Evaluating Software-defined Networking for Deterministic Communication in Distributed Industrial Automation Systems

Ben Schneider, Alois Zoitl, Monika Wenger
fortiss GmbH
An-Institut Technische Universität München
München, Deutschland
{schneider, zoitl, wenger}@fortiss.org

Jan Olaf Blech
AICAUSE/RMIT University
Melbourne, Australia
janolaf.blech@rmit.edu.au

***Abstract*—As networks continuously grow (keyword Internet of Things) the configuration and management of industrial networks gets more complex. Additionally the trend towards Ethernet networks in the domain of industrial automation leads to the adoption of new Ethernet based technologies for factory automation. One of these technologies is Software Defined Networking (SDN), which provides a programmable and more flexible method for network control through an abstract network view. This work focuses on the evaluation of SDN in the field of industrial automation. It provides the implementation of a prototype SDN controller for direct multicast routing of industrial traffic in a cyclic switched Ethernet network using the Ryu SDN framework. For evaluation, experiments about the performance penalty introduced by switch-controller-communication are performed. An IEC 61499 compliant development environment is used for the experiment definition. The experiment results and the prototype implementation of the SDN controller showed that SDN provides great opportunities for a flexible and reliable network setup in the automation industry and is suited for real-time traffic.**

***Index Terms*—SDN, UDP multicast routing, Ryu, IEC 61499, switched industrial Ethernet**

I. Introduction

The shift from mass production to individualized production increases the demand for higher flexibility of plant and network. SDN can be the future key technology to solve these upcoming problems by providing a flexible and reliable method for network control, which is achieved through the separation of data and control plane in forwarding devices. This leads to a programmable network and introduces additional networking information that can influence the forwarding decision-making.

The trend towards Ethernet based networks in industrial automation [1] [2] makes SDN a highly auspicious technology for networks of the fourth industrial revolution. The use cases of SDN in the field of industrial automation are very versatile. The main use cases are *traffic engineering*, e.g., for the separation of experimental and productive network traffic, or *security* such as the implementation of innovative, more advanced and highly customized security policies. The *programmability* of the network is also important for the reusability and reconfiguration of network nodes. In case the network requirements change, the only entity that needs to be reconfigured is the SDN controller, instead of every single state-of-the-art device in the specific network segment.

978-1-5090-6505-9/17/$31.00

A more specific use case for SDN in the field of industrial automation is the shortest path routing of User Datagram Protocol (UDP) multicast traffic in a ring topology, without packet flooding and duplication. The first **contribution** of this paper is an SDN controller implementation that fulfills these requirements. The routing application uses topology and multicast group information which are managed by the SDN controller and results in the shortest path routing of multicast packets to all hosts in the specific multicast groups. The second contribution of this work is the evaluation of the performance penalty introduced by the switch-controller-communication of the SDN controller that is evaluated based on networking experiments.

The paper is structured as follows: Section 2 gives essential background information on SDN and related work is summarized in Section 3. Section 4 describes the implementation of the SDN controller for direct UDP multicast routing. The performed experiments are summarized in Section 5 and Section 6 concludes with the results and gives starting points for future work.

II. Background

The main difference between a state of the art networking device and an SDN enabled networking device is the decoupled control and data plane. The data plane of a forwarding device is responsible for packet transmission according to forwarding rules installed on the plane. The control plane contains information and algorithms on how to modify the data plane entries (logic of the forwarding device). A state of the art network device contains both planes, whereas an SDN enabled forwarding device only contains a data plane. The control plane (SDN controller) is for example located on a general-purpose computer which is connected to the forwarding device. This approach reduces the complexity of networking device configuration by combining control planes of different forwarding devices in a single entity - the SDN controller. Another advantage of the centralized controller is

the enhanced network overview which enables more advanced algorithms for network control. More information about SDN can be found in "SDN: A Comprehensive Survey" [3].

The OpenFlow protocol [4] is an open source SDN protocol for switch-controller-communication (SouthBound Interface). It manages flow rules on OpenFlow enabled switches. The most important entries of a flow rule are *Match Fields* and *Instructions*. Match Fields specify the packet filter and Instructions contain the performed action set that is executed in case of a packet match. Such a match could for example be a match on multicast destination IP address 239.192.0.1 which could cause the actions "output at port 1" and "output at port 3" to be executed.

Ryu[1] is a component-based, open source SDN framework for programming network controllers in Python which supports various control protocols, e.g., OpenFlow v. 1.0 - 1.5, Netconf or SNMP. A Ryu application is a single-threaded event-handling entity which is able to send messages from one Ryu application to another or receive external messages for example events from an SDN enabled switch. Ryu is widely used for research projects and education and provides good sources for documentation which makes it very useful framework for rapid prototyping.

