©2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The definitive version of this paper has been published in 2018 28th International Telecommunication Networks and Applications Conference (ITNAC), 21-23 Nov. 2018, https://doi.org/10.1109/ATNAC.2018.8615342.

Towards an Active Probing Extension for the ONOS SDN Controller

Christopher Metter*, Valentin Burger*, Zheng Hu†, Ke Pei† and Florian Wamser*

*University of Würzburg, Institute of Computer Science, Würzburg, Germany
{christopher.metter|valentin.burger|florian.wamser}@informatik.uni-wuerzburg.de

†Huawei, Corporate Reliability Departement, Shenzhen, China
{hu.zheng|peike}@huawei.com

***Abstract*—Network monitoring is a complex task and is always a trade-off between granularity of information and the performance impact of the monitoring itself on the network. SDN controllers, such as the ONOS SDN controller make this challenge easier as they can supply centralized information over the whole network. In our previous work, we analyzed the built-in detection mechanism of ONOS and revealed a lack of detection performance and the vulnerability of this process to jitter. These results have also been verified by measurements. In this paper, we present an active probing extension for the ONOS SDN controller that overcomes these shortcomings by emitting probing packets that are transmitted through the network. Packet loss is detected by calculating statistics of a list which contains data from previous packets. The evaluation by the means of measurements proves the benefits in terms of detection performance and controller load of this application. Furthermore, our extension is able to detect jitter in the data plane and to automatically adapt the probing process to these conditions.**

***Index Terms*—SDN controller; Network monitoring; Detection; ONOS; Active Probing; Extension**

I. Introduction

With the introduction of Software-Defined Networking (SDN) in the late 2000s, network engineers have been given a new and powerful tool to operate and administrate their networks. SDN offers through its logically centralized and programmable controller flexibility and better availabilty with the decoupled control plane by enforcing rules independent of the data plane by the controller with a global view over the network. One beneficiary of this new paradigm is the complex area of failure detection and failure recovery in modern networks.

Currently, the common approach to detect failures in networks relies on the line rate protocol, which is already several years old. This technology monitors the port status of a direct link to another device. As soon as a port status change is detected, the controller is notified through special control messages. After the reception and evaluation of this message, the controller then calculates an appropriate reaction to this challenge, e.g. rerouting of flows that used the faulty link. Furthermore, common SDN controllers, e.g. ONOS [1] or OpenDaylight [2], use the so called Link Layer Discovery Protocol (LLDP) [3], to detect changes in the detected topology. By frequently sending LLDP packets on each link, the controller knows which device is connected to which other device. Each reception of such an LLDP packet resets a timer. If the timer expires, e.g. due to a sudden link failure, the controller assumes that its link is now faulty. This mechanism, whilst being good for the detection of a topology, is not suitable for the detection of link failures other than complete link outages. As our research in previous work has shown, this mechanism is especially vulnerable against packet loss in the data plane [4]. It is possible that the controller takes more than a minute to successfully detect that a link is faulty and recovery mechanisms are activated.

Therefore, our goal is to develop an extension that is able to reliably detect packet loss in the data plane for one of the common SDN controllers, namely ONOS. This extension uses provided knowledge of the controller and extends it by adding an active probing mechanism to its features. Furthermore, as has been brought up in [4], the whole probing process, as implemented by ONOS, is error-prone against jitter in the data plane. The effect is that, even when the link itself is still useable, the controller treats a link as a faulty one. This undesirable effect is considered by an extension to our probing application and implemented in it. Additionally, the extension takes current network conditions, e.g. packet loss, into account and optimizes its probing parameters. Our evaluation shows that our active-probing extension enhances the detection performance of the ONOS controller for the cases of packet-loss and jitter in the data plane.

This paper is structured as follows. Section II presents related work and the background on failure detection within SDN networks. Afterwards, in Section III, the active probing extension is presented. Section IV introduces our testbed, the measurement scenario, and the evaluation of the detection performance of the controller. Finally, Section V summarizes the content of this paper and gives an outlook to further work on this topic.

II. Background and Related Work

This section describes the basics of SDN failure detection necessary to understand the presented probing extension and its evaluation. We show how currently link errors are detected

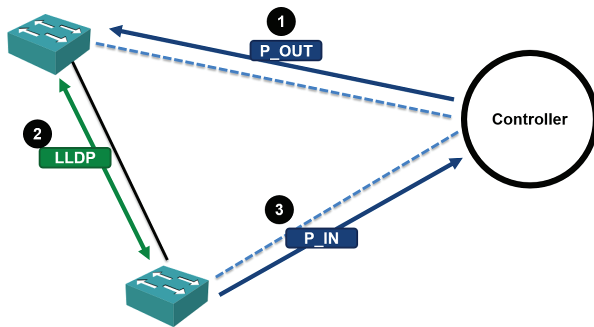

Fig. 1. LLDP Mechanism

and then discuss the disadvantages and challenges that currently prevail. We conclude the section with an overview of related work.

