Towards a Programmable Management Plane for SDN and Legacy Networks

Christian Sieber*, Andreas Blenk*, Arsany Basta*, David Hock†, Wolfgang Kellerer*
*Chair of Communication Networks, Department of Electrical and Computer Engineering,
Technical University of Munich, Germany
{c.sieber,andreas.blenk,arsany.basta,wolfgang.kellerer}@tum.de
†Infosim GmbH & Co. KG, Würzburg, Germany
hock@infosim.net

***Abstract*—Software-defined Networking (SDN) and one of its most known realization, namely OpenFlow, enabled wide-spread and vendor-neutral programmability of the control plane of modern network equipment. However, despite research and standardization efforts, the management plane is still eluding device-neutral and vendor-neutral programmability. Thus, innovation in the management plane is hampered by a dependency on human experts, domain knowledge that is hidden in human-centered manuals, and the huge amount of diverse device capabilities and configuration interfaces. Recent proposals for vendor-neutral data-models, such as OFCONF, are still lacking majority and do not provide a standardized way of device capability discovery and device status monitoring. Accordingly, we present an architecture that provides a unified interface to the management plane of heterogeneous devices, i.e., SDN and legacy devices. We discuss the properties of the chosen level of configuration abstraction and show how applications northbound of the abstraction layer are well prepared against undesired side-effects of management actions. By example of a popular approach for enabling OpenFlow in mixed-SDN/legacy networks, i.e., Panopticon, we provide a proof-of-concept implementation of the proposed architecture in a real test-bed. We show how a management application can use the abstraction layer to discover and configure QoS options in the network, monitor the devices, and prevent undesired traffic interruptions in the legacy domain, while being operated in parallel with an SDN controller.**

I. Introduction

For the control of communication networks, Software-defined Networking (SDN) opens up the devices and introduces a split between the device's control plane and the packet forwarding paths, i.e., the data plane. Through an open interface, e.g., the popular OpenFlow protocol, an external controller sets forwarding rules on the devices based on its global view of the network. In recent years, this vendor-neutral and device-neutral interface facilitated rapid innovation in the area of the network's control plane, such as virtualization of SDN networks [1]. In terms of network management, networks traditionally consist of independent and closed devices, managed by a logically centralized entity, i.e., the Network Management System (NMS). Through an NMS, a human network operator is able, for instance, to configure the control algorithms running on the devices, which in turn configure the packet forwarding rules.

978-1-4673-9486-4/16/$31.00 © 2016 IEEE

Despite proposals from research and standardization bodies, the management plane still lacks a comparable interface and protocol like OpenFlow for capability discovery, device management, and monitoring. Therefore, network management is still heavily human-centered with minor autonomous behavior. This leads to a high vulnerability to network failures of traditional networks. Surveys account the human factor for a large percentage of network outages, as a symptom of an overwhelming complexity of the system. [2] accounts configuration errors by humans for 50 % to 80 % of network outages. Although networks are planned and configured to provide a high redundancy, configuration errors can still lead to dramatic network outages [3]. Accordingly, we propose an architecture that is a first step towards a unified SDN-based management and control of communication networks. Such a software-defined management paves the way for eliminating the error-prone task of manual network configuration and enables innovation in the management plane.

This work is a first step towards a unified SDN-based management and control of communication networks. It introduces an architecture that allows SDN-based network management and also contains mechanisms allowing autonomous network management. Furthermore, it enables novel network virtualization techniques on top, which encompass the SDN and legacy domain. Figure 1 depicts the overall framework this work is embedded in. At the bottom, the physical networking infrastructure contains OpenFlow-enabled SDN switches and legacy switches, which is referred to as a hybrid or mixed-SDN/legacy network. On top, a future networking application conducts combined network control and management decisions enabled by our proposed architecture. The application interfaces with a Network Services Abstraction Layer (NSAL), as proposed in RFC 7426 [4]. Based on our previous work in [5], where we demonstrated the feasibility of such an NSAL, we focus in this work on the connecting link between the NSAL and the physical infrastructure, which we call the Management Abstraction Layer (AL). This layer unifies the control and management plane of the network and provides it as one interface to the application.

In detail, the proposed architecture provides a northbound interface via an open REST API to the NSAL. Further, it uses distributed agents with device-specific modules to con-

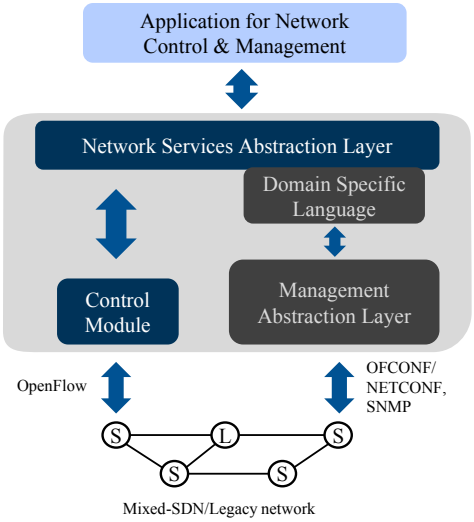

Fig. 1. Overall framework for unifying control and management for autonomous network operation. The management abstraction layer and the domain specific language are discussed in this work. A unifying Network Services Abstraction Layer and applications northbound are future work.

