Detection of Fog Network Data Telemetry Using Data Plane Programming

Zifan Zhou
DTU Fotonik, Technical University of Denmark, Lyngby, Denmark
zifz@fotonik.dtu.dk

Eder Ollora Zaballa
DTU Fotonik, Technical University of Denmark, Lyngby, Denmark
eoza@fotonik.dtu.dk

Michael Stübert Berger
DTU Fotonik, Technical University of Denmark, Lyngby, Denmark
msbe@fotonik.dtu.dk

Ying Yan
DTU Fotonik, Technical University of Denmark, Lyngby, Denmark
yiya@fotonik.dtu.dk

Abstract

Fog computing has been introduced to deliver Cloud-based services to the Internet of Things (IoT) devices. It locates geographically closer to IoT devices than Cloud networks and aims at offering latency-critical computation and storage to end-user applications. To leverage Fog computing for computational offloading from end-users, it is important to optimize resources in the Fog nodes dynamically. Provisioning requires knowledge of the current network state, thus, monitoring mechanisms play a significant role to conduct resource management in the network. To keep track of the state of devices, we use P4, a data-plane programming language, to describe data-plane abstraction of Fog network devices and collect telemetry without the intervention of the control plane or adding a big amount of overhead. In this paper, we propose a software-defined architecture with a programmable data plane for data telemetry detection that can be integrated into Fog network resource management. After the implementation of detecting data telemetry based on In-Band Network Telemetry (INT) within a Mininet simulation, we show the available features and preliminary Fog resource management based on the collected data telemetry and future telemetry-based traffic engineering possibilities.

2012 ACM Subject Classification Networks → Programmable networks

Keywords and phrases SDN, P4, P4Runtime, control planes, Fog, Edge

Digital Object Identifier 10.4230/OASIcs.Fog-IoT.2020.12

1 Introduction

Transmission latency for IoT applications such as health monitoring and emergency response has become crucial in providing reliable and efficient performance. However, due to the distributed location of IoT devices, some may be far from the core and Cloud networks. Considering that the latency of data communication is significantly impacted by the distance, IoT data can experience long propagation delays to reach the Cloud [3]. Besides, the centralized Cloud data centers often store and process a large amount of data from billions of IoT devices, the heavy workload can also cause a long processing delay for IoT data. Fog computing enables the distribution of Cloud services to the edge network, with a closer location to the devices. Therefore, the round trip latency from devices to the Fog and back to the devices is shorter. Fog networks are usually lightweight, and several Fog nodes could be placed at the edge network separately, the possibility of network congestion is lower, by

distributing data to these Fog nodes. Last but not least, regarding the energy needed to transmit a byte of data from IoT devices to computing nodes, processing applications expect that Fog Computing can also reduce the energy consumption in the network [11].

To maintain a high availability and performance of Fog computing, resource provisioning plays an essential part in network management. These decisions are usually made based on specific metrics to distribute workload optimally. Factors such as time, energy, user-application context, etc., have been mentioned in the literature of Fog computing. Since latency-critical applications have specified a tolerant maximum delay for finishing time, it is straightforward to provide service and resources according to network delay, i.e. the whole time for completing a task should be ahead of the deadline of the application. Unlike Cloud computing where most of the nodes are closely located in the data center, Fog nodes are often dispersed over the access network, which adds an extra transmission delay among Fog nodes, and that makes it more difficult for Fog nodes to make offloading decisions. Accordingly, to fulfill the transmission of Fog data, workload distribution has to take the offloading delay into consideration, which can be collected by the application of data plane programming when all nodes are time synchronized.

Integrating a centralized controller into a distributed Fog architecture further facilitates network management. Given the data telemetry of the Fog network, the controller sends real-time workload distributing instructions to linked nodes. The implementation of a centralized control plane assesses all delay factors and with the ultimate goal to fulfill transmission requirements, avoiding a lack of status information due to an isolated location in Fog networks.

In the existing literature, the research community has proposed several studies on the delay analysis of Fog computing, but there are not many discussions on detecting and monitoring the Fog network state in practice. In [10], mathematical formulas of computation and transmission delay in Fog computing and one offloading mechanism to optimize the resource allocation were introduced. Another mathematical model of latency in Fog architecture for 5G cellular networks was proposed in [4]. In [13], a software-defined embedded system for Fog computing optimization was presented. In [6], the authors proposed a delay-aware path finding algorithm based on a Software-Defined Network (SDN) framework and OpenFlow protocol to optimize network routing/rerouting performance.