A. SDN Failure Detection

SDN is a networking paradigm in which the data and control plane is decoupled from each other so that decisions can be made about where traffic is sent, independently from the forwarding itself [5]. In an SDN network the SDN controller constitutes the control plane of the network and is responsible for (1) maintaining a global view of network, (2) determining forwarding rules via one or more applications running on the controller, and (3) updating and requesting statistics and link status for the global view of the network. Switches constitute the data plane where the traffic is forwarded in a switch, at best at line rate.

The current prevailing approach for fault detection in an SDN-based network is the following. Detection is handled centrally by the controller by sending LLDP packets through the data plane. The general case is shown in Fig. 1. Here, the controller creates and sends an LLDP packet within an Openflow *packet out (P_OUT)* message to each connected switch (1), cf. Section III. The switch probes the links by forwarding the packet at all ports (2) or sending back the received packets to the controller (3). Thus, with this procedure, the controller receives every probing interval an LLDP packet from each link as a response to its initiation and detects where working links are in the network. If the SDN controller does not receive a return packet within a predefined timeout period, it can conclude that there is a link failure and a port within the network has failed.

Our previous work investigated the detection performance at an ONOS SDN controller under hazardous network conditions such as partial packet loss with LLDP [4]. A mathematical analysis of the built-in detection mechanism of ONOS reveals a lack of detection performance. For example, a packet loss value of 50 % is only detected after more than one minute has passed. These results were also verified by measurements. Furthermore, we showed that the ONOS SDN controller is also vulnerable against link impairing effects such as jitter in the data plane. With the default detection mechanism already small values of jitter can lead to a false positive detection of a link failure. In the worst cases this can lead to a constant flapping of links status, decreasing the overall availability and performance of a network. In this paper we now want to address these weak points by designing and implementing an active probing extension that overcomes them.

To sum it up, the current approach as described by current controllers such as ONOS has disadvantages. First, due to its nature as an active probing mechanism, it has the difficulty to choose the probing interval in a meaningful way. If the time interval is too long, the failure is detected too late. If the interval is too short, significant network overhead is generated. This problem gets even more severe with jitter in the network. Currently, the setting for the probing interval is set to 3 s as default value in ONOS controller. Second, the approach is that one can not monitor a link with partial packet loss, since the SDN controller would consider each partial recurring error as sequential failure, resulting in an oscillation of the detection of link failures.

B. Related Work: Existing Failure Detection Approaches

This section presents related work on the topic of SDN failure detection. Google's well-known B4 network presents one of the biggest SDN architectures with failure detection [6]. It uses OpenFlow to control low-cost switches, which leverage control at network edges and allow for multipath forwarding. They keep track of a link failure and enable a dynamic bandwidth reallocation in case of link or switch failure. Trinocular by Quan [7] is an outage detection system that uses active probing to understand reliability of edge networks. This approach learns the status of the Internet with probes driven by Bayesian inference. In [8], the authors propose a detection mechanism based on switches' internal periodic link probing for stateful SDN data plane abstractions like P4.

Another group of detection mechanisms is based on Bidirectional Forwarding Decision (BFD) protocol where the works propose to use the BFD daemon present in Open vSwitch to detect failures and ensure fast failure recovery [9, 10]. In [11], a controller-based monitoring scheme and optimization model is proposed in order to reduce the number of monitoring iterations that the controller must perform to check all links.

Furthermore, several commercial link detection mechanisms are built in switches for enterprise networks, including Cisco UniDirectional Link Detection (UDLD) Protocol [12], Juniper Monitor Ethernet Delay-Measurement [13]. Alternative solution might be passive probing via passive switch measurements [14].

III. Towards an Active Probing Application for the ONOS SDN controller

As observed in our previous work, ONOS, in its current version, is unable to meet service provider requirements in certain scenarios. As the approach of ONOS has several shown limitations, we create a new detection application for the ONOS controller that increases the detection performance for the case of packet loss in the data plane to enable carrier-grade networking.

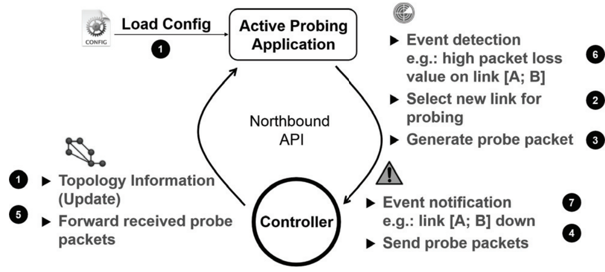

Fig. 2. Schematic Concept of the Application

First, before designing the application, required features have to be defined. The new probing application should be able to detect packet loss and link delay in the data plane by using an active probing mechanism. As we rely on the ONOS controller, the app should be designed to interact with it using the Northbound API. This API offers a well defined interface between additional applications on top, such as our detection mechanism, and the connected network devices on the bottom. The controller offers information on the status of the devices, for example flow traffic statistics, and transforms actions from an application into device-specific commands. As the controller is the ”brain” of SDN, and most network operation comes to a halt without it, the application should not generate a utilization overhead on the controller resources such as CPU load or memory utilization.