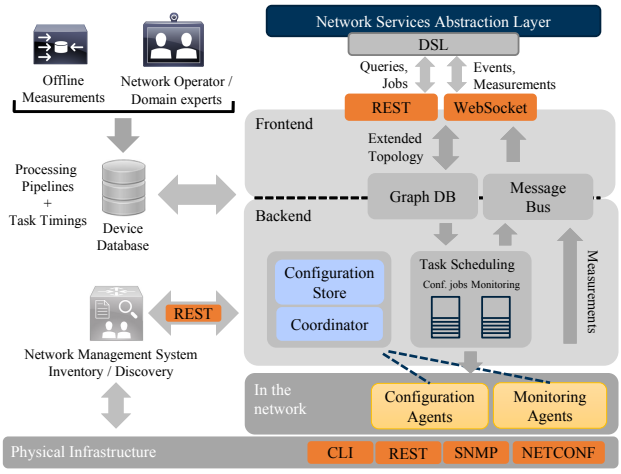

Fig. 2. The proposed architecture. The northbound-facing front-end consists of a synchronous REST and asynchronous WebSocket interface. The back-end consists of a messaging bus, distributed agents in the network and a task queue. A custom device database stores the devices timings and processing pipelines. A proprietary NMS provides discovery and inventory services.

figure and monitor the heterogeneous physical infrastructure. Furthermore, an implementation of a Python-based Domain Specific Language (DSL) on top of the REST interface is used to demonstrate how the abstraction layer can be used by a future NSAL implementation.

The implementation of the management abstraction layer (AL) consists of two key elements: an extended network topology and a database of management task timing characteristics. In the extended topology, the northbound interface includes the devices' processing pipelines to allow northbound applications to traverse different processing paths, such as different queuing strategies. The second important element of the architecture is the estimation of task timings. While hardware switches full-fill stringent performance requirements for packet forwarding, less is known about the timing characteristics of management operations. The importance of understanding the timing characters of management and control operations to avoid undefined forwarding states in the network is shown in [6]. Furthermore, while working on the architecture and experimenting with autonomous Quality of Service (QoS) configuration, we noticed non-inferable side-effects such as traffic interruptions in the range of seconds. For example, one device of the test-bed discards packets for 10 seconds if the scheduler type is changed from scheduler A to scheduler B, but not from scheduler B to scheduler A. Therefore, we allow northbound applications to *estimate* the timing, e.g., time to take effect or duration of traffic interruptions of management operations, before execution to prevent interruptions or flapping behavior. A device database stores the timings of device configurations.

The proposed architecture is evaluated based on a popular hybrid networking use case, called Panopticon [7]. Panopticon enables OpenFlow-control in hybrid networks by configuring VLAN tunnels on the legacy devices to connect two OpenFlow-enabled endpoints, which are separated by legacy devices. Additionally, a further evaluation shows the feasibility of a combination of the use case with QoS discovery and configuration, which provides differentiated service for a subset of the VLAN tunnels. The source code of the framework is released as open-source, together with a virtual environment showcasing the architecture [1].

This work is structured as follows. Section II gives an overview of the overall management architecture, including a discussion on the level of abstraction chosen and the extended network graph with devices' processing pipelines. Section III defines the term management task in the context of this work and the estimation of the task timings and side-effects such as traffic interruptions. Section IV evaluates the proposed architecture based on the mixed-SDN/legacy networking approach Panopticon. Section V discusses related work. Section VI draws conclusion.

II. Architecture

Figure 2 presents an overview of the proposed management plane. The Abstraction Layer (AL) and the Domain Specific Language (DSL) on top of the AL are the key elements of the architecture. The AL provides a unified interface to configuration and monitoring of the underlying network. Furthermore, it provides a stateless HTTP-based northbound interface and a WebSocket-based interface for asynchronous events, e.g., measurements and task completion events. The DSL is a Python dialect tailored to the AL. It simplifies the programming of the management module and coordinates the request/response model of the REST interface with the asynchronous events of an event bus.

In order to support the AL, multiple components are required southbound of the AL. In the network, distributed

[1] http://git.io/vB73A

TABLE I
AL DESIGN TRADE-OFFS

Model fidelity	low	medium	high
AL intelligence required	high	medium	low
Northbound intelligence required	low	medium	high
Device modeling effort	low	medium	high
Hardware feature utilization	low	medium	high
Practicality	low	high	low

configuration and monitoring agents execute configuration tasks, e.g., setting VLAN tagging through a Command Line Interface (CLI), and gather device statistics such as device ports. Each agent is responsible for one or more devices and uses device-specific modules to interface with the heterogeneous devices. Configuration tasks and monitoring configuration are distributed among the agents through a task queue and operational configuration is stored in a key-value store. A logically centralized event bus acts as a broker for measurements and task completion events.

The device database provides a model of the capabilities and the processing pipelines of a specific device type. A custom graph database stores the high-level network topology and the processing pipeline of the devices and makes it available through the AL. For the discovery of the network infrastructure, including the inventory with the vendor and the model of each networking device, we utilize a commercial Network Management System (NMS).

We implemented our proposed architecture relying on several open source projects, most notable Apache ZooKeeper [8] as configuration store, Apache Kafka[9] and crossbar.io [10] as backend and frontend messaging bus, and RabbitMQ [11] as messaging queue. For topology discovery and device inventory, we use the proprietary NMS StableNet [12].

A. AL Design & Trade-offs

In the following, we discuss the design trade-offs of the management AL and its provided data model. The key elements are the *model fidelity* and the *practicality* as shown in Table I. The data model fidelity describes how many details of the heterogeneous configuration interfaces and hardware features of the different devices should be exposed to the northbound management application. Practicality describes the qualitative result of the ”usefulness” of a combination of the three metrics *AL-intelligence*, *northbound-intelligence* and *hardware feature utilization*.

First, we comment on the case of a low model fidelity. A low model fidelity requires the AL to make more decisions on its own, as many parameters are hidden from the northbound application. This is comparable to an intent- or policy-based interface where the AL receives abstract requirements and translates them to device specific configurations. The metric *AL intelligence required* behaves inverse to the model fidelity. A low model fidelity requires the AL to make decisions on its own about parameters not visible to the northbound application. A high model fidelity requires less intelligence, as most decisions about parameters are the responsibility of the northbound application.