Following the motivations and the existing research, we propose an architecture using programmable data planes to collect data telemetry and conduct workload balance, moreover, with a SDN controller to perform centralized management of the Fog network. We use queuing delay as the metric for resource management in the architecture. Finally, we create a simulation of the network monitoring and load balancing in Mininet [8] for a proof of concept. The rest of the paper is organized as follows: Section 2 explains the data plane monitoring and the proposed architecture. Section 3 includes the implementation of the simulation in Mininet and the topology used for the simulation. In Section 4, we present the results of the collected data telemetry, and finally, in Section 5 we conclude the paper and discuss the future work.

2 Software-defined managing architecture

In this section, we briefly describe the architecture for data telemetry detection and Fog resource management. To keep track of the network state, a few protocols have been used to request and present the real-time state of the network devices, e.g. Simple Network Management Protocol (SNMP) [12]. In this proposal, we leverage the data plane programming

language P4 [1] for the devices used in Fog networks (i.e. gateways, switches, and Fog computing devices, etc.). P4 programs are used to define the pipeline behavior for packet processing in all the P4 programmable devices and the control plane can manage the data plane via a control-to-data runtime protocol (P4runtime).

By using P4 programs in Fog networks, specific network telemetry can be collected via data-plane operations. This is achieved via P4 programming and In-band Network Telemetry (INT) [7]. The devices create metadata of each packet within the pipeline, therefore, it is possible to collect internal timestamps when entering and leaving the queues of the pipeline. Meanwhile, the queue depth metadata changes along receiving and sending packets, which indicates the current workload of the device.

INT was derived from the Tiny Packet Program (TPP) that embeds programs into network packets from end hosts to query the state of network switches [5]. Similarly, INT headers that contain INT instructions and description of the header are added to normal network packets.

As shown in Figure 1, we define the INT domain which contains all the P4 programmed INT-capable devices. The process is initiated by the first node in the domain, so-called *INT source*, which adds the INT header and the required data telemetry to a particular packet passing being forwarded. The header can be encapsulated as a payload after several network protocols (e.g. after TCP or UDP and before payload, etc). The following devices on the path interpret the instruction contained in the INT header and then add an extra layer of telemetry data per device with the corresponding state information to the telemetry data within the packet. After the last device in the INT domain (i.e. the *INT sink*), the generates an *INT report* by encapsulating the packet that has just traversed the network. When the packet has been cloned in the switch pipeline, the *INT sink* also removes the telemetry headers and data to recover the original network packet from and sends it to the receiver. Therefore the *INT sink* generates 2 packets, one for the original traffic endpoint (without any telemetry data attached) and the other as an *INT report* packet for the telemetry collector.

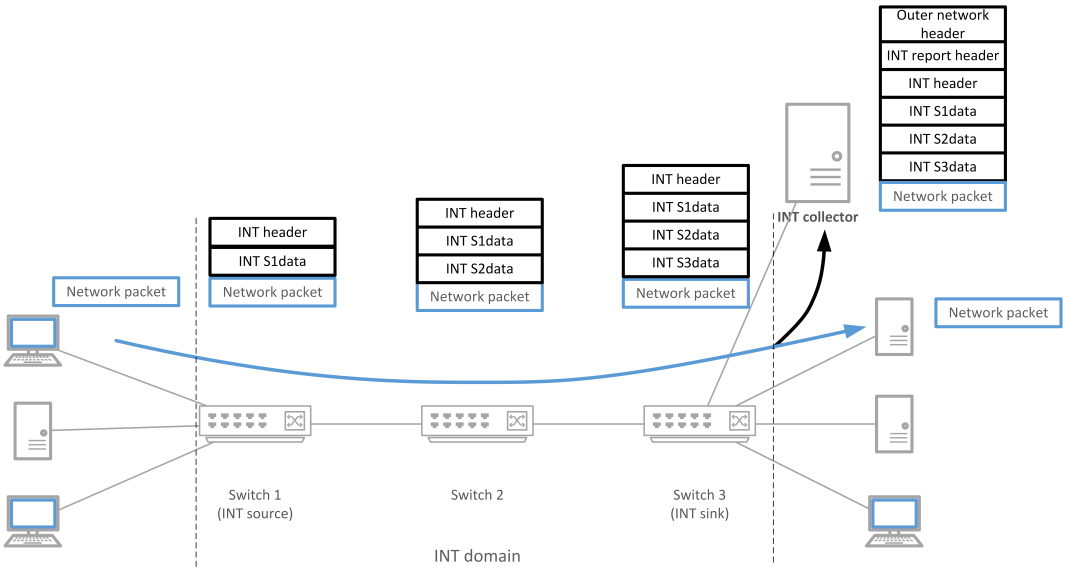

Figure 1 Illustrating of data telemetry collection in INT-capable P4 devices.

