Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Department of Telecommunications and Media Informatics

Traffic Measurements, Scheduling and Characterization of SDN-based Data Center Networks

Ph.D. DISSERTATION OF

Maiass Zaher

Scientific Supervisor

Dr. Sándor Molnár

December 8, 2021

Declaration

I, undersigned *Maiass Zaher* hereby declare that this Ph.D. dissertation was made by myself, and I only used the sources given at the end. Every part that was quoted word-for-word, or was taken over with the same content, I noted explicitly by giving the reference to the source.

Maiass Zaher _____ December 8, 2021

(signature)

Abstract

The dissertation presents different solutions in the field of traffic congestion control of data center networks. We aim to improve flow completion time of mice flows and preserve the throughput of elephant flows, since mice flows are dominant in such an environment. In this context, SDN represents an efficient paradigm to achieve our goals. In particular, we propose in-network solutions to control the congestion.

To achieve our aims, we analyze the characteristics of traffic flows in data center networks considering mainstream employed solutions in academic and business sectors. We study the risk associated with the different algorithms in the relation to elephant flows loss rates. The results motivated us to improve new solutions in this field.

Traffic congestion control solutions are classified into many classes. In this context, we propose a solution belongs to "TCP parameter modification" class. In particular, to improve page load time, our solution proposes a different reduction value from the default one applied upon congestion events in the network. This solution is proposed for QUIC protocol which is user-space transport protocol. QUIC is widely used transport protocol where it is employed when we use Google services.

We propose two solutions belonged to "Flow scheduling/rerouting" class. Both solutions aim to improve flow completion time of mice flows and maintain the throughput of elephant flows. In particular, the fist one employs QoS mechanisms to forward different flow types out of different queues to maintain specific transmission rates based on the network situation. The second solution schedules and reroutes flows based on available bandwidth on switch egress ports. Besides, we propose a new sampling algorithm which recognizes mice and elephant flows considering control plane overhead and sampling delay.

Furthermore, since all our solutions are SDN based, we study the available SDN solutions to be used in cloud data centers. Therefore, we study the characteristics and requirements of cloud data centers. Then, we study the specifications of many SDN solutions. In addition, we employ multi-criteria decision making model to find the SDN solution mostly satisfying the cloud data center requirements.

Acknowledgements

I would like to express my gratitude to my supervisor professor Sándor Molnár for his guidance and patience throughout my research. I would like to thank Stipendium Hungarian for funding during my PhD years. Last but not least, I would like to thank my parents, my family, my wife, my friends, for their unconditional love, support, understanding.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Description
DCN	Data Center Network
TCP	Transmission Control Protocol
FCT	Flow Completion Time
ECMP	Equal Cost Multi Path
DCTCP	Data Center Transmission Protocol
SDN	Software Defined Network
KS	Kolmogorov Smirnov
AD	Anderson Darling
N-B	Negative Binomial
P-D	Poisson Distribution
VaR	Value at Risk
cwnd	Congestion Window size
QUIC	Quick UDP Internet Connections
UDP	User Datagram Protocol
PLT	Page Load Time
HTTP	Hypertext Transfer Protocol
RTT	Round Trip Time
TLS	Transport Layer Security
CSS	Cascading Style Sheets
PLTC	PLT Change
ECN	Explicit Congestion Notification
AQM	Active Queue Management
SACK	Selective Acknowledgment
RTO	Retransmission Timeout
CDF	Cumulative Distribution Function
CT	Concentrated Traffic
UT	Uniform Traffic
MD	Multi Destinations
NOS	Network Operating System
CDC	Cloud Data Center
AHP	Analytical Hierarchy Process
VNF	Virtual Network Function

Chapter 1

Introduction

1.1 Overview

Quality of Service (QoS) is a main concern in the relation to network performance as networks have a limited amount of resources. Consequently, congestion problems negatively impact QoS. Therefore, extraordinary consideration of network delay should be conducted. Basically, the total delay is accumulated as a result of processing, propagation, queueing and transmission times. Since processing time can be disregarded and propagation time can be calculated, the maximum value of end-to-end delay over a path can be handled in terms of queueing and transmission times [1]. In particular, queueing delay is determined by scheduling algorithms employed on network elements along a path. Therefore, each link provides a specific level of QoS according to queues conditions and available bandwidth. In the context of network congestion yielded from traffic patterns, many-to-one flow pattern is considered as a one reason of network congestion events, particularly inside of data centers as congestion events might occur as a result of parallel connections generated from many sources forwarded to a single destination. Consequently, ports connecting the destination might overflowed, hence, packet retransmissions will negatively impact the overall performance [2].

In this context, two types of network congestion solutions: i) in-network based solutions and ii) end host based solutions, as illustrated in Figure 1.1 [3] are considered. Scheduling [4], rerouting and modifying TCP parameters solutions fall into the first solution category in which network elements, such as switch and router devices, host the solution logic to avoid or mitigate the congestion related problems. On the other hand, the second category solutions employ the end systems to hold and execute the solution logic and to react to congestion events in the networks. Basically, most of end host based solutions employ the congestion control methods provided by TCP for adjusting the transmission rate by modifying congestion window size according to network condition.

