# Evaluation of a Distributed Control Plane for Managing Heterogeneous SDN-enabled and Legacy Networks

Nicholas Gray, Stanislav Lange, Thomas Zinner
*University of Würzburg*
Würzburg, Germany
{nicholas.gray,stanislav.lange,zinner}@informatik.uni-wuerzburg.de

Benedikt Pfaff, David Hock
*Infosim GmbH*
Würzburg Germany
{pfaff,hock}@infosim.net

***Abstract*—With the increasing number of devices, protocols and applications, today's networks are becoming more and more complex. Hence, Software-defined Networking (SDN) tries to address this issue by separating the data from the control plane and by providing centralized interfaces for network configuration. As legacy devices cannot be replaced instantly due to high costs, both network segments have to be operated in coexistence with defined joints at their edges. To ensure a smooth operation, both controlling instances of these segments are required to exchange information. In this work, we design and implement a data model for storing the information needed to keep the controller and a Network Management System (NMS) synchronized, which are responsible for configuring the SDN-enabled network and the legacy devices respectively. For this, we implement and evaluate a total of three different synchronization strategies by the example of an SDN-based Bring Your Own Device (BYOD) use case.**

***Index Terms*—SDN, NMS, Management, Integration, Heterogeneous Networks, Distributed Control Plane.**

## I. Introduction

Today's networks face the challenge of a fast growing number of users and devices, resulting in changing traffic patterns and in fluctuating resource utilization. This establishes the need for not only a flexible traffic management but also for high bandwidths leading to more and more complex network infrastructures. As a consequence, operators are facing a high Capital Expenditure (CapEx) and Operational Expenditure (OpEx) as well as a rigid, inflexible network architecture. To tackle these drawbacks, the paradigm of Software-defined Networking (SDN) separates the data from the control plane, of which the latter is logically centralized within the SDN controller. The controller provides a holistic view on the network's topology as well as programmable interfaces, thus enabling an optimized resource utilization and cost savings for the network operator.

During the migration towards a fully softwarized network, existing legacy devices cannot be replaced instantly due to practical and financial limitations [11]. For enterprises a first adoption of this emerging technology leads to a coexistence of legacy and SDN enabled networks, which are connected at their edge. To establish a smooth conjunction between these two network segments a cooperation of their individual control instances, the SDN controller and a centralized Network Management System is required. To avoid system inconsistencies resulting in undefined system states, security flaws or massive packet loss, both control instances need to exchange information. In this work, this is done by synchronizing the networking state. Hence, it needs to be determined which information is required to be included into the state table and which synchronization method is most beneficial.

The contribution of this work is therefore (1) the conception of a data model designed for the shared network state synchronization, (2) the implementation of a prototype incorporating different synchronization strategies, and (3) the evaluation of distributed control planes for managing heterogeneous networks. To conduct the evaluation we base our investigations on a heterogeneous network scenario, consisting of an SDN and a legacy network segment. The scenario is based on an SDN-enabled Bring Your Own Device (BYOD) [7] use case, granting each connected and authenticated user its own personalized, virtual network. Requested services, residing in the existing enterprise legacy infrastructure, require the dynamic provisioning of an on demand path from SDN to the legacy segment. To quantify our results we focus on the window of inconsistency between the control planes and its impact on the data plane represented by the setup times and the overall packet loss of user to service connections.

The remainder of this work is structured as follows. In Chapter II we introduce the concept of *SDN* and detail the basis of *Network Management Systems (NMS)*. In addition we summarize an SDN-based *Bring Your Own Device (BYOD)* setup which we use as scenario in this work. This followed by a summary of related work in Chapter III. The derived data model and the implemented synchronization strategies as well as the test bed are detailed in Chapter IV. We proceed in Chapter V by evaluating the implemented strategies within the testbed, which is followed by a summary of our work in Chapter VI.

## II. Background

In the following, we provide background information to the paradigm of *Software-defined Networking (SDN)* and the fundamentals of *Network Managment Systems (NMS)*. In addition, we introduce an SDN-based *Bring Your Own Device (BYOD)* setup, used as a highly dynamic scenario for the evaluation of this work.