This section presents a description of the general concept of the application, and gives insights into the most important components, such as the structure of the probing packets and how the statistics of each link are calculated. Finally, its functionality and performance is evaluated with measurements in a testbed.

A. Concept of the Application

Fig. 2 shows the schematic concept of the application. On top, the active probing application is depicted, at the bottom the controller. After the activation of the probing application, the initial configuration is loaded and the current topology known to the controller is requested ❶. Based on this topology, the application now selects a new link between two data plane devices for the probing process ❷ and automatically generates a new probing packet ❸. This packet is then handed over to the controller via the Northbound API. The controller then injects this packet through its Southbound API into the data plane ❹. At the sink of the link, the probing packet is automatically redirected to the controller, which, in turn, passes this packet to the active probing application ❺. The application now processes this packet and updates the link properties, e.g. the packet loss and the link delay ❻. If one of these properties surpasses a threshold, for example the packet loss is greater than 10%, a notification is triggered and sent to the controller via the Northbound API ❼. The controller then is able to calculate an appropriate reaction, e.g. recalculating the routes of flows that traverse a link with bad quality.

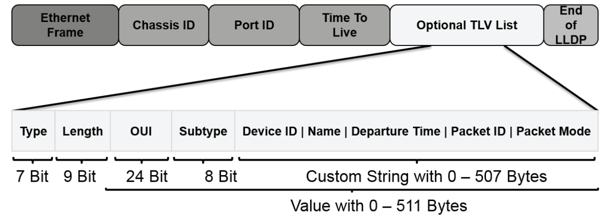

Fig. 3. LS3LLDP Packet Structure

B. Probing Packet Structure

For each link, an INFO3LLDP probing packet is sent each predefined probing interval. The packet structure is expanding the IEEE 802.AB LLDP standard [3]. The exact structure consists of an Ethernet frame, a number of mandatory LLDP TLVs (Type-Length-Value), an optional TLV list, here depicted as *LS3 Content* and finally an ”end of LLDP” TLV, see Fig. 3. Setting the destination MAC address of the Ethernet frame to a special multicast address, and the Ethertype of this packet to 0x88cc, ensures that network devices receiving this packet are able to handle it. Each of the TLVs, regardless of mandatory or optional, has three fields. Field 1 is the type of the TLV with 7 bits (e.g. 1 for chassis id or 2 for port id). Field 2 is 9 bits wide and describes the length of the following value. Field 3 is the actual value with a length ranging from 0 (minimum) to 511 (maximum) bytes. This structure is used for each TLV, even for the optional TLVs and the ”end of LLDP” TLV. The mandatory TLVs include the chassis id, the port id, and the time-to-live. For the advanced purposes of our probing application, we use optional TLVs with a unique OUI (organizationally unique identifier), in order to identify as packets generated from our application, and multiple subtypes followed by a value. These subtypes are *device id*, *name*, *departure time*, *packet id*, and *packet mode*. The device id identifies the source switch of this packet. The name field is used to identify this packet as an INFO3LLDP packet. The departure time field is set to the system time of the host of this packet at the time of creation and is used to calculate the transmission time of a packet upon receipt. The packet id is incremented for each packet and is used to determine packet loss on a link, as explained later on. The effect of the packet mode will also be described below. In order to transmit multiple values (e.g. the whole set) via one single LLDP packet, for each information a new TLV is created and attached to the packet.

C. Link Statistics Calculation

As soon as a probing packet is received by the probing application, the information of the packet is decoded, and the reception time is recorded. Out of this information, a new link object is created. The measurement algorithm now checks, to which link object in the previously mentioned queue it fits. This entry then is removed from the queue. Afterwards, the packet id is put into the list of the last received packet ids and the link latency now is calculated via the difference of reception and transmission time. Based on the list of received

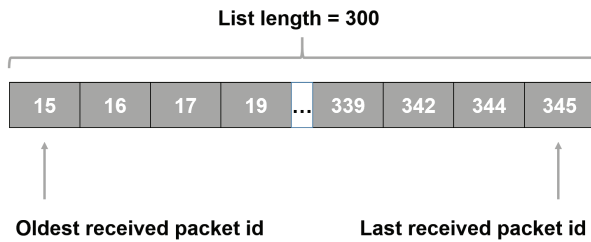

Fig. 4. Exemplary Packet History

packet ids, the packet loss rate of a link can be calculated. The list always contains 300 entries, where the first entry is the id of the oldest received packet, the last entry the id of the last received probing packet. As soon as a new packet is received, it is put at the end of the list and the first entry is removed. Now, based on the difference of the last packet id and the first id in the list, it can be calculated how many packets have actually been received.