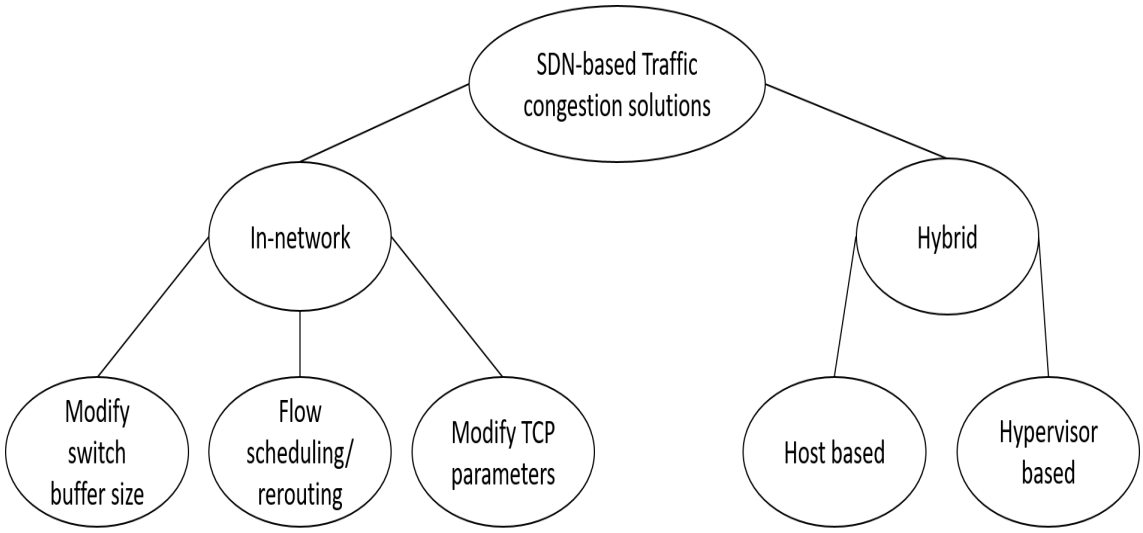


Figure 1.1: Categories of congestion control solutions in SDN based data centers

In this regard, SDN paradigm provides vital potentials to handle congestion problem as this paradigm provides central network control and monitoring. Furthermore, as SDN separates the data plane from the control plane, more solutions can be introduced in the network in virtue of SDN flexibility. As a result, solution categories can be agilely applied due to that SDN provides southbound and northbound APIs which can be employed to probe the network condition, to granular process flow packets and to instruct network elements in the data plane.

1.2 Research objectives

Basically, as a result of uses cases utilized in data centers as IoT, Big Data and IA related applications, many traffic patterns are co-existed in Data Center Networks (DCN). However, the traffic patterns are mapped to two types, which are elephant and mice flows. Every type has its unique characteristics and requirements where Mice flows are short flows and finish its transmission by sending a few number of packets. Therefore, this type is delay-sensitive in comparison to elephant flows type, which encompasses long flows and generates the most of the data traversing DCN. Therefore, it is the bandwidth demanding type. Although mice flows are prominent in DCN compared to the number elephant flows, elephant flows represent the majority of transmitted data in DCN [5] [6]. As a result, extraordinary effort should be conducted to address the probable competitions for resources to maintain the traffic pattern types requirements and consider their characteristics.

In this regard, SDN represents a strong potential to introduce new solution to optimize DCN resources sharing and controlling. Consequently, SDN is a crucial technology in DCN as new applications of IoT,

Big Data and virtual Reality entail processing huge data flows in real time [7]. Data center networks are typically designed as a multi rooted tree with several possible paths connecting each pair of end-hosts. As a result, determining the best route while avoiding any probable congestion is challenging. Typically, congestions significantly increase Flow Completion Time (FCT) which negatively impact mice flows. In this context, implementing static solutions employing static packet header hashing scheduling, as in ECMP (Equal Cost Multi Path), might yield packet collisions on egress ports in case of similar scheduling decisions. On the other hand, introducing the flow scheduling solutions inside network elements, e.g., end-host and switch is still difficult because it sometimes entails kernel or hardware modifications. Moreover, the entirely central solutions contribute to the burden the SDN controller should incur.

Therefore, the first goal of this research was to analyze and study the behaviour of commonly employed algorithms in DCN as presented in Chapter 2. In particular, a mathematical model is employed to present the probable loss rates of elephant flows under the investigated algorithms. We investigate the performance of Hedera, DCTCP and ECMP. In particular, we execute many experiments to measure the performance of the studied algorithms in terms of elephant flows loss rate, FCT of mice flows, throughput of elephant flows. For this sake, a stochastic performance analysis is applied to measure the likely maximum loss rates, elephant flows might incur under the studied algorithms, as well as, the stochastic study result have been compared to the empirical one.