As mentioned in the previous paragraph, the destination address in the outer network headers of the *INT report* refers to the collector where all the INT packets are sent to. The collector is an endpoint that decodes all the INT header stacks and, in our proposal, stores delay information and tracks the real-time changes. Furthermore, the collector can make decisions to distribute the workload among Fog nodes based on the delay information. Thus, it requires several modules and interaction among different tasks to assemble the collector, and in this architecture, we use a Software-Defined Network (SDN) controller that integrates all the modules and comes with the following benefits:

- **Centralized Fog management**: the control plane is separated from the data plane in Fog networks, the abstraction and control of the network are managed by the controller.
- **Real-time network management**: implementing the SDN controller and a programmable data plane in Fog networks enables controllers to modify flow tables in the devices at runtime.
- **Flexible application**: controller's network abstraction and shared control-plane software modules make it easier to program network applications such as routing algorithms and load balancers in the Fog network.

Figure 2 depicts the abstraction of an example Fog network with the architecture. As mentioned above, we used a P4 program for the data plane of the devices used in the Fog network. All of the devices are physically connected with the controller, thus the controller has direct access to modify the data plane actions of the devices. Historically, communication between the controller and the devices is achieved by a well-defined application programming interface (API), OpenFlow [9]. However, in our example, we use P4Runtime as the protocol between the centralized controller and switches. Using P4Runtime allows controller configurations to stay both centralized for various switches or work locally within a single switch (which can not be done using OpenFlow-enabled switches). Having control planes that can work independently within the same switch and pipeline can help in developing future applications for data plane telemetry and traffic steering. For instance, a local control plane can be in charge of small scale monitoring while a centralized control plane can be in charge of computing new paths using a centralized network view. The tasks can be properly organized among different controller configurations to achieve better performance.

By sending INT instructions from the controller, the *edge gateway a* can initiate the INT process, and the last-hop switch, e.g. top-of-rack (TOR) switch, to the Fog node, will generate the *INT report* or vice versa. To follow our future goals of using telemetry data, the queuing in both directions, as well as processing delay (within the pipeline) in the Fog nodes, are exposed to the controller, then it can improve traffic forwarding paths following a path optimization using telemetry data.

To collect the network state, monitoring protocols either create special probe packets (postcard mode) or add extra headers (integrated into traffic), both methods increase the overhead of the network. On the other hand, the precision of the detection is dependant on the frequency of collecting of network state, but frequent detection usually generates more overhead. In this architecture, we set the INT frequency adjustable considering the trade-off with overhead, thus, not all the network packets carry the INT header and the results can describe the network state. Only packets that match specific rules on a table at the beginning of the pipeline will ever be monitored, leaving the rest of the traffic not tracked.

In this architecture, data telemetry in the network is detected by the centralized accumulator, hence, the SDN controller could also distribute computational resources according to the real-time network state. Since the INT packets only traverse within the INT domain and through data plane, end devices and applications in the Fog nodes are unaware of the whole detecting process.

