A LINK DELAY COMPUTATION METHOD FOR THE QUALITY OF SERVICE SUPPORT IN SOFTWARE DEFINED NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EFE BALO

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2019

Approval of the thesis:

A LINK DELAY COMPUTATION METHOD FOR THE QUALITY OF SERVICE SUPPORT IN SOFTWARE DEFINED NETWORKS

submitted by **EFE BALO** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar Dean, Graduate School of **Natural and Applied Sciences**	____________
Prof. Dr. İlkay Ulusoy Head of Department, **Electrical and Electronics Engineering**	____________
Prof. Dr. Ece Güran Schmidt Supervisor, **Electrical and Electronics Engineering, METU**	____________

Examining Committee Members:

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı Electrical and Electronics Engineering, METU	____________
Prof. Dr. Ece Güran Schmidt Electrical and Electronics Engineering, METU	____________
Prof. Dr. İlkay Ulusoy Electrical and Electronics Engineering, METU	____________
Assoc. Prof. Dr. Ertan Onur Computer Engineering, METU	____________
Assist. Prof. Dr. Barbaros Preveze Electrical and Electronics Engineering, Çankaya University	____________

Date:

ABSTRACT

A LINK DELAY COMPUTATION METHOD FOR THE QUALITY OF SERVICE SUPPORT IN SOFTWARE DEFINED NETWORKS

Balo, Efe
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Ece Güran Schmidt

September 2019, 60 pages

Packet switched networks cannot provide tight delay bounds that are required by certain types of applications despite facilitating high throughput. Therefore, delay measurement techniques for packet-switched networks have always grabbed the attention of the community to both utilize advantages of packet-switched networks and provide a realistic end to end delay prediction of packets. Software Defined Networking (SDN) is a new paradigm of packet-switched networking which gathers management functionality of network in a logically single controller. SDN is thought to eliminate problems of legacy layered architecture by utilizing the control information coming from all network layers. However, in SDN topology, control plane and data plane are separated which implies control packets for network management flow in a different channel than datapath channel. Moreover, the SDN controller has to have a decision metric similar to legacy link-state computation approaches in order to calculate the most efficient route in the topology. All of these indicate that link delay computation in SDN needs new perspectives different than the legacy network to achieve its proper operation.

In this thesis, we propose a link delay computation method for SDN topologies. For this purpose, we construct a framework which uses standard OpenFlow messages and computes the switch queuing delay in run-time. In this framework, we model each queue in SDN switches as a G/G/1 queue and measure the ingress traffic with OpenFlow meters. Then, we utilize meter statistics to obtain mean and variance of interarrival times between packets. After finding the average state of the queues we eventually infer the respective queuing delay from Little's equation. We demonstrate our method in three cases which are single flow per queue, multiple flows per queue and a test application which uses our delay information to determine the fastest queue of an SDN switch port. Also, we discuss the accuracy and the application limitations of the proposed method.

Keywords: QoS, SDN, delay measurement, OpenFlow

ÖZ

YAZILIM TABANLI AĞLARDA SERVİS KALİTESİ DESTEĞİ İÇİN BAĞLANTI GECİKMESİ HESAPLANMASI

Balo, Efe
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ece Güran Schmidt

Eylül 2019 , 60 sayfa

Paket anahtarlamalı ağlar, yüksek verim elde etmeyi kolaylaştırmasına rağmen, belirli uygulama türlerinin gerektirdiği sıkı gecikme sınırları sağlayamaz. Bu nedenle, paket anahtarlamalı ağlar için gecikme ölçüm teknikleri, paket anahtarlamalı ağların avantajlarını kullanmak ve paketlerin gerçekçi gecikme tahminini yapabilmek için her zaman topluluğun dikkatini çekmiştir. Yazılım Tanımlı Ağlar (SDN), ağın yönetim işlevselliğini mantıksal olarak tekil bir denetleyicide toplayan paket anahtarlamalı ağ paradigmasıdır. SDN'in tüm ağ katmanlarından gelen kontrol bilgilerini kullanarak eski katmanlı mimari sorunlarını ortadan kaldırdığı düşünülmektedir. Bununla birlikte, SDN topolojisinde, kontrol düzlemi ve veri düzlemi ayrılır ve bu da ağ yönetimi için kontrol paketlerini veri yolu kanalından farklı bir kanaldan kontrol eder. Dahası, SDN denetleyicisi, topolojideki en verimli rotayı hesaplamak için eski link-durum hesaplama yaklaşımlarına benzer bir karar ölçütüne sahip olmalıdır. Bunların tümü, SDN'teki bağlantı gecikmesi hesaplamasının, düzgün çalışmasını sağlamak için eski ağdan farklı yeni bakış açıları gerektirdiğini göstermektedir.

Bu tezde, SDN topolojileri için bir bağlantı gecikmesi hesaplama yöntemi öneriyoruz.

Bu amaçla, standart OpenFlow mesajlarını kullanan ve çalışma sırasındaki anahtar sırası gecikmesini hesaplayan bir çerçeve inşa ediyoruz. Bu mimaride, SDN anahtarlarındaki her bir sırayı G / G / 1 olarak modelliyoruz ve OpenFlow sayaçlarıyla giriş trafiğini ölçüyoruz. Daha sonra, paketler arası geliş zamanlarının ortalama ve varyansını elde etmek için sayaç istatistiklerini kullanıyoruz. Ortalama sıra durumunu bulduktan sonra, Little denklemini kullanarak sıraya girme gecikmesine geçiş yapabiliyoruz. Yöntemimizi sıra başına tek akış, sıra başına çoklu akış ve SDN anahtarlama noktasının en hızlı sırasını belirlemek için gecikme bilgilerimizi kullanan bir test uygulaması olan üç durumda gösteriyoruz. Ayrıca, önerilen yöntemin doğruluğunu ve uygulama sınırlamalarını tartışıyoruz.

Anahtar Kelimeler: Servis Kalitesi, Yazılım Tanımlı Ağlar, Gecikme Ölçme, OpenFlow Protokolü

to humanity

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Prof. Dr. Şenan Ece Güran Schmidt, for her guidance throughout my Master's period as well as her positive and tolerant attitude towards me.

Secondly, I would like to thank my mum and dad whose support I always felt in my life. I love them so much and have always wanted to be a son whom they were proud of.

Thirdly, I am thankful to my brother Ege. He is such an energetic guy and helped me to relieve my stress with his motivating words. I hope he will be very successful man in his life like the words he tells for me.

Fourthly, I want to present my sincere thanks to my company ASELSAN A.Ş. and my collegues that tolerate me during my Master's period. I want to thank my managers Ulaş Kılıçarslan and Tuğrul Ertekin for their tolerance towards me; my team leaders Birol Keskin and Ahmet Emrah Demircan for their contributions to improve my technical knowledge as well as their patience to my labor; my collegues Mehmet Gökcan Bayrak, Yasin Çiftçi, Mehmet Bal, Ferhat Bozoklu, Erdem Çerkeşli, Kıvanç Kerse and Ercan Gürsoy for helping me to tackle problems in job when I had to share more time to my Master's than to my company.

Fifthly, I would like to thank my friends Mehmet Çetinkaya, Kerem Eyice, Onur Gülsem. We all performed similar stages of life around the similar ages in close dates. When we look back in future days, I hope these would be the days which we remember with a smile on our faces and would not remember any bad memories. While going through tough times, it is very lucky to have friends who do not understand your words but feel your emotions. I feel very honorable and lucky to spend my Master's period with you guys. I cannot wait to meet with you and drink beers after this hurry finishes.

At this part I want to make mention of my colleague Kerem Parıldar who committed suicide in November 19, 2017. I hope he has already found the peace which he had always been searching for. I wish there had been the opportunity to ask this to him.

Finally, during my Master's period lots of depressing moments were happened in Turkey such as July 15, 2016 Turkish Military Coup Attempt or August 10, 2018 rise of exchange rate of USD to TRY from 5.25 to 7.21 in one day. Even in that moments what whips me up was getting my Master's degree from METU and opening new horizons for my carrier. I want to thank myself and my desire to continue my Master's because being such a patient human knocked me out so much that it might have harmed to my health. I feel very sorry for my friends who had to abandon their Master's and appreciate the ones who were able to finish their Master's in spite of such hard times. Hopefully, after finishing my Master's, I am going to share more valuable time to myself and my beloved ones and do not put anything in front of body wellness and being healthy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	
API	Application Programming Interface
ARP	Address Resolution Protocol
BSP	Board Support Package
Dst	Destination
Eth	Ethernet
FCFS	First Come First Served
HTTP	Hyper Text Transfer Protocol
ICMP	Internet Control Message Protocol
ID	Identification
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
Kbps	Kilobits per seconds
LAN	Local Area Network
MAC	Medium Access Control
Mbps	Megabits per seconds
ms	miliseconds
ns	nanoseconds
NW	Network
OfSoftSwitch13	Open Flow Software Switch 1.3
OS	Operating System
OVS	Open Virtual Switch
OVSDB	Open Virtual Switch Database

QoS	Quality of Service
Proto	Protocol
REST	Representational State Transfer
RYU	RYU SDN Controller Framework
s	seconds
SDN	Software Defined Networking
SNMP	Simple Network Management Protocol
SNR	Signal to Noise Ratio
Src	Source
TCP	Trasnmission Control Protocol
ToS	Type of Service
Tp	Transport
UDP	User Datagram Protocol
UML	Unified Modelling Language
us	microseconds
VLAN	Virtual Local Area Network
WSGI	Web Service Gateway Interface

CHAPTER 1

INTRODUCTION

Compared to circuit-switched networks, packet-switched networks have lots of advantages such as high utilization rates of bandwidth capacity, efficiency or scalability. However, they do not provide delay guarantees. To this end, delay measurement is an important tool to support delay sensitive applications [2]. Previous work on providing delay guarantees in packet-switched networks [3] [4] [5] cannot guarantee an upper bound delay when network is overloaded since nodes in the network only communicate with their counterpart equivalents. SDN [6] has brought a new point of view to packet-switched networks in terms of their management. The information of the whole network stack could be seen by a single entity (SDN controller) which removes constraints of the layered structure. With the SDN technology, a controller can obtain information from different network layers and use this information to facilitate different network services such as computing the most efficient route in the network not to impede the delivery of packets.

SDN can provide delay guarantees [7] [8] by obtaining real-time network statistics and dynamically change the routes of packets if their routes are overloaded with traffic. In order to provide delay guarantee in SDN networks, we model problem in two parts which are monitoring delays of links and efficient routing. In the scope of this thesis, we focus on monitoring delays of links which provides an alternative metric for efficient routing of packets. Note that efficient routing not only depends on delay but depends on other factors such as bandwidth or hop count. The advantages of our proposal are that our computation method uses standardized protocol of SDN called OpenFlow [1]; moreover, our architecture does not need to send redundant link delay computation packets to the datapath channel.

When a packet passes through a node in the network, its delay is modeled in four components which are processing, queuing, transmission and propagation. [9] All of these components except queuing delay are deterministic and assumed not to be changed during transmission of a packet. Queuing delay, however, has a stochastic nature and it is mostly mentioned as average queuing delay. Average queuing delay depends on the state of network with Little's Law [10].

To this end, determining the average queuing delay on the links is the most significant component of providing service quality to the delay sensitive applications in the packet-switched networks. Average queuing delay computation is not straightforward. One has to choose the correct Markovian model in order to compute the average state of a queue. Moreover, the correctness of computation is an important criterion. One can determine so loose bounds that they might result in inefficient route calculation even there exists enough capacity for satisfactory service. Furthermore, in SDN, the control plane and data plane are separated from each other. This requires control packets flow from a separated channel. Hence, while measuring queue state redundant traffic affecting data plane shall not be generated.

In legacy packet-switched networks, queuing delay computation is used in applications such as traffic management or load balancing [11]. However, in SDN, there exists no straightforward protocol similar to OSPF [12] or standard link delay computation technique which enables the controller to obtain states of links from SDN switches. If states of links can be computed correctly in realtime, SDN becomes more agile in terms of traffic management such that it can shape, limit or redirect traffic when there exists bottleneck point in the network. Moreover, besides its technical advantages, efficient usage of resources influences all stakeholders; for example, customers are satisfied thanks to high-quality service whereas service providers are pleased due to reduced maintenance costs.

For the evaluation of queuing delay computation, the first performance metric is accuracy. In order to conclude about the accuracy of delay computation, one can observe the difference between real packet delays and samples that queuing delay method computes. Moreover, SDN has a single point (SDN controller) for managing the network. Such an organization brings a burden on the controller and limit the scalability.

Hence, the second metric is the scalability of the method.

Existing solutions in the literature for queuing delay compuation on SDN controllers mostly concentrate on generating redundant traffic from SDN controller which passes through the path of both data plane and controller plane. However, such solutions contradict with SDN's separation of control and data planes. Moreover, some solutions such as OpenNetMon [13] cannot provide accurate delay compuatation results.

On the one hand, some techniques are restricted to compute the delay of a certain type of packets such as only TCP Flows [14]. Moreover, most type of delay computation techniques require customization on network interfaces or they depend on the certain type of hardware. [15]. However, the solution that we propose for queuing delay computation utilizes standard statistics defined in OpenFlow 1.3 [1] standard. First, we calculate the inter-arrival times between packets coming through a queue of a port by measuring ingress traffic with OpenFlow meters [1]. We created a sampling mechanism to find the mean and variance of the inter-arrival times through the queue. Output rate has predefined mean and variance values thanks to the popular traffic generator tool iPerf [16] that we use. Since we model queues as G/G/1 we utilize Marchal's method [17] to compute the state of the queue from mean and variances of ingress and egress traffic. After finding the queue state, we reach queuing delay from Little's equation [10].

We use software tools to demonstrate our proposed solution. One of these tools is Mininet [18] which enables to construct SDN network for simulation purposes. Moreover, we selected RYU as SDN Controller that utilizes OpenFlow 1.3 [1] interface and provides a WSGI(Web Service Gateway Interface) API with JSON(JavaScript Object Notation) format to extract obtained OpenFlow data such as meter and queue statistics. We realized our solution as a software application that communicates with RYU controller through WSGI interface and extracts queuing delay from meter statistics.

After simulation runs, we collect two types of data which are real-time Wireshark captures that contain time tags when a packet passes from an interface of SDN switch and data from our proposed solution periodically updating the computed average queuing delay. To demonstrate the accuracy of our method, we compare two sets of data. Moreover, limitation is determined by the amount of time passes between controller requests statistics from SDN switch and obtains reply of it. This depends on the bandwidth of control links between the SDN controller and SDN switches. On the other hand, the controller has to satisfy a minimum sampling frequency which depends on the bandwidth of data plane traffic links. In order not to miss samples between packets, the sampling rate has to exceed the maximum packet per unit time rate that can pass from a data plane interface.

While performing simulations we have encountered various issues. Firstly, the simulation tools that we use provide timing sensitivity in microseconds scale. Such timing scale is not enough to demonstrate real networks [19]. Therefore we used 1 Mbps links to prove our method which guarantees a maximum of 1 bit passes in 1 us. Moreover, comparison of Wireshark captures and simulation data required timing synchronization for comparison. We used the time of Operating System which provides synchronous time for two distinct components inherently. Furthermore, in some scenarios, we observed inconsistent outcomes. These inconsistent outcomes could not be solved but they depend on the instantaneous performance of the computer which they operate on.

Results of our implementation are promising and our proposed method is open to improvements. We evaluate our method in three cases which are single flow, multiple flows and a test application Best QoS Selector that utilizes our calculated queuing delay from statistics to decide which queue of a port has the lowest delay. For the single flow case our application measures queuing delay with high accuracy. However, our solution has scalability problems on our simulation platform. This is because of the bandwidth limitation of southbound links between SDN controller and SDN switches. Bandwidths of these links limit the maximum number of queues queried in a minimum sampling period of statistics. Moreover, for multiple flow case, our method computes link delay correctly after an adaptation period. During adaptation period, variance of inter arrival traffic is high and we cannot find true mean value. When high variance component of inter arrival time becomes close to zero and true mean value is found our method becomes stable. Furthermore, we evaluate test application Best Qos Selector from jitter measurements of iPerf [16] data generation application.