In chapter 3, we present a solution to manage congestion by modifying the reduction rate of congestion window upon congestion events. We adopt SDN based congestion control solutions classification as present in [3]. In particular, this solution belongs to *"TCP Parameters modification"* that we investigate the influence of a different reduction value on the page load time of QUIC protocol. QUIC protocol is a user-space transport protocol employs QUIC TCP variant and it is used when access Google services. For this sake, we study the effects of different reduction values in different scenarios which have varied page sizes, buffer size and bandwidth. As a result, we propose a new reduction value can be used instead of the default one, which yields better page load time in specific scenario parameters.

We propose two SDN based solutions belong to *"Flow scheduling/rerouting"* in chapter 4. The first solution aims to improve FCT of mice flows and maintain the throughput of elephant flows. For this sake, we propose and create a framework employing QoS mechanisms by creating different queues for different flow classes. This framework manipulates the transmission rate of queues dedicated for elephant flows on the upstream switch proportional to the queue length of mice flows on the downstream switch. On the other hand, we design and create another framework which is *"In-network"* solution as well. This framework schedules traffic flows based on Dijkstra algorithm and it reroutes elephant flows only in case of congestion indications. We propose and implement a sampling algorithm to distinguish between mice and elephant flows by employing Openflow bucket. In particular, this framework deals with about 50% of the total flows in DCN to protect the control plane from being overwhelmed and to mitigate the packet

collisions yielded from merely use of ECMP. As a result, about 50% of traffic flows will be scheduled by ECMP that preserves the short time of scheduling process and improves the path selection process by employing our framework in the control plane for scheduling the rest part of the flows. Both frameworks have been evaluated using Mininet emulator under different scenarios. In addition, both frameworks require no modifications to hardware, TCP suite or OS kernel.

Finally, due to the new capabilities provided by SDN, many Network Operating Systems (NOS) have been designed and released. In particular, each NOS provides different functions and services which could serve specific use cases. On the other hand, each NOS consider different set of non-functional requirements. As a result, choosing the right NOS depends on the required use case and the specifications provided by NOS that imposes analyze the available NOSs. In the final chapter, chapter 5, we present a comparative study of six open-source NOSs to choose the most suitable NOS to be used in Cloud Data Center (CDC). For this sake, we employ the mathematical model of Analytical Hierarchy Process (AHP) to decide on the most suitable NOS. The targeted use case has been undiscussed before in the literature. We first investigate and identify the functional and non-functional requirements of CDC. Then, we study the suitability of the studied NOSs according to CDC requirements using pair-wise comparisons.

1.3 Dissertation structure

This dissertation is structured as follows: In Chapter 2, we present an overview on three flow scheduling and control algorithms employed in DCN, the related works, present the experimental methodology followed by the results and discussion. In Chapter 3, we introduce QUIC protocol, our research motivations and the related works, the testbed followed by the results and the discussion. In Chapter 4, we introduce our research methodologies and the related works. Then, we present the architecture of the first solution and its results. Next, we present our second solution and its motivations, its architecture and design aspects followed by its results. Finally, Chapter 5 introduces SDN alternatives specifications, the related works and research motivation. Then, we introduce the characteristics of cloud based data center followed by the decision support system model and the results. Finally, we conclude our contributions in Chapter 6 and present our future works.

Chapter 2

Performance analysis of flow scheduling algorithms in data center network[1]

Most of the flows in DCN are short-lived and small (i.e., mice), and only very few flows are long-lived and large (i.e., elephants). Mice flows are latency-sensitive, such as Voice over IP. On the other hand, elephant flows are throughput sensitive, such as flows transfer data backups. SDN has been employed for routing mice and elephant flows in DCNs. Elephant flows tend to almost fully utilize the switch buffers, yielding delay to mice flows. Many solutions route mice and elephant flows in SDN-based DCNs by dynamically installing forwarding rules for the elephants and schedule mice flows by using switch local static scheduling, (e.g., ECMP). Therefore, these solutions introduce new impacts on mice and elephant flows. In this chapter, we evaluate a Monte Carlo model predicting the loss rate of two state-of-art flow scheduling algorithms which are ECMP, Hedera. Besides, a flow congestion control algorithm which is Data Center TCP (DCTCP). Although these algorithms have been widely employed in the academic researches and the industry, their impacts have not been inspected and analyzed in relation to preserving elephant flows in data center networks. Therefore, we investigated many experiments in fat-tree DCN to evaluate the algorithms efficiency under different network scenarios to validate Monte Carlo risk statistical analysis by experimental evaluation.

[1]This chapter is based on the published article in [J1]

2.1 Background

Nowadays, many enterprises leverage data center fabrics to manage highly-demanded bandwidth applications. Applications like Hadoop [8] and MapReduce [9] rely on hundreds or thousands of servers to provide high availability and scalability; therefore large data is transferred through the data center network to achieve these requirements. Other types of data center applications such as regular web services are hosted inside the data center as well, due to the guaranteed availability and reliability. Because of these substantial requirements, many data center topologies evolved like hyperx [10], flattened butterfly [11], and fat-tree [12]. Many traffic management techniques emerged, like throughput-based forwarding and load balancing [13]. Many applications, like data mining, machine learning, and data analysis [6] [14] demand intensive data transmissions. Therefore, elephant flows must be forwarded through appropriate routes with sufficient bandwidth. In this context, static forwarding techniques like ECMP [15] could yield network congestions since collisions on a specific switch port might occur due to static hashing [16] [17]. Hence, enhancing flow scheduling in data center networks would improve throughput and FCT of both flow types.