### A. Software-defined Networking

*Software-defined Networking (SDN)* is a paradigm, which aims for reducing the management overhead of network operations while simultaneously increasing the flexibility [9]. For this, SDN promotes the separation of the control and data planes, which are unified on legacy devices. While the data plane is responsible for simply forwarding the packets to their destination and remains in the switching device, the control plane is outsourced to a logically centralized SDN controller, which runs on standard server hardware. The SDN controller is responsible for determining where the packets are sent to and provides a holistic view of the network as well as the opportunity to configure the network via interfaces. Therefore, it enables the operator to automate configuration changes via central point within the network.

### B. Network Management Systems

With a growing number of network devices, the configuration, management and troubleshooting can be an exhaustive task for a network operator. For this reason network operators often rely on *Network Management Systems (NMS)* to maintain and supervise their networks. A *NMS* achieves this by implementing the *Fault, Configuration, Accounting, Performance and Security (FCAPS)* model [6] as standardized by *ISO*. The FCAPS model includes tasks for network device discovery and configuration, which are done by either the *Simple Network Management Protocol (SNMP)* [5] or through proprietary protocols. Therefore, the *NMS* is able to automatically configure the legacy network from a central point in the network and is thus a suitable counterpart to the SDN controller.

### C. Sardine Bring Your Own Device

To investigate the effectiveness of a distributed control plane, a highly dynamic scenario is required to provide sufficient state changes, which need to be synchronized. In this work the *SARDINE Bring Your Own Device (S-BYOD)* [7] setup has been chosen and is introduced in the following. *BYOD* enables end users to bring their own devices to a company or campus network and gains more and more importance [1]. Existing solutions are often based on *VLAN* and do not allow for a fine-grained flow control, thus softening security policies. The authors of the paper [7] overcome this drawback by employing *SDN* to deploy a fine-grained, on demand, personalized virtual network initiated per device and requested service. For this, the newly connected device remains isolated from the network and has initially only access to the captive portal. Here, the user is required to authenticate and to actively request a service. Once the service is requested, the SDN controller provisions a private path from the device to this service. Therefore, the overall attack surface is limited to the active services and the fine-grained flow control enables a more flexible security policy enforcement.

## III. Related Work

This section provides a brief overview of related research and details how the method followed in this work differs from the presented approaches.

### A. Sweet Little Lies

The paper *Sweet Little Lies: Fake Topologies for Flexible Routing* [10] focuses on a method for fine-grained traffic shaping within legacy networks using only link-state protocols. To achieve this, the authors introduce an approach called *Fibbing*, which manipulates a router's view of the network by sending fake packets to perform tasks like traffic steering, traffic engineering, load balancing and fast failover. The authors contend that a centralized control instance, like it is the case in SDN, could then be used to manage a heterogeneous network, consisting of SDN switches and legacy routers. However, it can be argued that computing an augmented topology is simply overhead in comparison to configuring a device by the use of *SNMP*. Also it should be taken into account that manipulating the *FIB* without being logged into the device can be seen as critical since no authentication on the device is needed.

### B. SDNMP

The paper *SDNMP: Enabling SDN management using traditional NMS* [11] covers a method for enabling an SDN controller to provide OpenFlow data via *SNMP*. This in turn allows the monitoring and management of SDN devices, using traditional network management software. To achieve this SDN management, the authors present a function called *Data Acquisition* that runs inside the SDN controller, as it holds all the needed information of the SDN network, i.e., the network topology and related resources. In addition, it stores the obtained data inside a *MIB* structure. The component called SDNMP collects data from the controller via *SNMP*, which is then converted into JSON format and outputted via a web page. Accessing topology and OpenFlow data by using SNMP is a promising feature. Yet, implementing this approach would mean to build a coherent module for each SDN controller. Therefore, in this work the NMS obtains its data over a SDN controller's *REST API*, as this module is already provided by most controllers.