Thesis from this point is organized as follows. In chapter 2, we introduce key concepts that are used in the explanation of our solution. Moreover, we define performance metrics which will be used in the evaluation of our method. In chapter 3, we represent literature contributions about queuing delay monitoring in SDN networks. In chapter 4, we present our contribution; that is, our framework for collecting OpenFlow statistics as well as the model that we use for inferring queue delay from statistics. In chapter 5, we first introduce the emulation tools that we use for demonstrating our solution. Then, we present the results of experiments which are evaluated in terms of evaluation criteria. In chapter 6 we conclude our approach and mention the future work for improving this study.

CHAPTER 2

BACKGROUND

This chapter explains the necessary concepts, glossary terms, problems and performance metrics which are used in further chapters. In the Software Defined Networking (SDN) section, we explain the concept of SDN and elaborate differences from the traditional layered packet-switched networks. In the OpenFlow section, we explain the OpenFlow protocol. It is a defacto standard protocol used in SDN to exchange network information between SDN entities. Since OpenFlow consists of abundant message types, we only explain messages which we used in testing our architecture. In Quality of Service section, we explain a glossary term Quality of Service (QoS). In Delay Modelling in Computer Networks section, we explain four components of node delay in a computer network. Before finishing this chapter we indicate problems encountered in SDN delay measurement. Finally, we define performance metrics which are used in the evaluation of our framework architecture.

2.1 Software Defined Networking (SDN)

In traditional computer networks, there exists a concept called five-layered stack design [9] where each layer has its own functionality. The most top layer is layer five or called the application layer, which produces data exchanged between hosts. Layer four is the transport layer. It provides a unique interface to applications in end hosts. The responsibilities of this layer extend to reordering packets at the receiving end for delivering application layer data correct order or controlling the flow of application data not to bottleneck communication. Most popular transport layer protocols are User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). Layer three is the network layer. The purpose of the network layer is to identify network devices from their unique network numbers and route packets properly. Internet Protocol version 4 (IPv4) is commonly agreed on protocol for exchanging data in the network layer. Routers are network devices which provide service in the network layer. Layer two is the data link layer. This layer provides service for connecting two machines standing in the same medium. A popular protocol for this layer is Ethernet. Ethernet switches manage the connection between two nodes in the same medium. Finally, the lowest layer is the physical layer. It is responsible for the transportation of packets in the medium.

In packet-switched networks, multiple packets are required to be forwarded through the same port of router or ethernet switch simultaneously. If input traffic rate exceeds output line rate then packets need to be buffered, or queued, to get service when the link is available. Organization of queue determines the performance of the network towards services. For example, First Come First Serve (FCFS) does not have any performance differentiation; however, priority queuing forwards an important packet prior to other waiting packets in a node. Therefore, prioritized packets are not influenced by regular traffic which increases the performance of a network in terms of given service to important applications.

Each layer in traditional networks is responsible for giving service to its upper layers and takes service from lower layers. However, layer in a node can only communicate with its corresponding layer of a node in the network. Because of this fact, in traditional computer networks there exist two types of packets which are datapath and control packets. Datapath packets encapsulate valuable information coming from the most top layer which is converted mostly to audio, video or other valuable signals at the end hosts. On the other hand, control packets are transferred between nodes to exchange their states and maintain network operation. Since each layer are only allowed to interact with its equivalent, each layer has a separate control mechanism. After receiving states of other nodes, a network element such as ethernet switch or router decides how to take action on packets on its own. Therefore the logic of network is dispersed to every node on the network.

Software Defined Networking (SDN) [6] concept has emerged in order to unify the dispersed control logic of the network. It allows control packets transmitted from a different channel rather than transmitting them from the same channel with datapath packets. It takes control plane from network elements and puts control plane to a logically single controller which undertakes control functionality of network by its own. In SDN there exists no different hardware for network layers. All hardware of traditional networks such as ethernet switch and routers are put in a single SDN switch. SDN Controller can communicate with SDN switches from its southbound API; on the other hand, network admins or user applications can intervene to control of the network from northbound API of SDN controller. Note that a controller in SDN is logically single which implies that there might be more than one physical device handling the controller functionalities. However, they have to appear logically as single to counterparts they serve. In Figure 2.1 an example of SDN model is shown.

In the next section, we are going to elaborate the standard (OpenFlow) which not only defines Southbound API communication between SDN controller and SDN switch but also processing pipeline of SDN switch for packet forwarding.

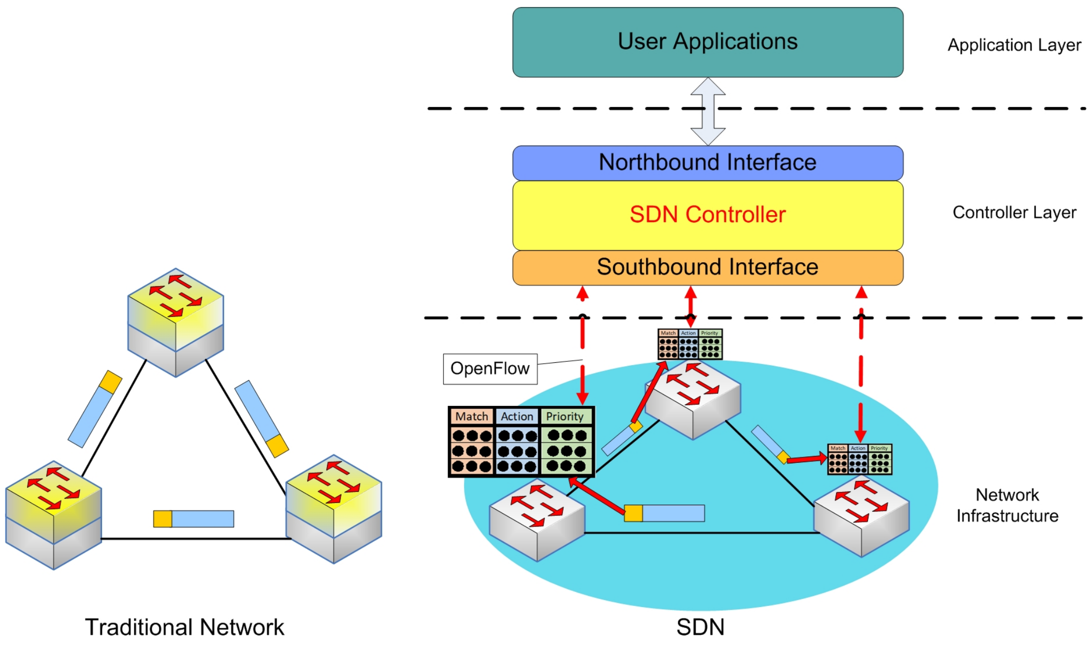


Figure 2.1: Example of SDN Architecture

2.2 OpenFlow

In order to have a common southbound communication in SDN, OpenFlow [20] was introduced by ONF [6]. OpenFlow is a standard in which both OpenFlow pipeline processing and messaging protocol are explained. Before explaining OpenFlow, let us mention glossary terms used in the definition of it. A flow is glossary term which is used in order to identify processing rules of SDN switch. A flow table is a collection of flows which SDN switch looks at when processing a packet. An SDN controller controls SDN switches by modifying flow tables of them. Each flow consists of three parts which are priority, match and actions. A visual model of a flow is given in Figure 2.2. Priority determines the place of a flow in the flow table. The highest priority indicates that it is the first flow to be checked when processing a packet for that table. Furthermore, match determines which sections of packet headers to check while processing and their predetermined values. An example is given in Figure 2.3. One can specify a match as ethernet type equals 2048 and network protocol equals 17. This indicates that when processing a packet for this flow the searched sections of a packet will be ethernet type and network protocol. Their values have to match with 2048 and 17 respectively. Multiple match definitions for a flow increase precision which provides deeper flow differentiation. Moreover, action determines the behavior of SDN Switch when a packet matches with that flow. For example, the packet could be forwarded to the next flow tables if action part of a flow consists go to table type of action. Multiple action definition for a flow is allowed in OpenFlow. Also, actions are allowed to be forwarded between flow tables since they are implemented after processing finishes. Matching sections for a packet and action types that SDN switch can implement are given more detail in standards OpenFlow 1.0 [20] and Openflow 1.3 [1].

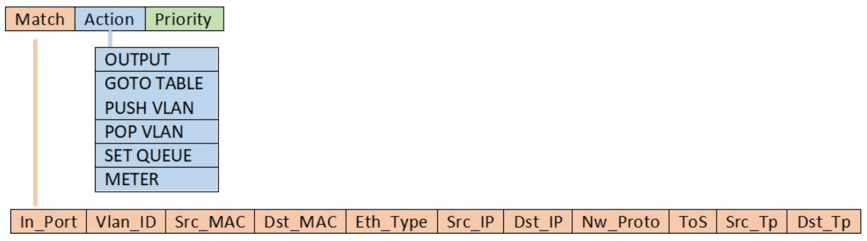


Figure 2.2: Flow Table Example

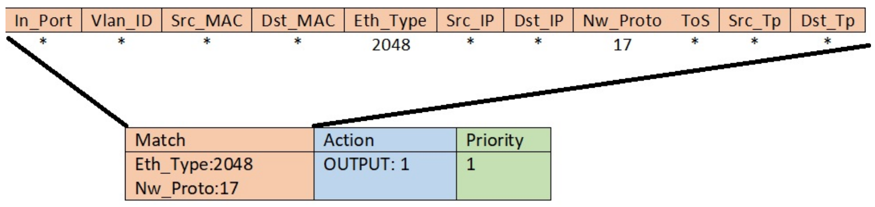


Figure 2.3: Match of a Flow Example

OpenFlow pipeline processing defines how to process packets in SDN switches. In Figure 2.4 pipeline processing mechanism is shown. When a packet first enters OpenFlow processing pipeline, an action set is automatically constructed for it. Processing in OpenFlow pipeline starts with the table which has the lowest identification (ID 0). During processing, a packet is passed from low numbered to high numbered flow tables and its action set is filled with actions of matching flows. Forwarding a packet to a flow table which has a lower table ID than current table ID is not allowed. Finally, if there exist no tables left to be forwarded, the action set is executed for that packet. After a packet enters OpenFlow processing pipeline from table 0, if it does not match with any flows then it is dropped. On the other hand, in SDN switch, one has to define a flow with send to controller action for packets needed to be forwarded to SDN Controller. This creates a `packet in` message sent to SDN Controller. Depending on the implementation of SDN switch, `packet in` might consist either whole packet or a small description defining header fields of the packet as well as buffer id where that packet is hosted in SDN Switch. If the controller wants an action applied to this packet, such as forwarded from an output port or dropped, it informs SDN switch with a message having the same descriptions values with the `packet in` message. An SDN controller can create, delete or modify a flow entry of SDN switch. Moreover, in OpenFlow, it is allowed to define idle time and hard time for flows to be removed automatically without the intervention of SDN Controller. If a flow is removed then `flow removed` message, sent by SDN switch informs the SDN Controller.

The messaging protocol of OpenFlow provides a common language by which SDN switches and SDN controller can exchange information. In the scope of this thesis, we are interested in Multipart type OpenFlow messages whose header sections related to OpenFlow are summarized in Figure 2.5. OpenFlow header is common for all type of messages. Then comes Multipart header. An SDN controller can query statis-

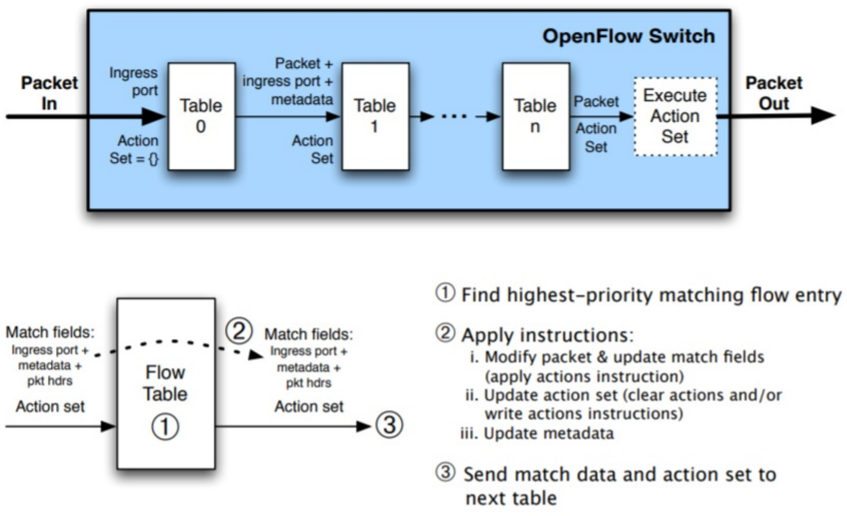

OpenFlow Switch
Packet In
Ingress port
Action Set = {}
Table 0
Packet + ingress port + metadata
Action Set
Table 1
Table n
Packet
Action Set
Execute Action Set
Packet Out
Match fields: Ingress port + metadata + pkt hdrs
Action set
Flow Table
①
②
Match fields: Ingress port + metadata + pkt hdrs
Action set
③
① Find highest-priority matching flow entry
② Apply instructions:
i. Modify packet & update match fields (apply actions instruction)
ii. Update action set (clear actions and/or write actions instructions)
iii. Update metadata
③ Send match data and action set to next table

Figure 2.4: OpenFlow Pipeline Processing Mechanism [1]

tics by sending `multipart request` message; on the other hand, SDN Switch replies this request message with a `multipart reply` message. Multipart type messages further have a header to indicate their payload type which could be meter, queue, table or port statistics. Meter is an ingress traffic counting mechanism introduced in OpenFlow 1.3 [1]. It counts packets which are directed to meter table (itself) by the OpenFlow actions in terms of bytes and packet counts. In an OpenFlow switch, firstly, a meter is defined via OpenFlow protocol then, a flow is bound to the meter by defining `METER` action in action list of a flow. All type of statistics of OpenFlow including meter could be obtained with Multipart request/reply messaging mechanism between SDN Controller and SDN Switch.

All OpenFlow messages start with an OpenFlow header given in Figure 2.6. In this header, `version` represents which version of OpenFlow is used. Moreover, `type` represents the content of rest of message. `length` indicates total size of OpenFlow message including OpenFlow header. Finally, `xid` represents the transaction number. This number is important since when switch constructs a reply message for request, it indicates which query number is it responding with the transaction number.

Multipart type of messages were introduced in OpenFlow 1.3 standard [1] instead of statistics type in OpenFlow 1.0 standard [20]. Headers of `multipart request` and `multipart response` are given in figures 2.7 and 2.8 respectively. In these

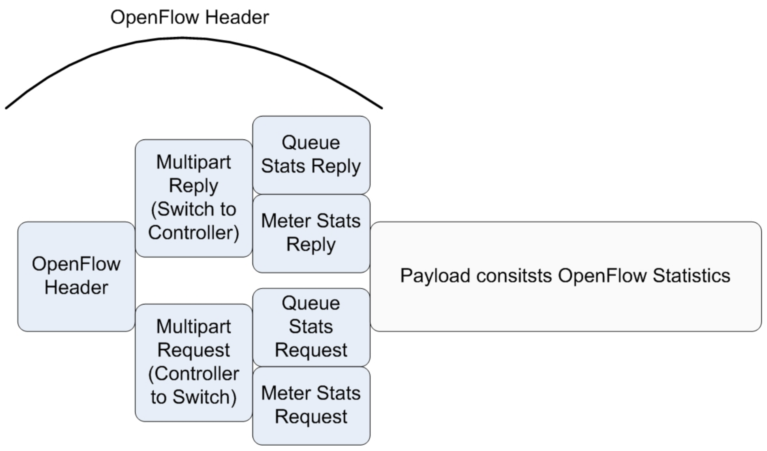
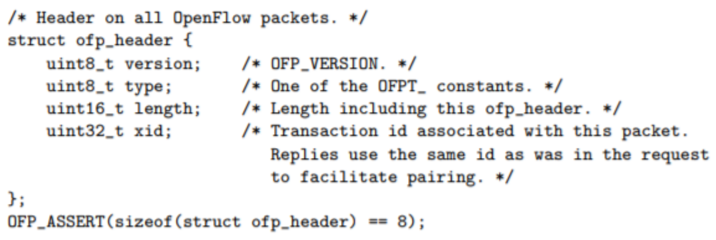

OpenFlow Header
OpenFlow Header
Multipart Reply (Switch to Controller)
Multipart Request (Controller to Switch)
Queue Stats Reply
Meter Stats Reply
Queue Stats Request
Meter Stats Request
Payload consitsts OpenFlow Statistics

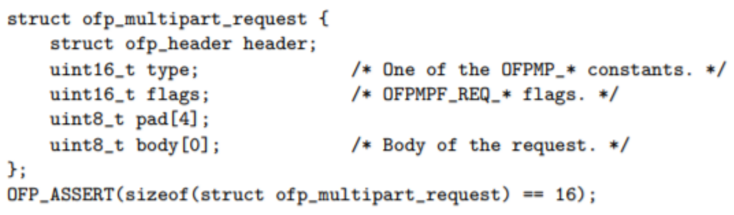
Figure 2.5: OpenFlow Statistics Packet



```
/* Header on all OpenFlow packets. */
struct ofp_header {
    uint8_t version;    /* OFP_VERSION. */
    uint8_t type;       /* One of the OFPT_ constants. */
    uint16_t length;    /* Length including this ofp_header. */
    uint32_t xid;       /* Transaction id associated with this packet.
                           Replies use the same id as was in the request
                           to facilitate pairing. */
};
OFP_ASSERT(sizeof(struct ofp_header) == 8);
```

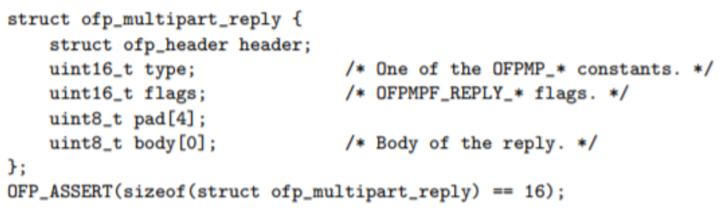
Figure 2.6: OpenFlow Header [1]

headers, type section indicates which multipart type message is sent. The types are given in figure 2.9. Note that `type` number for queue statistics is 5 whereas it is 9 for meter statistics. Next field is `flags`. In the standard there exists only one flag which is send more. Controller can query more than one `type` of statistics with same `xid` of Openflow header if it sets send more flag. By the same mechanism, switch can respond with same OpenFlow `xid` by setting send more flag in its `multipart reply` headers.