In today's data centers, SDN [18] plays a vital role in network resource allocation, traffic monitoring, and classification [19]. The paradigm has significantly employed by the research community for flow scheduling, and traffic load balancing [20] [21] since the implementation of real-time applications is delicate without adequate resource and traffic management [9]. The standard design of a data center network includes multi-rooted trees that have multiple paths between every pair of hosts [17]. As a result, the challenge is to identify the suitable path for flows according to the current load of the paths and to avoid network congestion. However, most of the existing flow scheduling solutions like Hedera [17] forward both flow types on the same paths; hence, flow competitions and bottlenecks are inevitable [22]. Furthermore, rerouting the elephant flows might yield delay, packet reordering, and retransmission.

In this section, we evaluate the efficiency of Monte Carlo model in terms of loss rates of ECMP, Hedera, and DCTCP. Particularly, we experimentally inspect effectiveness of the algorithms to address the following problems:

1. How accurate are the predicted percentages of elephant flows at the risk of loss generated by Monte Carlo model?

2. How could the FCT and throughput of mice and elephant flows be under different algorithms?

Therefore, our main contributions are:

1. Validating the Monte Carlo based stochastic analysis in relation to elephant flows loss rate and throughput of the investigated algorithms.

2. Measuring FCT and throughput of mice and elephant flows of the investigated algorithms in DCN.

2.2 Related work

Many works tackled performance of the traffic flows in data center networks. In this context, the work in [23] proposed an adaptive multipath forwarding solution for elephant flows in data center networks. The study in [20] identified elephant flows by tracking the buffer size of TCP sockets at end hosts. The solution added flag to flows whose size hits the threshold; Hence, the weighted based algorithm will route the tagged flows on suitable path. It reroutes flows based on the link load. It routes mice flows using ECMP. Devoflow [24] provided a flow control mechanism in data center networks by rerouting elephant flows whose sizes are more than 1 MB. Authors in [25] employed OpenFlow group to manage the routes in data center networks by inspecting link loads so that the solution forwarded flows on different routes balancing the load. This solution processed elephant and mice flows similarly. However, upon congestion events, it selects the flow with most bandwidth consumption. However, it did not provide any evaluation of the mice flows performance criteria. The solution presented in [26] proposed TSACO, which employed sFlow to detect elephant flows then it routed them using an multipath algorithm. The solution measured the delay and available bandwidth of routes and scheduled elephant flows on multiple routes whose available bandwidth are considerably sufficient. On the other hand, it scheduled mice flows on the other flows which have low delay values. As a result, the solution achieved more throughput and less delay for elephant flows and mice flows, respectively.

2.3 Experimental methodology

In this section, we describe our experimental methodology, including our system setup, network setup, and applications workloads employed in our empirical study.

2.3.1 System setup

K-4 fat-tree data center topology was built by using Mininet 2.2.2 SDN emulator [18] installed on Ubuntu 16.04 machine provided with Intel Core i5-8400 CPU 2.80 GHz with 16 GB of RAM.

2.3.2 Flow scheduling algorithms

- **Hedera:** estimates the demand for elephant flows then reroute them to a path with sufficient bandwidth by installing new flow entries on the switches. Particularly, flows will be forwarded through one of the equal-cost paths by applying a static hashing based technique as in ECMP until they grow beyond the predefined threshold which is 10% of the link capacity [17].
- **ECMP:** switches are statically configured with several forwarding paths for different subnets. The

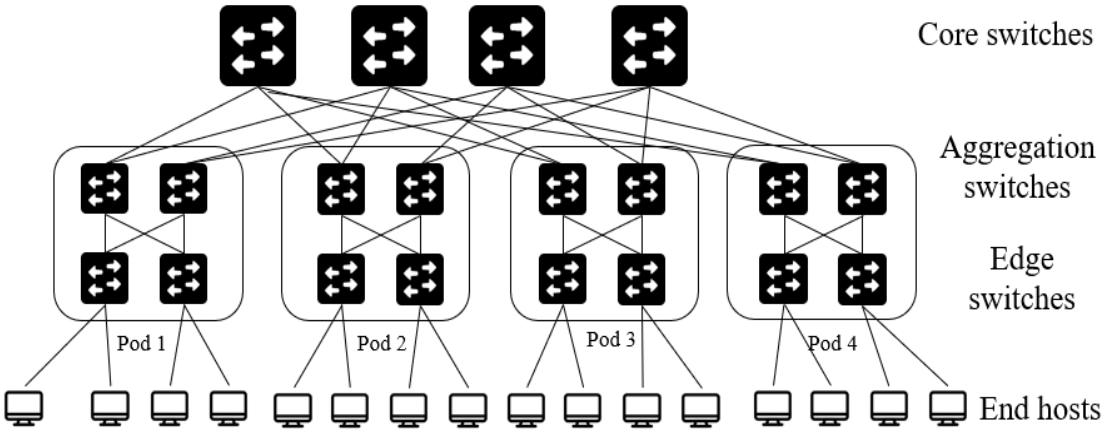


Figure 2.1: Data center network topology

forwarding is based on the hash value of specific fields of packets header modulo the number of paths for spreading the load across many paths [15].