### C. i-NMCS

The authors of the paper *Enhancing Network Management Frameworks with SDN-like Control* [4] present a framework called *i-NMCS* for managing legacy devices and softwarized networks. Providing traditional *NMS* functions like discovery and fault management for SDN-enabled networks, the framework collects network and device information e.g. network topology, from the *NMS*. Furthermore, the framework provides the core components with network policy requirements and translates the obtained information from the other *i-NMCS* modules into OpenFlow flow rules. The authors implemented two different scenarios. Whereas the first takes *QoS* provisioning algorithms and the global network state into account, to configure network devices, the second scenario performs flow provisioning on a user-identity basis. In both scenarios the authors do not specify precisely whether the configured devices consist merely of SDN devices or if a heterogeneous network is configured. Moreover, the authors do not discuss in detail how the state between the *NMS* and the core *i-NMCS* component is kept consistent.

### D. Incremental SDN Deployments

In the paper *Incremental Deployment of SDN in Hybrid Enterprise and ISP Networks* [8] the authors present an algorithm which selects a subset of legacy devices to be upgraded to SDN-enabled devices. The authors show that this incremental upgrade results in an increase of the network's resiliency and a decrease in the link utilization, due to advanced forwarding mechanisms, which are enabled by the global view of SDN controller. To maintain the global view in this hybrid SDN/legacy scenario, the presented architecture requires the placement of SDN-enabled switches directly within the legacy network. These intercept packets from classical routing protocols, which are then forwarded to the controller, from which it is able to deduce the topology of the legacy network. In contrast, this work does not require the placement of additional SDN-enabled switches within the legacy network for means of topology discovery, as this information is directly provided by the *NMS*. Thus, our focus resides on the consistency constraints imposed by the interoperation of the *NMS* and the SDN controller as well as the execution of configuration changes within both networks.

## IV. Design & Implementation

In this section we describe the implementation for synchronizing the SDN-enabled and legacy networks. For this, we describe the testbed on which we performed our evaluations and derive a common data model to hold the required information. We proceed by detailing the synchronization process and conclude by explaining the related synchronization strategies.

### A. Testbed

To evaluate the implemented synchronization strategies, we use the testbed as illustrated in Figure 1. Here, we extend the existing *BYOD* setup to be attached to a legacy network for the overall *heterogeneous network scenario*. On the datapath we deploy one OpenFlow switch and one legacy device. Both devices are implemented by a *VM* running inside OpenStack. Furthermore, the *User*, shown on the left-hand side in Figure 1 and the *Network services* on the right-hand side are implemented as *VMs*. In addition we use ONOS as SDN controller, to manage the OpenFlow switch and to provide fine-grained policy enforcement for each BYOD device and available service. Finally. we implement Infosim's StableNet as *NMS*, which monitors the entire setup, aggregates the required information regarding the legacy network and dynamically reconfigures the legacy device if needed.

OpenStack runs inside the Ubuntu OrangeBox which is a portable box containing ten Intel *NUC* micro computers connected to a cluster. Each *NUC* contains an Ivy Bridge i5-3427U dual core *CPU* with 16GB of *RAM* and a 128GB *SSD* [2]. The OpenStack version, running on the OrangeBox is Mirantis OpenStack version 9.0.1 Mitaka. The OpenFlow switch is realized by a *VM* with 1GB of *RAM* running Ubuntu 14.04 Trusty and the *OpenVSwitch (OVS)*. The legacy device consists of the same hardware and operating system configuration, but instead of running *OVS*, we implemented the packet forwarding with *TC* to resemble the functionality of a legacy router. *TC* is shipped with Linux and is used to configure network traffic within the Linux kernel [3]. Depending on the selected synchronization strategy, either the *NMS* or the SDN controller defines rules for ingress traffic to enable the basic *BYOD* functionality.

### B. Data Model

To enable the network state exchange between the SDN controller and the *NMS* we implemented the data model as displayed in Figure 2. This data model incorporates the information of a user, a requested service as well as any established connection. Hence, all relevant information for the *BYOD* scenario can be synchronized by the *NMS* instance.

To hold both control instances synchronous, each time a user requests or removes a service, the *BYOD* application, running on the controller, maps this information into the data model. To make this data model accessible to the *NMS* we extended the *REST* interface of the *BYOD* application, which provides the required information as *JSON* format.