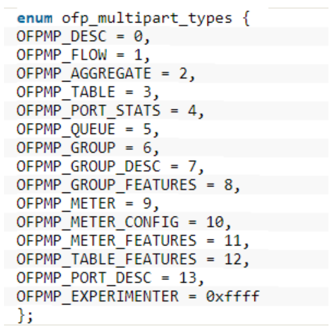
```
struct ofp_multipart_request {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPMP_* constants. */
    uint16_t flags;             /* OFPMPF_REQ_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the request. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_request) == 16);
```

Figure 2.7: Multipart Request Header Structure [1]



```
struct ofp_multipart_reply {
    struct ofp_header header;
    uint16_t type;              /* One of the OFPMP_* constants. */
    uint16_t flags;             /* OFPMPF_REPLY_* flags. */
    uint8_t pad[4];
    uint8_t body[0];            /* Body of the reply. */
};
OFP_ASSERT(sizeof(struct ofp_multipart_reply) == 16);
```

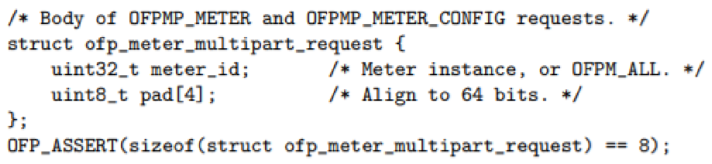
Figure 2.8: Multipart Reply Header Structure [1]



```
enum ofp_multipart_types {
OFPMP_DESC = 0,
OFPMP_FLOW = 1,
OFPMP_AGGREGATE = 2,
OFPMP_TABLE = 3,
OFPMP_PORT_STATS = 4,
OFPMP_QUEUE = 5,
OFPMP_GROUP = 6,
OFPMP_GROUP_DESC = 7,
OFPMP_GROUP_FEATURES = 8,
OFPMP_METER = 9,
OFPMP_METER_CONFIG = 10,
OFPMP_METER_FEATURES = 11,
OFPMP_TABLE_FEATURES = 12,
OFPMP_PORT_DESC = 13,
OFPMP_EXPERIMENTER = 0xffff
};
```

Figure 2.9: Types of Multipart Messages [1]

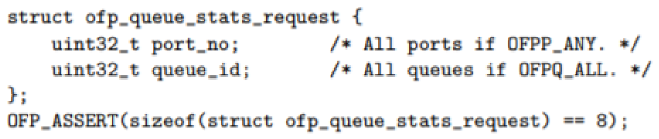
Multipart request messages that we are interested in are Meter and Queue messages. For querying meter statistics meter type multipart request message is shown in figure 2.10. The only parameter which we need to construct this message is `meter id`. Moreover, we ask queue statistics with queue type multipart request message. In figure 2.11 body of queue type multipart request is shown. In order to ask the statistics of a queue, one has to provide `port id` and `queue id` in the body of the message.



```
/* Body of OFPMP_METER and OFPMP_METER_CONFIG requests. */
struct ofp_meter_multipart_request {
    uint32_t meter_id;          /* Meter instance, or OFPM_ALL. */
    uint8_t pad[4];             /* Align to 64 bits. */
};
OFP_ASSERT(sizeof(struct ofp_meter_multipart_request) == 8);
```

Figure 2.10: Meter Statistics Request Structure [1]

On the other hand, messages that an SDN switch responds to `multipart request` are `multipart reply`. Type in the header of reply message indicates the body of reply to the controller. Types of replies are the same with requests and seen in Figure 2.9. Body of the queue statistics reply could be seen in Figure 2.12. In this

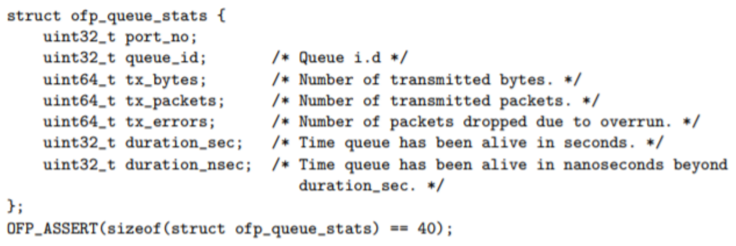


```
struct ofp_queue_stats_request {
    uint32_t port_no;          /* All ports if OFPP_ANY. */
    uint32_t queue_id;         /* All queues if OFPQ_ALL. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats_request) == 8);
```

Figure 2.11: Queue Statistics Request Structure [1]

Figure, available parameters for queue statistics are given. `duration_sec` and `duration_nanosec` is used to calculate total time passed since queue was first introduced to give service whose formula is `duration_sec` + `duration_nanosec`$*10^{-9}$. Furthermore, Meter statistics reply body is given in Figure 2.13. In meter statistics, one can infer how many different flows, bytes and packets were associated with meter since it was first introduced to the system. Duration mechanism is same with queue statistics. One can create a meter with a `metermod` message in OpenFlow 1.3 [1]. However, queue creation is out of the scope of OpenFlow [20] [1]. Queues are created via protocols such as Ofconfig [6] or vendor-specific mechanisms such as serial consoles or SNMP.

In the next section, we focus on Northbound communication and SDN Controller and RYU SDN Controller which we use in our testing setup.

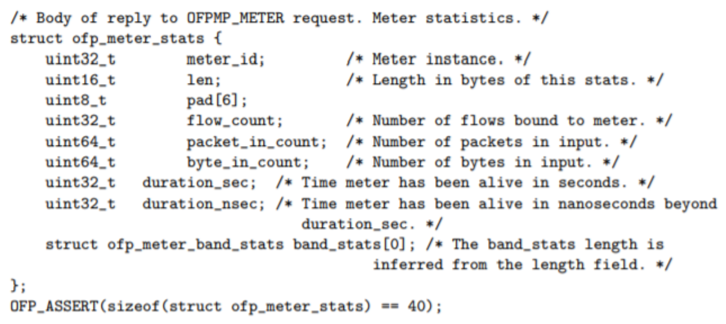


```
struct ofp_queue_stats {
    uint32_t port_no;
    uint32_t queue_id;         /* Queue i.d */
    uint64_t tx_bytes;         /* Number of transmitted bytes. */
    uint64_t tx_packets;       /* Number of transmitted packets. */
    uint64_t tx_errors;        /* Number of packets dropped due to overrun. */
    uint32_t duration_sec;     /* Time queue has been alive in seconds. */
    uint32_t duration_nsec;    /* Time queue has been alive in nanoseconds beyond
                                  duration_sec. */
};
OFP_ASSERT(sizeof(struct ofp_queue_stats) == 40);
```

Figure 2.12: Queue Statistics Response Structure [1]

2.3 SDN Controller

In this section, we explain the SDN Controller and its Northbound operations by giving examples from RYU SDN Framework [21]. In SDN topology, SDN Controller provides an abstraction between SDN switches and external entities. Therefore, in



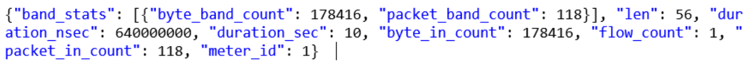
```
/* Body of reply to OFPMP_METER request. Meter statistics. */
struct ofp_meter_stats {
    uint32_t        meter_id;         /* Meter instance. */
    uint16_t        len;              /* Length in bytes of this stats. */
    uint8_t         pad[6];
    uint32_t        flow_count;       /* Number of flows bound to meter. */
    uint64_t        packet_in_count;  /* Number of packets in input. */
    uint64_t        byte_in_count;    /* Number of bytes in input. */
    uint32_t   duration_sec;  /* Time meter has been alive in seconds. */
    uint32_t   duration_nsec; /* Time meter has been alive in nanoseconds beyond
                                 duration_sec. */
    struct ofp_meter_band_stats band_stats[0]; /* The band_stats length is
                                        inferred from the length field. */
};
OFP_ASSERT(sizeof(struct ofp_meter_stats) == 40);
```

Figure 2.13: Meter Statistics Response Structure [1]

order to send message to SDN switches, an external entity has to communicate them over the SDN controller. For this purpose SDN controllers provide two different interfaces. First one is a northbound network interface. This interface does not have a common protocol and it depends on the SDN controller. From this interface SDN controller and external entity exchange information without having a dependency relation. On the other hand, the second controller interface is module callback interface. For this interface, a module integrated to the SDN controller is programmed and it cannot exist without the SDN controller. Module subscribes to events of the controller and it is notified when subscribed event occurs. Event types depend on the controller and they are not allowed to be masked or closed by the external entity during runtime.

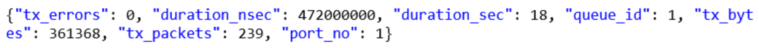
Our SDN controller RYU provides common Web Service Gateway Interface (WSGI) for its components that want to exchange data from Northbound interface with external entities. From this northbound API, OpenFlow messages are formatted in the Javascript Object Notation (JSON) format. The format of meter statistics and queue statistics in the JSON format is given in Figures 2.14 and 2.15 respectively. Note that this format has all required sections parallel with OpenFlow Multipart Reply Statistics. Moreover, RYU has built-in function callback modules for handling events. In our simulations, we use two built-in modules which are OpenFlow module and Routing module. OpenFlow module works as an adapter module between an external entity and SDN switches. When an external entity wants to create an OpenFlow message, it communicates with the OpenFlow module of SDN RYU Controller. Then, OpenFlow module of RYU converts message of the external entity to OpenFlow message and sends it to the SDN Switch. OpenFlow asynchronous events are not subscribed in this module; therefore, external entity is not notified when asynchronous of OpenFlow events occur (asynchronous events will be explained in SDN Topology Management and Delay Measurement Issues section). Moreover, the routing module operates an SDN switch as a legacy router whose router functionality is conducted with static routes. For this operation, IP addresses, network masks of ports as well as static routes of the SDN switch have to be provided to routing module.

In the next section, we elaborate a commonly used term in packet-switched networks for indicating the performance of the network.



```
{"band_stats": [{"byte_band_count": 178416, "packet_band_count": 118}], "len": 56, "dur
ation_nsec": 640000000, "duration_sec": 10, "byte_in_count": 178416, "flow_count": 1, "
packet_in_count": 118, "meter_id": 1}
```

Figure 2.14: RYU Northbound JSON Meter Statistics



```
{"tx_errors": 0, "duration_nsec": 472000000, "duration_sec": 18, "queue_id": 1, "tx_byt
es": 361368, "tx_packets": 239, "port_no": 1}
```

Figure 2.15: RYU Northbound JSON Queue Statistics

2.4 Quality of Service (QoS)

Quality of Service (QoS) in computer networks is used for the notion that is "serving unlimited demands with limited resources". Limited resources could be processing power, bandwidth, system capacity whereas unlimited demands might be low delay, high throughput, low cost and high efficiency. Since full satisfaction rate is impossible, the network has to prioritize some packets according to needs of it. For example, services such as telephony or video need a delay-sensitive link quality where companies are willing to pay more money for a qualitative videoconferencing experience that saves time and money for them [22]. Therefore, if the delay is an important parameter then priority scheduling of delay-sensitive packets in the network is more important than bandwidth assigned to them. On the other hand, if bandwidth is critical then how much packet is transferred per amount of time is a performance criterion. Community puts emphasis on works [23] [24] [25] which increase end to end delay performance of a packet. It is the amount of time that a packet spends on the network before reaching its destination. Furthermore, one of the issues that packet-switched networks face is that they cannot guarantee packets go out from a certain port in a specific time [26]. Also, QoS environment of network changes dynamically. Static approaches fail when providing service for such environment [27]. Therefore dynamically measuring the delay of packets in the network has valuable input for QoS management of delay critical applications. Delay measurement is a complex issue since all layers in the network have their own control mechanism. On the other hand, SDN enables to improve legacy QoS approaches since control plane information could be gathered from whole layers of network in SDN. By this way, SDN networks can both provide prioritized scheduling mechanism as well as high bandwidth utilization which creates efficient networks.

In the next section, we are going to define the model of delay in legacy packet-switched networks. We also use this model as reference in our SDN topology.

2.5 Delay Modeling in Computer Networks

While packets pass from a node, they have entrance and leaving times. Time difference between the first bit of packet is introduced to a node and last bit leaves from it is node delay of a packet. This node delay is modeled in four components. [9] The first component is the processing delay. Processing delay is the amount of time passes until that packet is switched to its egress port. [9] This type of delay is very small thanks to hardware; therefore often neglected when calculating total delay. Moreover, propagation delay is the amount of time needed for an electrical wave traveling in a media reaching its destination. [9] It is related to the distance between transmitting and receiving ends also electromagnetic properties of the medium in which electrical wave travels. Propagation delay is assumed to be constant during the transmission period. Furthermore, the transmission delay is the time necessary for a node to encode all bits of a packet. [9] This delay depends on the length of the packets and bandwidth of the interface. Lastly, the fourth delay which packet is exposed in a node is queueing delay. If two or more packets are switched to the same egress port simultaneously then one of them has to wait in a queue before it gets service. Amount of time a packet waits in a queue is the queuing delay. [9]

Additional to node delay in SDN, packets which are forwarded to and taken back from SDN Controller add additional burden on latency. Since multiple packets are forwarded to one single node, this will add additional controller processing queuing delay for packets of which controller is involved in the processing chain. [28] This model becomes significant when traffic between switch and controller is intense. However, controller delay could be avoided if looser match fields are defined for flows of background traffic.