- **DCTCP:** employs Explicit Congestion Notification (ECN) to estimate the fraction of bytes that encounter congestion rather than directly detecting that congestion has occurred. Then, DCTCP scales the size of the TCP congestion window accordingly. This method provides low latency and high throughput with shallow-buffered switches where they can be used in large data centers to reduce the capital expenditure. In typical DCTCP deployments, the marking threshold in the switches is set to a deficient value to reduce queueing delay, and a relatively small amount of congestion will cause the marking. During the blockage, DCTCP will use the fraction of marked packets to reduce the size of the congestion window more gradually than that in case of conventional TCP [2].

DCTCP and Hedera algorithms are implemented and tested as SDN applications by using Ryu controller whereas, ECMP is implemented statically in switches.

2.3.3 Environment setup

In this section, we present the conducted experiment to evaluate the results of the proposed evaluation model. Fat-tree topology is used since it is considered one of the essential topologies for building efficient, scalable, and cost-effective data centers. Fat-tree topology constructed from three main layers of connected switches located in core, aggregate, and edge layers. However, K-4 fat-tree data center topology has been built in Mininet with 10 Mbps links for each as shown in Figure 2.1. The conducted scenarios

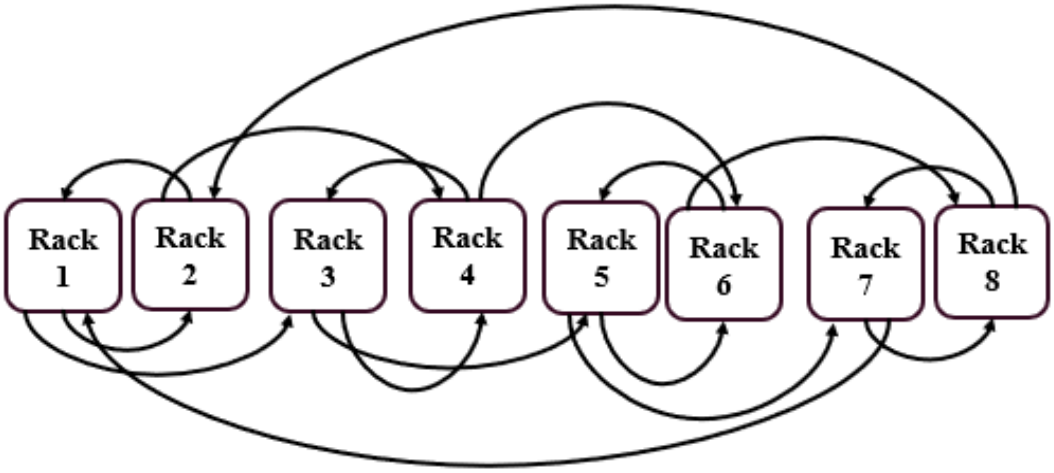


Figure 2.2: Traffic pattern

have two patterns; the first one generates connections that span all topology layers while the second one generates connections span the switches in edge and aggregation layers only as depicted in Figure 2.2. In these patterns, all of the end hosts in each rack employed to generate the traffic for each of the proposed scenario. To generate the required elephant and mice flows, we employed iperf for generating elephant flows, whereas the traffic of mice flows was generated by requesting specific files whose sizes are 10 Kbyte by applying an Apache server repeatedly in a random fashion as reported in [5]. However, the performance of the proposed model will be evaluated under high load scenarios where the mice flows are synchronized with the elephant flows to simulate bursty traffic and introduce congestion in the network as a result. The evaluation process includes three different scenarios with different workloads, a mix of elephant and mice flows, whose time span are varied from 1 to 15 seconds in case of elephant flows to evaluate the investigated algorithms with different sizes of elephant flows. In the first scenario, 1:1 ratio, we generated 120 concurrent connections, so mice and elephant flows have an equal proportion (e.g., 60:60) respectively. In the second scenario, the 1:2 ratio, where we increased the number of the elephant flows to 80, and reduced mice flows to 40. Finally, in the third scenario, 2:1 ratio, where we have 40 elephant flows to 80 mice flows. However, each scenario has been executed twenty-five times, and during each repetition, the throughput has been measured between hosts 1 and 16 by creating a 20 seconds connection using *iperf* to reflect the impact of different algorithms on the throughput of a specific elephant flow.

On the other hand, for the sake of measured the loss rates of the investigated algorithms using Monte Carlo model, the probability distributions of their throughput and measurement errors were identified by Kolmogorov Smirnov and Anderson-Darling statistical tests [27]. As a result, and as presented in [J1],

Table 2.1: Elephant flows parameters

Elephant flow	Size(S)	Volume(V)
large	1.25 MB	100
normal 1	0.75 MB	85
normal 2	0.5 MB	65
small 1	0.12 MB	45

their throughput has the geometric distribution whereas their measurements error of Hedera, ECMP and DCTCP followed discrete uniform, negative binomial and Poisson distributions, respectively.