In the following, we discuss the structure of the data model in more detail. The root of the model is a *Connection* object. It is specified by a particular *User*, a *Service* the user is requesting, and all *SDN Devices* in the path between. The *User* object, shown on the left-hand side of the figure, holds information about the users *ID*, *MAC address*, *IP addresses*, *VLAN ID*, and *Location*. The *Location* object in turn is composed of a *Device ID* and the corresponding *Port* to which an user is connected. The *Service* object, displayed in the center, is composed of *ID*, *TP port*, the services *Name*, and the services *IP addresses*. On the right-hand side of Figure 2 the *Devices* object is shown. This object holds the *IDs* and *Flows* for all devices lying on the path from user to service.

### C. Synchronization Process

The synchronization of the shared network state of the *NMS* is done via a *Business Process Script*. This script provides means to measure applications and services and can be extended to use an *External Measurement Script* which is designed to import structured data from the *BYOD* application.

The *Business Script* used for this work is written in *Java* and is imported as *JAR* file into the *NMS*. For each execution the script performs two steps, i.e., *Discovery* and *Measurement*. In the first step all devices and services are discovered and imported into the data model. Once this process is completed the script schedules a measurement for each discovered entity. This process is then repeated in defined intervals.

Using the *Business Process* and *External Measurement Script* the *NMS* is able to monitor and display the current state

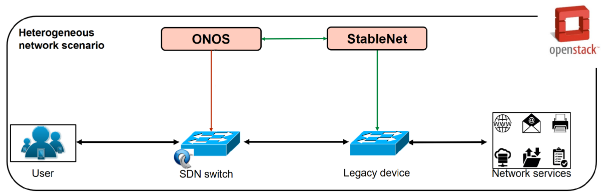

Fig. 1. Heterogeneous network scenario.

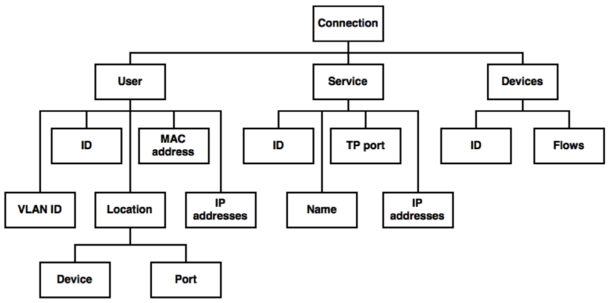

Fig. 2. Data model utilized to hold the network state.

of the *BYOD* network in a dynamical manner. The current state of each user's connection to a service can be tracked and whenever a state change is detected an internal alarm is triggered, which we used to initiate the configuration of the legacy devices if required.

### D. Synchronization Strategies

In the following, we discuss a total of three different data exchange strategies implemented in the prototype, i.e., *Polling*, *Pushing* and *Direct*.

The implemented polling method consists of the *NMS* periodically requesting two *REST* interfaces of the *BYOD* application running on the controller. This is done via the external measurement script, hence the *NMS* updates the current network state regularly within a certain time frame. Based on the updated networking state the *NMS* is able to determine if a configuration change of the legacy device is required and will trigger an alarm script to initiate the process if needed. The main advantage of this method is that it enables the network operator to regulate the induced load on the *NMS* by altering the interval in which the script is executed. This is especially useful in large networks, which produce many state changes. Yet, a higher interval will ultimately result in a higher window of inconsistency between the SDN controller and the *NMS*.

For the pushing method the external measurement script still actively polls the *BYOD* application for the current network state. Yet, instead of waiting for an alarm script being triggered and leading to a configuration change of the legacy device, the SDN controller directly triggers the *NMS*. Therefore, we extend the *BYOD* application to send *REST* calls, in case a user requests or cancels a service. Hence, in addition of deploying the corresponding flow rules to the SDN-enabled switches, the *BYOD* application also transmits the state change to a configuration job available by the *NMS*. Whereas the pushing method decreases the window of inconsistency between the SDN controller and *NMS*, it also significantly increases the load on the *NMS*, as state changes can no longer be aggregated to batches.