An example is illustrated in Fig. 4. Here, the last packet id is 345, the oldest is 15. The difference of both ids is 330. As the length of the list is 300, the packet received rate is $300/330 = 0.91$. Therefore, the packet loss rate is $1 - (\text{packet received rate}) = 1 - 0.91 = 0.09$.

If the application does not receive a probing packet for a predefined timeout θ, the link is put under investigation and a link failure event is triggered: probe packets with the aforementioned packet mode set to ”special” are generated. If these packets are received, the above progress comes into play. If not, this link is considered down and its updated status is forwarded to the controller which then is able to take appropriate actions, e.g. adopt the routing of the installed flows.

D. Evaluation of the Benefits of the Active Probing Extension

After designing and implementing this application its functionality and performance has to be evaluated. Therefore, measurements comparing the detection performance of ONOS with its own detection application and with our new active probing application have been conducted. For this purpose the testbed of Fig. 5 is used. One physical server is running Mininet to emulate a network topology of four switches. The topology of the testbed is a ring-topology with four switches. To each of the switches one simulated host is connected. Two physical servers form the ONOS controller cluster. The switches are load-balanced between these two nodes, i.e. one controller node controls two switches.

In order to test the detection capabilities of ONOS for the packet delay, both with and without our extension, packet loss is configured on the data plane link between switch 1 and 2. To be able to determine the reactions and their delay, the signaling traffic between the controllers and their connected switches is recorded and evaluated after each run. Analyzing these traces allows us to calculate the mean reaction time and the detection probability of each scenario. For scenarios evaluating the active probing application, the vanilla detection application

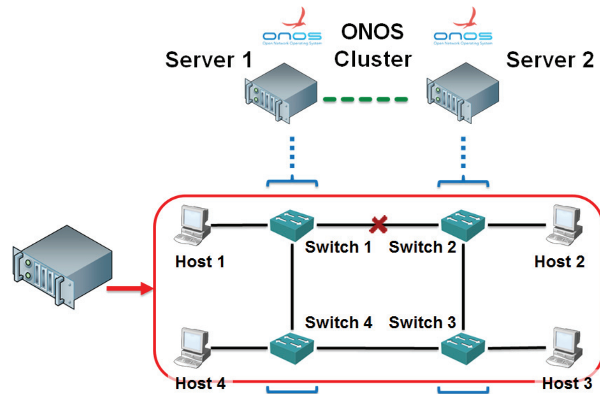

Fig. 5. Testbed Overview

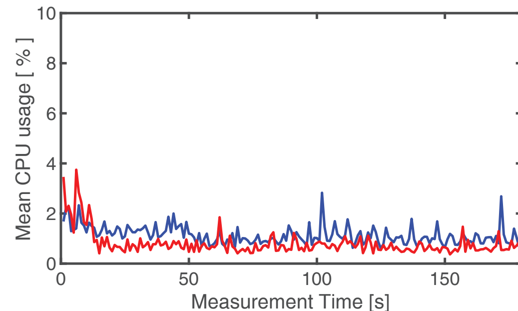

Fig. 6. Mean CPU usage with and without the Active Probing Application enabled

has been disabled and, accordingly, the new probing app has been installed and activated.

At first we will investigate the resource utilization of the application in comparison to a vanilla ONOS configuration. Afterwards, the enhanced detection performance is evaluated using different scenarios and configurations. The presented data represents the mean values of 10 repetitive measurement runs. If not mentioned separately, a packet loss value of 10% has been configured in the data plane.

1) Resource Utilization: Fig. 6 and 7 show the resource utilization on side of one of the controller hosts with and without using the active probing application. Of course, the measurements involved the collection of the statistics from both servers. However, our investigations have shown that their behavior only differs marginally, and, therefore, it is sufficient to display only the results of one of the hosts. For this scenario the application has been configured with a probing interval of 30 ms and a timeout of 100 ms. The red line displays the usage without the application, the blue line the usage with the application installed and activated. Fig. 6 shows the mean CPU utilization during the whole time of the measurement. The x-axis depicts the measurement time in seconds, the y-axis the mean CPU usage in percent. Both measurements show only a small CPU usage varying between 1 and 3% throughout the measurement. Additionally, the measurement with the extension enabled is only insignificantly higher than the one without.

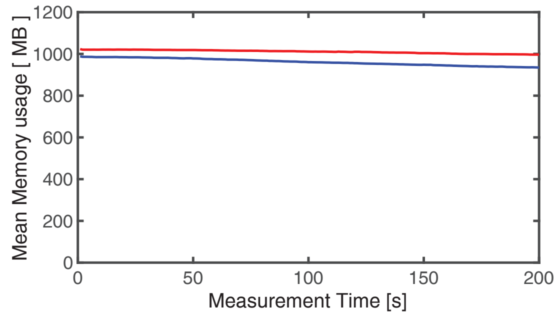

Fig. 7. Mean RAM usage with and without the Active Probing Application

Fig. 7 shows the mean memory usage with and without the active probing application. The x-axis shows the measurement time in seconds, the y-axis the memory usage in megabytes from 0 to 1200 MB. Again, only a small difference is noticeable: Whereas the measurement without the probing application starts at 1000 MB and slowly decreases to a final value of 950 MB, the measurement with the app activated requires 1050 MB and slowly decreases to 1000 MB. Nevertheless, as the host of this controller instance has had a total of 16 GB of memory available, both memory usages do not carry any weight.