Since queuing is modeled as a stochastic process that is, it depends on probabilistic arrival and departure rates, when referring to queueing delay one has to know it is the average queueing delay. Average queuing delay depends on average system state that is defined by the number of packets in the system by Little's Law [10] given in equation 21. In this equation L is defined as average system state, W is given as average delay and λ is the mean packet arrival rate to the queue.

$$L = \lambda \cdot W \tag{21}$$

Defining a queueing system has a special notation called Kendall's notation [29]. An example to Kendall's notation is A/B/s. In this notation, "A" defines the shape of arrival, "B" defines the shape of departure, "s" defines the number of serving capability in the queuing system. Some letters have special meaning indicating distribution such as "M" indicates memoryless distribution and "G" or "GI" indicates general distribution. Average system state equation is extracted from shapes of arrival and departure processes. Average system state equation for G/G/1 queue has an approximated equation given in 22 proposed by Marchal [17]. For this equation λ is average arrival rate whose unit is packets per seconds (or just s^{-1}), $1/\lambda$ is the average interarrival time between packets whose unit is seconds s and σ_a^2 is the variance of interarrival time. Furthermore, similarly μ is defined as average service rate, $1/\mu$ is the average departure time between packets and σ_s^2 is the variance of average service rate. ρ is the utilization of the queueing system which is defined in equation 23. C_a^2 is defined in equation 24 as well as C_s^2 is defined in equation 25. By using L_q one can extract wait time in the queue W_q by Little's equation [10] 21.

$$L_q \approx \frac{\rho^2(1+C_s^2)(C_a^2+\rho^2 C_s^2)}{2(1-\rho)(1+\rho^2 C_s^2)} \tag{22}$$

$$\rho = \frac{\lambda}{\mu} \tag{23}$$

$$C_a^2 = \frac{\sigma_a^2}{(1/\lambda)^2} \tag{24}$$

$$C_s^2 = \frac{\sigma_s^2}{(1/\mu)^2} \tag{25}$$

2.6 SDN Topology Management and Delay Measurement Issues

In traditional networks, in order to provide low delay as a service quality, network nodes have to schedule packets of delay-sensitive service with a prioritized fashion. Such architecture requires custom implementation on switching hardware. Moreover, in traditional networks, there exist messaging protocols such as RIP [30] or OSPF [12] in the control plane to build network topology. These protocols share their own messages which indicate the state of links in the network. With these messages, a network node could construct a topology view of the network by its side and calculate the shortest path or the most efficient path to the destination. However, in SDN, the control plane is separated from the data plane. This separation brings some problems to SDN in terms of topology management.

Firstly, in OpenFlow, there exists a limited set of asynchronous messages between the controller and the switches. In OpenFlow protocol, four types of asynchronous messages initiated by switches are defined. These are `packet in`, `flow removal`, `port status` and `error` messages. `Packet in` is a type of message which indicates that a packet, matching with a flow that has an action type send to the controller is received. The controller can respond to this packet in order to drop or allow it from an out port. `Flow removal` indicates that a flow has been removed and it is no more accepted as a valid flow. `Port status` indicates a change in the status of a port. It might be added, shutdown or modified by the user or because of a problem in the network. `Error` indicates there occurred a problem in the processing of received controller packets. [1] These messages have limited span on network situations which happens simultaneously most of the time. For example, a network load change on some port could not be informed to the controller. Thus, the SDN controller has to poll every node in the network and determine whether the link state is enough to handle capacity. Such behavior brings a problem that dynamic behavior of legacy networks in terms of rerouting could not be achieved standardly in SDN unless there exists a network monitoring module in an SDN controller.

Secondly, in OpenFlow, there exist no standard topology building mechanisms. When SDN switches contact with SDN controller for the first time, they send `Hello` packets and receive `Hello` message from SDN controller. After exchanging symmetrical Hello packets, SDN controller can send `features request` message to learn what features are implemented on SDN switch such as how many ports it has or how many queues created for that switch. Neither `features reply` nor `Hello` messages coming from switch contain how SDN switches are organized in a network. Which port of SDN switches is connected to where is understood from the behavior. For example, SDN switch floods ARP query packets which have an unknown destination. SDN controller receives the same packet again from adjacent SDN switch with a `packet in` message. For second SDN switch controller could understand from which port SDN Switch has received the packet (because it is defined in `packet in` message of OpenFlow); however, first SDN switch could not indicate controller from which port it outputs the packet. Although there exist link discovery and topology viewer mechanism in SDN controllers, most of them either do not build a complete view that is which IP address or MAC address is behind which port or does not provide up to date view of topology. Therefore, most of the time routing is done by flooding the packet and waiting for a return link to establish. In order not to be affected from controller processing delay, some SDN controller inserts a flow automatically after it learns a certain MAC address is behind of its port. This leads to static routing to destination in practical since alternative paths could not be discovered for that flow.

In the next section, we define performance metrics for evaluating our proposed architecture.

2.7 Constraints and Performance Metrics in SDN Network Monitoring

The first constraint is the average delay definition of the system. Since queuing delay is a stochastic process, one cannot define an instantaneous delay for packets. This average delay definition has to represent the average performance of the system which is the concern of controller in terms of performance metrics. Furthermore, since the topology of SDN is controlled by a single controller and controllers initiate link-state computation by polling, delay computation techniques would have an upper bound in terms of the number of link-state that it can compute. Also, in order not to miss any sample one has to define a minimum polling rate for a switch.

Until this point, we indicated that a delay computation technique has to be accurate, scalable.

Next is the definition of performance metrics:

Relative Percentage Correctness:

We define the relative correctness of the delay as in equation 26. In this equation, m is the sample space that holds computed delay and r is the sample space that holds average real-time delay measurements. Since computed delay and real-time delay measurements are not obtained at the same time, computed delay data have to be mapped to closest real-time delay measurements. That is the reason why t_1 and t_2 values are different. Moreover, t_1 has to be greater (later in time) than t_2. Mean of relative correctness function has to be close to 100% to have accurate results.

$$a(t1) = \begin{cases} (1 - \frac{|m(t_1)-r(t_2)|}{m(t_1)}) * 100 & \text{, if } 0 \leq a(t1) \leq 100 \\ 0 & \text{, otherwise} \end{cases} \tag{26}$$

Scalability:

We define scalability s in equation 27 of the network monitoring system as the maximum number of links that a controller can monitor. For our system explained in the next chapters this is limited by meter id size of OpenFlow Protocol [1] and the maximum number of queries n that a controller can send by polling. Query number depends on polling period of switch $t\_sampling$, size of packets exchanged between for query and response $l_{query} + l_{response}$ as well as link rate C between controller and switch. This equation is defined in 28. Maximum possible polling period of a switch, given in equation 29, depends on the minimum length of a packet that can pass from datapath l_{min} and bandwidth of datapath B.

$$s = min(meter\_id, n) \tag{27}$$

$$n = \frac{t\_sampling}{(\frac{(l_{query}+l_{response})}{C})} \tag{28}$$

$$t\_sampling < t\_max_{switch} = \frac{l_{min}}{B} \tag{29}$$

CHAPTER 3

RELATED WORK

In this chapter, we present selected delay measurement solutions in literature for providing QoS in SDN. Some delay measurement methods are declared implicitly in papers while others have an explicit definition of their used techniques. The reason behind the implicit declaration of measurement techniques in papers is that emphasis is not put on delay measurement. In those papers, authors use delay measurement as an input to QoS evaluation. Delay measurement is an important metric for service differentiation in SDN. At the end of this section, we conclude the results of literature papers in a table to provide a summary of related works.

SDN allows installing QoS policies in the form of applications that run on the SDN controller. In [31] authors represent their QoS architecture, OpenQos which runs the shortest path algorithm with the real-time statistics that can be collected from the SDN switches. Their contribution has two main components which are route management and route calculation. Route management module checks whether there exists congestion by utilizing Feature Request message defined in OpenFlow 1.0 standard [20]. According to congestion, the cost of each link is determined. Route calculation module collects real-time cost of links and applies constrained shortest path algorithm to possible routes of flow after a packet arrives in the controller. The paper presents test results collected from a real network which consists of three SDN switches and a controller. Nodes are connected with a triangular topology to provide at least two different path options for forwarding. The main performance metric is the SNR values of flows. However, the SNR values do not provide how many bits are transferred during communication instead they show media capacity in order to transfer how much bits in one symbol. There is no information about the number of bits transferred and the average data rate during the test. Moreover, authors should clarify how congestion data is obtained from the controller. Inference of congestion information is given so implicitly that one interrogates the credibility of this contribution.

The work presented in [32] brings a complete top-down approach to service differentiation in SDN. The authors express their motivation as making flow centric, dynamic, scalable and easily deployable QoS structure which interacts both with network administrators and network elements such as controllers and SDN switches. They define their structure in three planes and each plane is further divided into smaller parts having smaller duties while the whole system is in service.

The first plane in this model is the data plane. Data plane constituted from switches and their interfaces communicating with SDN controllers which are named as southbound APIs. The most important southbound API is OpenFlow which mainly led to the concept of SDN. Moreover, there exists other southbound APIs to enrich the feature set of network nodes. The second plane is the control plane. Besides having a complete view of network resources, the controller has four extra modules in this topology. They are admission control, routing, device tracker, statistics collector and rule database. Admission control makes the decision whether to accept new flow to a system by checking resources from southbound API. Routing module constantly tracks routes of flows defined in the rule database. It utilizes device tracker and statistics collector modules. Device tracker simply tracks switches in the network. Statistics collector collects the information available from network nodes. Finally, the rule database keeps the rules which will be enforced on nodes of the interconnected system.

The main contribution of this QoS scheme is the third plane which is the management plane. First two planes are almost standard in SDN concept; however, management plane provides a semi-automatic control layer between network administrators and SDN controller. Semi-automatic means that in some situations program directs the problem to human but it also has an automatic block which can handle variations in the network. This block consists of two modules and they are further divided in terms of their duties. Note that management plane elements interact with four control plane components to have their job done and a special northbound API is implemented to establish connectivity between two planes.

The first module of the management plane is policy validator. The main response of it is to ensure that everything is normal in the network. Policy validator is further divided into three subparts which are traffic monitor, policy checker and event handler. Traffic monitor observes the network and feeds information to policy checker. Policy checker then compares these values with database and calls event handler if there occurs a violation. Event handler then evaluates message coming from policy checker. It further directs the request to second management plane module which is policy enforcer. Policy enforcer module has four subparts. While the topology manager keeps track of the network devices, the resource manager checks network resources. Policy adaptation determines the type of violation and resource provisioning takes action according to violation type. This topology is tested in terms of link failure and flow admission. In order to test it, designers have built OVS to five different computers. They are interconnected with GRE tunnels and Floodlight controller is chosen as SDN controller. Overall I think this architecture is satisfying but complicated to implement however such a complete approach might be preferred for those who construct their own system.

Another perspective to service differentiation in SDN mechanism presented in [33]. In this model, authors extend traditional controller datapath model and bring a concept which exchanges QoS information between controllers. Each controller manages one domain whose responsibilities are topology management, resource management and route calculation. Also, authors claim that a controller should support flow management, queue management, call admission and traffic policy enforcement mechanisms for increasing the functionality. There exists quality of server levels in each domain and controllers calculate paths iteratively starting from the most prior QoS level. Intra domain information is gathered from modules which interact with southbound API. On the other hand, inter-domain statistics are shared between controllers and this is the point where distributed architectures for QoS provisioning is introduced. The first controller to controller architecture is named as fully distributed model. In this model, controllers share information between them directly. The second architecture is hierarchically distributed model in which controllers have hierarchy and they only share information with their hierarchical super controllers. This architecture resembles the layered architecture of traditional networks in which one layer needs to utilize sub-layers in order to share information; they do not interact directly in the physical sense but in the logical sense. In order to test these two architectures, creators have build three algorithms. The first algorithm runs in each controller of fully distributed architecture while the other two runs in the super controller of hierarchically distributed architecture and domain controllers of the same architecture. They compared two models in terms of distance and amount of traffic they served with traditional BGP directed services. Fully distributed model wins the comparison of three models while hierarchical model gives better result than traditional BGP routing. From my personal point of view, the authors conducted an outstanding job in this paper. However, this approach needed to be standardized by the community or adopted generally by vendors to get more respect. In the community, there exists no standard controller to controller protocol so this paper gives a possible requirement set to designers of such protocol in terms of QoS.

Collaborators of paper [34] have come up with an idea which is named as HiQos. It mainly consists of two modules named as differentiated services module and multipath routing module. Differentiated services module identifies flows according to their match parameters. Each type of flow is then assigned to a queue. On the other hand, multipath routing module identifies possible paths for flows and chooses the best option by calculating the sum of weights for each link in a route which is determined by a network monitoring thread that observes bandwidth utilization periodically. Moreover, each flow inserted by multipath routing module has five seconds idle time meaning that their rules vanish if there exists no packet in the network for more than five seconds. This enables dynamicity for flows staying idle for more than five seconds. Tests are conducted with a topology consisting of two servers eleven clients and five switches. When the paper was written, authors could not find models to compare their architecture and introduced two extra contributions for comparison purposes. First of these architectures is LiQoS. This architecture always selects the shortest path to the destination as well as it does not differentiate services by using different queues. The second option is MiQos. It chooses the shortest path to the destination; however, it utilizes queues for service differentiation. Comparison of three architecture in terms of the response time of servers, throughput and recovery in case of path loss resulted in the success of HiQos. The most appreciated point about this paper is that even two of them is for testing purposes, authors have created three different QoS architectures and compared them. Nevertheless, scalability is an issue which was not discussed in the paper. This architecture needed to be reprogrammed when a new switch or new server needed to be added to the topology. Topology is given with a hash map to HiQos. Moreover, there exists no information about how queues are configured in the system. However, how bandwidth utilization is queried for queues is explained clearly. Benefits of such querying method are open to debate but there exists no other option in OpenFlow 1.0 standard [20] or OpenFlow 1.3 standard [1] as an OpenFlow switch asynchronously sends congestion report to the controller. In conclusion, HiQos could be an example for network administrators who design a QoS system for their SDN networks. Also, it could inspire academics in their future studies.

In [35] designers have designed an architecture such that controller periodically injects test packets to measure delays of links. Since these packets would be forwarded to the controller at the receiving node controller will measure the delay between injection time and its current time to understand link delay. Note that while sending test packet to the network, controller encapsulates a reference time in the packet to calculate time difference upon receiving. While packet injection mechanism is implemented by OpenFlow Packet Out Message [20] asynchronous Packet In [20] mechanism enables the controller to get test packets from switches. Besides obscuring the implementation details and test results, authors of this paper might have missed one of the important points of SDN concept. In SDN networks datapath traffic is separated from controller path traffic. This implies that any traffic which does not carry user data should not be forwarded from the datapath channel. However, in this architecture, redundant test packages are transmitted. Moreover, these test packages might not be exposed to the same QoS scheme with real traffic since they are different flows. To conclude, despite its violation of the SDN concept, this idea is still applicable and could be preferred although it might deteriorate performance in the network.

While modeling delay of a switching node, queueing delay is the most prominent among four delay types which are queueing, processing, propagation and transmission since its nondeterministic nature. In [14] authors model the queuing delay of

TCP flows. In this model they assumed processing delay of a node is negligible, propagation delay is constant during transmission and queueing delay dominates delay of a node. Before transmission starts, authors measure propagation delay from round trip time extracted from ICMP packets. Then, they estimate the average queuing delay by deducing the average length of a queue. Length of a queue is assumed to be related with the window length of a TCP connection because of the fact that TCP window length is determined by unacknowledged packets which are still in the network. For the multiple TCP flow case, mean of the delay for one TCP flow is modeled as Gaussian random variable and queueing delay is extracted from variations in the length of the queue. The testing environment consists of switches which are cascaded to each other. After proofing their ideas, authors suggest queue switching model in order to provide service differentiation. This idea will be mentioned in next chapters since it has some benefits when compared with routing from another path. Nevertheless, the delay measurement technique in this paper has some crucial dilemmas which should be talked about. Firstly, this model is limited to TCP applications. Although TCP encapsulates most of the application types, the transport layer is not built just on TCP. Secondly, TCP is a complex protocol which is affected by lots of factors. Moreover, window length of a TCP is affected by the TCP/IP stack of OS on which TCP socket is opened. This information could only get from the user platform which modifies user end systems. To sum up, besides the disadvantageous delay measure method, switching between queues when delay increases is a valuable solution which will be discussed in other chapters.

In [13] authors have come up with a monitoring architecture and named it Open Network Monitor (OpenNetMon). Designers concentrated measuring throughput, packet loss and delay accurately. In order to measure throughput, they utilized flow statistics of OpenFlow 1.0 protocol [20]. Because of the fact that flow statistics consist amount of bytes transferred and the amount of time passed since the first flow was introduced throughput of flow could be deduced. They assumed that last switch on which the packet passes before it reaches its destination is the point where flow statistics must be queried. Query period of statistics is an important parameter which brings accuracy in return of processing overhead in controller to switch (outband) network. In order to reduce overhead in processing, they developed an algorithm which sends more frequent queries when flow first introduced but reduce query time after variance of throughput results decrease. On the other hand, for packet loss, they claim that each port statistics [20] of SDN switches from which flow passes have to be queried in order to understand if any drops occurred. Finally, to measure delay they injected test packets with the help of packet out action in [20] on the network. After test packets are routed controller gets these test packets from the last switch by the packet in action [20]. From the time difference of sending and receiving times, controller understands the delay in the network. Authors have tested their topology in a real network. Where throughput and packet loss measurements were satisfying, delay measurements were not accurate which was mentioned by authors too.