From the point of view of the modeling effort per device type, a low fidelity AL design requires the least modeling effort, as one has to create a basic common model for all switching devices only, e.g., to distinguish OpenFlow from legacy devices. However, a low model fidelity is not able to fully leverage all features of the hardware. For example, the OFCONF management data model for OpenFlow does only support data-rate based scheduling, even if the hardware could support mechanisms like Priority Queuing (PQ) or input queuing. Furthermore, specific hardware features outside of a simple model cannot be considered from the global perspective of the northbound management application.

A high model fidelity in turn only requires minimal AL intelligence. In the best case, one northbound request translates exactly to one required configuration change, e.g., changing a weight of a queue. On the other hand, this increases the complexity of the decision logic in the management module, as it has to be able to handle many options and also device-specific exceptions and limitations, e.g., one feature blocks another switch feature. Then again, a high model fidelity in combination with a complex management module algorithm allows to globally utilize a large portion of the features of specific device types, e.g., all the different scheduling strategies, with the cost of a high modeling effort per device type.

The design of our AL for the management plane aims for a medium practicality. Whenever possible, we design the AL in a way that one northbound change requests translates to one southbound configuration job. Furthermore, we do not expose all hardware features of the heterogeneous devices to the management module to facilitate rapid development of novel management plane algorithms. Device-specific features and parameters not covered by the AL are silently set to sensible default values by the configuration agents, still based on hand-crafted rules by domain experts.

B. Monitoring

We see network monitoring in the management abstraction layer as an enabler for northbound applications to verify the configuration. Hence, traffic engineering techniques, e.g., load-balancing, are not in the focus, but possible with the provided primitives. Figure 3 depicts the two modes of monitoring supported by the platform, namely task-based on-demand monitoring and continuous monitoring. Both modes are defined based on the unified AL specification, but executed in the context of the device-specific modules in the monitoring agents. The choice of how to implement the monitoring, e.g., SNMP traps or CLI polling, is up to the device-specific module and based on human domain experts implementation.

The **on-demand monitoring** creates monitoring tasks consisting of the device and component to monitor, e.g., interface X of device A, the metric to monitor, e.g., interface status or received bytes, a condition which terminates the task, e.g., average of received bytes samples is smaller than

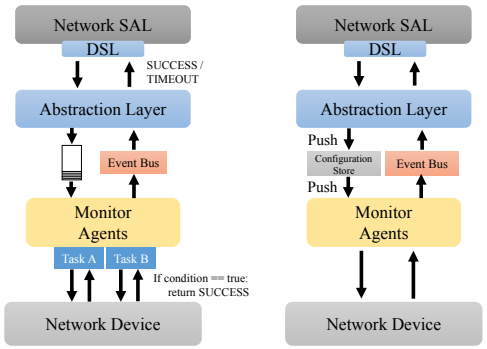

Fig. 3. The two modes of monitoring supported by the proposed architecture. On-demand monitoring executes arbitrary monitoring tasks close to the device with high frequency and defined stop criteria. Long-term monitoring data configuration is pushed in the configuration store and from there to the agents.

1000 Bytes, and a maximum execution time of the whole task. The condition is expressed as arbitrary (Python) code that is executed in an isolated execution environment with access to numerical statistic programming packages and the collected samples. On-demand monitoring tasks allow for high frequency monitoring without burdening the northbound interfaces with unnecessarily frequent samples and are designed for distributed synchronous operation (*do - wait for condition - continue*).

The **continuous monitoring** mode is designed for long-term data acquisition and asynchronous events. Continuous monitoring is not expressed as a monitoring task, but as a permanent configuration in the configuration store. The application on top of the AL can set an interval and optional threshold for all metrics defined in the AL. The configuration is saved to the configuration store, which triggers the monitoring agent to update its local configuration. If the metric exceeds the threshold, the agents send the measurement sample to the message bus where the application on top of the AL, i.e., the management application, can listen for the stream of samples.

C. Extended Graph & Device Models

In the following, we describe the structure of the (extended) topology graph, which in-cooperates the processing pipelines of the devices into the high-level topology. We use three devices from our testbed as reference, an NEC PF5240F, a Cisco Catalyst 4503-E and an HP-V1910 switch. The devices are given anonymized as *X*, *Y* and *Z* in the remainder of the paper. The association between the letters and the switches is random and not given here. We define the graph as a list of components with attributes and unidirectional relationships/links between the components. The components can be put into two categories; components that describe the processing pipeline, e.g., a queue or scheduler, and components that describe switch features, e.g., an OpenFlow interface component. First, we introduce the components associated with the processing pipeline. Afterwards, we discuss further relevant components and relationships. To describe the processing pipeline, we use an *input* and *output* relationship. Depending on the type

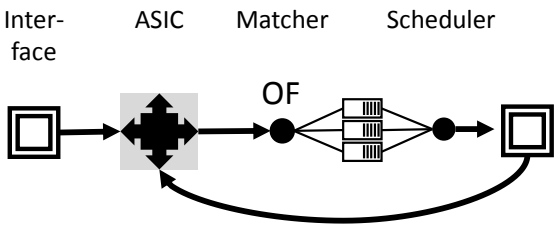

Fig. 4. Example of a simple processing pipeline of an OpenFlow switch with two connected ports, an ASIC, a matcher, three queues, and a WFQ scheduler.