Looking at the resource utilization, the goal of not overloading the controller host with our application has been accomplished. For the case of the CPU load, the utilization more or less remains the same. For the case of the RAM load, the usage even is smaller than before. The reasons for this behavior are simple: Whereas the vanilla ONOS probing mechanism creates instances for each connected switch, our application only runs one instance per controller host. Therefore, less RAM is required.

2) Detection Performance: Fig. 8 shows the detection rate of the active probing interval after 10 runs. The x-axis displays multiple configured data plane packet loss values, ranging from 3 to 50%. The y-axis displays the detection rate in percent. For the lowest packet loss values of 3 and 5%, the probing application offers a detection rate of 10%. For 10% and 15% packet loss the detection rate increases to 70%. A detection rate of 90% has been measured for the packet loss values between 20 and 30%. Beyond that, the application is in 100% of the cases able to detect the configured packet loss in the data plane.

In Fig. 9 the reaction times of the active probing application for multiple packet loss values are depicted. The x-axis shows the packet loss values ranging from 3 to 50%, the y-axis the reaction time in seconds. Beginning with a detection time of around 96 seconds for 3%, the detection time continuously decreases with an increasing packet loss value. For 5% the reaction takes place within 85 seconds. 10% of packet loss is detected within 47 seconds, 15% within 42 seconds, 20% after 40 seconds. Roughly 37 seconds required to detect 30% of packet loss, 29 seconds for 40%, and, finally, 50% of packet loss are detected within 19 seconds.

In summary, the results of Fig. 8 and 9 demonstrate

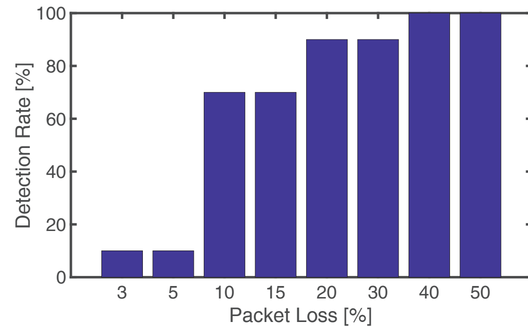

Fig. 8. Detection rate of the Active Probing Application

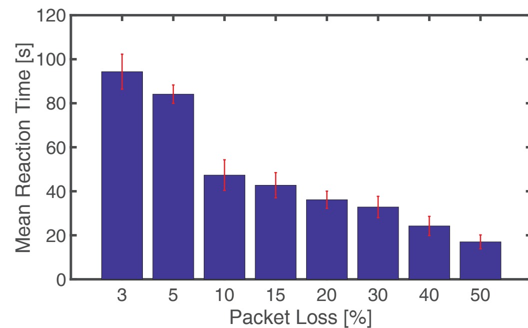

Fig. 9. Reaction times with the Active Probing Application enabled

the increased detection performance of the active probing application. In terms of detection rate, the new application surpasses the performance of the native ONOS detection mechanism. Our model, and the provided measurements, in our previous work show a detection probability of 10% for a packet loss probability of 40% [4]. The new application, in comparison, already offers this detection rate for packet loss values of 3%. Additionally, the detection times have decreased as well. Before implementing the active probing mechanism, the detection time for packet loss value of 40% was around 100 seconds. After the successful implementation of the app, the detection of this link status is already possible after 29 seconds. Reviewing the resource utilization on side of the controller host also reveals that the goals in terms of overhead have been kept. The new application offers a better detection performance whilst requiring the same or even less CPU or RAM usage.

IV. Adapting to Change

The proposed mechanisms for the active probing application does not yet consider one parameter that requires consideration: the constant change in the network, especially the variance in the packet inter-arrival times. Especially the impact of jitter in the data plane has been introduced and analyzed in [4].

Taken from this work are Equations 1 and 2 which we are used again for the optimization algorithm. Equation 1 is used

to determine the mean detection time of a link failure with packet loss p

$$t_{detect}(p,\theta,\lambda) = \frac{1}{p^{\lfloor \theta \cdot \lambda \rfloor}} min(\frac{1}{1-p} \cdot \frac{1}{\lambda}, \theta) \quad . \quad (1)$$

The minimum value for the timeout θ can be calculated by evaluating the inverse cumulative distribution function of a normal distribution $N^{-1}_{1/\lambda,\theta}(p)$ with mean $\frac{1}{\lambda}$ and standard deviation σ depending on the tolerated false positive rate $\bar{f}_{fail}$

$$\theta_{min}(\lambda,\sigma,\bar{f}_{fail}) = N^{-1}_{1/\lambda,\sigma}(1-\bar{f}_{fail}) \quad . \quad (2)$$

A false positive in this case describes the event that a link failure is considered as faulty due to jitter in the data plane.