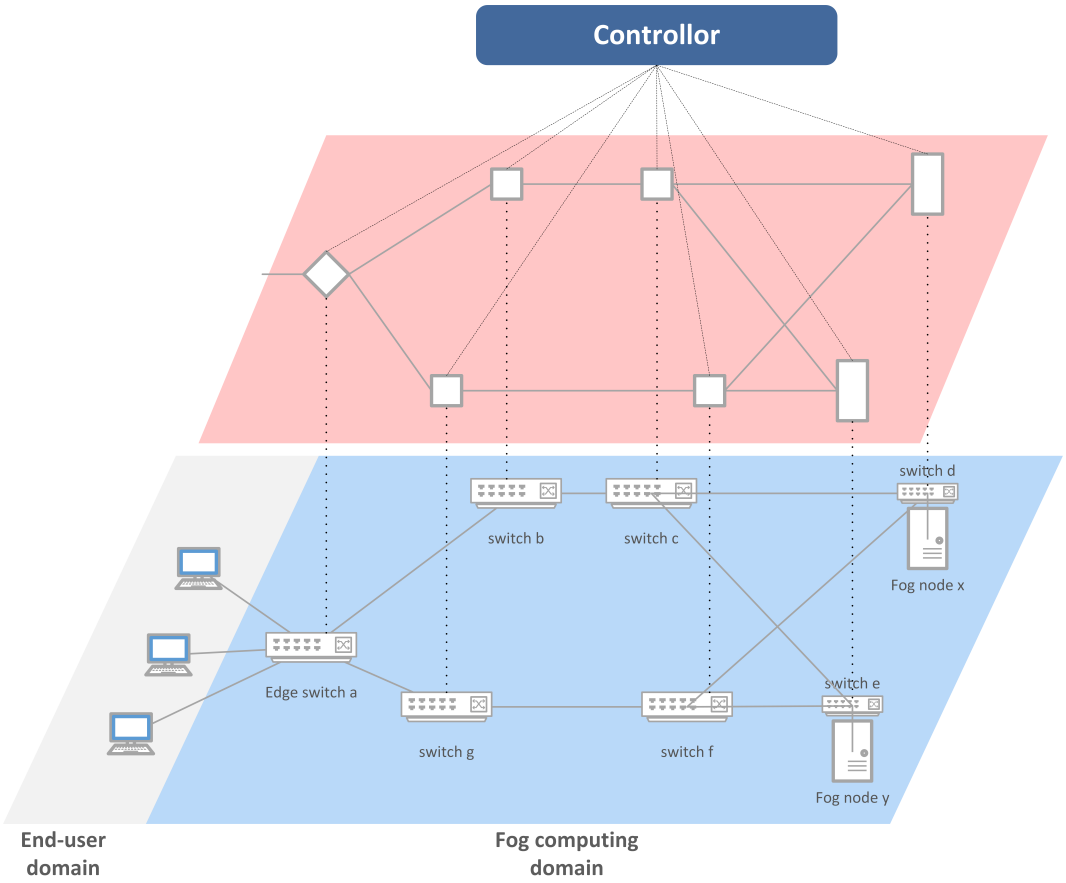

Figure 2 All devices in the Fog computing domain are connected to the controller, and all the INT reports are sent to the controller, thus the controller has the abstraction of the network and the state of each link and device.

3 Simulation

In this section of the paper, we have tried to demonstrate, within the Mininet simulation environment, the telemetry capabilities of INT and P4 programmable devices. In Mininet we set up virtual hosts as nodes that generate and receive packets. P4-compatible BMv2 [2] software switches are used as programmable switches in the simulations.

In these tests, we have set up 4 switches in between one Fog node (Host 2) and an end-user (Host 1) to test the time that packets spend in switches' queues. The first test shows attempts with continuous ICMP traffic exchange so we can observe a specific period of the time that packets spend in queues. In the second test, we conduct the same simulation but test with UDP traffic exchange using iperf. We try to load the switches with a relatively higher demand for traffic (1 Mbit/s of bandwidth, 1000 bytes per packet) and present the performance of the switches from queuing delay telemetry. We have to lower the maximum packet size in the generated UDP stream in order to be able to attach telemetry data to the network packets (without surpassing maximum Ethernet packet size).

In the last test, we demonstrate the preliminary redirecting of network flow in the simulation when one switch on the path starts to show poor performance. During the whole simulation period, Host 1 keeps sending ICMP traffic to Host 2 through the default path:

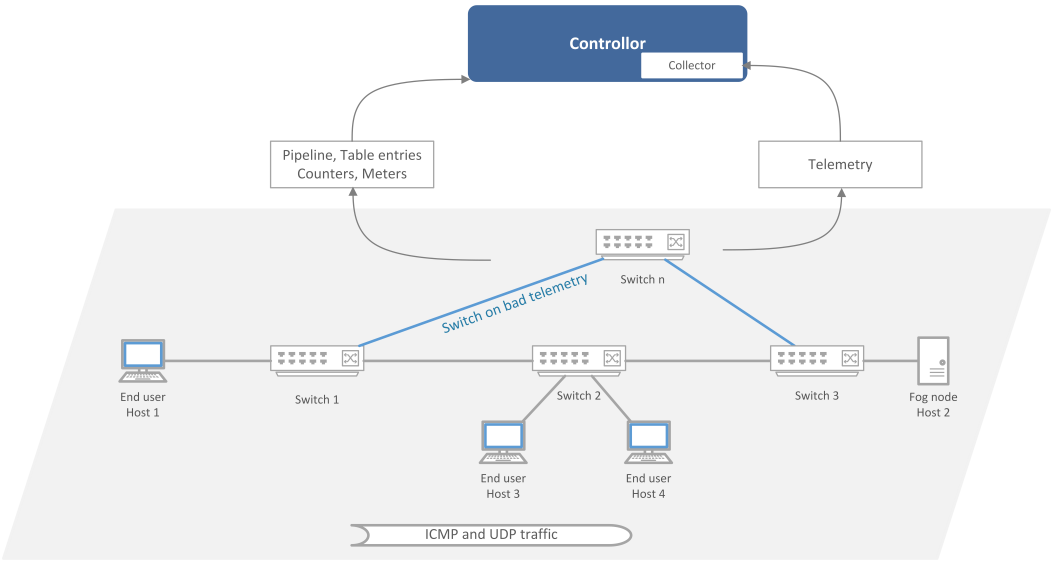

Figure 3 Simulation topology to monitor traffic between edge switch and end user. This telemetry data can be reused to update the traffic forwarding at runtime.

switch 1 – switch 2 – switch 3. Then we create the scenario when switch 2 has a descent of performance by sending interfering traffic from Host 3 to Host 4 through switch 2. The queuing delay is monitored along with the simulation and after reaching a certain threshold, i.e. bad telemetry, switch 1 will redirect the ICMP flow to switch n, hence offloading from switch 2.

