Scalable Messaging for Java-based Cloud Applications

Kevin Beineke, Stefan Nothaas and Michael Schöttner

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

***Abstract*—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 a Java.nio transport for DXNet, a low latency and high throughput messaging system which goes beyond message passing by providing a fast parallel object serialization. The proposed design uses a zero-copy send and receive approach avoiding copying data between de-/serialization and send/receive. It is based on Java.nio socket channels complemented by application-level flow control to achieve low latency and high throughput for >10 GBit/s Ethernet networks. Furthermore, a scalable automatic connection management and a low-overhead interest handling provides an efficient network communication for dozens of servers, even for small messages (< 100 bytes) and all-to-all communication pattern. The evaluation with micro benchmarks shows the efficiency and scalability with up to 64 virtual machines in the Microsoft Azure cloud. DXNet and the Java.nio-based transport are open source and available on Github.**

Keywords–Message passing; Ethernet networks; Java; Data centers; Cloud computing;

I. INTRODUCTION

Big data processing is emerging in many application domains whereof many are developed in Java or are based on Java frameworks [1][2][3]. 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 very important. Fortunately, public cloud data centers already provide 10 GBit/s Ethernet and faster.

Java applications have different options for exchanging data between Java servers, ranging from high level Remote Method Invocation (RMI) [4] to low-level byte streams using Java sockets [5] or the Message Passing Interface (MPI) [6]. However, none of the mentioned possibilities offer high performance messaging, elastic automatic connection management, advanced multi-threaded message handling and object serialization all together. Therefore, we proposed DXNet [7], a network messaging system which meets all of these requirements. DXNet is extensible by transport implementations to support different network interconnects.

In this paper, we propose an Ethernet transport implementation for DXNet, called EthDXNet. The transport is based on Java.nio and provides high throughput and low latency networking over Ethernet connections.

The contributions of this paper are:

- the design of EthDXNet and practical experiences including:
  - scalable automatic connection management
  - zero-copy approach for sending and receiving data over socket channels
  - efficient interest handling
- evaluations with up to 64 VMs in the Microsoft Azure cloud

The evaluation shows that EthDXNet scales well while per-node message throughput and request-response latency is constant from 2 to 64 nodes, even in an high-load all-to-all scenario (worst case). Furthermore, high throughput is guaranteed for small 64-byte messages and saturation of the physical network bandwidth (5 GBit/s) with 4 KB messages. The latency experiments also show that EthDXNet efficiently utilizes the underlying network as long as the CPU does not get overstressed by too many application threads leading to a natural increase in latency.

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 sending and receiving procedure of EthDXNet, followed by a presentation of the connection management in Section V. Section VI focuses on the flow control implementation and Section VII on the interest handling. The evaluation is in Section VIII. The conclusions can be found in Section IX.

II. RELATED WORK

In this section, we discuss the related work for this paper.

A. JavaRMI

Java's RMI [4] 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. Missing classes can be loaded from remote servers during RMI calls, which is very flexible but introduces even more complexity and overhead. Additionally, the built-in serialization is known to be slow and not very

 ISBN: 978-1-61208-633-0

space efficient [8][9]. Furthermore, method calls are always blocking.

B. MPI

MPI is the state-of-the-art message passing standard for parallel high performance computing. It is available for Java applications by implementing the MPI standard in Java or wrapping a native library. However, MPI was designed for spawning jobs with finite runtime in a static environment. DXNet's and EthDXNet's main application domain are ongoing applications with dynamic node addition and removal (not limited to). The MPI standard defines the required functionality for adding and removing processes, but common MPI implementations are incomplete in this regard [10][11]. Furthermore, job shutdown and crash handling is still improvable [11].

C. 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 [12][5]. 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 [13] introduced with `Java.nio` [12][14]. (1) Instead of byte arrays, the read/write 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. Obviously, this relieves the garbage collector as well 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, afterwards.

D. Java Fast Sockets

Java Fast Sockets (JFS) is an efficient Java communication middleware for high performance clusters [15]. 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 as well. EthDXNet is an Ethernet transport implementation for DXNet.