The remainder of this section focuses on proposing an algorithm that is able to cope with these conditions. Afterwards, the implementation and combination with our probing extension is evaluated.

A. Automated Probing Rate Adaptation in the App

As shown in the previous work [4] the timeout θ has to be increased depending on the jitter on the link to fulfill a certain target rate of false positive link failure events. To fulfill a certain target detection time, the timeout θ can be reduced or the probing rate λ can be increased, which, in turn, affects the rate of false positive link failures. Hence, there is a trade-off between a low rate of false positive link failure events and a low failure detection time. To keep the load on the controllers low, the probing rate θ_{min} has to be minimized after the formula of Equation 7 of [4]. Therefore, we propose an algorithm for a self-optimized process that meets the target rate of false positive link failure events $\bar{f}_{fail}$ and the target mean detection time $\bar{t}_{detect}$ (c.f. Equation 4 of [4]) given the tolerated packet loss rate $\bar{p}$ and minimizes the controller load. The term false positive link failure describes the event when a link failure is triggered due to an exceeded timeout due to jitter instead of lost probing packets (cf. [4]).

Algorithm 1 Probing Rate Adaption

1: **procedure** PROBING RATE ADAPTION
2: **Input Parameters:**
3: tolerated packet loss rate $\bar{p}$
4: tolerated false positive rate $\bar{f}_{fail}$
5: target detection time $\bar{t}_{detect}$
6: **Variables:**
7: jitter σ
8: probing frequency λ
9: timeout θ
10: *start*:
11: Determine $\theta(\lambda,\sigma,\bar{f}_{fail})$ based on Equation 2
12: Calculate expected detection time $t_{detect}(\bar{p},\theta,\lambda)$ based on Equation 1
13: Set probing frequency $\lambda := \lambda \cdot t_{detect}(\bar{p},\theta,\lambda)/\bar{t}_{detect}$
14: Go to *start*

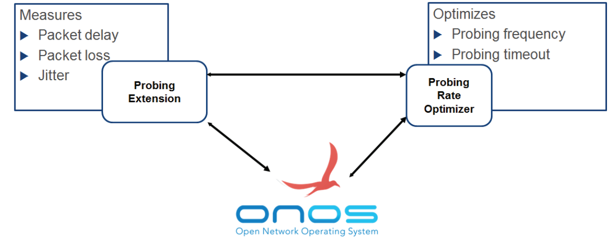

Fig. 10. Schematic Integration of Algorithm 1 into the Active Probing Extension

At first, the new timeout θ is calculated according to Equation 2. Afterwards, by using the new timeout, the new expected detection time is calculated according to Equation 1. Finally, the new probing frequency is set to the old probing frequency times the quotient of the expected and the old detection time. Afterwards, the process restarts at the beginning, and, therefore, is always adapting to the current network situation.

B. Integration with the Active Probing Application

In order to take advantage of the proposed auto-adapting algorithm, it has to be implemented and integrated with the already existing active probing application. The only missing information inside the application is the current network jitter. The probing frequency and the timeout are both configuration parameters that can be read from the application. The target variables tolerated packet loss rate $\bar{p}$, tolerated false positive rate $\bar{f}_{fail}$, and the target detection time $\bar{t}_{detect}$ are input parameters that will be loaded upon start-up of the application.

The app already measures the one-way latency of each connected link with each transmitted probing packet. Therefore, to determine the network jitter, only a new data field has to be created that stores the difference in link delay from one measurement point to the next one. The remainder of the implementation is putting together the data and implementing Algorithm 1.

Figure 10 shows the new optimization cycle of the application. The active probing extension measures the packet delay, the packet loss and the jitter of a link of the network topology. These values are then put into the algorithm. Based on the predefined configuration parameters, the algorithm then calculates a new probing frequency and a new probing timeout for the current environment conditions. This output is then used to modify the actual probing frequency and the probing timeout of the active probing extension. As network environment parameters such as link delay, network jitter, and packet loss tend to change over time, this whole optimization process is repeated regularly.

C. Evaluation

In order to highlight the advantages of the self-optimization algorithm, measurements with and without activated self-optimization have been conducted. In order to proof the functionality of the auto-optimization feature of the active probing application new measurements have been performed. The testbed of Section III-D has been used again.