Table 3.1: Summary of Delay Measurement Techniques in SDN

Algorithm, Year	Delay or Link State Measurement Technique	Target, Performance Metrics	Comments
[31],2012 OpenQos	OpenFlow 1.0 Feature Request Message [20] to Controller	Peak SNR Values	poor performance metrics, wrong message type
[32],2013 PolicyCop	Special interface between Management Plane and Controller Module which collects statistics	not defined	
[33],2014	Special interface which collects statistics	not defined	
[34],2015 HighQoS	Proactive and periodic query of OpenFlow Queue Statistics	response time of servers, throughput, recovery in case of path loss	
[35],2016	Test Packet Injection, Measurement from Controller	not defined	controlpath datapath violation
[14],2016	from TCP buffer deplation queue size estimation	accuracy	needs user side modification, only covers TCP connections
[13],2014 OpenNetMon	Test Packet Injection, Measurement from Controller	accuracy	low accuracy of results, controlpath datapath violation

CHAPTER 4

PROPOSED LINK DELAY COMPUTATION METHOD FOR SDN

In this thesis, we propose a method to compute the average delay of links in an SDN Network from the statistics collected by the SDN controller. Note that the link delay depends on the queuing delay which is the only nondeterministic component in delay model of packet-switched networks. For queuing delay measurement we propose the set-up shown in Fig. 4.1. When a packet enters a switch, it eventually goes out from a queue of a port. We model the switch queues as G/G/1 queues and bind a meter to each queue for measuring ingress traffic rate. For G/G/1 queue model we need mean and variance values inter arrival times of ingress and inter departure times of egress traffic for calculating average state of the queue. Mean and variance components of inter arrival times are found by meter statistics. On the other hand, we calculate mean and variance of inter departure times from packet lengths and bandwith of links in datapath. After we find average states of queues which is defined as the average number of packets in a queue, we calculate queuing delay from Little's equation.

In order to implement our idea, we design an application which uses the Northbound interface of RYU controller. We periodically poll OpenFlow meter statistics via OpenFlow module of RYU controller. This module further communicates via OpenFlow protocol with SDN switches and it reports results of statistics query via Northbound interface. Our application is operated independently from RYU controller; however, they run in the same Virtual Machine. Note that environment where our application is realized and communication interfaces will be explained in Implementation section of Chapter 5. We only present the inner architecture and the idea of link delay computation method in this chapter.

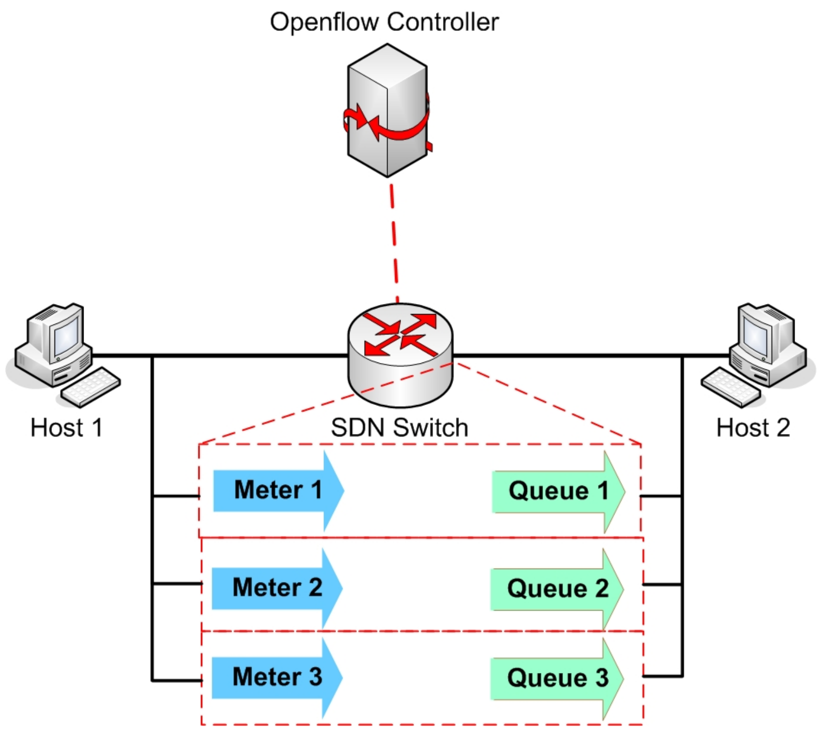


Figure 4.1: Proposed Queueing Delay Measurement Methodology

4.1 Sample Creation Mechanism

For G/G/1 queueing model, one has to assume that shape of ingress and egress traffic could be any type. Hence, for the proposed model we have to extract the mean and variance of interarrival time between packets as well as the time between departed packets from the queue.

For measuring the interarrival time of packets we create a sampling mechanism that creates samples of interarrival times from meter statistics. Sampling mechanism is visualized in Figure 4.2. Meter statistics of SDN switch is periodically requested. Reply of meter statistics consists of how many packets are counted to that time and how much time has passed since meter started its operation. They are called `packet count` and `duration` respectively. Moreover, the algorithm of how we extract a sample is visualized as a flowchart in Figure 4.3. Sampling mechanism holds states for `packet count` and `duration`. If the `packet count` increases, we take the difference of `durations` that comes from statistics message and the value sampling mechanism holds. This creates a time difference. Then we divide this time value to the difference of `packet counts`. This division gives us one sample of interarrival time of packets.

Let us clarify this with an example. Suppose our sampling mechanism holds values 50 ns for duration and 10 for `packet count`. If a statistics message is received with values 110 ns for `duration` and 12 for packet count then we first look the difference between packet counts. Since packet count is increased, we take time difference between durations which is 60 ns. Then we divide it by difference of packet counts. In the end, we get one sample for interarrival time between packets which is 30 ns. We accumulate these samples to calculate mean and variance of interarrival time of packets. In order to get accurate measurements sampling period of statistics by the SDN controller is an important parameter. We define the minimum sampling period as given in Equation 41. In this Equation l_{min} is the minimum length of a packet that can ingress through the switch. C is the bandwidth of the port. If $\tau_{min\_sampling}$ is satisfied then no false samples will be fed to mean and variance calculating process.

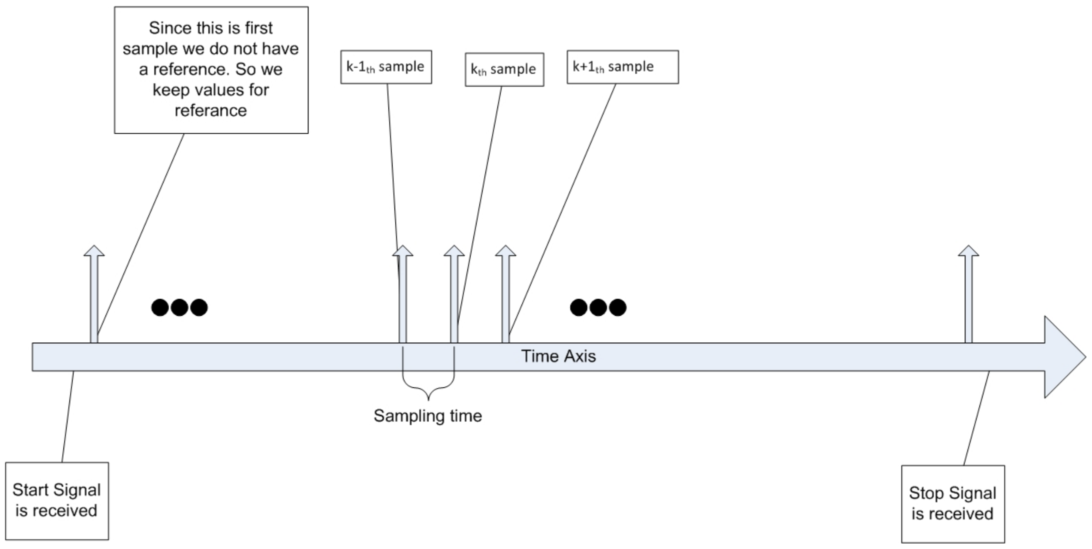


Figure 4.2: Sampling Mechanism

$$\tau_{min\_sampling} = \frac{l_{min}}{C} \tag{41}$$

Departure rate of a queue is defined as how many packets it can transmit in a unit of time. This depends on the bit rate of the service link and length of packets. If the bitrate is assumed to be constant, what changes the departure rate is the length of

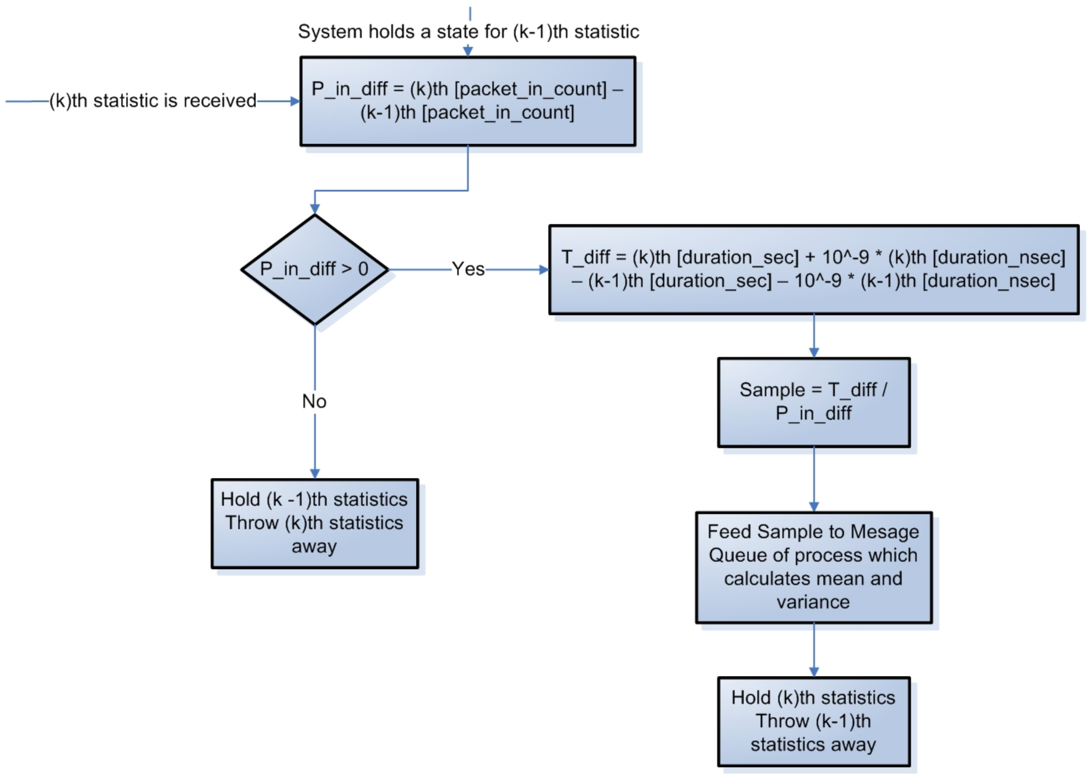

System holds a state for (k-1)th statistic
(k)th statistic is received
P_in_diff = (k)th [packet_in_count] – (k-1)th [packet_in_count]
P_in_diff > 0
Yes
No
T_diff = (k)th [duration_sec] + 10^-9 * (k)th [duration_nsec] – (k-1)th [duration_sec] – 10^-9 * (k-1)th [duration_nsec]
Sample = T_diff / P_in_diff
Feed Sample to Mesage Queue of process which calculates mean and variance
Hold (k)th statistics Throw (k-1)th statistics away
Hold (k -1)th statistics Throw (k)th statistics away

Figure 4.3: Sample Creation Algorithm

packets. Since there exists no statistics about the length of packets, we used a predefined departure rate in delay calculation whose mean and variance are hardcoded in application. Mean and variance values do not change during the simulation thanks to iPerf [16] application. Mean of packet lengths is 1512 bytes and variance is assumed to be zero. We also observe correctness od this decision by checking the length of packets from Wireshark.

After samples are created they are transferred to a process where their mean and variances are calculated. Note that samples are collected in a serial fashion. One cannot calculate mean and variance by using standard mean and variance equations when the sample count reaches to large amounts. Calculating mean and variance from serial samples are given in Equations 42 and 43 respectively. In these Equations $\bar{x}_k$ is the mean of k samples, x_k is k_{th} sample and σ_k^2 is the variance of k samples.

$$\bar{x}_k = \frac{(k-1)\bar{x}_{k-1} + x_k}{k} \tag{42}$$

$$\sigma_k^2 = \frac{(k-1)}{k}(\sigma_{k-1}^2 + \frac{(x_k - \bar{x}_{k-1})^2}{k}) \tag{43}$$

4.2 Implemented Application for Delay Measurement

We design an application which communicates with the SDN controller and collects statistics periodically. In Figure 4.4 we show the modules and their relations on a UML diagram. Our application mainly consists of seven types of threads working concurrently. First of these threads is the main thread called as `environment manager`. This thread is responsible for creating and stopping all the other threads, inserting flows which are necessary for queueing mechanism and forwarding requests to the related threads which are created by itself. Furthermore, second and third are statistics collecting threads which are `Meter Statistics Collector` and `Queue Statistics Collector`. They periodically poll Meter and Queue Statistics and pass them to the Queues of Statistics Calculator Threads. Forth and fifth `Statistics calculator` threads process messages and calculate mean and variances between statistics messages by using algorithms described previously. The sixth type of thread is `Query manager`. It creates collector and calculator threads as well as calculates the queue state and queue delay when environment manager wants the report for delay of a specific queue. Seventh thread is `Console Thread` which enables user to interact with the program. It is built for debugging, starting and stopping the application.

The purpose of `SwitchBSP` class in figure 4.4 is abstracting out the southbound interface from our application. By this way, this application could support various type of interfaces that SDN controllers implement. Furthermore, `Application` interface and `Environment Manager` classes in figure 4.4 provide the interface for programs which can pull information created from our application. These interfaces are not vital for our applications to work but they enhance their adaptability and scalability.

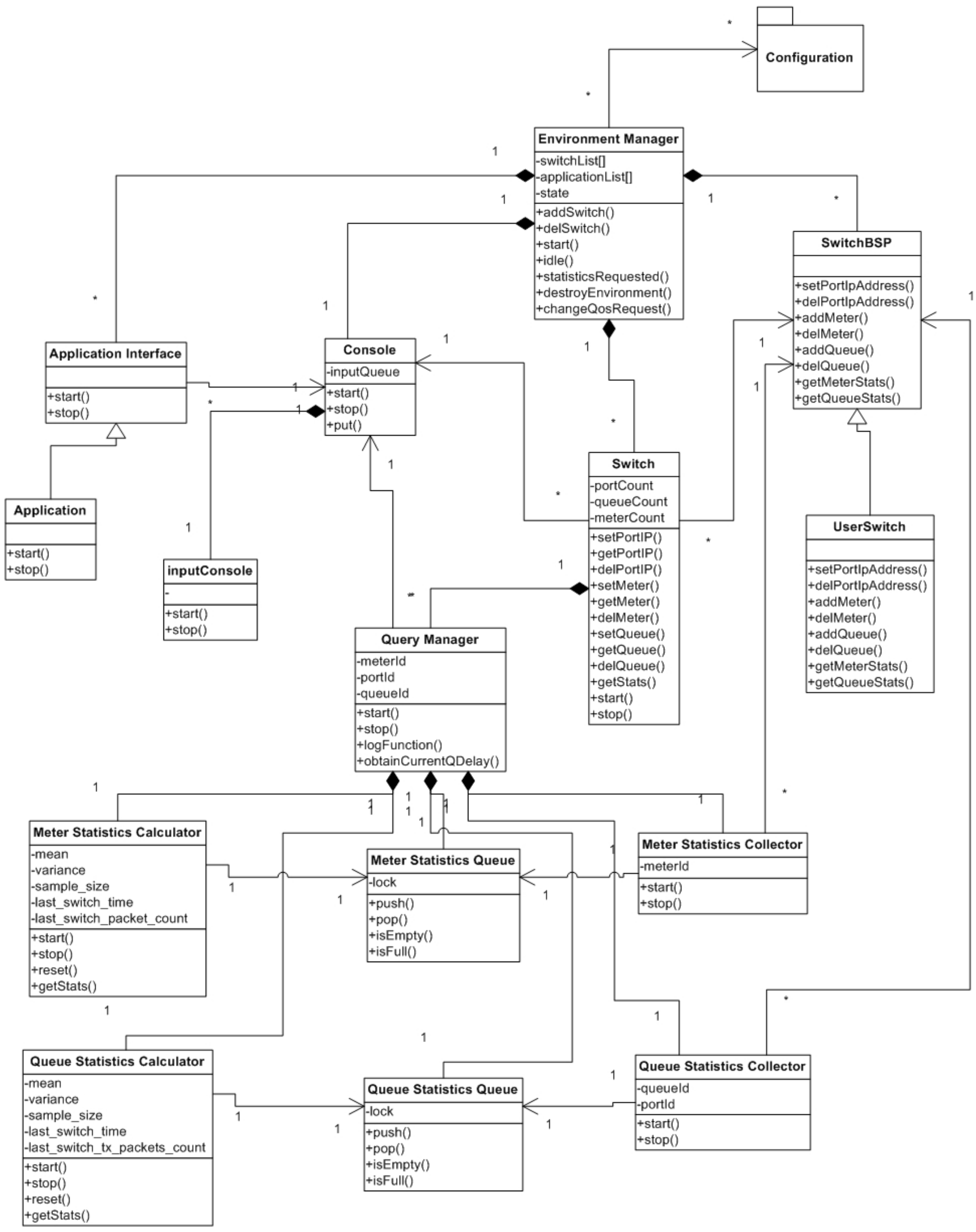


Figure 4.4: UML Diagram of Proposed Method

CHAPTER 5

IMPLEMENTATION AND EVALUATION

5.1 Implementation

We evaluate and test our application in a simulation environment. Figure 5.1 shows the components of the evaluation set-up and how they are organized. All of the tests are conducted on a Linux Virtual Machine. The testing environment has mainly three components. First of these components is Mininet [18] which creates a virtual network on Linux machine. This network has real interfaces meaning that one can share and observe real data while the network operates. Also, it enables its user to create SDN switches such as Ofsoftswitch13 [36] or Open Virtual Switch (OVS) [37]; moreover, it could direct controller interfaces of switches to out of Mininet.