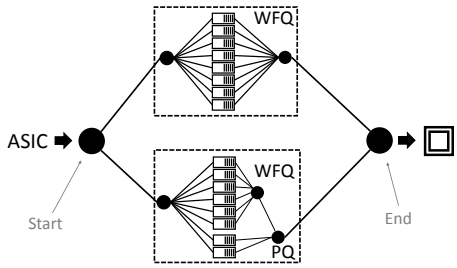

Fig. 5. Example of an alternative output graph path from the switch's application specific integrated circuit (ASIC) to an interface of a *X* device. The top path offers eight queues scheduled using WFQ. The bottom path two queues with PQ and six queues with WFQ.

of component, a component can have zero or more inputs and zero or more outputs, e.g., a scheduling component has multiple inputs and one output. Figure 4 gives an example model of a simple OpenFlow switch with two connected ports. The interface shown to the left is directly connected to the application specific integrated circuit (ASIC), the switching engine. The input path from the interface to the right to the switching engine does not contain any components. Therefore it has no input shaping mechanisms. The output path to the interface to the right offers three queues, which are connected to a Weighted Fair Queuing (WFQ) scheduling component and a matcher.

Some of the switches in our testbed allow to adapt the processing pipeline to specific use cases. We model this by introducing an alternative path start and end component in the graph. This enables applications on top of AL to traverse the different alternatives like they would traverse links and nodes in a standard high-level topology graph. Furthermore, this simplifies modeling the processing pipeline, as it does not inflate the definition of generic scheduler and generic queue components. Figure 5 presents a simplified model of the processing pipeline of the *X* switch in our testbed using alternative path components. The processing path on the top provides 8 queues scheduled by a WFQ scheduler, while the path on the bottom schedules only 6 queues with a WFQ scheduler and two 2 queues, plus the output of the WFQ, with a Priority Queuing (PQ) scheduler.

Table II summarizes the relationships between the switch components used in the extended network graph. There are the *input* and *output* relationships, which are used to specify the

TABLE II
RELATIONSHIPS BETWEEN SWITCH COMPONENTS

Type	Description
managedBy	Refers to the component/device which accepts the configuration changes for the component.
containedIn	Specifies which logical or physical component houses the component.
input & output	Specifies the packet processing workflow for a component.

processing pipelines. *managedBy* allows to delegate the configuration interface of a component to a different component. For example, a hardware interface of a switch does not have its own configuration interface, but is configured through the CLI or NETCONF configuration interface. *containedIn* specifies the physical or logical dependency of a component and allows to deduce failure or maintenance impacts from the graph. For example, an interface restart results in unavailability of the whole processing pipeline inside the interface.

III. TASK COMPOSITION & TIMING ESTIMATION

In this section, we discuss how an application on top of the AL can perform management operations and query an estimation of the timing characteristics of the tasks to be executed. Management operations are expressed as atomic configuration *tasks*. One task, e.g., assigning a VLAN tag to an interface, either succeeds or fails and can be characterized by its timing, e.g., the time it takes for the desired configuration to put into effect, and side-effects, e.g., a necessary interface restart. In the following, we define a configuration task in detail. Afterward, we introduce the methodology for estimation of task timings and side-effects based on an offline measurement set-up in our test-bed. We give a generic solution to encompass a variety of use cases from enterprise networks to networks with strict timing characteristics. In the use cases presented in this paper we reduce the complexity as some aspects are not needed to describe the selected use cases.

A composite task T is defined as a set of atomic tasks t ($T := t_1, ..., t_n$). Each atomic task is defined by four types of delay. t^0 denotes the time the atomic task is executed, t^t the transport delay caused by the AL, i.e., the task forwarding to the agent, the processing time by the agent and the physical propagation delay between agent and the target device, t^p the time of acknowledgment (or processing time on device), t^b the blocking time, e.g., interface restart as cause of atomic task execution, and t^d the time difference between task reception at the device and the time the effect is measurable on the data plane. We assume t^d and t^b to be mutually exclusive (disjoint). t^d, t^b, and t^p are relative to $t_0 + t^t$ and there is no strict ordering between the three types of delay, e.g., a device can acknowledge a change after or before the data plane effect is measurable.

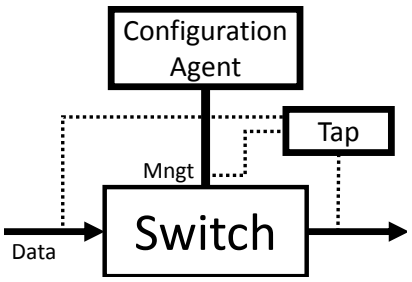

Fig. 6. Offline measurement set-up consisting of a high-precision tap device, a configuration agent and the switch to be measured. Timing characteristics are measured by tapping the management and the data plane interfaces.

$$estimate(T) := \begin{array}{c} \\ t_1 \\ t_.. \\ t_n \end{array} \begin{array}{c} \begin{array}{cccc} t^t & t^p & t^b & t^d \end{array} \\ \begin{pmatrix} t^t_1 & t^p_1 & t^b_1 & t^d_1 \\ .. & .. & .. & .. \\ t_n & t^p_n & t^b_n & t^d_n \end{pmatrix} \end{array} \quad (1)$$

$$exec(T) := \begin{array}{c} \\ t_1 \\ t_.. \\ t_n \end{array} \begin{array}{c} \begin{array}{c} t^t + t^p \end{array} \\ \begin{pmatrix} t^t_1 + t^p_1 \\ t^t_.. + t^p_.. \\ t^t_n + t^p_n \end{pmatrix} \end{array} \quad (2)$$

A composite task can either be *estimated* or *executed* through the AL. An estimation tries to estimate t^t, t^d, t^b, and t^p based on the offline measurements. In our prototype implementation, the estimation returns the average values of the offline measured delays and in case of the transport delay, the average measured transport delay (round-trip) of previously executed tasks on a specific device. Note that the device database does not only contain measurements, but additional knowledge of domain experts being still required and implemented to capture corner cases, e.g., changing from scheduler A to B restarts the interface, changing from B to A not. Equation 1 and 2 summarize the output of the estimation and execution function, respectively. We focus on t^b in the remainder of this work, as interruptions are most relevant to our investigated use cases.