As seen in Figure 3, we have used the Mininet emulator to create a virtual network that connects virtual hosts (Edge/Fog devices) with virtual P4 programmable switches. This enables us to create custom topologies and test traffic between hosts with a custom network pipeline. In this way, we have been able to customize the pipeline and track which network flows we want to monitor. In this case, we have both tracked the ICMP requests from Host 1 (IP=10.0.1.1) to Host 2 (IP=10.0.2.2) and also the specific *iperf* generated UDP traffic from Host 1 (iperf client) towards port 4444 on the server. In order to monitor these flows, we program the tables of the pipeline by defining which parameters of the flows we monitor. Specifically, fields of IPv4, ICMP and UDP headers.

When we detect a packet from a type of traffic that we have to monitor, we add the telemetry headers and extract them at the end of the path, sending the whole packet and telemetry headers to a collector. The data stored by the collector is the preferred data needed by the controller to, possibly, change traffic flows at runtime.

4 Results

In this part of the paper, we focus on bringing simulation and results for INT monitoring. As we have set up the environment to monitor the state of P4 programmable switches, now we present the statistics of telemetry to demonstrate how we can visualize the internal state of forwarding devices to possibly further use it with traffic engineering purposes.

In Figures 4 and 5, giving the results when the ICMP request packets circulating over S1 and S2 from Host 1 to Host 2, different behavior of both switches are shown. In Figure 4 we can observe 3 outlining spikes that define a long time of packets within the switch but overall

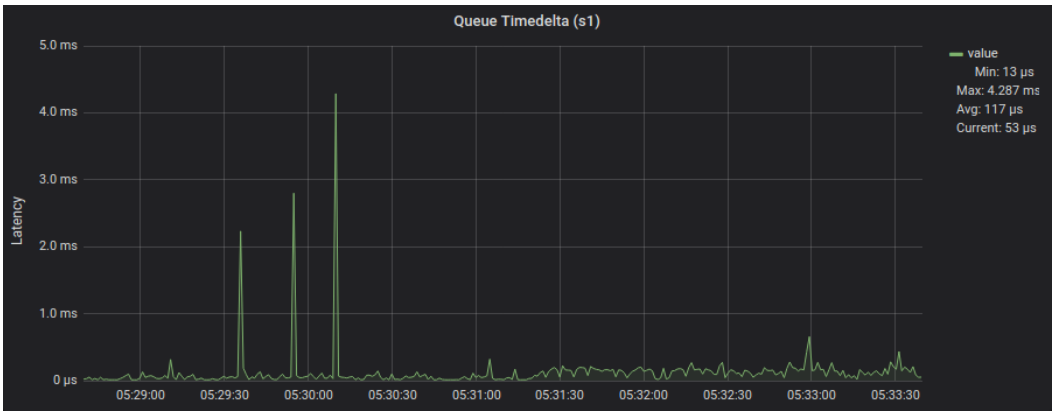

Figure 4 Time that packets spent in the queue on Switch 1 for ICMP traffic exchange. Represents ICMP requests from Host 1 to Host 2.

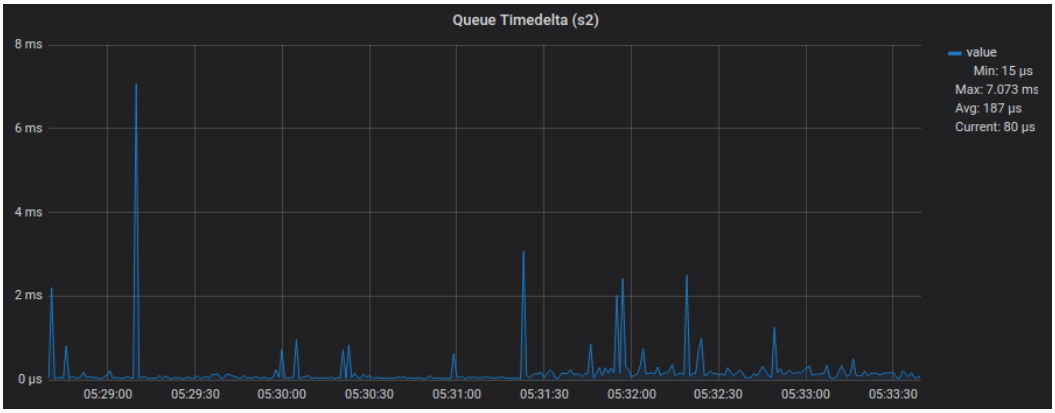

Figure 5 Time that packets spent in the queue on Switch 2 for ICMP traffic exchange. Represents ICMP requests from Host 1 to Host 2.