III. Related Work

The conceptual overview of SDN in the field of industrial automation is given Cronberger et al. [1] and Khandakar et al. [5]. [1] shows an overview of SDN in a conceptual way in industrial automation with Cisco's proprietary SouthBound protocol onePK which is an alternative for OpenFlow. [5] proposes the adoption of SDN in the field of Cyber-physical System (CPS) by providing an architecture for software-defined networks in industrial automation using an SDN adoption of PROFINET called SDNPROFINET which supports flexible communication and maintenance and allows easier reconfiguration of factories.

Herlich et al. [6] evaluate SDN for industrial real-time Ethernet traffic by a proof of concept on a virtual platform showing typical use cases in the field of industrial automation like traffic separation and reconfiguration of the network.

Henneke et al. [7] analyse requirements of SDN for the realization of future industrial networks, for example real-time monitoring of the network, stable and secure configuration platforms and effective operation on legacy hardware, among others.

Thiele et al. [8] provide a model for formal verification of real-time capabilities in software-defined networks and shows the synergies of Time-Sensitive Networking (TSN) and SDN. The verification model is evaluated for a number of SDN requests per switch, the SDN frame size, the SDN processing time and the SDN traffic priorities. The formal analysis showed that an SDN network latency well below 50ms is possible in Ethernet networks.

[1] https://osrg.github.io/ryu/

Corici et al. [9] implemented the SDN controller framework openSDNCore and evaluated its performance on virtual hardware using cbench for latency and throughput of the controller.

Pfrommer et al. [10] propose a model for static code analysis with Frama-C/Para-C. They use Lua and OPC-UA as means for ensuring a flexible runtime and support network reconfiguration by the integration of an SDN controller.

Schweissguth et al [11] implemented an SDN controller that uses a Time Division Multiple Access (TDMA) approach with topology information and application requirements to route traffic and guarantee real-time capabilities.

Jiang et al [12] propose an extended implementation of the Dijkstra Shortest Path algorithm which takes the weight of nodes into account (calculated from the statistics of the SDN switch).

The related work mentioned here is either theoretical and conceptional (e.g., [1], [5], or [6]) or running in an virtual environment (e.g., [9], [12] or [12]). This work focusses on a use case in the field of industrial automation which is evaluated on real hardware.

IV. An SDN Controller for Direct Routing of Multicast Traffic

This section presents the concept for shortest path routing of UDP multicast traffic in a network that contains a ring. This concept is implemented and evaluated with an example network managed by an SDN controller. The requirements for the use case is given first, followed by the concept description and the implementation including an example.

A. Use Case and Requirements Analysis

A typical use case for a network in the field of industrial automation is publish/subscribe (UDP multicast) based communication in a circular network topology [13].

Traditional switches flood (forward a packet on all ports) multicast packets to make sure that every interested host receives the messages. Message flooding introduces the problem of packet duplication for networks containing a ring. Consider Figure 1 which shows an example network topology for this use case. If H 1 sends a multicast packet to H 2 and H 3, S 1 forwards the packet to S 2 and S 4 which would forward the packet to H 2 and H 4 and twice to S 3. The problem is not only that Host 3 receives the packet twice, but also that Switch 3 forwards the packet from Switch 2 to Switch 4 and from Switch 4 to Switch 2 again because of the flooding strategy which causes an infinite loop on the network. Every host on this loop receives the packet multiple times, endlessly.

The specific **use case** and the resulting requirements for an SDN controller which manages an industrial network can be described like follows: The network controller, connected to all SDN enabled switches, shall be able to directly route UDP multicast traffic to the desired receivers via the shortest paths without the need of flooding packets at any time, as flooding causes packet duplication in a ring topology. The routing of messages is based on multicast group management and the topology discovery of the network. The forwarding

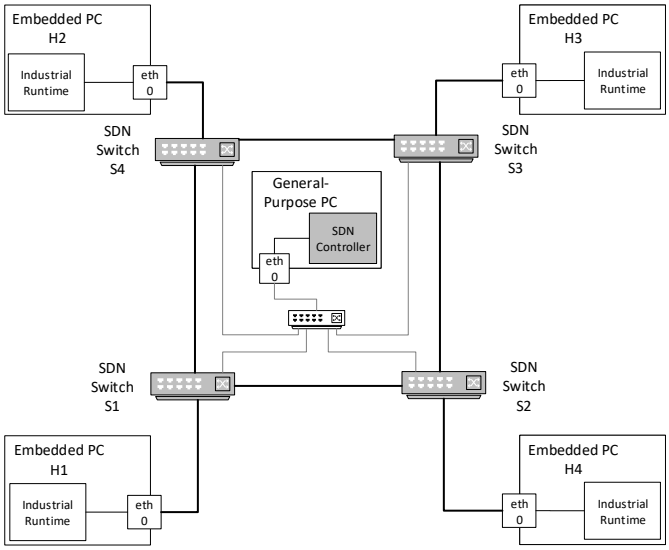

Fig. 1. Example network topology

delay of the traffic caused by the controller needs to be as low as possible, which shall be achieved by preferring proactive forwarding decisions to reactive decisions. The networking scope is limited to switched Ethernet networks and all switches need to support the OpenFlow protocol.