Figure 6 describes the offline measurement set-up. The management port of the switch is directly connected to a configuration agent. A UDP stream of packets with a packet size of 1400 Bytes is sent to a specific port and forwarded to a configured output port. We use a high-precision hardware tap device with three ports to capture the management traffic and the data streams before entering and after leaving the device. For the delay of the change of a VLAN tunnel, we only measure the output data at the new destination port. After the capture process, the measurements are analyzed offline and the results are stored in the device database.

Figure 7 gives an example for a measurement of a task where we select an alternative path in the processing pipeline, i.e., changing the QoS scheduler of the *X* device in our testbed. At $t_0 + t^t$ with t^t being zero, the switch receives the configuration command through the management interface. As we define the blocking delay t^b and the processing on device

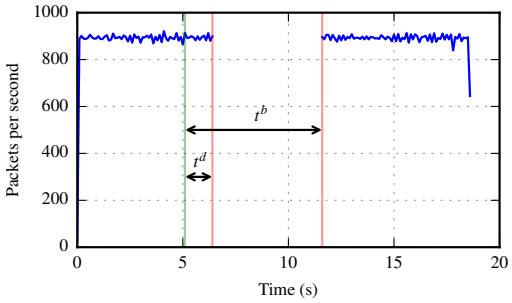

Fig. 7. A packet stream traversing an interface of a *X* device while the processing pipeline, i.e., the QoS scheduler of the interface is changed. t^b is on average about 6 seconds.

TABLE III
OFFLINE MEASUREMENT RESULTS FOR VLAN TAGGING

Switch	t^p	t^b	t^d
X	648.9 ms	125.8 ms	316.2 ms
Y	4 ms	5.6 ms	8.1 ms
Z	24.4 ms	0.3 ms	15.2 ms

t^p independent of each other, t^b is the time from $t_0 + t^t$ until the interface is up again and transmits data. For this task and device type, t^b is about 6 seconds on average. t^t is the time between the reception of the configuration command and the switch's confirmation.

Table III gives a summary of the results of the offline measurement of the devices for the VLAN tagging. Each measurement was repeated between 50 and 100 times. The standard deviation is omitted in the table. For the *X* device, the standard deviation of t^b is about 20 ms, for the *Y* and *Z* device 1.4 ms.

IV. USE CASES & PROTOTYPE EVALUATION

In the following, we introduce and evaluate two use cases implemented on our proposed architecture. Both use cases are from the domain of SDN-hybrid networking. In the first use case we evaluate the VLAN-based Panopticon [7] approach from the perspective of the management plane and show how the domain-specific knowledge provided by the AL can reduce or prevent short-term interruptions of virtual networks. In the second use case, we combine the first use case with Quality of Service discovery and configuration. We use the AL to discover alternative processing pipelines with low latency scheduling to prioritize traffic of specific VLAN tunnels. Furthermore, in this use case, we deploy task prediction and the on-demand monitoring to prevent mid-term, i.e., about 6 seconds, service interruptions of virtual networks.

A. Use Case: VLAN Tunneling

Figure 8 depicts the test-bed topology for the VLAN tunnel use case. Two SDN domains are connected by a legacy network consisting of five switches. The five switches allow three paths between two SDN domains where the two legacy edge nodes are shared by all three paths. For sake of simplicity, the relevant hardware interfaces are numbered from 1 to 14.

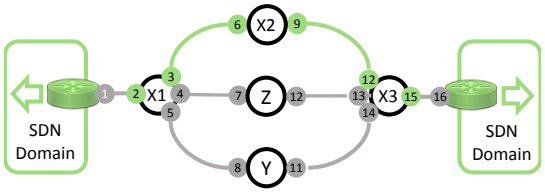

Fig. 8. Two SDN domains in our test-bed connected by VLAN tunnel. Green interfaces indicate the configured VLAN tunnel on the top path. *X*, *Y* and *Z* denote the type of the switch (see Table III).

TABLE IV
TASK EXECUTION ORDER EFFECT ON INTERRUPTIONS

ID	Execution order	$\sum^{\forall t \in T} t^b$
1	$\{t_2, t_7, t_{13}\}, \{t_4, t_{12}, t_{15}\}$	251.6 ms
2	$\{t_4, t_7, t_{12}, t_{13}\}, \{t_2, t_{15}\}$	125.8 ms
3	$\{t_2\}, \{t_4\}, \{t_7\}, \{t_{12}\}, \{t_{13}\}, \{t_{15}\}$	1984 ms

t_x denotes the atomic task to change the VLAN tagging of interface x.

Now, we assume device *X2* on the top path is scheduled for maintenance, e.g., a firmware upgrade. Without any further information about the devices and without advanced monitoring, a naive approach could be to execute the tasks $t_x, \forall x \in \{2, 4, 7, 12, 13, 15\}$ in parallel to create a new VLAN tunnel between the SDN domains to steer the traffic through device *Z*. Note that $t_x, \forall x \in \{2, 4\}$, $t_x, \forall x \in \{7, 12\}$ and $t_x, \forall x \in \{13, 15\}$ cannot be executed in parallel as they require changes to the same device. A more advanced approach could first execute $t_x, \forall x \in \{4, 7, 12, 13\}$ to configure a new tunnel and afterward, when the device acknowledged the command, change the steering by executing $t_x, \forall x \in \{2, 15\}$.