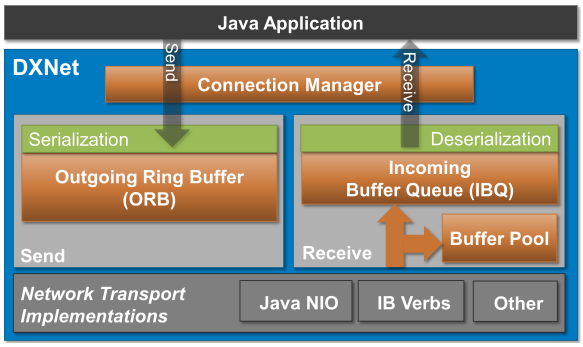

Figure 1. Simplified DXNet Architecture (from [7])

III. DXNET

DXNet is a network library for Java targeting, but not limited to, big data applications. DXNet implements an **event driven message passing** approach and provides a simple and easy to use application interface. It is optimized for highly multi-threaded sending and receiving of small messages by using **lock-free data structures, fast concurrent serialization, zero copy and zero allocation**. Split into two major parts, the core of DXNet provides automatic connection management, serialization of message objects and an interface for implementing different transports. Currently, an Ethernet transport using Java.nio sockets and an InifiniBand transport using ibverbs is implemented.

This section describes the most important aspects of DXNet and its core (see Figure 1) which are relevant for this paper. However, a more detailed insight of the core is given in a dedicated paper [7]. The source code is available at Github [16].

A. Connection Management

All nodes are addressed using an **abstract 16-bit node ID**. Address mappings must be registered to allow associating the node IDs of each remote node with a corresponding implementation dependent endpoint (e.g., socket, queue pair). To provide scalability with up to hundreds of simultaneous connections, our event driven system does not create one thread per connection. A **new connection is created automatically** once the first message is either sent to a destination or received from one. Connections are closed once a configurable connection limit is reached using a recently used strategy. Faulty connections (e.g., remote node not reachable anymore) are handled and cleaned up by the manager. Error handling on connection errors or timeouts are propagated to the application using exceptions.

B. Sending of Messages

Messages are Java objects and sent asynchronously. A message can be targeted towards one or multiple receivers. Using the message type **Request**, it is sent to one receiver. The

 ISBN: 978-1-61208-633-0

sender waits until receiving a corresponding response message (transparently handled by DXNet) or skips waiting and collects the response later.

One or multiple application threads call DXNet (concurrently) to send messages. Every message is automatically and concurrently serialized into the **Outgoing Ring Buffer (ORB)**, a natively allocated and lock-free ring buffer. When used concurrently, messages are automatically aggregated which increases send throughput. The ORB, one per connection, is allocated in native memory to allow direct and zero-copy access by the low level transport. The transport runs a decoupled dedicated thread which removes the serialized and ready to send data from the ORB and forwards it to the hardware.

C. Receiving of Messages

The network transport handles incoming data by writing it to **pooled native buffers**. We use native buffers and pooling to avoid burdening the Java garbage collection. Depending on how a transport writes and reads data, the buffer might contain fully serialized messages or just fragments. Every buffer is pushed to the ring buffer based **Incoming Buffer Queue (IBQ)**. Both, the buffer pool as well as the IBQ are shared among all connections. Dedicated **message handler threads** pull buffers from the IBQ and process them asynchronously by de-serializing them and creating Java message objects. The messages are passed to pre-registered callback methods of the application.

D. Flow Control

DXNet implements its own **flow control (FC)** mechanism 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.

E. Transport Interface

DXNet provides a transport interface allowing implementations of different transport types. One of the implemented transports can be selected on the start of DXNet. The transport and its specific semantics are transparent to the applications.

The following methods must be implemented for every transport:

- Setup connection
- Close and cleanup connection
- Signal to send data available in the ORB of a connection (callback)
- Pull data from the ORB and send it
- Push received raw data/buffer to the IBQ