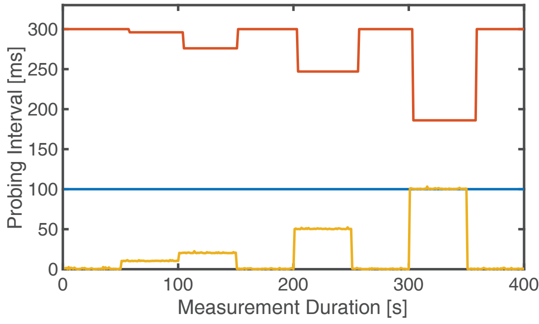

Fig. 11. Impact of Jitter on Probing Interval

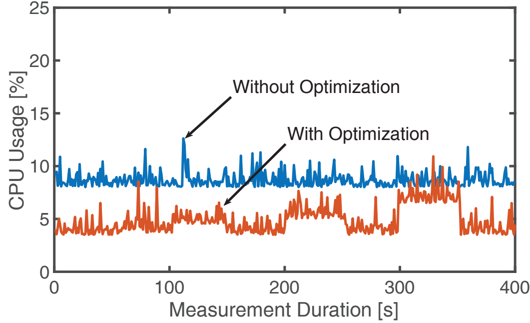

Fig. 12. CPU Usage both with and without the Optimization Algortihm enabled

At first the actual adaption of the probing rate under changing jitter levels is investigated. In this scenario, the probing interval for the probing extension without the optimization has been set to 100 ms and the timeout-factor to 3, resulting in an expected detection time of 300 ms. Fig. 11 shows the jitter interval in orange, the probing interval for measurements without the optimization algorithm enabled in blue and the probing interval for measurements with the optimization algorithm enabled in red. In this scenario, the jitter level changes each 50 seconds.

With the probing interval of the extension without the optimization, the change in the jitter level does not impact it in any way. The probing interval is constantly at 100 ms. The optimized version, in contrast, shows an adoption to the changed networking environment. Already small changes in the probing interval can be a drastic difference in the actual reaction time if a failure happens with such environment parameters. As previously introduced, a small change in the jitter level can have a huge impact if the probing extension does not react to it [4]. As soon as the jitter increases, the timeout factor increases as larger timeouts lead to a reduced false positive rate of link failure events. Additionally, as demonstrated later on, this change in the probing frequency has an impact on the resource utilization of the controller host. Throughout the whole measurement the adaption time of the optimization mechanism is on average 3 seconds. This factor is influenced by two factors: First, the probing extension has to detect the jitter level, which takes some time, as probing

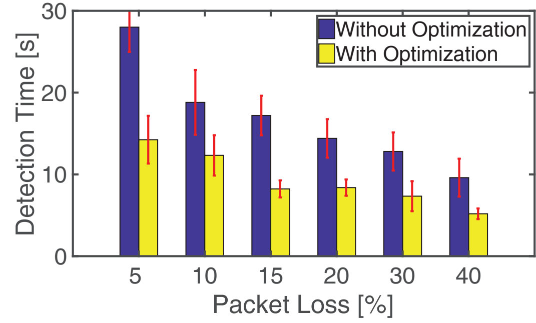

Fig. 13. Reaction times to packet loss events both with and without the Optimization Algorithm enabled

packets have to be transmitted, received and analyzed. Second, as we use multiple controllers, a synchronization between these two nodes is required. For this setup, the synchronization time has been set to 5 seconds. Decreasing these values is possible, but would not necessarily increase the detection performance.

Fig. 12 shows the CPU usage for above scenario. The CPU usage for the controller host without the optimization algorithm enabled is shown in blue, the usage with the optimization Algorithm enabled is shown in red. Here, the previously mentioned difference in the CPU usage is visible. Without the optimization, the CPU usage is always in between 8 and 11%, with the exception of a few peaks. With the optimization enabled, the CPU usage is lower throughout the measurement, with values between 3 and 8%. Each time the probing interval is increased, the CPU load is reduced, and vice versa. The memory utilization for both with and without the optimization has also been analyzed. But, as the measurement showed little to none difference in the results, they are not shown here. The memory utilization was at levels that have been shown in Seczion III-D.

In order to measure the detection time for the case of link failure in the data plane, the scenario settings have been changed. The optimization algorithm calculates a probing interval of 208 ms and a time-out factor of 1.2. For the non-optimized variant, a probing interval of 100 ms and a time-out factor of 3 is chosen. As the actual packet loss in the data plane has no impact on the optimization process, the calculated parameters remain constant. In Figure 13 the reaction times for both with and without the optimization algorithm are compared. For this scenario, the topology is set up and running without any packet loss for 150 seconds. Afterwards, as introduced in Section III-D, packet loss is added to the link between switch 1 and switch 2. Finally, the network is set back to 0% packet loss for the next measurement. The x-axis of Figure 13 shows multiple packet loss values from 5 to 40%, the y-axis shows the detection time to a packet loss event in seconds. The values for the probing extension without the optimization algorithm are depicted in blue, the results with the optimization algorithm in yellow, respectively. In red 95% confidence intervals are shown. Overall, the measurements

present the expected results. In all cases increasing the packet loss decreases the reaction time. Enabling the optimization mechanism decreases these times to 7 and 5 seconds.