B. Concept Description

The SDN controller's execution can be grouped into three phases: configuration, operation and reconfiguration.

A communication pattern for UDP multicast traffic is Publish/Subscribe, with Internet Group Management Protocol (IGMP) for multicast group management. IGMP messages are sent during the initialization of the communication in order to configure the network. An IGMP packet does not only contain information on which host belongs to which multicast group, it also guarantees that the receiving hosts of multicast traffic are discovered and added to the network topology before the first multicast packet is sent, as the SDN controllers detect hosts in their network segment when the first communication happens.

An example ring topology can be seen in Figure 1 which contains four embedded PCs (hosts H1, H2, H3 and H4), connected to four SDN switches (S1, S2, S3 and S4) via their Ethernet Interface (eth0) and one SDN controller (C1) that is connected to all SDN switches via a traditional, commercial off the shelf (COTS) switch.

The operation phase is characterized by the shortest path routing of multicast traffic. Routing is especially important for networks with a ring topology, as traditional switches flood multicast packets to make sure that all hosts receive the packets. Flooding packets in a ring topology causes packet duplications without special switching hardware.

The shortest path routing solves this problem, by calculating the path to every host in the multicast group and writing the flows to all switches in the path. A packet from Host 1 to Host 2 and Host 3 is sent via Switch 4 and Switch 3 and is not duplicated on the ring, as the packet is not even forwarded to Switch 2.

The second aspect of the shortest path routing is the delay that is introduced by the switch-controller communication. This delay can be reduced by not calculating the flows for a single switch once the packet arrives at the switch (reactive forwarding strategy), but for calculating the path for a packet when the first switch receives it and modifying all flow tables on switches in the packet's path. This strategy reduces the overhead for computation, as the path only needs to be calculated once and not on every switch and makes the forwarding pattern proactive except for the very first switch in a path which means that the flows for a specific packet are written to the switches before the actual forwarding happens.

The reconfiguration of the network can be realized by flow modifications. In case the network topology (e.g., host connects or link failure) or the multicast group information (e.g., hosts join or leave a group) changes, the routing can be adopted by removing the specific flows from the switches. This will forward the following multicast packet affected by the change to the SDN controller (table-miss flow entry) and a new shortest path for the new network setup can be calculated and written to the switches.

The steps the SDN controller needs to perform in order to successfully route multicast packets in a network containing a ring is summarized in the following listing. The only assumption is an existing table-miss flow entry that forwards packets to the SDN controller.

1) Perform topology discovery for switches, links and hosts
2) Parse incoming IGMP packets and extract multicast group information (join or leave multicast groups)
3) Parse the multicast packet and extract relevant information (e.g., Datapath ID of the switch, Source Media-Access-Control (MAC) address, Destination multicast MAC address)
4) Generate the match for following packets (e.g., destination matching on the multicast MAC address)
5) Look-up all hosts in the multicast group of the destination MAC
6) Calculate the shortest path to all hosts in the multicast group
7) Extract the port forwarding information for all hops in the calculated shortest path
8) Assemble flow entries (match and actions) for each switch in the path
9) Write the generated flows to all switches in the path
10) Send the packet back to the switch which results in the execution of the generated flow rules (e.g., fowarding the packet at a specific port)

The reconfiguration of the network is performed by executing step 3) to 10) again which happens automatically if the flows for the appropriate packet transitions are removed from the SDN switches, e.g., when topology or multicast group information changes.

C. SDN Controller Implementation for Direct, Shortest Path Multicast Routing

IEC 61499 is used as reference standard for future distributed industrial automation systems. The standard provides software reusability through the use of event-triggered function blocks which encapsulate automation software and interoperability through the use of a standardized data exchange format that enables the execution of a program written in IEC 61499 on any IEC 61499 compliant runtime environment. The example applications have been implemented using Eclipse 4diac™ [2] and have been tested on 4diac Runtime Environment (FORTE), where the hosts communicated with UDP multicast messages via Publish and Subscribe Service Interface Function Block (SIFB)s.

The software of the SDN controller is based on Ryu and its topology Application Programmable Interface (API) for topology discovery, Python's matplotlib[3] for the graphical representation of the network and Python's networkX library[4] for the graph representation (data structure) and shortest path algorithms.

The SDN controller consists of three components, which are an IGMP Handler for controlling the multicast group members, a Topology Handler that discovers the topology of the underlying network and the Routing Application which calculates the shortest paths and writes the appropriate flow rules to the switches depending on network topology and multicast group information that are provided by the other two parts of the controller application.