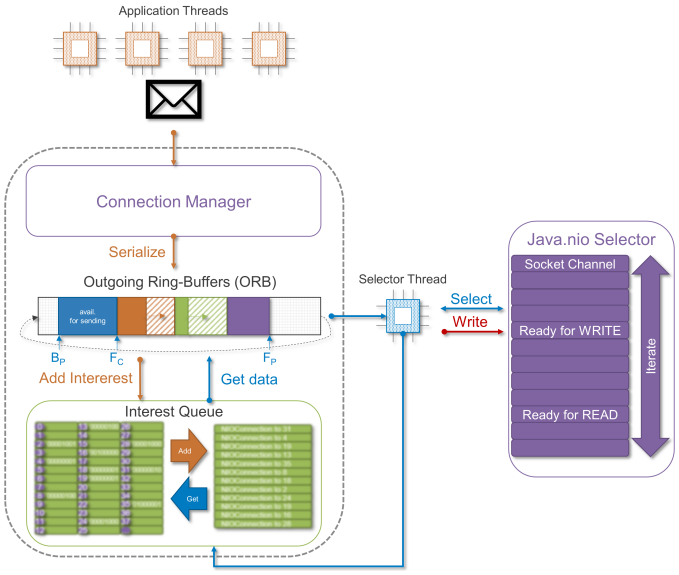

Figure 2. Data structures and Threads. Details of the Interest Queue can be found in Figure 4.

IV. 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 depict in Figure 2.

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 transfered. 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 [7]). 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 I) for given connection in the **Interest Queue** (see Section VII). 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 transferring 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

 ISBN: 978-1-61208-633-0

messages are naturally aggregated to data blocks in the ORBs, which is a performance critical aspect. The write method is called repeatedly 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 afterwards, 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 V). 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 are able to 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 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 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 described in [7].

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, configuring the kernel socket buffer sizes (one for sending and one for receiving) correctly has a great impact on performance. We empirically determined setting both buffer sizes to the ORBs' size offers a 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 [7].

In order to read from the socket channel, a ByteBuffer is required to write the incoming data into. Constantly allocating new ByteBuffers decreases the performance drastically. Therefore, we implemented a **buffer pool**. The buffer pool provides ByteBuffers, allocated in native memory (which are DirectByteBuffers), in different configurable sizes (e.g., 8×256 KB, 256×128 KB and 4096×16 KB). The SelectorThread pulls DirectByteBuffers using a worst-fit strategy as the amount of bytes ready to be received on the stream is unknown. It can also scale-up dynamically, if necessary. The buffer pool management consists of three lock-free ring buffers optimized for access of one consumer and N producers [7].

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 within the ByteBuffer.

V. Automatic Connection Management

For sending and receiving messages, we have 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 3 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

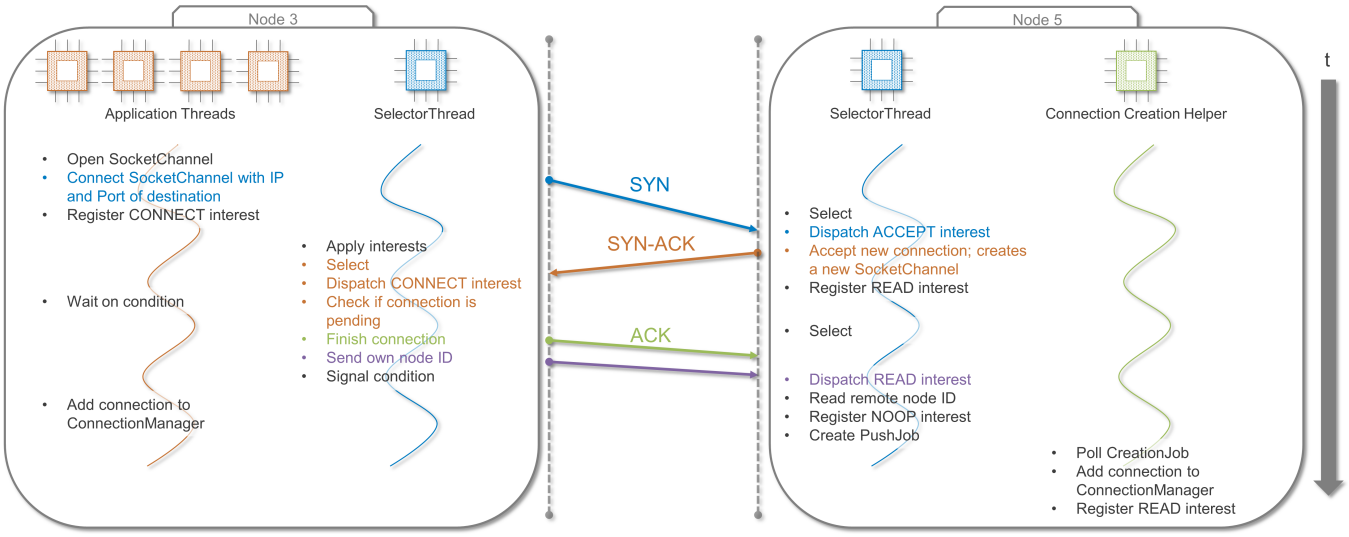

Figure 3. Connection Creation

the connection creation was aborted. 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 on the newly created channel. Furthermore, the SelectorThread marks the PipeOut as connected and signals the condition so the application thread can continue. The application thread adds the connection to the connection manager, increments the connection counter and starts sending data, afterwards.