Table IV lists the overall blocking for different task execution orders. We assume that each set is executed in the order given in the table and that the tasks in a set are executed in parallel. The orders with ID 1 and 2 are the worst and best case task execution orders for parallel task execution, respectively. Order 3 represents the worst case for sequential task execution, when the management application waits for confirmation before continuing to the next task. From the table we conclude that a planned task execution decreases the traffic interruptions on average by factor 16. Although the absolute interruption time of up to 2 seconds seems negligible for low frequencies of configuration changes, we argue that a future autonomous decision entity on top of the NSAL makes use of the AL with a higher frequency than a static VLAN tunnel set-up. For example, this can be for load-balancing reasons and, therefore, can result in frequent interruptions.

B. Use Case: QoS Discovery & On-demand Monitoring

In the second use case, we show how a management application uses the AL to first discover low-latency options and second, configure low-latency tunneling through a legacy network. Figure 9 depicts the test-bed for the second use case. The network consists of an SDN and legacy domain. Two (physical) paths through the legacy domain are connecting the

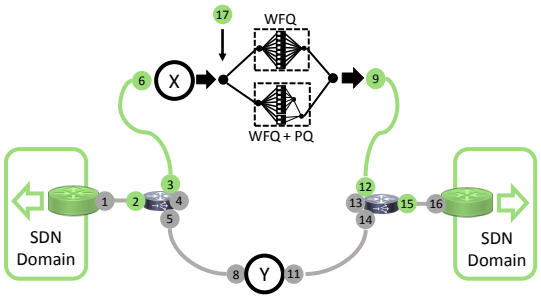

Fig. 9. Set-up for the second use case consisting of two SDN domains, two named switches, *X* and *Y*. Switch *X* supports an alternative QoS processing pipeline on the output interface as shown in Figure 5

two SDN domains. One path traverses the *X* switch on top and the second path traverses a simple legacy switch with VLAN configuration options on bottom. The *X* allows to switch to alternative packet processing pipelines as hinted in the upper part of the figure and illustrated in detail in Figure 5.

In order to provide low-latency virtual networks, the management application first has to discover the *extended* network graph through the AL. The discovery takes a path of nodes and edges from the high-level topology and returns an extended path consisting of the switch components on the path, including the alternative processing selectors. In the extended graph, it can calculate all shortest paths where priority queuing is available and based on the calculated paths, determine the alternative path selectors to configure. If the path selector is configured without further analysis, i.e., without calling *estimate* of the task to be executed, the virtual networks are interrupted for about 6 seconds as illustrated in Figure 7.

In the following, we show how the domain knowledge of the device database provided by the AL allows the management application to predict the interruption, monitor it on-demand, and reroute the traffic accordingly. We denote $t_x, e \in \{2, 3, ..15\}$ as the atomic task to configure the interfaces $\{2, 3, .., 15\}$ and the alternative path selector $\{17\}$. A management application aware of the prediction provided by the AL, first gathers a set of tasks required for changing to an alternative processing pipeline, denoted as T, with $T := \{t_{17}\}$ in this case. Second, it calculates the combined blocking delay by $t^b_{max} := max(\{t^b_x\}, \forall x \in T\})$ on the path. If the t^b_{max} is larger than a defined threshold t^b_{thress} of acceptable interruption, it decides to re-route the traffic before the change of the processing pipeline. For example, the interruptions observed in Figure 7 can be expressed by $estimate(t_{15})$, which returns an average t_b of 6 seconds.

Next, we introduce the implementation of this use case in the Python-based DSL. We assume the reader to be familiar with the general Python3 syntax. Listing 1 gives the implementation in source code and the subsequent enumeration summarizes the steps with corresponding line numbers in square brackets. Please note that the listing is over-fitted to the scenario due to space constrains, i.e., additional loops and checks are required to make it work under different scenarios and environments.

Step 1 [4] Retrieve high-level topology
Step 2 [5] Calculate shortest paths
Step 3 [8] Request extended graph for shortest paths
Step 4 [15] Select path with priority scheduler
Step 5 [19] Predict t^b for alternative processing path
Step 6 [20] Re-route traffic to second shortest path
Step 7 [22] Execute conf. job for alternative processing
Step 8 [23] Monitor interface, wait until available again
Step 9 [25] Re-route traffic to first shortest path again

First, we retrieve the high-level topology as a graph of connected interfaces. This requires one call to the AL. Second, we calculate the path options between the two domains and afterward, in Step 3, we query the AL to retrieve the extended graph including the interface processing pipelines for each relevant path, i.e., the upper and lower one in this use case. Subsequently, in Step 4, we traverse the extended graph to search for priority schedulers. Next, we check if the scheduler is on a selected path segment by calling *is_selected* on the priority queuing switch component. As the path segment is not selected, we create an anonymous task object and retrieve t^b by calling *estimate*. Note that this does not execute the task. Afterward, as $t^b > 5s$, we use VLAN tunneling to create a linear VLAN broadcasting domain (excluding the SDN edge interfaces) to steer the traffic to the bottom path. This implicitly creates a list of configuration jobs, which are forwarded to the configuration agents. In Step 7, we execute the configuration job to select the priority scheduling. This creates one configuration job, which is executed on the *X* switch and results in an interface restart, thus, a blocking time. In Step 8, we create a monitoring job to check every 250 ms for the interface status and return when the interfaces is available again. In the final step, we change the VLAN configuration to steer the traffic again to the default upper path.

Figure 10 depicts the stream of packets affected by the described configuration changes. The traffic is interrupted two times, but only for a brief duration. This demonstrates how the interplay of discovery, management, and monitoring enables an autonomous management application on top of our proposed architecture which can correctly predict and reduce traffic interruptions.

V. Related Work

Network management in general has received a lot of attention in the past and in the present and is comprised of different topics. We see our work in the context of configuration management, e.g., setting configuration options, monitoring, e.g., on-demand monitoring tasks, and device capability discovery, e.g., QoS options. In the following, we introduce publications most relevant to the topics of our work.