2.3.4 Monte Carlo Simulation

The Monte Carlo method is a technique for simulating a system's stochastic behavior or evaluating a collection of uncertain inputs. Basically, the outcomes of black box systems are unpredictable [28]. Hence, the Monte Carlo method was employed to simulate many predicted scenarios based on the probability distribution of the input. Consequently, this operation had been repeated thousand times to simulate the probable scenarios. For this purpose, samples of throughput and error were generated based on the algorithms' probability distributions to be the input of Monte Carlo simulation. Eq. 2.1 proposed by Aymen [J1] which represents Monte Carlo model to simulate various sizes and volumes of elephant flows along with the risk values.

$$pre(V, S, A, E) = B_m = V_i \times (S_i - (A_k + E_l)) \tag{2.1}$$

Table 2.1 presents the parameters value of the model. B_m is the probable data amount at the risk of loss, V_i is the volume of the elephant flow, S_j is the sizes demanded by the elephant flows, A_k is the available throughput, and E_l is the error factor. The value of S varies from link capacity (i.e., 10 Mbps) to the size threshold of elephant flows, which is 10% of link capacity as proposed by Al-Fares et al. in [17]. The value of V indicates the amount of a flow, and it is measured in seconds.

2.4 Results and discussions

The proposed model performed on the algorithms to investigate how they will preserve the elephant flows. Therefore, Eq. 2.1 was repeated one million times.

Table 2.2: The probable elephant flow loss rate computed by Monte Carlo of the algorithms

Algorithm	maximum probable loss rate
Hedera	64%
ECMP	68%
DCTCP	77%

2.4.1 Loss rate distribution

Table 2.2, conducted by Aymen [J1], presents the overall estimation of Monte Carlo model for the input of all algorithms. DCTCP provided the highest probability of loss that DCTCP is a flow control mechanism and does not provide any flow scheduling algorithm. On the other hand, Hedera and ECMP provide flow scheduling.

2.4.2 Value at Risk analysis

Using Monte Carlo investigation results, Value at Risk (VAR) analysis [29] had been employed to provide more evaluation using different confidence levels. 95% confidence level was adopted, to consider varying different values. VAR has been calculated as in Eq. 2.2 [29].

$$VaR = -\mu_n + \varnothing^{-1}(1-u)\sigma_n \tag{2.2}$$

This type of analysis explores a dynamic assessment of elephant flows throughput under the investigated flows schedulers or congestion control methods. Aymen [J1] computed VAR and depicted its values for various confidence levels as shown in Figure 2.3. The probable maximum data size at the risk of loss in the case of Hedera is lowest with 112 MBps for the total number of elephant flows depicted in Table 2.1. The loss of the ECMP and DCTCP were 116 and 117 MBps, respectively. These values represent the probable maximum loss amounts, elephant flows might incur in the lifespan of data center network.

2.4.3 The probability distribution of the whole workload

In this section, we present the probability distribution of the entire workload (i.e., 120 connections) of each scenario for all algorithms. We evaluate the performance of DCTCP, ECMP, and Hedera in terms of throughput of elephant flows and flow completion time of mice flows. All figures show the fact that Hedera and ECMP have very similar performance regarding flow completion time of mice flows and the throughput of elephant flows. Where Hedera employs ECMP for forwarding the mice flows, and ECMP performs well when there are no packet collisions on switch ports what makes its performance in terms of