The second component of the testing environment is Opensoftswitch13 [36]. It is an SDN switch and supports OpenFlow 1.3 [1] protocol. For our application, meter and queue support of switch play a vital role. Except its stable features and being widely accepted from community we did not choose Open Virtual Switch (OVS) [37] for our tests because it does not implement meters although it can negotiate with SDN Controllers via OpenFlow 1.3 protocol [1].

In OpenFlow 1.3 protocol meter creation, modification and deletion is defined [1]. Also, Multipart request/response messages are defined in order to query meter statistics. On the other hand, queue creation is out of the scope of OpenFlow 1.3 protocol [1]. Therefore, manufacturers have special queue creation mechanisms.

For queue creation, OVS implements its protocol OVSDB management protocol [38] via which user can define the minimum and the maximum rates of queues. However,

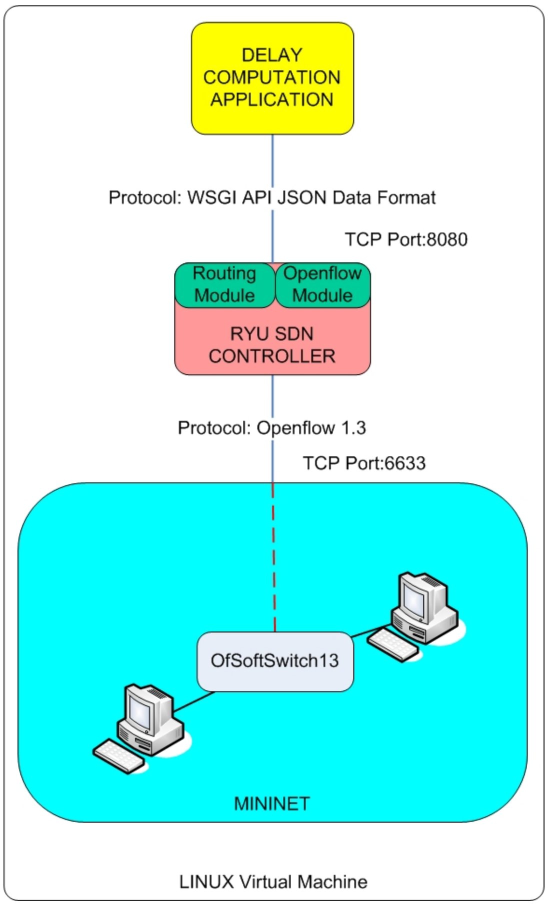


Figure 5.1: Testing Environment

Ofsoftswitch13 implemented its own command-line tool 'dpctl' which interacts with Ofsoftswitch13 and configures the queues of it. 'dpctl' only enables the user to configure minimum bandwidth feature of Ofsoftswitch13 queues. Minimum bandwidth is defined by one of thousand of the bandwidth of the link. For example, when user enters 10 for queue 1 it means, queue 1 should have minimum $10/1000$ of the link bandwidth. When multiple queues exist for a port, a priority mechanism is not defined. Both OVS and OfSoftSwitch13 exchange their queue statistics via Multipart request/response messages [1]. However, Ofsoftswitch13 [36] lacks a communication protocol for creating queues. For queue creation and deletion, we have implemented a command-line queue adding and deleting mechanism using 'dpctl' which communicates Ofsoftswitch13 via the command line of Linux Virtual Machine.

The third component is RYU SDN Framework [21] which is an SDN controller. It has a modular structure and one can include those modules if they are needed. Since we restrict our scope with link delay measurement, we choose [21] controller which has a built-in routing module that routes packets. However, built-in route module inserts and deletes flows from table 0 of SDN switches. On the other hand, we have the requirements for ingress traffic passes from meter to be counted as well as they are directed to queues to count egress traffic. Therefore, we changed the standard routing module of RYU, which inserts routing flows to table 1 of SDN switch instead of table 0. Moreover, we insert flows for link delay measurement to table 0 of SDN switch which direct ingress traffic to meters and output egress traffic from predefined queues. Also, we pass packets to table 1 for routing purposes. This modification brings an advantage which is the separation of concerns of two problems in QoS, routing and queuing.

5.2 Evaluation

In this section, we present experimental results of our approach. Firstly, we examine the performance when a single flow passes through the queue whose queuing delay is observed. Secondly, we analyze the performance of our approach when two different flows are multiplexed into the queue under test. We also evaluate our approach in terms of performance metrics defined in Section 2.

5.2.1 Single Flow

In this experiment we test our proposed method for queuing delay computation in Mininet [18] when a single flow arrives through the queue under test. Our topology is seen in Figure 5.2. This experiment has stages which must be accomplished in order to obtain proper measurements. The first part of the stages is preparation. In this part, we insert flows to the SDN switch that enable datapath packets pipe through a certain queue of SDN Switch. Then, we start Wireshark captures from port 1 and port 2 of SDN Switch to measure time values when packets reach to port 2 of SDN Switch and when they depart from port 1 of SDN switch.

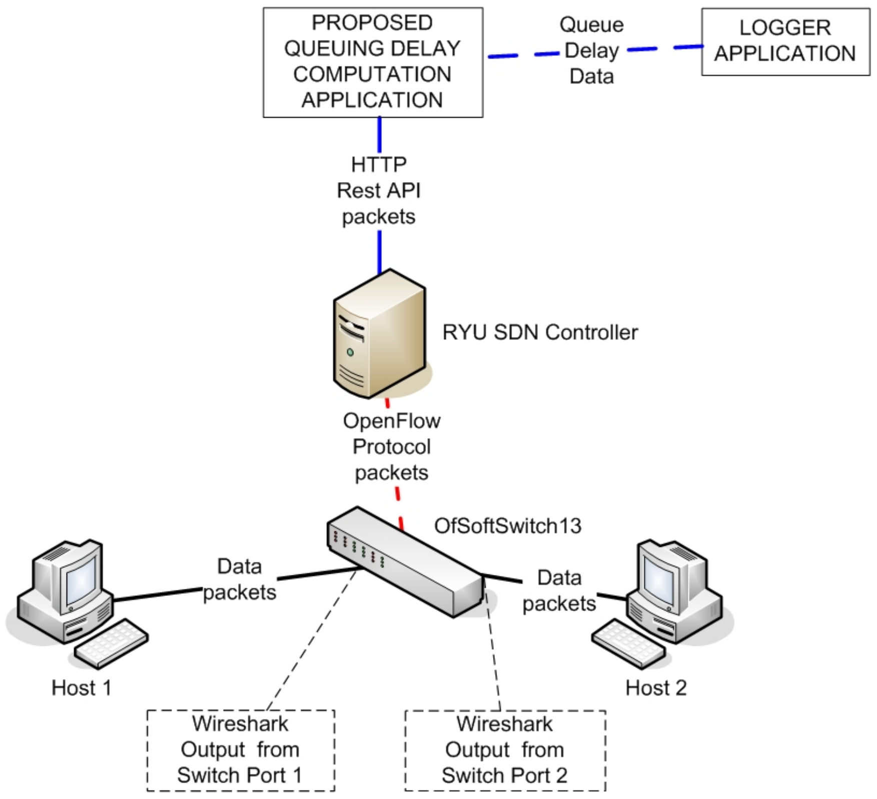

PROPOSED QUEUING DELAY COMPUTATION APPLICATION
Queue Delay Data
LOGGER APPLICATION
HTTP Rest API packets
RYU SDN Controller
OpenFlow Protocol packets
OfSoftSwitch13
Data packets
Data packets
Host 1
Host 2
Wireshark Output from Switch Port 1
Wireshark Output from Switch Port 2

Figure 5.2: Test Topology

Next, we move to the testing stage. We start our proposed application and logger application. Our proposed application periodically polls meter statistics and runs our proposed method which updates queuing delay from meter statistics. Logger application is an embedded thread of our proposed application. It polls queuing delay with one-second intervals and logs computed value of queueing delay as well as the time when it was obtained. Then, we start traffic from hosts. We use iPerf [16] to generate UDP traffic. Host 1 in Figure 5.2 is an iPerf [16] UDP server which accepts UDP traffic from UDP port 5001. On the other hand, Host 2 is a UDP client which sends UDP packets to Host 1. After we run tests for 10 minutes we have two Wireshark captures files of port 1 and port 2 of SDN switch as well as a log file. Logfile consists computed value of queueing delay and time tag when it is logged.

The final stage is the evaluation stage. Firstly, from two Wireshark capture files of SDN Switch, we first construct interarrival delays between packets and the individual queuing delay time series of packets during the simulation. Interarrival times between packets are given in Figure 5.3. Since iPerf [16] do not change packet lengths during the simulation, we do not graph the change of packet lengths versus time. Packet lengths for calculating queuing delay is constant which is 1512 bytes. Moreover, individual queuing delay time series of packets are given in Figure 5.4. We assume that the queuing delay a packet experiences while passing through the queue under test is differences of time tags in the port 1 and port 2 Wireshark capture files. In Figure 5.4, we see that each packet faces different amount of queuing delay during the simulation. Therefore, average queuing delay of time series needed to be calculated for time t_1 in order to compare Wireshark data with the computed value of our proposed application. Average queueing delay of time t_1 is given in Equation 51. In Figure 5.5 we compare average queuing delay of Wireshark data calculated with Equation 51 and the computed queuing data we obtain from our proposed application.

$$d_{average}(t_1) = \frac{\sum\limits_{i=t_{starting}}^{t_1} \quad delay(i)}{n} \tag{51}$$

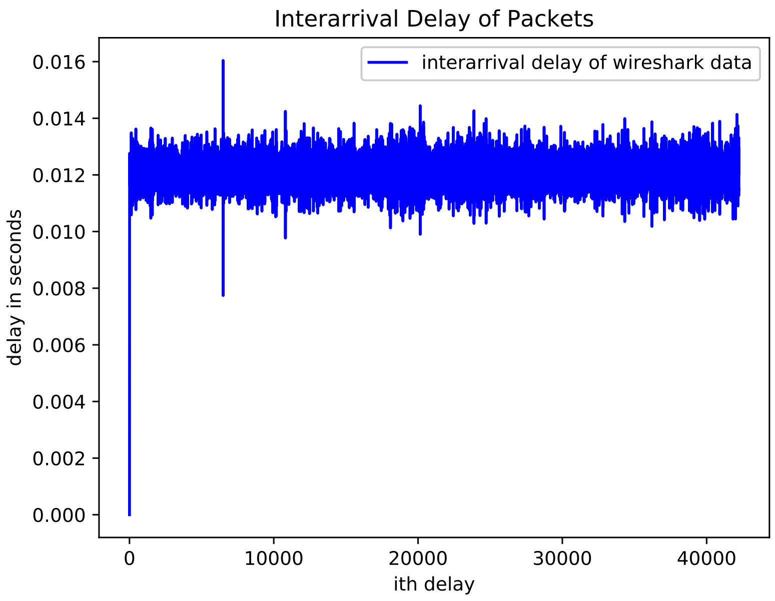


Figure 5.3: Interarrival times between packets to the SDN Switch for Single Flow Case

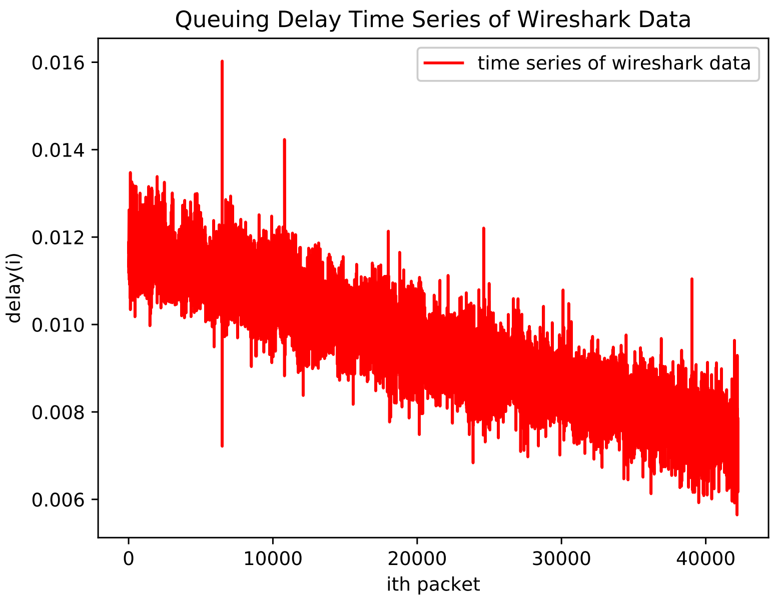


Figure 5.4: Queuing Delay Time Series of Wireshark Data for Single Flow Case

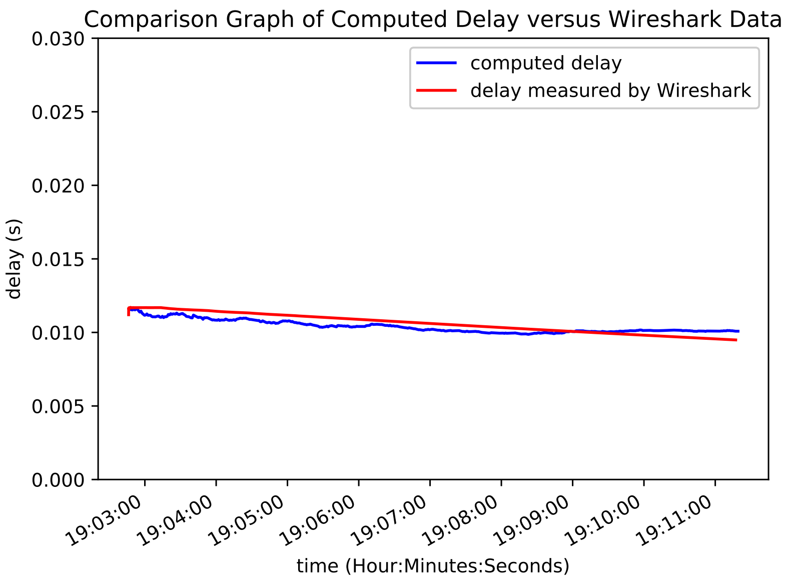


Figure 5.5: Average Queueing Delay of Wireshark Data versus Computed Queuing Delay of Proposed Application for Single Flow Case

Note that, since every component uses the time of virtual machine that it runs on, time tags for comparing average queueing delay and the computed queuing delay data we obtain from our proposed application are time-synchronized. The Relative Percentage Correctness changing with the time is given in Figure 5.6. It is calculated according to Equation 26 which is defined as a performance metric while evaluating performance of our proposed delay computation method. Moreover, our sampling period of statistics $t\_sampling$ is $5 * 10^{-4}s$. This number is selected according to Equation 29 where l_{min} of packets in network is 64 $bytes$ and bandwidth is 1 $Mbps$. n in Equation 28 is calculated as 2.74 where $t\_sampling$ is $5 * 10^{-4}s$, l_{query} is 90 $bytes$ $l_{response}$ is 138 $bytes$ and C is 10 $Mbps$. Therefore our scalability index in Equation 27 equals to 2.74 (n) which is smaller than maximum $meter\_id$ field (2^{32}) of OpenFlow Protocol.