The last synchronization strategy implemented in this work is the direct method. Equally to the above described polling and pushing methods, the external measurement script polls the *BYOD* application to receive the current network state. However, for this method every legacy device is directly configured by the SDN controller. For this, we extend the *BYOD* application and enable it to directly access the legacy device. In case a user requests or cancels a service, the *BYOD* application installs the flow rules on the SDN device and executes the extended code, which connects to the legacy device and configures it accordingly. The advantage of this method is that internal processing times of the *NMS* are skipped and no additional load is introduced to this system. However, the main drawback is that provided configurations and mechanisms to access legacy devices implemented within the *NMS* have to be ported to the SDN controller, which results in additional development costs as well as inducing further load to the controller.

## V. Evaluation

For the evaluation of the different synchronization and configuration methods we focus on system critical parameters, namely *interconnection time*, *window of inconsistency* and *time to first packet out*. The interconnection time is the time measured from sending a service request until the chosen configuration method got executed and the service is ready for use. The window of inconsistency is the time span in which both control instances are asynchronous. This is equal to the time it takes to synchronize the SDN controller and the *NMS*. In addition we monitor the ingress as well as the egress port on each network device between the user and the service and log the time at which the first packet has been seen on the device. This gives us a more insights of where in our network the configuration process is stalling and where packets are being dropped.

We evaluate each synchronization method by five individual test runs of which each consisted of requesting and declining a service 30 times, thus resulting in a total of 60 synchronization attempts per test run. For each established connection we sent UDP packets for 20 seconds to measure the packet loss during the connection initiation. At last we configure the external measurement script to run every five seconds.

In the following the chosen three methods are compared by their measurement results. For the comparison we first focus on the interconnection time $t_i$ of the different methods. Figure 3 displays the empirical *Cumulative Distribution Function (CDF)* for the interconnection time $t_i$ for all methods. The

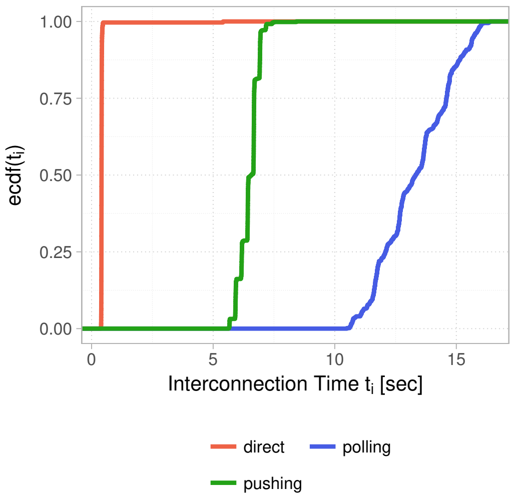

Fig. 3. Empirical *CDF* of the Interconnection Time $t_i$

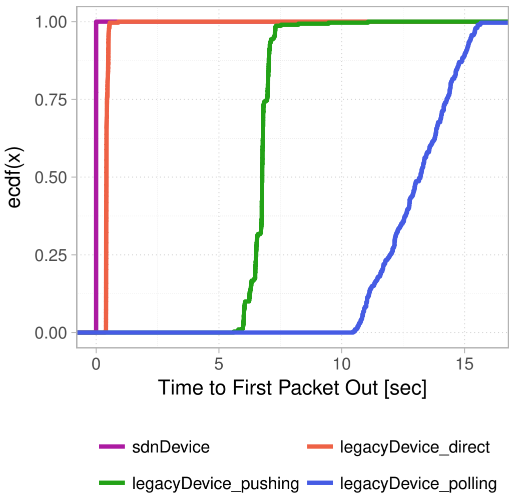

Fig. 4. Empirical *CDF* of the Time to First Packet Out

x-axis shows the interconnection time $t_i$ from 0 seconds to 17 seconds. On the y-axis the empirical *CDF* of the interconnection time $t_i$ can be seen. The red line on the left-hand side shows the measurement results for the direct method, the green line in the center of the figure shows the results for the pushing method, and the blue line on the right-hand side shows the results for the polling method. From the right to the left, the methods lines become steeper, indicating a decreasing dispersion of the measured values.