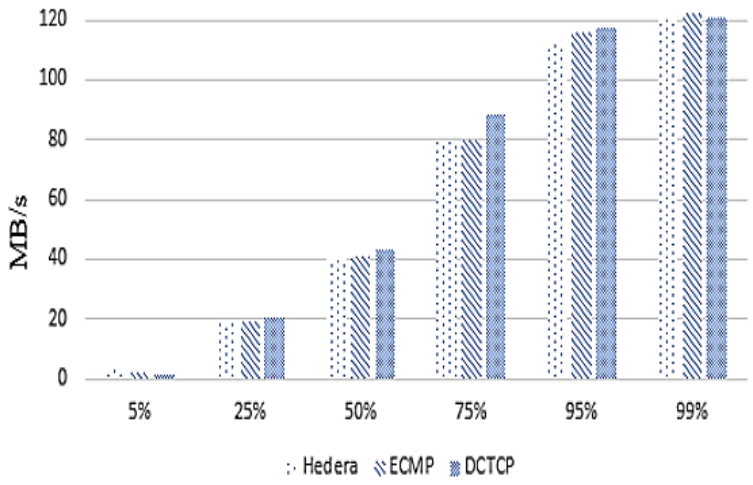
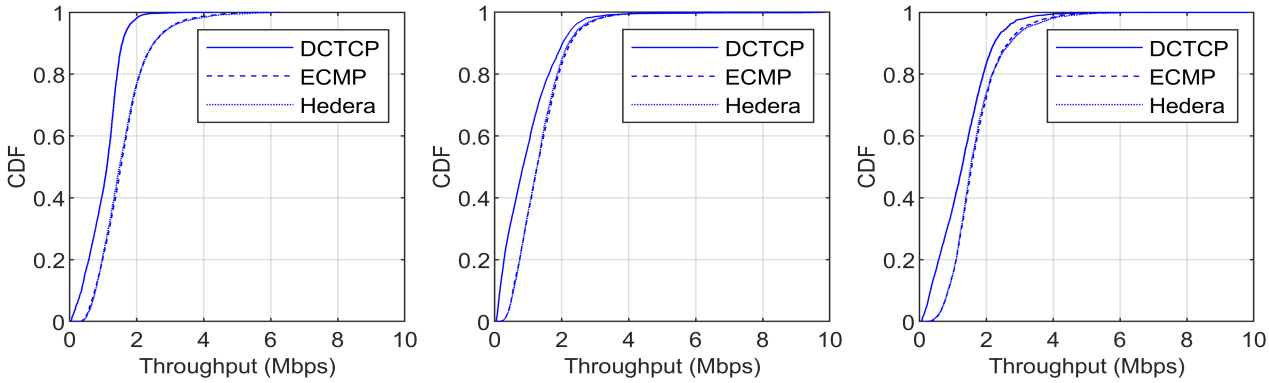


Figure 2.3: Maximum probable amount of lost data of all algorithms according to confidence levels of VaR

elephant flows throughput closely approaches that of Hedera as shown in Figures 2.4 a-c. On the other hand, Figures 2.5 a-c depict the performance of DCTCP where its FCT of mice flows is more significant than that of Hedera and ECMP because of that DCTCP employs shallow threshold to trigger the marking event. Consequently, the transmission rate will be throttled by sources where mice flow is delay-sensitive traffic, as well as elephant flows have worse throughput than that of ECMP and Hedera where DCTCP provides flow control mechanisms, but it does not provide scheduling technique.

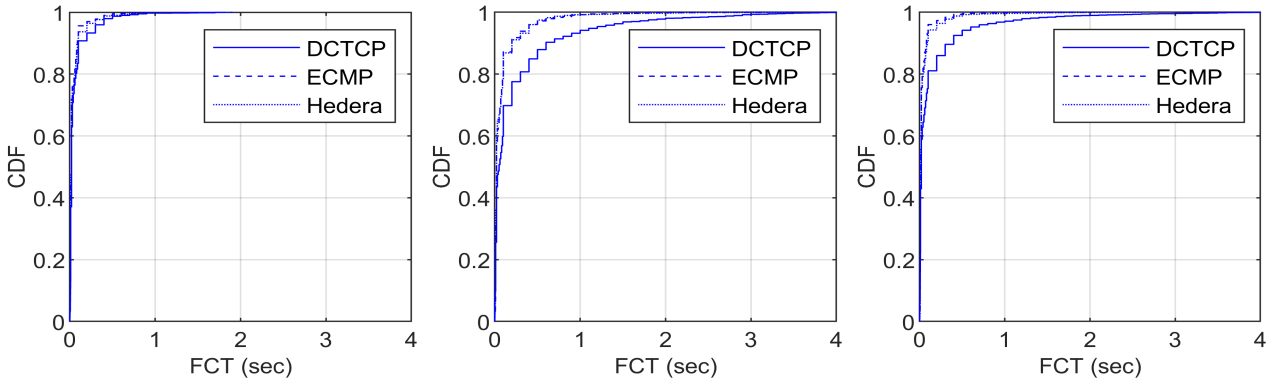
In a nutshell, Hedera provided a smaller data amount at risk of loss than ECMP as expected, but with higher variance for the error factor as presented in [J1]. This factor makes the Hedera does not much outperform over ECMP. As for the response time, Hedera and ECMP achieved better flow completion time due to the static hashing between every source and destination on the network. In the case of flow congestion control in DCTCP, it has achieved its best in the 2:1 scenario. This indicates that the algorithm suffers in case of high elephant flow loads and this fact is presented as well in the associated loss rates of each algorithm, as shown in Table 2.3. Regarding data center applications that demand high bandwidth and low latency, every TCP loss causes bursty retransmission and that what makes queues length of the data center switches bloat frequently. Therefore, applications like MapReduce cannot make incremental progress without limiting the number of contending flows. Therefore, we suggest that some fairness should be considered by providing a balance between link utilization, congestion control. As for the performance evaluation methods of new algorithms that handle traffic flows, we recommend considering the uncertainty behaviors of the tested network and predict their loss rates.