Consider Figure 1 as an example network topology again. In this case, host H1 sends a multicast packet to multicast group G1 which contains H2 and H3. The following paragraphs describe the implemented SDN controller application in detail with the aid of this example.

The controller is started and connects to the SDN switches S1, S2, S3 and S4. This OpenFlow handshake adds the switches to the topology of the network managed by the controller. After the connection of the switches has been established, the controller discovers the links between the switches by sending Link Layer Discovery Protocol (LLDP) packets. The discovery of the receiving hosts is the next step. When a Subscriber SIFB is initialized, it sends an IGMP join report to the all-multicast-group-address (224.0.0.1), which is snooped by the SDN controller through the table-miss flow entry. The first effect of this message is that the controller discovers the host which is done, when the first packet of the host is received by an SDN controller and adds it to the topology of the network. The second effect is that the SDN controller parses the IGMP join report and adds the sender's MAC address to the multicast groups that are listed in the payload of the message.

Therefore, the first messages the SDN switches receive are IGMP join reports from H2 to S4 and H3 to S3 containing the information that H2 and H3 want to join the IGMP multicast group (in this case called G1). The result of these messages is that the sending hosts are added to the network topology and the hosts join the multicast group with the sent MAC address G1. The IGMP packets are dropped afterwards, as there is no router available in the example network (scope of the implementation limited to switched Ethernet networks) which could be interested in the multicast group information. The configuration phase of the SDN controller is finished, when all the IGMP received from Subscriber SIFBs are parsed.

The first step that is performed in the operation phase of the application is the discovery of the host that sends the first UDP multicast packet which is host H1. The next step is the actual packet transmission that is initiated when H1 sends the packet. The controller receives this multicast packet (table-miss) from the first switch, calculates the shortest paths using a Dijkstra shortest path algorithm to all Subscribers in the network that are currently known to the SDN controller and writes the appropriate flow entries to all switches on the calculated paths which are S1, S4 and S3. The resulted path is afterwards transformed into the specific flow rules for every switch in the path. When the packet is forwarded to switch S4 and S3, they already have a matching flow rule and can proactively forward the received message without the need for an additional switch-controller-communication which means that all multicast packet forwarding is proactive except the transmission at the very first switch S1.

Reconfiguration of the paths is performed when topology or multicast group information of the SDN controller is updated. This can happen in two cases. First, a host connects to the network and is discovered by an initial IGMP packet that is transmitted to an SDN switch. Second, a host is removed from its belonging IGMP multicast group via an IGMP leave report. Both cases result in the deletion of the flow rules associated with the updated multicast group. The next time a UDP packet is sent by a Publisher SIFB, the shortest paths for the Subscribers in the multicast group are recalculated and new flows are written to the SDN switches. A silent leave of hosts is currently not supported. This can for example happen when a host disconnects from the network without sending an IGMP leave report.

D. Implementation Results

The result of the controller implementation is shown in Figure 2 which is based on the output of the SDN application. Hosts are blue circles labeled with H and SDN switches are red circles labeled with S. The ports for a specific switch are labeled with p. Every switch is connected to the same SDN controller which is not shown in the figure. Datapath IDs of switches and MAC addresses of hosts have also been annotated after the execution of the program.

The SDN controller for the direct, shortest path routing of UDP multicast traffic showed that the problem of packet duplication in a network containing a ring topology is solved like described in the section above. The arrows in Figure 2 show the shortest paths that are taken by the packets from H1

[2] http://www.eclipse.org/4diac/
[3] http://matplotlib.org/
[4] https://networkx.github.io/

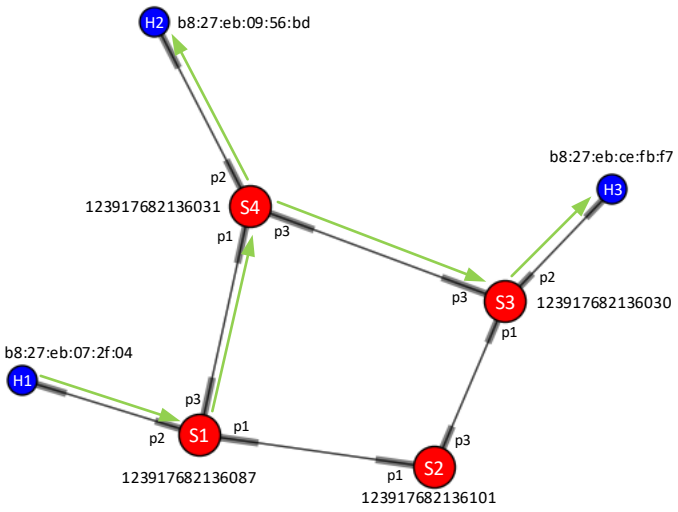

Fig. 2. Implementation result for the SDN controller that diretly routes multicast traffic to all interested hosts

TABLE I
GENERATED FLOW RULES FOR UDP MULTICAST TRAFFIC FROM H1 TO H2 AND H3

Switch	MAC dst	...	Instructions
S1	G1	*	Apply-Actions: Output: p3
S4	G1	*	Apply-Actions: Output: p2, Output: p3
S3	G1	*	Apply-Actions: Output: p2

to H2 and H3. The forwarding delay is minimized by proactive flows for all switches except the very first (in this example S4 and S3 are proactive, S1 is reactive). For this example the resulting flow rules are listed in Table I. The host connected to switch S2 was not recognized by the SDN controller as it did not send an IGMP join report for the example packet transmission from H1 to H2 and H3.