Comparing the three methods by their interconnection time $t_i$ median, significant differences can be seen. The polling time with its interconnection time $t_i$ median of approximately 13.4 seconds is twice the interconnection time $t_i$ median of the pushing method, which is at approximately 6.7 seconds. Further, the interconnection time $t_i$ of the direct method with approximately 0.41 seconds is by a factor of about 16 and 33 smaller than the values for the pushing and polling method respectively. The same applies for the mean values which correlate with the corresponding median values.

Examining Figure 4 similar results as discussed above can be observed. The y-axis shows the empirical *CDF* of the time to first packet out for the SDN and the legacy device, split by method. The x-axis displays the time to first packet out from 0 to 17 seconds. The purple line on the left-hand side displays the SDN device measurements. The SDN device measurement is displayed only once as the measurements are independent from the synchronization method. The red, green, and blue lines display the legacy devices time to first packet out measurements for the direct, pushing, and polling method.

Comparing the time to first packet out median values, it can again be seen that the direct method performs best followed by the pushing method. As before, the polling method achieves the weakest results. In 50% of all cases a packet is forwarded in less or equal than 0.42 seconds with the direct method, less or equal than 6.8 seconds with the pushing method, and less or equal than 13.1 seconds with the polling method. Based on the steepness, the dispersion of the measured values is the lowest with the direct method, followed by the pushing method and finally the polling method with the highest dispersion.

Table I summarizes the results from above and also incorporates the measured values for the window of inconsistency as well as the amount of dropped packets on the data plane.

Reviewing the results of the polling method, it can be said that it is inapplicable for the chosen heterogeneous *BYOD* scenario. Despite the moderate window of inconsistency $t_w$ of about 2.5 seconds, a user has to wait more than 13 seconds on average until a requested service is ready for use. Hence, this time is far too high and results in a noticeable service degradation in a real world deployment. With 10 seconds the biggest impact on this waiting time is caused by the StableNet implementation of the alarm script execution. This is due to the intentionally set interval in which alarms are correlated and presented in an aggregated manner to the network operator. Yet, due to this aggregation alarm scripts are not applicable for real time configuration routines.

The pushing method meets these requirements more closely. A user has to wait 6.5 seconds on average until a requested service is ready for use. In a *BYOD* use case where a user enables services over a captive portal and afterwards switches to a desired application, this time may be sufficient to not result in a decreased QoE. However compared to the window of inconsistency $t_w$ with 170 milliseconds on average, the majority of the interconnection time $t_i$ is caused by the StableNet *REST* call and the processing time of the legacy device configuration.

The direct method performs best in terms of interconnection time $t_i$. With an interconnection time $t_i$ mean of 410 milliseconds it's by orders of magnitude better than the other two methods. Yet, the window of inconsistency $t_w$ is equal to the window of inconsistency $t_w$ for the polling method. Since

TABLE I
COMPARISON OF DIFFERENT METHODS

| **Method** | **Interconnection Time $t_i$** | | | | **Window of Inconsistency $t_w$** | | | | **Time to First Packet Out** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Quartiles in [sec] | | | Mean | Quartiles in [sec] | | | Mean | Quartiles in [sec] | | | Dropped Packets |
| | 25% | 50% | 75% | in [sec] | 25% | 50% | 75% | in [sec] | 25% | 50% | 75% | Mean |
| *Polling* | 12.1 | 13.4 | 14.6 | 13.4 | 1.3 | 2.5 | 3.6 | 2.5 | 11.8 | 13.1 | 14.4 | 1325 |
| *Pushing* | 6.2 | 6.6 | 6.75 | 6.5 | 0.15 | 0.16 | 0.18 | 0.17 | 6.6 | 6.8 | 7.2 | 680 |
| *Direct* | 0.405 | 0.41 | 0.415 | 0.41 | 1.3 | 2.5 | 3.6 | 2.5 | 0.41 | 0.42 | 0.43 | 8 |

the legacy device is configured by the controller and not by the *NMS* the window of inconsistency only affects the monitoring of the SDN network. However this performance is bought by the direct configuration of the legacy device through the SDN controller. This in return results in additional implementation effort for the SDN controller whereas the *NMS* is already equipped with these capabilities by design.