The Forwarding and Control Element Separation (ForCES) protocol [13], which is positioned as alternative to OpenFlow, shares similar concepts with our work. In ForCES, the hardware of the forwarding elements, e.g., switches, are modeled

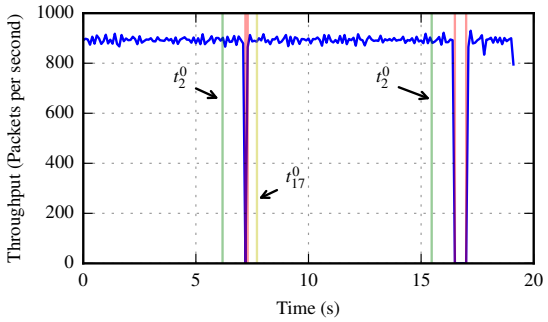

Fig. 10. Use case 2 implemented in our testbed. The short traffic interruptions are caused by rerouting of the traffic by changing the VLAN configuration. t_2^0 and t_{17}^0 are the times when the management application created the configuration tasks and before the task is executed by the agents.

```
1  import nmdsl as nm
2  import networkx as nx
3
4  topo = nm.topology()
5  sp = nx.all_shortest_paths(topo,source=1,target=16)
6
7  for s in sp:
8    et = nm.ext_topo(s)
9
10   pq = [e for e in et if e.type() == "PriorityQueuing"]
11
12   if not len(pq): continue
13   break
14
15 pq = pq[0] # only one pq scheduler here
16
17 if not pq.is_selected():
18
19   if pq.select().estimate()['t_b'] > 5:
20     nm.execute([e.set_vlan(12) for e in sp[1][1:-1]])
21
22   pq.select().execute()
23   pq.contained_in().wait_for('intf_status=="up"',
24                              freq = 4)
25   nm.execute([e.set_vlan(10) for e in sp[0][1:-1]])
```

Listing 1. Working (but not optimally) implementation of use case 2

as processing pipelines of so called Logical Function Blocks which can be discovered and configured. In our work, we aim for less complexity and model the switches configuration interfaces as easy to use elements and abstract many low-level details of ForCES. At [14], an informal working group of large network operators headed by Google is working on transferring software-defined principles to the management plane. This could greatly simplify the implementation of the device-specific modules in our configuration agents. OFCONF [15], a data model for NETCONF, is the counterpart of OpenFlow for the management plane. However, OFCONF is still new and the provided data models are of limited scope. Furthermore, it is tailored to the specific needs of OpenFlow and does not consider switch features outside of the scope of OpenFlow, e.g., input shaping. In [16], the ONF is working on a Core Information Model (CIM) which specifies physical, logical and virtual switch components, relationships and protocols in great detail. Our abstraction aims for a representation closer to the physical switch and does not consider higher layer relationship between protocols. The defined relationship in CIM can be implemented on top of our AL. In [17], OASIS is working on a Topology and Orchestration Specification for Cloud Applications (TOSCA). However, the abstraction focuses on higher level network services such as database management systems (DBMS) and their relationship to other entities, not on details of the individual forwarding elements.

In [18], the authors describe how domain knowledge is required for network configuration, but hidden in domain experts and human-centered switch manuals, thus, way inaccessible for network automation. The authors introduce COOLAID, an interface similar to a database API to create a logically centralized abstraction of the network configuration. In [19], the authors argue that the management plane is too complex due to devices exposing all their internal details and parameters. This leads to error-prone configuration, fragmentation of management tools, and hard to understand configuration parameters. They introduce CONMan, an abstraction layer which exposes device configuration with inter-connected protocol configuration modules and dependencies. In [20], the authors introduce PACMAN, a platform for automated operation and configuration management. The work defines active documents, which describe an abstract configuration task. One active document represent higher-level abstractions, spanning multiple actions and one or more devices. The work represents a vertical subset of each of the NSAL and AL in our work, but designed without northbound interface and focused on composed atomic tasks. In [21], the authors introduce SWItch, a framework for the management of data center networks. SWItch uses namespaces trees, similar to our switch component graph, to model the devices. COOLAID, CONMan, SWItch and PACMAN are designed to be operated by humans and to be responsible for the management of the legacy control algorithms. This differs from our work as we see and design the management plane as a building block underneath a network services abstraction layer tailored for automation. Furthermore, monitoring in the abstraction layer and the management task timings in terms of delay and blocking are not part of their work.

In [22], the authors depict Statesman, a network-state management service deployed in the Microsoft Azure cloud. Statesman offers a graph abstraction northbound and is designed to resolve conflicts between different management applications accessing the graph. Compared to our work, the abstractions are not as detailed, e.g., no QoS support, and the focus is on network states instead of atomic tasks. Domain knowledge regarding the cost of change, i.e., the blocking time due to configuration change, are not available to northbound applications.

In [23], the authors conclude that monitoring the frequency of atomic management tasks, e.g., configuration of a VLAN on an interface, can be used to classify seldom touched configuration options as important and dangerous. This could be an extension to the estimation in the abstraction layer introduced in our work. A way of automatically learning the capabilities of a certain device is introduced in [24]. The authors show how an ontology-based information extraction system can deduce the capabilities of a device by analyzing

the CLI of the device. This could allow rapid prototyping of the device models discussed in our work. In [25], the authors introduce Hybnet, a network manager for a hybrid SDN/legacy networks. The work is related to the second use case in our work and to the Panopticon approach. We see this as complementary to our work and as part of the NSAL on top of our AL. The same applies to [26], [27], where the authors show OpenFlow agents and control for legacy devices.