good behavior considering the rest of the data. The switch shows 3 spiking queue delays over 2 ms but an overall correct queue delay. We have a similar result in Figure 5, but this time, there is an outlining spike at the beginning of the figure and then a few less well-performing spikes exist in the rest of the figure. ICMP packets are not very demanding for switches so packets spend a very short time in queues, graphs show an average time of 117 and 187 μs, respectively. Seeing these average results and Figures 4 and 5 we can observe that S1 has performed slightly better than S2. As we have not stress-tested the switches, the spikes on S1 and S2 only refer to the unexpected bad performance that software switches could have.

Looking into Figures 6 and 7, we can demonstrate the ability to monitor and signalize badly performing moments of the switches. For instance, Figure 6 shows 2 specific moments when packets experience relatively long queuing time in the switch. The peek in the result gives a worst-case queuing time for over 2 seconds, which can result in a poor experience for a realistic latency-critical application that might be harmful. Due to the amount of traffic, switches need to perform tasks quicker and faster than in the previous case, having specific bad performing moments.

Similar problems appear in Figure 7, where the worst-case results show a few cases of 1.5 seconds queuing time. Because the BMv2 switch is a virtual switch, it shares some capabilities with the virtual machine (VM) that hosts the testing process. We can expect a

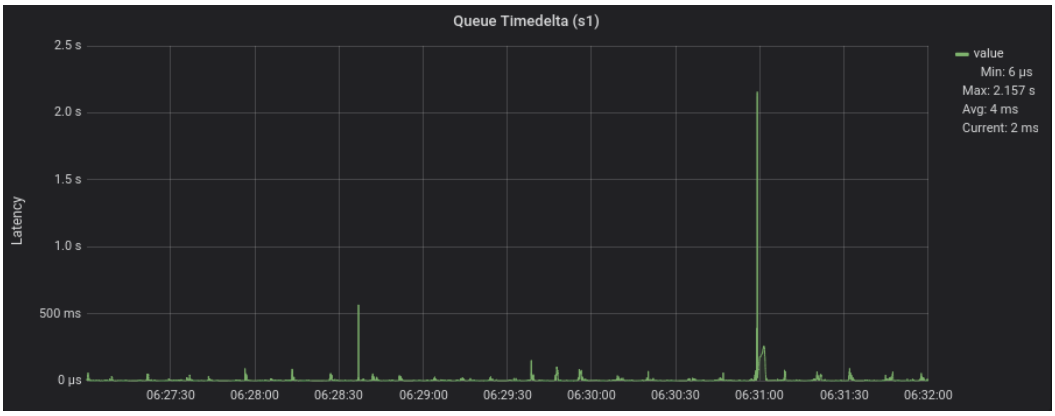

Figure 6 Time that packets spent in the queue on Switch 1 for UDP traffic exchange. Represents packets from Host 1 (iperf client) to Host 2 (iperf server).

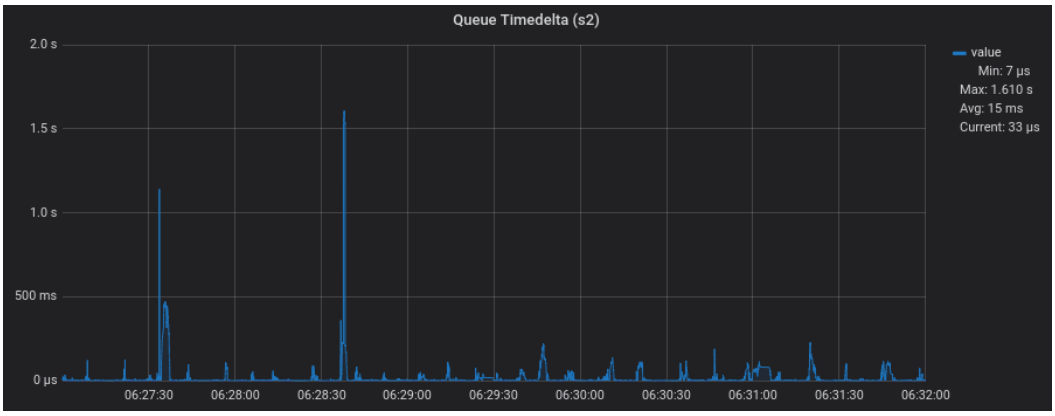

Figure 7 Time that packets spent in the queue on Switch 2 for UDP traffic exchange. Represents packets from Host 1 (iperf client) to Host 2 (iperf server).

few cases that the host VM might affect the switch performance in the simulation process, like the highest peaks shown in Figures 6 and 7, whereas learning from most other packets, switches have performed correctly. Besides having a higher amount of traffic compared to Figures 4 and 5 shows an expected worse performance in Figures 6 and 7.