The next section gives first insights on the performance penalty introduced by the switch-controller-communication of the SDN controller.

V. EVALUATING LATENCY OF SDN SWITCH-CONTROLLER-COMMUNICATION

This section focusses on the performance penalty introduced by the SDN switch-controller-communication. The performed experiments are seen as a point of reference for the evaluation of SDN in the field of industrial automation, for example by a comparison between the delay of proactive and reactive forwarding strategies. The scope of the experiments is reduced to the communication between two devices, connected via different network interfaces. This setup is expected to produce more meaningful results for comparing reactive and proactive forwarding strategies.

The **experiments** are briefly described as follows:

- Two hosts directly connected via an Ethernet cable (no forwarding device)
- Two hosts connected via a traditional low cost COTS desktop switch (learning switch)
- Two hosts connected via an OpenFlow enabled enhanced learning switch which operates a completely proactive forwarding strategy
- Two hosts connected via an OpenFlow enabled enhanced learning switch that follows a completely reactive forwarding strategy

The first part of this section provides detailed information about the hardware and software, followed by the description of the network experiments and the discussion of results.

A. Hardware and Software Configuration

This section focusses on the hardware and software that is used for the SDN experiments and their configuration. The following devices are used in the network experiments:

- Raspberry Pi 3 Model B V1.2 with Raspbian 8.0
- SDN switch Zodiac FX with firmware version 0.63
- low cost COTS desktop switch Edimax ES-5800G V3 (8 port, layer 2)

The Ethernet port's bandwidth of the Raspberry Pi and Zodiac FX are limited to maximum 100Mbps, which sets the upper bound of the bandwidth for the experiments to 100Mbps even if the Edimax desktop switch provides 8 1000Mbps Ethernet ports.

The configuration of the Zodiac FX SDN switches is straightforward. All switches communicate to the same SDN controller (Ryu running on a Raspberry Pi). The OpenFlow version is forced to 1.3 and the fail state of the OpenFlow devices is set to either `secure` (experiment four) which means that the switch stops sending packets if the controller disconnects, or `safe` (experiment three) which means that the switch still forwards packets in case the controller disconnects from the device.

The software used for performing the experiments is Ryu 4.5 as SDN controller framework and FORTE 1.8 as IEC 61499 compliant runtime environment for industrial automation systems. Ryu's learning switch example for OpenFlow 1.3 is used as basis for the SDN experiments (simple_switch_13.py[5]). The learning switch code is enhanced by the IGMP Handler described in Section IV which parses IGMP packets and manages the multicast groups of the network segment the SDN switch is responsible for. Instead of the standard layer 2 forwarding based on the destination MAC address (flooding in case of UDP multicast on a traditional switch), the destination multicast MAC address is looked up in the multicast group table managed by the controller and the packet is sent to all hosts that are registered in the specific multicast group (messages are not flooded).

The Raspberry Pis automatically start FORTE after booting by a start script. A boot file, containing the experiment definition written as an IEC 61499 compliant function block network, is loaded when FORTE starts. This approach ensured the repeatability of the experiments. Only the experiment setup

[5]https://github.com/osrg/ryu/blob/master/ryu/app/simple_switch_13.py

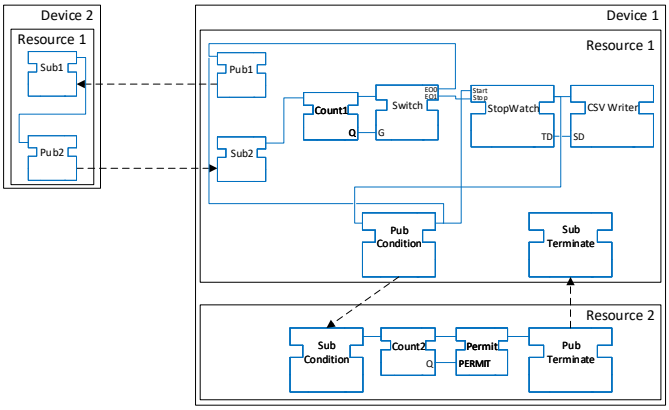

Fig. 3. Schematic and simplified IEC61499 application for measuring RTTs of packets with different network setups

(hardware composition and network setup) are changed among the four experiments. Any restart of the Raspberry Pis starts an experiment from scratch with the physically configured network setup.