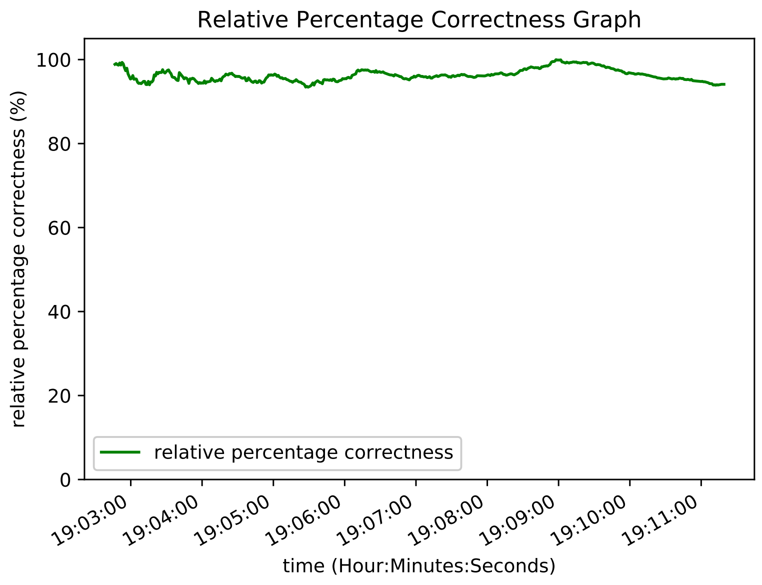


Figure 5.6: Relative Percentage Correctness Graph of Single Flow Case

Scalability depends on the bandwidth ratio between links of controller path traffic and links of datapath traffic. For this case, the bandwidth of controller path traffic links is $10Mbps$ and bandwidth of datapath traffic links are $1Mbps$. For example, if the bandwidth of controller path links becomes $100Mbps$, scalability value n automatically becomes 27.4 increasing ten times of our testing case. This introduces the necessity for our system to run in such SDN topologies whose controller traffic links have higher bandwidth rates than datapath traffic links.

5.2.2 Multiple Flows

Multiple flow case test topology is shown in Figure 5.2 and information flowing from links are same with Single Flow case. Furthermore, starting stages are same except we start two iPerf [16] UDP flows between Host 1 and Host 2. These flows are transmitted from the same queue in SDN Switch. We perform this test with the same procedures applied. Note that Scalability is same with Single Flow test case since we do not change testing topology. Results of multiple flow case are given as follows.

In Figure 5.7 interarrival times between incomming packets are shown. Moreover, Wireshark time series of queuing delay is given in Figure 5.8. In order to compare time series we calculate average queuing delay for time t_1 with Equation 51. The comparison graph of average queuing delay and computed queuing delay of our proposed application is given in Figure 5.9. Finally, Relative Percentage Correctness graph for Multiple Flows case is given in Figure 5.10.

Firstly, in Figure 5.7, it is seen that interarrival packet times are consistent with the Single Flow case. By consistency we mean that mean component of interarrival times are gathered around 0.012 seconds. This value is forced by the bandwidth link of datapath traffic which is $1Mbps$. Further, it is converted to 82.67 $packets\ per\ second$ link capacity by dividing link bandwidth to the length of packets in bits (12096 $bits\ per\ packet$). $1/82.67$ is the amount of time between packet interarrival time which is 0.012 seconds. Since we utilize queue under test close to 100%, the mean component of packet interarrival times being around value 0.012 seconds is reasonable. However, the variance component differs (as we expected) because of the fact that multiple flows have different paces while sending packets.

Moreover, in Figure 5.9 computed delay of our proposed application converges to average queuing delay of Wireshark capture slowly. The reason behind this is the high variance value component of packet interarrival times is significant in queuing delay calculation. After the variance component of packet interarrival times approaches to the minimum value as well as the mean component of packet interarrival times becomes more stable, the relative percentage correctness becomes close to 100% compared to start of application. Note that our computation method of mean and variance of interarrival times influences the convergence time of computed delay. We compute mean and variance from all collected samples; meaning that from start of application all samples collected until time t_1 have impact on computed delay. If there exists a sample whose value differs too much from average, it affects the convergence result of computed delay whose effect is seen in Figure 5.6. On the other hand, mean and variance values of interarrival delay of packets could have been computed with most recent k samples for which we only calculate mean and variance of recent k samples. However, computationally this method requires more processing time as we calculate new mean and variance from scratch. Moroever, k samples method requires more memory space since we hold states for k samples; however, with the method we use, it is only required to hold state for mean, variance and sample count of interarrival delays of packets.

Outcome of multiple flows testing, results have consistency with single flow test. Next we demonstrate our proposed application in a possible usage scenario of QoS differentiation of a flow.

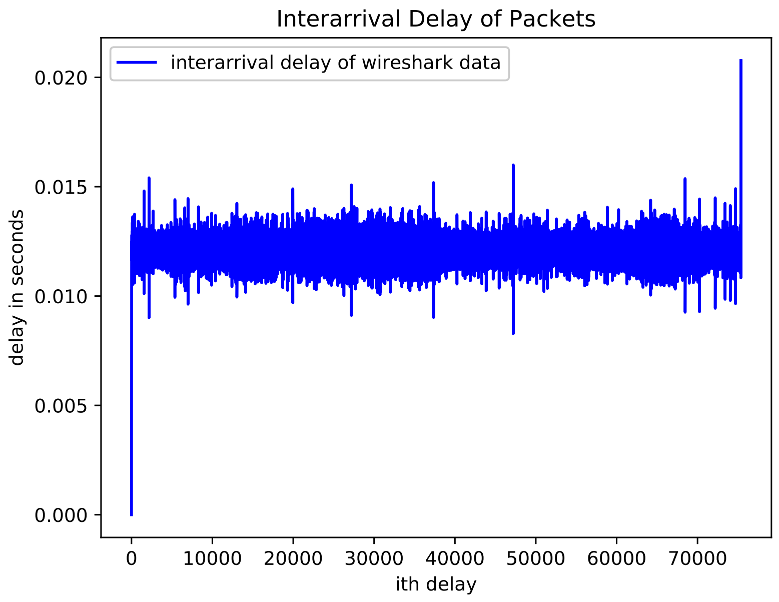


Figure 5.7: Interarrival times between packets to the SDN Switch for Highly Accurate Multiple Flow Case

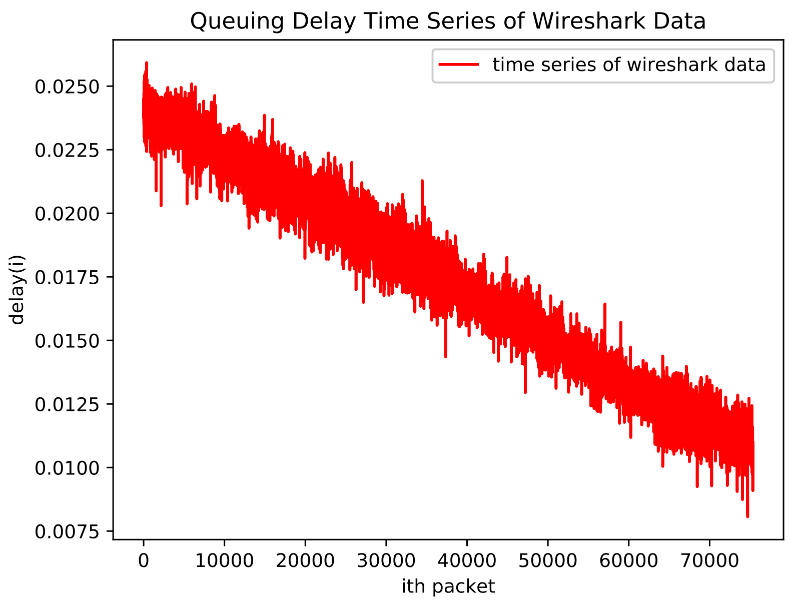


Figure 5.8: Queuing Delay Time Series of Wireshark Data for Highly Accurate Multiple Flow Case

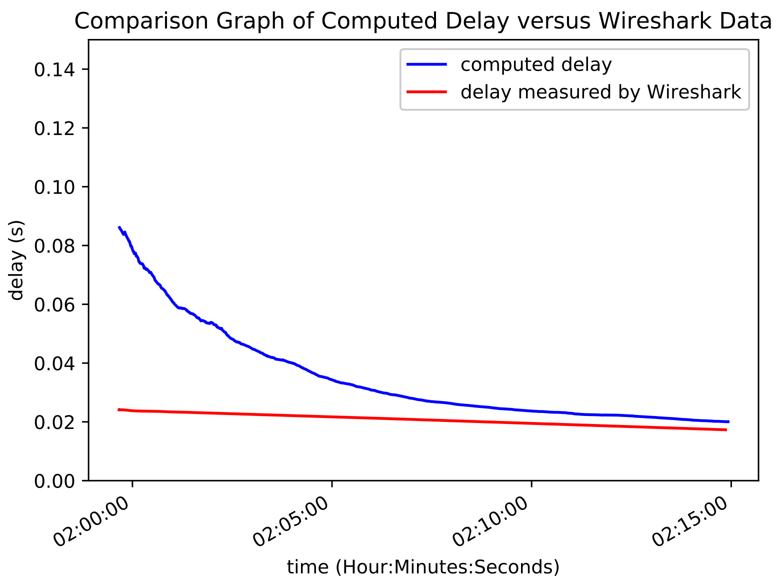


Figure 5.9: Average Queueing Delay of Wireshark Data versus Estimated Queuing Delay of Proposed Application for Highly Accurate Multiple Flow Case

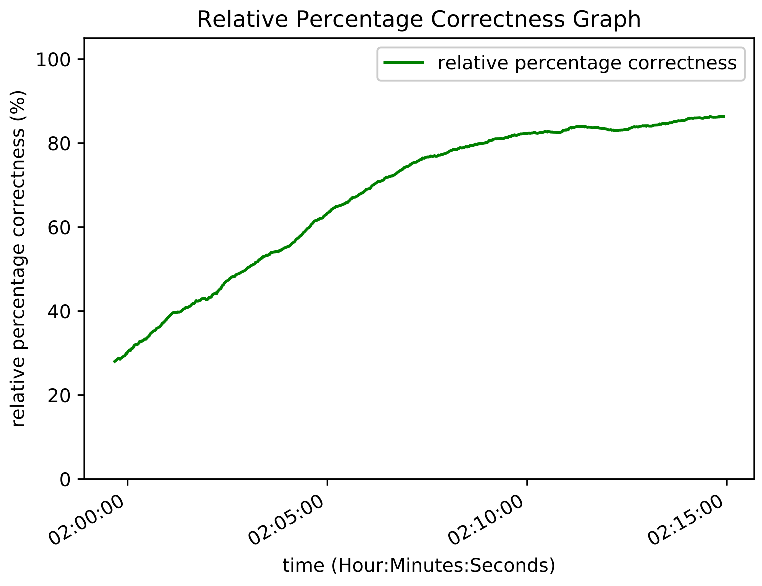


Figure 5.10: Relative Percentage Graph of Highly Accurate Multiple Flow Case

5.3 Best Queue Selector Application

In this section, we demonstrate a possible usage scenario of our proposed method for providing QoS to a predefined flow. For this purpose, we constructed the topology given in Figure 5.11. In this topology, two different UDP iPerf [16] flows flow from Host 2 to Host 1. Host 1 is UDP port 5001 and port 5002 iPerf [16] server, whereas Host 2 behaves as UDP client. At the start of the test, all flows are transmitted from one single queue. In further part, we have 2 scenarios no QoS and with QoS. In no QoS scenario, our proposed method is not operated and all flows are continued to be transmitted from a single queue. On the other hand, with QoS scenario, our proposed method monitors delay value for all queues in port 1 of SDN Switch which is connected to Host 1. Moreover, in with QoS scenario, there exists a Best Queue Selector Application which is able to communicate with our proposed method to get estimated queuing delay values. Furthermore, Best Queue Selector Application provides a delay critical service to UDP dst port 5001 flow by monitoring estimated delays of our proposed application and determines the fastest queue for UDP dst port 5001 flow. After determining the fastest queue, it inserts necessary flows to SDN switch. In order not to jump QoS flow unnecessarily, we put a queuing delay threshold value for changing the queue of QoS flow which is $10ms$. If queue delay of UDP dst port 5001 flow is greater than $10ms$ then Best Queue Selector Application searches the best queue and changes flows in SDN switch if it is necessary.

For evaluation of no QoS and with QoS, we compared jitter measurements of iPerf [16] servers in Host 1 which listens UDP port 5001. Jitter measurements are declared in one-second intervals. In Figure 5.12 jitter measurement values, as well as their trend lines, are shown. Diamonds are jitter measurements of no QoS scenario and Squares are jitter measurements of with Qos Scenario. The dashed line is the trend line of no Qos and the continuous line is trend line of with QoS scenario. Since queues of SDN switch does not provide a priority mechanism, we could not provide a significant improvement; however, from Figure 5.12 with Qos scenario has more stable fashion in terms of jitter compared to no QoS.

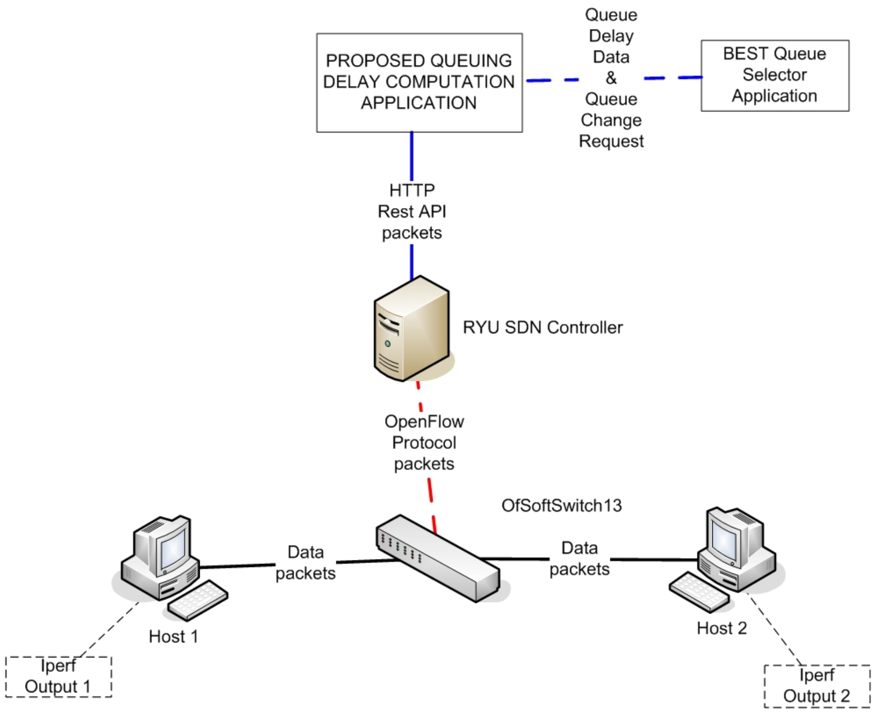

PROPOSED QUEUING DELAY COMPUTATION APPLICATION
Queue Delay Data & Queue Change Request
BEST Queue Selector Application
HTTP Rest API packets
RYU SDN Controller
OpenFlow Protocol packets
OfSoftSwitch13
Data packets
Data packets
Host 1
Host 2
Iperf Output 1
Iperf Output 2