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. Afterwards, the interest `READ` is registered in order to receive the node ID of the remote node. After selecting and 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 ConnectionObject has yet to be created and pushed to the connection manager. This process is rather time consuming and might be blocking if an application thread creates a connection to the same node at the same time (connection duplication is discussed in Section V-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 V-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 opened and closing the other one. Obviously, the other node must decide consistently which can be done by including 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 initially to be sent over the closing connection has 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

 ISBN: 978-1-61208-633-0

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 very small socket receive buffer for the outgoing socket channel and a very small 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 VI).

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 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. Afterwards, a CLOSE interest is registered to close the socket channels asynchronously. The SelectorThread does not shut down the socket channels on first opportunity but postpones shutdown for at least two RTT timeouts to ensure all responses are received for still outstanding requests.

VI. Flow Control

DXNet provides a flow control on application layer to avoid overwhelming slower receivers (see Section III). 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 throughout 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 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 actually sending the flow control data, the threshold has been exceeded again. For example, if the current counter is 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

 ISBN: 978-1-61208-633-0

TABLE I. 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

TABLE II. 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

the limit (i.e., the flow control window is full) are blocked until message receipt is acknowledged by the receiver. The read flow control value is used to decremented 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 IV-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 a contiguous high throughput.

VII. 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 has to register 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 I shows all interests specified by Java.nio and Table II 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.

B. Interest Queue

None of the interests in Table II 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 a rather high overhead and a complicated work flow. Instead, **we address this problem with an Interest Queue** (see Figure 4) 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 4 shows the Interest Queue consisting of a byte array storing the operation interests of all connections (left side in Figure 4) and an ArrayList of ConnectionObjects containing connections with new operation interests sorted by time of occurrence (right side in Figure 4).

The byte array has one entry per node ID allowing access time in O(1). The node ID range is limited to 2^{16} (allowing

 ISBN: 978-1-61208-633-0

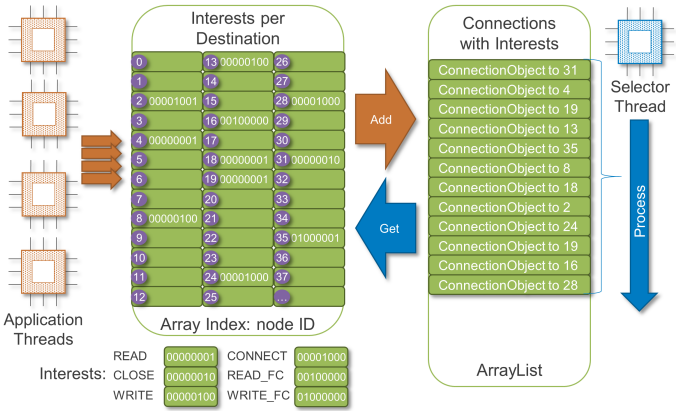

Figure 4. Interest Queue: the application threads add new interests to the Interest Queue. If interest was 0 before, the ConnectionObject is added to an ArrayList.

max. 65,536 nodes per application) which results in a fixed size of 64 KB for the 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 when processing them.

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 important to call it if absolutely 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). If the SelectorThread is currently blocked in the select call, it returns immediately and can process the pending operation interests.