A custom Function Block (FB) (StopWatch) for time measuring is implemented. This function block uses the C++ `chrono::high_resolution_timer` which can be started and stopped by two input events and provides a data output for the time difference between the start and stop event. This time difference is the input for a CSV Writer FB which records the different times and writes it to a comma-separated value file.

The IEC 61499 compliant experiment definition is illustrated in Figure 3. The applications running on two devices are simplified (e.g., all event connections responsible for the correct initialization of the FBs are removed) in order to decrease the complexity of the figure. Resource 1 of Device 1 contains the main part of the application. The Publisher sends the UDP multicast messages which starts counter and timer for the first message of a batch. Device 2 immediately answers the message with another UDP multicast sent to Device 1. The number of messages is counted and in case the size of the message batches is reached the time is stopped by the StopWatch FB. In other words, the application measures the Round Trip Time (RTT) for UDP message batches. Resource 2 of Device 1 is responsible for counting the number of these batches and stopping the experiment when the configured number of repetitions is reached.

Five network experiments have been performed in order to determine the performance of a basic software-defined network setup. The first experiment was used to evaluate on the measurement accuracy and the measuring method that suite the experiments best. Therefor one Raspberry Pi with two resources in one FORTE was used. FORTE was transmitting the messages from one resource to the other and back, measuring the RTTs. The smallest time that was measured was 3μs. This time can be seen as lower bound for the other experiments as sending a network packet within a device is considered faster than the packet transmission via

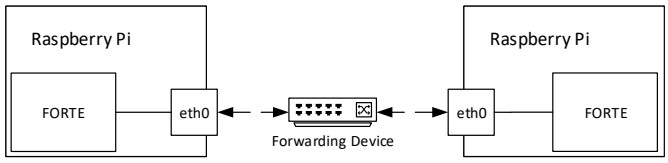

Fig. 4. General experiment structure

a network connection. This lower bound also changed the measuring strategy from measuring the time of single packets to measuring the time of multiple packets and calculating the mean for a single packet transmission from this results. In the following experiments the time for 130000 (10000 for the reactive learning switch experiment) UDP packets was measured 1000 times. This ensured a high measurement accuracy by reducing the uncertainty of the timer.

A meaningful payload size for messages in the automation industry is 4B [13]. This payload is stuffed with zeros as the minimal payload size of an Ethernet packet with UDP header and without the optional header tag for 802.1Q is 18B. Hence, the message size does not influence the network performance as long as it is smaller than 18B. The experiments are performed with a boolean value as payload.

B. Experiment 1: Direct Cable Connection

The general composition of the experiments is depicted in Figure 4. The Raspberry Pis are running the same application (Figure 3) for all the experiments and only the network infrastructure is changed which ensures the repeatability of the experiments and the comparability of the measurment results. The following paragraphs give an overview of the four different experiments.

1) Composition: A sending and a responding Raspberry Pi device is directly connected via an Ethernet cable on their interfaces eth0. The responding device is started before the sending device.

2) Statement: This experiment was performed to get the fastest possible network transmission, for an estimation and comparison of the performance of the upcoming experiments.

C. Experiment 2: Traditional Desktop Switch Connection

1) Composition: Two Raspberry Pis are connected via the Edimax ES-5800G commercial off the shelf desktop switch. The responding device is started before the sending device.

2) Statement: This experiment is intended as a comparison of the traditional desktop switch against the SDN enabled switch (proactive and reactive learning switch approach) described in the next two sections.

D. Experiment 3: Proactive Learning Switch

1) Composition: The forwarding device is a Zodiak FX SDN switch directly connected to an SDN controller running on an additional Raspberry Pi. When the first packet transmission was successful, the appropriate flows are written to the switch. The fail state of the switch is set to `safe` (packets will be forwarded according to existing flows even if the

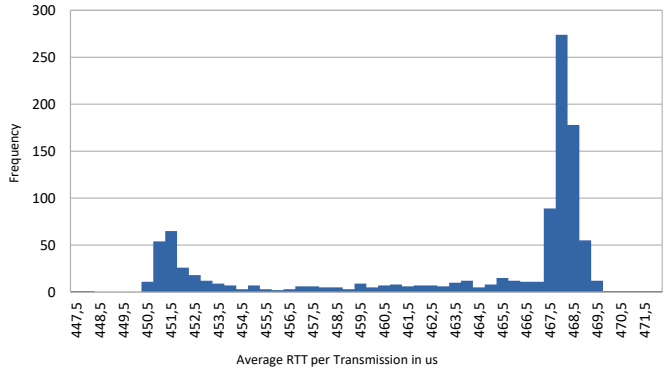

Fig. 5. Proactive experiment results (histogram)