Figure 5.11: Test Topology of Best Qos Selector Application

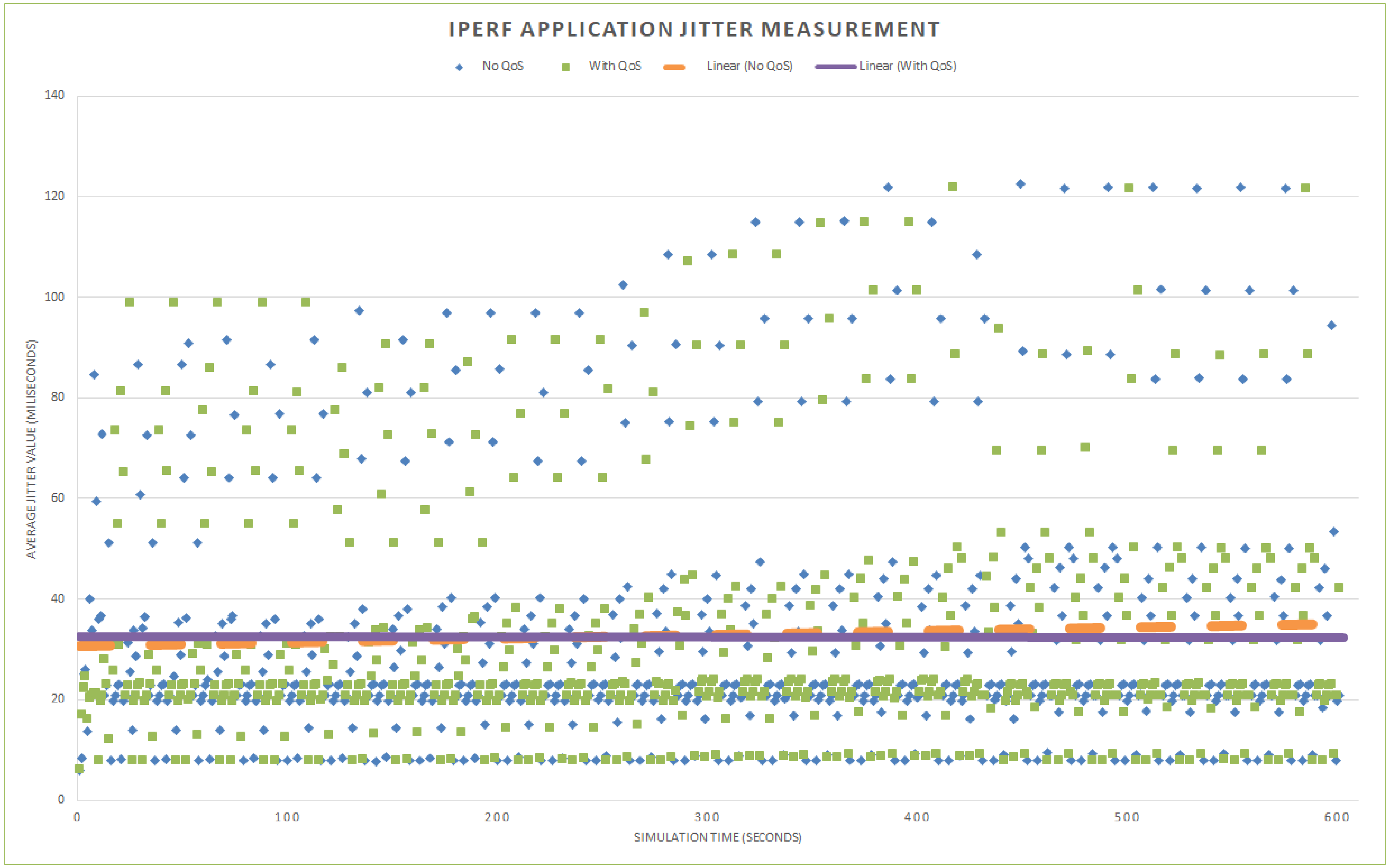


Figure 5.12: Best QoS Selector Application Iperf Measurements

5.4 Sampling Period effect on Relative Percentage Correctness

In this section, we demonstrate the effect of sampling period of statistics $t\_sampling$ on Relative Percentage Correctness. The purpose of this section is to investigate whether $t\_sampling$ period defined in Equation 29 is a sufficient factor that limits our scalability. In another words, "Can we still provide high relative percentage accuracy by reserving little amount of resources such as low computational power on SDN Controller or less bandwidth between SDN controller and SDN Switches?". We set the sampling period of statistics $t\_sampling$ to $10^{-4}\ s$, $10^{-3}\ s$, $10^{-2}\ s$, $10^{-1}\ s$, $1\ s$, $10\ s$ respectively and conduct Single Flow tests with these periods. Note that, in our test setup the minimum required sampling period $t\_sampling$ is calculated as $5 * 10^{-4}\ s$. Furthermore, to evaluate sampling period $t\_sampling$ effect, we calculated the average relative percentage correctness for each tests. Average relative percentage correctness is calculated according to Equation 52. In this Equation, $a_{average}$ is average percentage correctness parameter which will be used in order to evaluate sampling period effect on relative percentage correctness. Moreover, $a(i)$ is Relative Percentage Correctness metric which is defined in Equation 26. Finally, $t_{starting}$ and t_{ending} times are the start time and the ending time of the multiple flow test respectively. The comparison graph of each tests are given in Figure 5.13. Note that, we have conducted tests with iPerf [16] which generates constant packet length traffic. Packet lengths are fixed to $1512\ bytes$. Hence, minimum sampling period required is calculated as $1.2 * 10^{-2}\ s$ according to Equation 29 (Bandwidth B is $1\ Mbps$ and l_{min} is $12\ kilobits$). In Figure 5.13, average relative percentage starts to deteriorate after $t\_sampling$ is greater than $10^{-2}\ s$. The required sampling time for proper operation is $1.2 * 10^{-2}\ s$. We utilize links close to 100% in tests. Note that, minimum sampling time for proper operation is between $10^{-2}\ s$ and $10^{-1}\ s$. This explains the deteriorate of average relative percentage correctness in Figure 5.13 after $10^{-2}\ s$. When sampling period is increased further, we obtain so much errorneous statistics while computing queuing delay that relative percentage becomes almost zero. Therefore, we conclude that introducing the minimum sampling period requirement to the system is a sensible idea that this period has to be satisfied in order to compute queuing delay correctly.

$$a_{average} = \sum_{i=t_{starting}}^{t_{ending}} a(i) \tag{52}$$

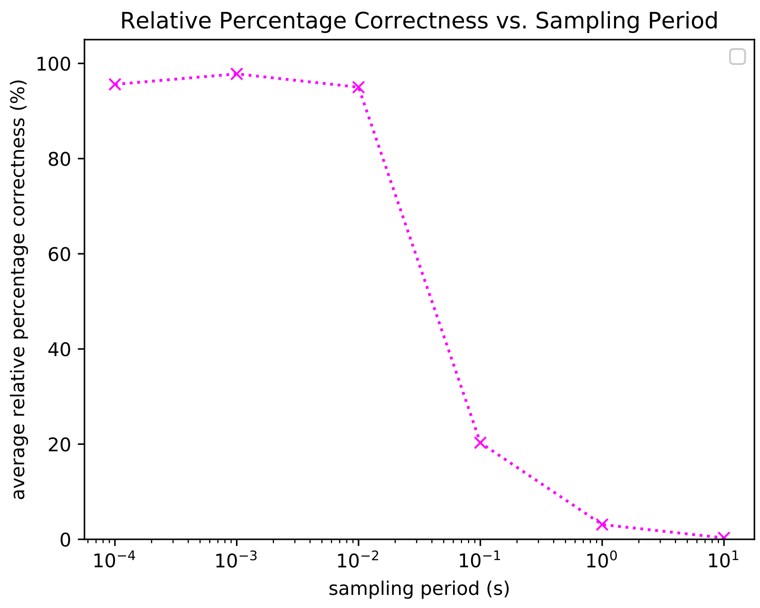


Figure 5.13: Sampling Period effect on Relative Percentage Correctness

CHAPTER 6

CONCLUSION

In this thesis, we concentrate on queuing delay measurement in SDN networks which could provide a decision metric for efficient route calculation. We used OpenFlow meters to measure the ingress rate through a queue and calculate the mean and variance of arrival rate from OpenFlow meter statistics. We used G/G/1 model to extract average state of the queue. Then we find delay from Little's equation that converts queue state to queuing delay.

When compared with works in literature, our solution uses data defined in OpenFlow 1.3 standard [1] which increases adaptability to different environments. OpenFlow 1.3 [1] is a well known and widely accepted standard in the community. Therefore, when a design is built on top of raw mechanisms of OpenFlow, that design becomes realizable in most of SDN topologies. Moreover, the work presented in this thesis does not violate the SDN control plane and data plane separation. We do not intervene in datapath traffic by generating redundant traffic to measure queue delays.

We conduct three experiments to prove the functionality of our work. In the first scenario, we transfer single flow per queue. This experiment presents relatively correct realtime insight about the delay of observed queues; however, maximum queue number that our model can query is limited by packet sizes of OpenFlow Meter Statistics and bandwidth between the SDN controller and SDN switches. In the second scenario, we transfer multiple flows from the observed queue. For this case we observed the fact that our method needs an adaptation time to get accurate results; however after computing a stable mean and variance close to zero, our compuatational results achieves high relative percentage correctness rates which is close to 100%. Furthermore, we made an instance application that uses computed queuing delay of our approach as a metric to determine the emptiest queue of a port. This application does not provide significant QoS since queues of SDN switch is organized to provide fair bandwidth and it is not changeable. However, it provides an improvement in average delay trend which is more stable compared to standard traffic transmission. Finally, we investigate minimum sampling period on correctness of delay computation. After conducting multiple flow tests with different sampling periods, we conclude that minimum sampling period is required to compute queuing delay relatively correct.

We test our solution only on soft environment. Because of this reason our experiments are conducted in suboptimal conditions. For example, due to the fact that the virtual machine has microscale time sensitivity, we could not measure queueing delay of links over 1 Mbps which require nanoscale time resolution in OpenFlow statistics. Moreover, since lots of concurrent module runs in single hardware, our simulation stops when sampling frequency of controller exceeds 10^4 query per seconds. Therefore, for future work, our contribution needs to be tested under real hardware which eliminates mostly capacity issues of the simulation environment. However, measurement techniques of real queuing delay changes as timing synchronization are lost between Wireshark captures and our application.

REFERENCES

[1] B. Pfaff, B. Lantz, et al., “Openflow switch specification version 1.3.1 (wire protocol 0x04),” September 6, 2012. https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.3.1.pdf.

[2] P. Goyal, S. S. Lam, and H. M. Vin, “Determining end-to-end delay bounds in heterogeneous networks,” in International Workshop on Network and Operating Systems Support for Digital Audio and Video, pp. 273–284, Springer, 1995.

[3] R. L. Cruz, “Quality of service guarantees in virtual circuit switched networks,” IEEE Journal on Selected areas in Communications, vol. 13, no. 6, pp. 1048–1056, 1995.

[4] P. Goyal, H. M. Vin, and H. Cheng, “Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks,” IEEE/ACM Transactions on networking, vol. 5, no. 5, pp. 690–704, 1997.

[5] G. G. Xie and S. S. Lam, “Delay guarantee of virtual clock server,” IEEE/ACM transactions on Networking, vol. 3, no. 6, pp. 683–689, 1995.

[6] “Open network foundation.” https://www.opennetworking.org/sdn-definition/.

[7] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagam, S. Mohan, and R. B. Bobba, “End-to-end network delay guarantees for real-time systems using sdn,” in 2017 IEEE Real-Time Systems Symposium (RTSS), pp. 231–242, IEEE, 2017.

[8] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sanso, “Spider: Fault resilient sdn pipeline with recovery delay guarantees,” in 2016 IEEE NetSoft Conference and Workshops (NetSoft), pp. 296–302, IEEE, 2016.

[9] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach (6th Edition). International Edition, Pearson, May 1, 2012.

[10] A. Leon-Garcia, Probability, statistics, and random processes for electrical engineering. Prentice Hall, 2008.

[11] M. Ben-Akiva, M. Bierlaire, D. Burton, H. N. Koutsopoulos, and R. Mishalani, "Network state estimation and prediction for real-time traffic management," Networks and spatial economics, vol. 1, no. 3-4, pp. 293–318, 2001.

[12] R. Coltun, D. Ferguson, J. Moy, and A. Lindem, "Ospf for ipv6," 2008. https://tools.ietf.org/html/rfc5340.

[13] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," IEEE Network Operations and Management Symposium (NOMS), 2014.

[14] M. Haiyan, Y. Jinyao, P. Georgopoulos, and B. Plattner, "Towards sdn based queuing delay estimation," China Communications Vol. 13, No. 3, 2016.

[15] M. Karakus and A. Durresi, "Quality of service (qos) in software defined networking (sdn): A survey," Journal of Network and Computer Applications, vol. 80, pp. 200–218, 2017.

[16] "iperf - the ultimate speed test tool for tcp, udp and sctp." https://iperf.fr/.

[17] W. G. Marchal and C. M. Harris, "A modified erlang approach to approximating gi/g/1 queues," Journal of Applied Probability Vol. 13, No. 1, 1976.

[18] B. Lantz, B. O'Connor, et al., "Mininet an instant virtual network on your laptop (or other pc)." http://mininet.org/.

[19] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, "Planck: Millisecond-scale monitoring and control for commodity networks," in ACM SIGCOMM Computer Communication Review, vol. 44, pp. 407–418, ACM, 2014.

[20] B. Pfaff, B. Heller, et al., "Openflow switch specification version 1.0.0 (wire protocol 0x01)," December 31, 2009. https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf.

[21] T. Fujita, Y. Kaneko, and Y. Isaku, "Ryu component-based sdn framework," 2014. https://osrg.github.io/ryu/.

[22] X. Xiao and L. M. Ni, "Internet qos: A big picture," IEEE Network, 1999.

[23] E. Crawley, R. Nair, B. Rajagopalan, and H. Sandick, "A framework for qos-based routing in the internet," 1998. https://tools.ietf.org/html/rfc2386.html.

[24] H. Badis, A. Munaretto, K. A. Agha, and G. Pujoll, "Qos for ad hoc networking based on multiple metrics: Bandwidth and delay," 2003.

[25] P. Mohapatra, J. Li, and C. Gui, "Qos in mobile ad hoc networks," IEEE Wireless Communications, June 2003.

[26] S. Ran, "A model for web services discovery with qos," Communications of the ACM, 2003.

[27] L. Zeng and et. al., "Qos-aware middleware for web services composition," IEEE Transactions on Software Eengineering. Vol. 30, No. 5, May 2004.

[28] S. Muhizi, G. Shamshin, A. Muthanna, R. Kirichek, A. Vladyko, and A. Koucheryavy, "Analysis and performance evaluation of sdn queue model," in International Conference on Wired/Wireless Internet Communication, pp. 26–37, Springer, 2017.

[29] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain," The Annals of Mathematical Statistics Vol. 24, No. 3, 1953.

[30] G. Malkin, "Rip version 2," 1998. https://tools.ietf.org/html/rfc2453.

[31] H. E. Egilmez, T. S. Dane, T. K. Bagci, and M. A. Tekalp, "Openqos: An open-flow controller design for multimedia delivery with end-to-end quality of service over software-defined networks," Proceedings of The 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference, 2012.

[32] B. M. Faizul, C. S. Rahman, A. Reaz, and B. Raouf, "Policycop: An autonomic qos policy enforcement framework for software defined networks," 2013 IEEE SDN for Future Networks and Services (SDN4FNS), 2013.

[33] H. E. Egilmez and M. A. Tekalp, "Distributed qos architectures for multimedia streaming over software defined networks," IEEE Transactions on Multimedia, 2014.

[34] Y. Jinyao, Z. Hailong, S. Qianjun, L. Bo, and G. Xiao, "Hiqos: An sdn-based multipath qos solution," China Communications Vol. 12, No. 5, 2015.

[35] M. Selmchenko, M. Beshley, O. Panchenko, and M. Klymash, "Development of monitoring system for end-to-end packet delay measurement in software-defined networks," 13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET), 2016.

[36] E. L. Fernandes et al., "Basic openflow software switch (bofuss)." https://github.com/CPqD/ofsoftswitch13.

[37] "Open virtual switch." https://www.openvswitch.org/.

[38] P. B. and D. B., "The open vswitch database management protocol," 2013. https://tools.ietf.org/html/rfc7047.