Listing 5 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.

```
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 5. Workflow of SelectorThread

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 V-3).

The order is based on following rules: (1) a connection must be connected 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 connected before. Therefore it is not possible to register CONNECT and CLOSE together.

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

VIII. Evaluation

We evaluated EthDXNet using up to 65 virtual machines (64 running the benchmark and one for deployment) connected with 5 GBit/s Ethernet in Microsoft's Azure cloud in Germany Central. The virtual machines are Standard_DS13_v2 which are memory optimized servers with 8 cores (Intel Xeon E5-2673), 56 GB RAM and a 10 GBit/s Ethernet connectivity, which is limited by SLAs to 5 GBit/s. In order to manage the servers, we created two identical scale-sets (as one scale-set is limited to 40 VMs) based on a custom Ubuntu 14.04 image with 4.4.0-59 kernel and Java 8.

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

 ISBN: 978-1-61208-633-0

TABLE III. 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

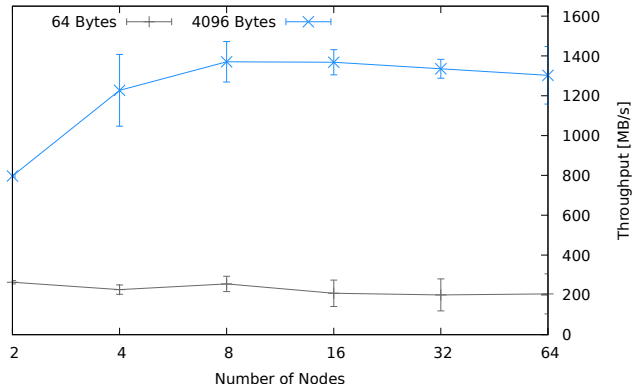

Figure 6. Message Payload Throughput per Node. 1 Application Thread, 2 Message Handler Threads

be sent on IP layer. Additionally, all runs with DXNet's benchmark are **full-duplex** showing the aggregated performance for concurrently sending and receiving messages/requests.

A. 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 III.

Figure 6 shows the average payload throughput for single nodes and Figure 7 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 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.

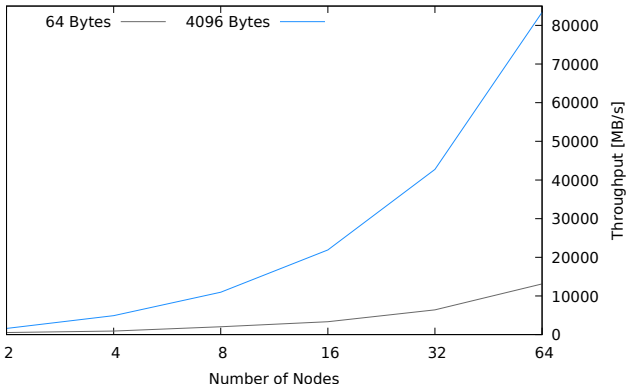

Figure 7. Aggregated Message Payload Throughput. 1 Application Thread, 2 Message Handler Threads

B. Request-Response Latency

The next benchmarks are used to evaluate request-response latency by measuring the **Round Trip Time** (RTT) which includes sending a request, receiving the request, sending the corresponding response and receiving the response. Figure 8 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.

The latency of the Azure Ethernet network is relatively high with a minimum of 352 µs measured with DXNet and one application thread (Figure 8). 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 but is still good.

 ISBN: 978-1-61208-633-0

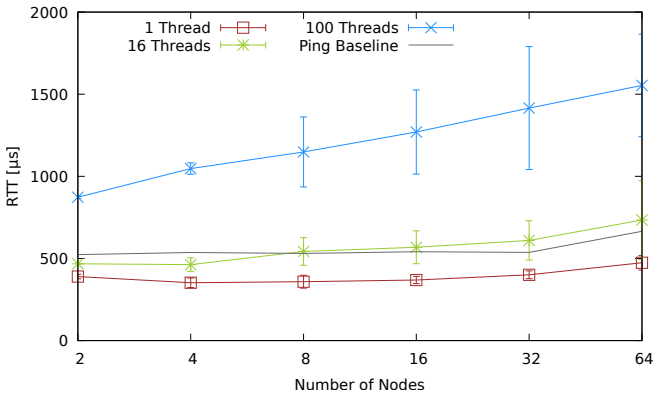

Figure 8. Average Request-Response Latency. 1 to 100 Application Threads, 2 Message Handler Threads

IX. Conclusions

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 very important for these application domains.

In this paper, we described our practical experiences in designing a transport implementation, EthDXNet, based on Java.nio, 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.

Evaluation with micro benchmarks 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.

References

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

[2] 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.

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

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

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

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

[7] K. Beineke, S. Nothaas, and M. Schoettner, "Efficient messaging for java applications running in data centers," Feb. 2018, preprint on webpage at https://cs.hhu.de/en/research-groups/operating-systems/publications.html.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

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

 ISBN: 978-1-61208-633-0