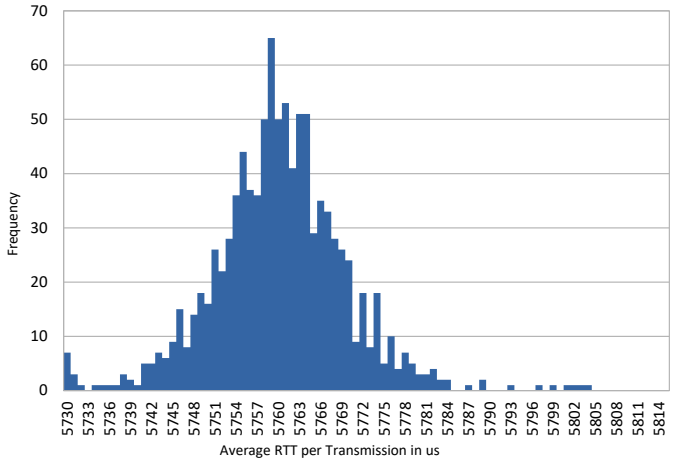

Fig. 6. Reactive experiment results (histogram)

TABLE II
EXPERIMENT RESULTS

	Min	Max	Mean	Std. Deviation	Median	MMTest	Required Sample Size
Direct cable	189,86μs	316,05μs	196,66μs	9,63μs	195,53μs	0,574%	301
COTS switch	211,13μs	344,96μs	222,45μs	12,88μs	220,25μs	0,991%	356
Proactive	447,2μs	471,55μs	463,47μs	6,72μs	467,55μs	—	—
Reactive	5684,8μss	6525μs	5760,75μs	28,95μs	5759,5μs	0,218%	30

SDN controller disconnects). The controller is disconnected from the switch after the flows are written and the sending and receiving Raspberry Pis are restarted in order to start the experiment again with the proactive flows (flows have no time out in this case).

2) Statement: The measurements are estimated to be similar to the results of the COTS desktop switch because the SDN switch is now emulating a COTS switch except that it does not flood packets but forwards them to specific ports (depending on IGMP multicast group information).

E. Experiment 4: Reactive Learning Switch

1) Composition: The structure of this experiment does not differ from the proactive experiment performed before. The only difference is the controller which is programmed to forward the packets without writing flows to the switch and is not disconnected from the SDN switch. Every packet is forwarded to the controller which decides the port a message is forwarded. The controller does not write flows to the SDN switch, but directly forwards the packet (fully reactive forwarding). The controller's fail state is set to `secure`.

As this experiment is estimated to be significantly slower than the experiments before (switch-controller-communication for each packet), the number of the messages for a single sample is reduced to 10000 (instead of 130000).

2) Statement: This experiment is meant to give first insights on the delay caused by the switch-controller-communication and the execution time of the controller code. The results are expected to be much higher, than the results of the measurements before.

F. Comparison and Discussion of Results

This section focuses on the evaluation and interpretation of the measurements. An analysis for the meaningfulness of the experiments is given first. Therefore, the statistical best practices for normal distributed experiment samples have been applied:

- Samples are normal distributed if Mean-Median-Test (MMTest) returns a value smaller than 1%
- Confidence interval and required sample size for a set of samples are calculated

The characteristics for the experiments (except experiment 3 - proactive learning switch) show that they are normal distributed (MMTest $<$ 1%) and meaningful because of the calculated required sample size, which is significantly lower as the real number of samples. The histogram for the only experiment that does not follow a normal distribution is shown in Figure 5 which means that the number of required samples could not be calculated for the proactive experiment. The results for the reactive learning switch in Figure 6 are shown as comparison and reference for the graphs of experiments one and two, which showed a similar normal distributed character.

An overall comparison of all four experiments can be seen in Table II.

The mean RTT value for a single packet can be considered as best possibility for a performance comparison (calculated from the overall time for transmission of one batch and the number of packet transmissions). As expected, the direct cable connection of two devices is the fastest possible connection with a mean RTT of 196,66μs, directly followed by the COTS switch a mean RTT of 222,45μs. This also shows the perfor-

mance of the hardware based COTS switch which is highly optimized for less complex tasks and layer 2 forwarding. The COTS switch also does not forward packets to a specific port, but floods them in case of multicast packets. This can be considered as a reason for the performance difference between the COTS switch and the proactive switching experiment, where the OpenFlow switch emulated the functionality of a COTS switch, but with additional forwarding decision making knowledge. The second reason why the OpenFlow switch is significant slower than the COTS switch is that the Zodiak FX SDN switch is software based instead of hardware based. This increases the RTT in the performed experiments by a factor of around 2 from 222,45μs (COTS switch experiment) to 463,47μs (proactive experiment).