VI. Conclusion

In order to facilitate innovation in the management plane of modern network equipment, a device-neutral and vendor-neutral unified abstraction for discovery, monitoring and configuration of device resources is required. Such a unified abstraction is a first step towards combining network control with network management in a programmable manner. Thus it unifies the feature set of network control with the feature set of management actions. For example by making port-based packet schedulers available through a programmable abstraction layer, we can increase the number of possible Quality of Service options compared to the OpenFlow and OFCONF data model. Additionally, the abstraction has to provide ways to capture domain knowledge from human experts and human-centered manuals, such as device specific limitations and non-inferable side-effects of management actions.

This work proposes an architecture that represents an extended network topology that provides a hardware detail level that includes even the processing pipelines of devices. Via a northbound interface, management applications can run networking tasks through a vendor- and device-neutral abstraction layer. By example of Panopticon, an approach for OpenFlow networking in mixed-SDN/legacy networks, we show how a management application, which is actually triggered by an SDN controller, can run configuration tasks. These configuration tasks, which were not accessible before, include, e.g., VLAN tagging on non-SDN devices, discovering of QoS options in the network, or changing the packet scheduling. Furthermore, the architecture provides mechanisms to estimate traffic interruptions due to the triggered management actions, e.g., due to the change of the scheduler configurations. By example, we show how a smart task scheduling based on the known timings of the management actions mitigates service interruptions. Besides, the proposed architecture enables novel management applications capable of autonomous network management, e.g., plug and play for network devices. In the future we plan to extend the abstraction layer to a larger variety of use cases. In particular, we target use cases for holistic network virtualization, i.e., combined control and management virtualization, and time-critical use cases such as networks intended for real-time communication.

VII. Acknowledgment

This work has been partially funded by the German Federal Ministry of Economic Affairs and Energy (BMWi) under the grant numbers KF3157502LF3 and KF3157602LF3 as part of the ZIM program. This project has received funding, in part, from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 647158 FlexNets). This work reflects only the authors' view and the funding agency is not responsible for any use that may be made of the information it contains.

References

[1] A. Blenk, A. Basta *et al.*, "Survey on Network Virtualization Hypervisors for Software Defined Networking," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 655–685, 2016.
[2] Juniper Networks, "What's behind network downtime?" 2008.
[3] P. Gill, N. Jain *et al.*, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. of ACM SIGCOMM*. New York, NY, USA: ACM, 2011, pp. 350–361.
[4] E. Haleplidis, K. Pentikousis *et al.*, "Software-Defined Networking (SDN): Layers and Architecture Terminology," RFC 7426 (Informational), Internet Engineering Task Force, Jan. 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7426.txt
[5] C. Sieber, A. Blenk *et al.*, "Network configuration with quality of service abstractions for sdn and legacy networks," in *IFIP/IEEE International Symposium on Integrated Network Management (IM) 2015*, May 2015.
[6] X. Jin, H. H. Liu *et al.*, "Dynamic scheduling of network updates," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014.
[7] M. Canini, A. Feldmann *et al.*, "Software-defined networks: Incremental deployment with Panopticon," *Computer*, vol. 47, no. 11, Nov 2014.
[8] "Apache ZooKeeper," https://zookeeper.apache.org/.
[9] "Apache Kafka," http://kafka.apache.org/.
[10] "crossbar.io," http://crossbar.io.
[11] "RabbitMQ," https://www.rabbitmq.com/.
[12] Infosim GmbH & Co. KG, "StableNet," https://www.infosim.net, 2015.
[13] A. Doria, J. H. Salim *et al.*, "Forwarding and Control Element Separation (ForCES) Protocol Specification," RFC 5810 (Proposed Standard), Internet Engineering Task Force, Mar. 2010, updated by RFCs 7121, 7391. [Online]. Available: http://www.ietf.org/rfc/rfc5810.txt
[14] OpenConfig, "OpenConfig," http://www.openconfig.net/, 2015.
[15] Open Networking Foundation, "OF-Config," https://www.opennetworking.org/, 2015.
[16] ——, "Core Information Model (CoreModel)," https://www.opennetworking.org/, 2015.
[17] OASIS, "TOSCA," https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca, 2015.
[18] X. Chen, Y. Mao *et al.*, "Declarative configuration management for complex and dynamic networks," in *Proc. of ACM CoNEXT*, 2010, p. 6.
[19] H. Ballani and P. Francis, "CONMan: a step towards network manageability," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, 2007, pp. 205–216.
[20] X. Chen, Z. M. Mao *et al.*, "PACMAN: A platform for automated and controlled network operations and configuration management," in *Proc. of ACM CoNEXT*, New York, NY, USA, 2009, pp. 277–288.
[21] C.-C. Chen, P. Sun *et al.*, "SWIM: A switch manager for datacenter networks," *Internet Computing, IEEE*, July 2014.
[22] P. Sun, R. Mahajan *et al.*, "A network-state management service," in *Proc. of ACM SIGCOMM*, 2014, pp. 563–574.
[23] H. Kim, T. Benson *et al.*, "The evolution of network configuration: a tale of two campuses," in *Proc. of ACM SIGCOMM IMC*, 2011, pp. 499–514.
[24] A. Martinez, M. Yannuzzi *et al.*, "An ontology-based information extraction system for bridging the configuration gap in hybrid SDN environments," in *Proc. of IFIP/IEEE Symposium on Integrated Network Management (IM)*, May 2015, pp. 441–449.
[25] H. Lu, N. Arora *et al.*, "Hybnet: Network manager for a hybrid network infrastructure," in *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, 2013, p. 6.
[26] C. Jin, C. Lumezanu *et al.*, "Telekinesis: Controlling legacy switch routing with openflow in hybrid networks," in *Proc. of 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, New York, NY, USA, 2015, pp. 20:1–20:7.
[27] A. Zaalouk and K. Pentikousis, "Network configuration in OpenFlow networks," in *Mobile Networks and Management*. Springer, 2015, pp. 91–104.