The best results in terms of window of inconsistency $t_w$ are attained by the pushing method, since the controller directly informs the *NMS* in case of network state changes. Hence, the vital information, on which basis the *NMS* performs the legacy device configuration, are present within an instance.

## VI. CONCLUSION

During the migration from legacy to SDN-enabled networks, a coexistence of both technologies is inevitable. For this reason we discussed and evaluated possible realizations of a distributed control plane for heterogeneous SDN and legacy networks. The challenge of this heterogeneous network scenario was to keep the SDN and legacy control instances synchronous which in turn enables a seamless flow of traffic between both network domains.

As a basis for this heterogeneous network scenario we chose the SarDiNe *BYOD* setup which provided the required dynamics for our evaluation. We extended this setup by a legacy network domain, comprised of a legacy device and the commercial *NMS* StableNet, which acted as the control instance for the legacy network. Further, we designed a data model which is used to build a common data basis for a network state exchange between both control instances and discussed the synchronization process between the SDN controller and the *NMS*. For this we implemented three different methods, namely polling, pushing, and direct.

We evaluated these three methods by three metrics, interconnection time , window of inconsistency , and time to first packet out. The first two metrics provide insights into the control plane's behavior whereas the last metric resembles the behavior of the forwarding plane.

In future work we focus on the optimization of the NMS's implementation regarding the polling and pushing method. For the polling method a prioritized alarm execution, which is not queued when emerging, but directly executed, might reduce the interconnection time. The window of inconsistency for the polling method could be reduced by directly implementing the functionality into the *NMS*'s code and not relying on an external measurement script. This allows for more frequent measurements and for more sophisticated scheduling techniques. A reduction in the interconnection time for the pushing method could possibly be achieved by implementing a prioritized *REST* endpoint for the configuration job execution and a increasing the job's priority. For these and further investigations the implemented prototype provides a solid foundation.

## ACKNOWLEDGMENT

This work has been performed in the BMBF framework KMU-Innovativ in the project SarDiNe (Project ID 16BP12308). The authors alone are responsible for the content of the paper.

## REFERENCES

[1] Bring Your Own Device. last visited on 2017/02/04.
[2] Canonical's cloud-in-a-box: The Ubuntu Orange Box. Last visited on 2017/02/13.
[3] Linux Traffic Control man page. last visited 2017/02/14.
[4] W. J. Buchanan, M. Naylor, and A. V. Scott. Enhancing network management using mobile agents. In *Proceedings Seventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2000)*, pages 218–226, 2000.
[5] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple network management protocol (snmp). Technical report, 1990.
[6] A. Clemm. *Network management fundamentals*. Cisco Press, 2006.
[7] S. Gebert, T. Zinner, N. Gray, R. Durner, C. Lorenz, and S. Lange. Demonstrating a personalized secure-by-default bring your own device solution based on software defined networking. In *2016 28th International Teletraffic Congress (ITC 28)*, volume 01, pages 197–200, Sept 2016.
[8] D. K. Hong, Y. Ma, S. Banerjee, and Z. M. Mao. Incremental deployment of sdn in hybrid enterprise and isp networks. In *Proceedings of the Symposium on SDN Research*, page 1. ACM, 2016.
[9] M. Jarschel, T. Zinner, T. Hossfeld, P. Tran-Gia, and W. Kellerer. Interfaces, attributes, and use cases: A compass for sdn. *IEEE Communications Magazine*, 52(6):210–217, June 2014.
[10] S. Vissicchio, L. Vanbever, and J. Rexford. Sweet little lies: Fake topologies for flexible routing. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 3:1–3:7, New York, NY, USA, 2014. ACM.
[11] Y. Zhang, X. Gong, Y. Hu, W. Wang, and X. Que. Sdnmp: Enabling sdn management using traditional nms. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 357–362, June 2015.