However the most interesting part of the comparison of measurement results is the difference between proactive and reactive packet forwarding. The delay caused by the switch-controller-communication and the execution time of the controller's algorithm is a factor of around 12,5, from 463,47μs (proactive experiment) to 5760,75μs (reactive experiment).

A more complex routing strategy than the learning switch approach used in the experiments like for example the shortest path direct routing of UDP multicast traffic (Section IV) would also increase the RTT. This emphasizes the need for proactive instead of reactive forwarding strategies in software-defined networks which is also shown by the 4 times higher standard deviation of reactive forwarding compared to the proactive strategy. The complexity of the implemented controller has a great impact on the delay and the determinism of the communication.

VI. Conclusion

This work showed how the concept of SDN can be adopted to the field of industrial automation. Therefore, an SDN controller was implemented for the direct routing of UDP multicast traffic in a ring topology. The controller follows a mainly proactive forwarding approach, calculates the shortest paths for all members of the appropriate multicast groups and writes the specific flows to the SDN switches. Four network experiments were performed in order to give first insights into the performance penalty of the switch-controller-communication. These experiments resulted in a 12,5 times higher delay for a reactive forwarding compared to proactive forwarding on the Zodiac FX OpenFlow-enabled switch. In addition, the standard deviation for reactive forwarding is 4 times higher than the proactive standard deviation. Hence, a mainly proactive approach needs to be chosen for deterministic communication in the field of industrial automation.

Future work can be dedicated to the enhancement of the concept, for example by implementing a prototype for load balancing in industrial networks or the implementation of an SDN controller that is able to communicate with a communication coordination layer extension of embedded devices as proposed by Schimmel et al. [13] or by reducing the reconfiguration overhead of the implemented SDN controller. Rerunning the experiments described in Section V on productive SDN hardware may yield different results for the performance of network latencies.

Another promising approach for the industrial networks of the future is the new TSN standard which aims at providing real-time capabilities to standard Ethernet networks. The combination of SDN and TSN needs to be evaluated in the future in order to better understand the synergies between SDN (flexibility and reconfiguration) and TSN (performance and real-time).

Acknowledgment

This work was accomplished in collaboration with RMIT in Melbourne. Special thanks to Prof. H. Schmidt, Dr. I. Peake and Dr. A. Khandakar for their support and very valuable feedback. It is partially funded by the German Federal Ministry of Education and Research (BMBF) under grant no. 01IS16022N through the project BaSys 4.0 http://www.basys40.de/.

References

[1] D. Cronberger, "The software defined indsutrial network," in *Industrial Ethernet Book, The Journal of Industrial Network Connectivity*, October 2014, pp. 8–13.

[2] T. Moore, "Growth of industrial ethernet," in *Industrial Ethernet Book, The Journal of Industrial Network Connectivity*, October 2014, p. 4.

[3] Kreutz, Ramos, Verissimo, Rothenberg, Azodolmolky, and Uhlig, "Software defined networking: A comprehensive survey," June 2014, survey available online at http://arxiv.org/abs/1406.0440 last accessed on October 15th 2016.

[4] ONF, "Openflow switch specification - version 1.3.0," June 2012, available online at https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf last accessed on October 15th 2016.

[5] A. Khandakar, J. O. Blech, M. A. Gregory, and H. Schmidt, "Software defined networking for communication and control of cyber-physical systems," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2015, pp. 803–808.

[6] M. Herlich, J. L. Du, F. Schrghofer, and P. Dorfinger, "Proof-of-concept for a software-defined real-time ethernet," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2016, pp. 1–4.

[7] D. Henneke, L. Wisniewski, and J. Jasperneite, "Analysis of realizing a future industrial network by means of software-defined networking (sdn)," in *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, May 2016, pp. 1–4.

[8] D. Thiele and R. Ernst, "Formal analysis based evaluation of software defined networking for time-sensitive ethernet," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 31–36.

[9] M. Corici, B. Reichel, B. Bochow, and T. Magedanz, "An sdn-based solution for increasing flexibility and reliability of dedicated network environments," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2016, pp. 1–6.

[10] J. Pfrommer, M. Schleipen, S. Azaiez, M. Boc, and X. Klinge, "Deploying software functionality to manufacturing resources safely at runtime," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sept 2016, pp. 1–7.

[11] E. Schweissguth, P. Danielis, C. Niemann, and D. Timmermann, "Application-aware industrial ethernet based on an sdn-supported tdma approach," in *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, May 2016.

[12] J. R. Jiang, H. W. Huang, J. H. Liao, and S. Y. Chen, "Extending dijkstra's shortest path algorithm for software defined networking," in *The 16th Asia-Pacific Network Operations and Management Symposium*, Sept 2014.

[13] A. Schimmel and A. Zoitl, "Real-time communication for iec 61499 in switched ethernet networks," in *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on*, Oct 2010, pp. 406–411.