.17.

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

[8] 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.

[9] S. Nothaas, K. Beineke, and M. Schöttner, "Distributed multithreaded breadth-first search on large graphs using dxgraph," in *Proceedings of the First International Workshop on High Performance Graph Data Management and Processing*, ser. HPGDMP '16, 2016, pp. 1–8.

[10] K. Beineke, S. Nothaas, and M. Schoettner, "Dxnet project on github," https://github.com/hhu-bsinfo/dxnet, accessed: 2018.11.17.

[11] 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.

[12] R. Noronha and D. K. Panda, "Designing high performance dsm systems using infiniband features," *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, pp. 467–474, 2004.

[13] 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.

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

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

[16] 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.

[17] 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

[18] 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.

[19] 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: 2018.11.17.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] K. Beineke, S. Nothaas, and M. Schoettner, "Fast parallel recovery of many small in-memory objects," in *2017 IEEE 23rd International*

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Conference on Parallel and Distributed Systems (ICPADS), Dec. 2017, pp. 248–257.

[25] Oracle, "Java i/o, nio, and nio.2," https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html, accessed: 2018.11.17.

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

[27] 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.

[28] 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.

[29] W. Pugh and J. Spacco, *MPJava: High-Performance Message Passing in Java Using Java.nio*. Springer Berlin Heidelberg, 2004, vol. 16.

[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: 2018.11.17.

[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.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org