Furthermore, the optimization algorithm constantly outperforms the non-optimized version of the probing extension. The decrease of the reaction time for increasing packet loss can be explained by the mechanism of the active probing extension. In order to detect packet loss on a link three consecutive packets have to be lost. Therefore, increasing the packet loss increases the probability of three consecutively dropped packets on a link. This phenomena has been introduced an explained in our previous work [4]. It also explains the performance gain of the optimization algorithm. The optimization algorithm analyzes the configuration parameters and calculates a suiting probing interval and timeout. In order to optimize the reaction time for the case of packet loss, it reduces the timeout. With this setting less than two consecutive packets have to be lost to trigger a detection, which has a higher probability of three consecutively dropped packets.

Overall, the performance gains of the optimization mechanism are as expected. With normal network environment parameters, close to 0% packet loss and 0 ms of jitter, the algorithm relaxes the probing interval and, therefore, requires less CPU resources. Furthermore, under harder network conditions, the optimization algorithm detects a challenging situation and decreases the probing interval in order to increase the detection performance.

V. Conclusion

With the introduction of Software-Defined Networking and its decoupling of the control from the data plane, network engineers have been given a new, flexible and central tool to monitor networks. An open source tool that allows for SDN-enabled deployments is the ONOS SDN controller. According to its developers, it offers reliability, scalability, and is already production ready. The question whether this interplay of technology is really working in the field is of high interest for service providers and requires thorough analysis before migration. Our previous work [4] has shown that ONOS is unable to reliably detect link impairing effects such as packet loss in the data plane. An exemplary result is that ONOS in its default configuration requires more than one minute to detect packet loss values of 50%. In order to improve the detection performance multiple options exists.

The approach presented in this paper introduces an active probing mechanism for the ONOS SDN controller as a Northbound API application. It generates, transmits, and receives probing packets through and from the network. Therefore, it is possible to derive network data on a link level, e.g. the packet loss rate, or the the link delay. The evaluation performed in a testbed shows that this application is to increase the detection performance, both in rate and time, whilst only marginally increasing the load on the controller resources, such as CPU or RAM.

An additional feature presented in this paper is to implement an self-adapting active probing mechanism into the ONOS controller that adapts to the changes in the network conditions, e.g. jitter. Furthermore, in order to tradeoff controller load to detection performance, the time between two succeeding packets and the internal timeouts are adapted to the current network conditions. An algorithm that optimizes the whole probing process during run-time has been presented. This algorithm analyzes the current network environment parameters and calculates new probing intervals and probing timeouts, which have an impact on the detection performance. Afterwards, the algorithm is implemented as part of the already existing probing extension. Measurements demonstrating the difference between an enabled and a disabled optimization algorithm are presented. Furthermore, under normal environment parameters, the probing interval will be increased and, therefore, the CPU utilization on side of the controller can be reduced.

References

[1] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "Onos: towards an open, distributed sdn os," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014, pp. 1–6.

[2] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *2014 IEEE 15th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)(WOWMOM)*, June 2014, pp. 1–6. [Online]. Available: doi.ieeecomputersociety.org/10.1109/WoWMoM.2014.6918985

[3] I. C. Society, "Station and Media Access Control Connectivity Discovery," *IEEE Std. 802.1ab*, vol. IEEE Standard for Local and Metropolitan Area Networks, pp. i–204, 2009.

[4] C. Metter, V. Burger, Z. Hu, K. Pei, and F. Wamser, "Evaluation of the Detection Capabilities of the ONOS SDN Controller," in *7th IEEE International Conference on Communications and Electronics (IEEE ICCE 2018)*. IEEE, 2018, pp. 1–6.

[5] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[6] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[7] L. Quan, J. Heidemann, and Y. Pradkin, "Trinocular: Understanding internet reliability through adaptive probing," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 255–266.

[8] C. Cascone, D. Sanvito, L. Pollini, A. Capone, and B. Sansò, "Fast failure detection and recovery in sdn with stateful data plane," *International Journal of Network Management*, vol. 27, no. 2, p. e1957, 2017.

[9] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 61–66.

[10] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, no. 6, pp. 656–665, 2013.

[11] S. S. Lee, K.-Y. Li, K.-Y. Chan, G.-H. Lai, and Y.-C. Chung, "Path layout planning and software based fast failure detection in survivable openflow networks," in *Design of Reliable Communication Networks (DRCN), 2014 10th International Conference on the*. IEEE, 2014, pp. 1–8.

[12] M. Foschiano, "Cisco systems unidirectional link detection (udld) protocol," Internet Requests for Comments, RFC Editor, RFC 5171, April 2008.

[13] "Juniper Monitor Ethernet Delay-Measurement." [Online]. Available: https://www.juniper.net/documentation/en_US/junos/topics/reference/command-summary/monitor-ethernet-delay-measurement.html

[14] A. Dusia and A. S. Sethi, "Recent advances in fault localization in computer networks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 3030–3051, 2016.