Figure 8 illustrates the queuing delay from switch 1 in the flow redirecting scenario. As we can see, ICMP flow sent by Host 1 gets received in the queue of switch 1 at around time 17:44:40. The spikes in the queuing delay of switch 1 are mainly caused by the instability of the software switch used in the simulation. The ICMP flow shows up at switch 2 right after the moment it leaves switch 1, as given in Figure 9. From time 17:45:20, Host 4 starts the transmission of interfering flow to switch 2, and the queuing delay shows three significant increases due to each time the burst of interference. After the last spike at time 17:46:10, the redirecting threshold is reached and the ICMP flow no longer appears in the queue of switch 2. Instead, as shown in Figure 10, since time 17:46:10, the flow is redirected to switch n, indicating the workload has been distributed from switch 2 at runtime.

These tests demonstrate the ability to monitor the switches' performance in a per-packet way (in any pipeline we have programmed) and the fundamental workload-distributing ability based on the collected telemetry. The ultimate goal is to demonstrate how P4 devices and software-defined management can accelerate data exchange between end-users and Edge/Fog nodes by monitoring per-flow performance.

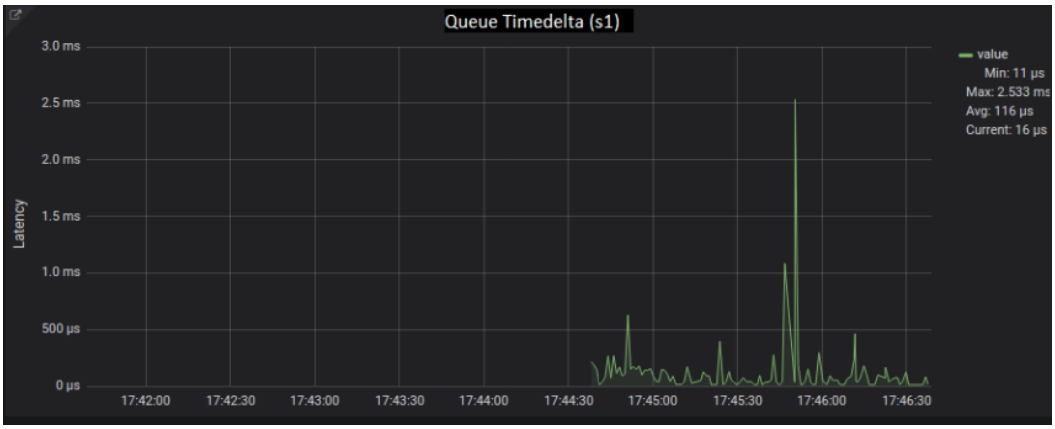

Figure 8 Time that packets spent in the queue of switch 1 for traffic redirecting.

5 Conclusion

In this paper, we have shown how data telemetry detection using data plane programming works, and how can software-defined managing architectures fit into Fog computing. Although SDN technologies have not primarily been focused on Edge/Fog networks, we have seen more and more researches working in this area. The fact that the SDN data plane has evolved from the typical OpenFlow switches to P4 programmable ones creates a huge space for new protocols, data plane telemetry measurement, and network management. We have also demonstrated how INT-capable P4 switches work within networks, and how to collect telemetry data, demonstrating with simulations on how to collect per-packet queuing delay. Furthermore, a preliminary workload distribution based on the collected data telemetry illustrates how data plane programming can be used in network resource management. We propose the network managing architecture of this paper to utilize data telemetry for optimization of Fog network, because we strongly believe that the data-plane operation will conform to the distributing nature of Fog network. This also enables new traffic engineering ways of distributing computational resources at runtime. We believe that our work will lead to new methods that achieve a lower network latency, by altering traffic forwarding

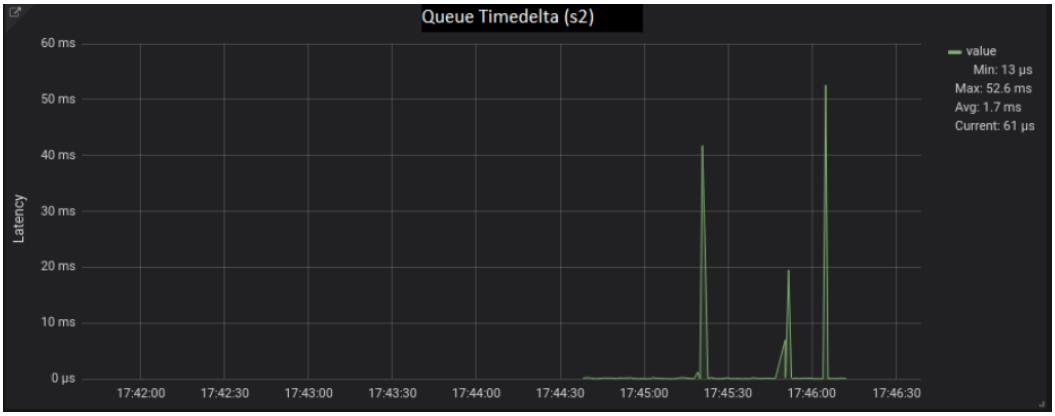

Figure 9 Time that packets spent in the queue of switch 2 for traffic redirecting.

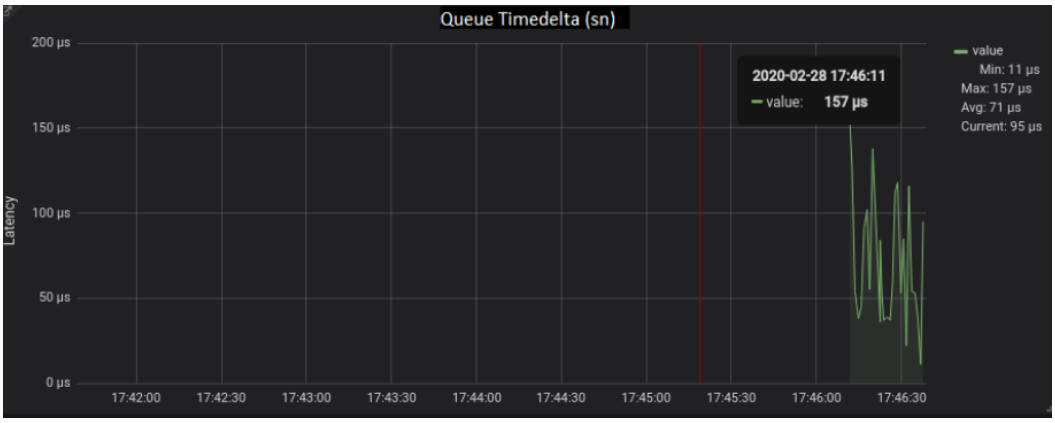

Figure 10 Time that packets spent in the queue of switch n for traffic redirecting.

paths based on real-time network state. Furthermore, this method can offer third party applications (Inter-SDN controller communication, Time-Sensitive Networks, etc.) to benefit from it.

References

1 Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014. doi:10.1145/2656877.2656890.

2 P4 Language Consortium. Behavioral model (bmv2). URL: https://github.com/p4lang/behavioral-model [cited 2020-01-21].

3 Amir Vahid Dastjerdi, Harshit Gupta, Rodrigo N Calheiros, Soumya K Ghosh, and Rajkumar Buyya. Fog computing: Principles, architectures, and applications. In *Internet of things*, pages 61–75. Elsevier, 2016. doi:10.1016/B978-0-12-805395-9.00004-6.

4 Krittin Intharawijitr, Katsuyoshi Iida, and Hiroyuki Koga. Analysis of fog model considering computing and communication latency in 5g cellular networks. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–4. IEEE, 2016. doi:10.1109/PERCOMW.2016.7457059.

5 Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. Millions of little minions: Using packets for low latency network programming and visibility. *ACM SIGCOMM Computer Communication Review*, 44(4):3–14, 2014. doi:10.1145/2740070.2626292.

6 Rutvij H Jhaveri, Rui Tan, Arvind Easwaran, and Sagar V Ramani. Managing industrial communication delays with software-defined networking. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11. IEEE, 2019. doi:10.1109/RTCSA.2019.8864557.

7 Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

8 Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010. doi:10.1145/1868447.1868466.

9 Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008. doi:10.1145/1355734.1355746.

10 Mithun Mukherjee, Suman Kumar, Mohammad Shojafar, Qi Zhang, and Constandinos X Mavromoustakis. Joint task offloading and resource allocation for delay-sensitive fog networks. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2019. doi:10.1109/ICC.2019.8761239.

11 Subhadeep Sarkar and Sudip Misra. Theoretical modelling of fog computing: a green computing paradigm to support iot applications. *Iet Networks*, 5(2):23–29, 2016. doi:10.1049/iet-net.2015.0034.

12 William Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addison-Wesley Longman Publishing Co., Inc., 1998.

13 Deze Zeng, Lin Gu, Song Guo, Zixue Cheng, and Shui Yu. Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system. *IEEE Transactions on Computers*, 65(12):3702–3712, 2016. doi:10.1109/TC.2016.2536019.