To the best of our knowledge, most of the developed heuristic algorithms for flow scheduling are



(a) Scenario 1:1 (b) Scenario 1:2 (c) Scenario 2:1

Figure 2.4: Throughput of elephant flows



(a) Scenario 1:1 (b) Scenario 1:2 (c) Scenario 2:1

Figure 2.5: Flow completion time

evaluated using the average values for the obtained data without employing the probability distribution function. Note that the expected value for random variables does not exist for some distributions that have a long tail [30]. Consequently, considering the average for any sample of the data may not actually describe the expected value of the measured data, especially if the number of samples is limited. Such as in the case of Hedera [17] and Mahout [20] where the average value is taken for different performance evaluation objectives without identifying the proper probability distribution. Nevertheless, the essence of the prediction produced by independent and random variables relies on current observations to predict future performance. Accordingly, the model and assumptions need to be accurate enough.

Table 2.3: Loss rate of the algorithms in simulation environment

Algorithm	1:1	1:2	2:1
Hedera	0%	0%	0%
ECMP	0%	0%	0%
DCTCP	22.22%	75.55%	11.39%

2.5 Summary

In this chapter, we experimentally validated Monte Carlo model employed to find value at risk of elephant flows loss rate in DCN. For this purpose, we conducted many experiments in different traffic scenarios in fat-tree DCN. The experimental results empirically validate the Monte Carlo model results since Hedera and ECMP provided similar throughput and outperformed DCTCP. On the other hand, the empirical loss rates of the different algorithms consistent with the empirical throughout results and with Monte Carlo model results as well. As a result, stochastic based analysis such as Monte Carlo simulation can be employed to evaluate and predict flow scheduling and control algorithms performance.

Chapter 3

TCP parameters modification based congestion control[1]

Congestion control solutions are classified as "In-network" solutions in case they employ network elements to detect, avoid and respond to congestion events in the data center. This class does not require any changes to either the end-host TCP/IP stack or the hypervisor. Basically, such TCP-agnostic solutions have no limitation on OS or TCP variant provided in the production environment. One subclass of this solution class modifies TCP congestion control parameters to change the transmission rate of TCP senders. The most common TCP parameter that can be manipulated is Congestion Window size (cwnd). Since TCP senders rate is limited by the reduction amount upon congestion events, senders rate can be mitigated by smaller cwnd value. In this chapter, we investigate the impact of two different values of multiplication decrease factor of QUIC on the Page Load Time (PLT).

3.1 Introduction

Conventional network stack is the most deployed model for manipulating the network functions. However, one of the reasons which makes the introduction of new solutions in the network difficult is that the network stack resides in the kernel space of the operating systems and it requires a lot of time to adopt changes. This situation motivated network engineers to introduce the network stacks and protocols as a part of the user-space of the operating systems in order to simplify the developing of new solutions. As a result, today there are some user-space network stacks and protocols such as Quick UDP Internet Connections (QUIC) [31], mTCP [32]. Furthermore, the increasingly expanding architectures such as IoT

[1] This chapter is the published paper in [C1]

and the cloud computing encouraged network engineers to employ a user-space network stack to mitigate the inefficiency of the conventional general-purpose kernel network stack for providing scalability for service providers. The user-space network stacks have been proven to be developed and configured in a very flexible way [33].

Google proposed QUIC over User Datagram Protocol (UDP) to reduce the latency generated by TCP. Google deploys QUIC in its servers and the Chrome browser. Google decided to design a new protocol rather than modify TCP because modifying TCP would take years due to TCP is built in the kernel of the operating system which requires a long life-cycle in the market to adopt. In contrast, QUIC is built in the user-space of the operating system, so it is flexible to deploy and fast to update [34]. Although QUIC employs UDP, QUIC provides a connection-oriented and reliable transfer [35]. However, user-space network protocols and stacks could implement the congestion control algorithms provided for TCP and it encouraged us to study the robustness and sensitivity of the multiplication decrease factor β of the CUBIC congestion control algorithm and its impact on the performance of QUIC. In this chapter, we investigate the impact of two different values of β of CUBIC algorithm, which is implemented by QUIC as the default congestion control algorithm. Note that, this question is practically also motivated since QUIC changed the congestion window reduction as compared to CUBIC from 30% to 15%. QUIC has been chosen for this research because it is the most well designed and deployed user-space network protocol at the time of writing. The main contribution of this chapter is providing an experimental investigation to check the impact of two different values of β on Page Load Time (PLT) of QUIC in case of a lossy network where we compare the impact of $\beta = 0.7, 0.6$.

3.2 Background and related works

3.2.1 QUIC

Google has developed QUIC to speed up the web traffic more than that with SPDY [36] and Hypertext Transfer Protocol (HTTP) [37]. To achieve this goal QUIC employs UDP as a transport protocol, but at the same time QUIC provides a connection oriented and reliable transmission [34]. QUIC significantly reduces the connection startup delay where it offers 0- RTT (Round Trip Time) connection establishment if there was a connection established between the client and the server before, as shown in Figure 3.1. This 0-RTT latency for connection establishment is the result of that QUIC applies a dedicated ID to identify a connection instead of using source and destination IP addresses and port numbers. Therefore, the dedicated ID mechanism provides a persistent connection when mobile clients change their access points and get a new IP address [35]. In contrast, TCP generates 1-RTT connection establishment and 3-RTT in case of using Transport Layer Security (TLS) due to key exchanging, as shown in Figure 3.1 [34].

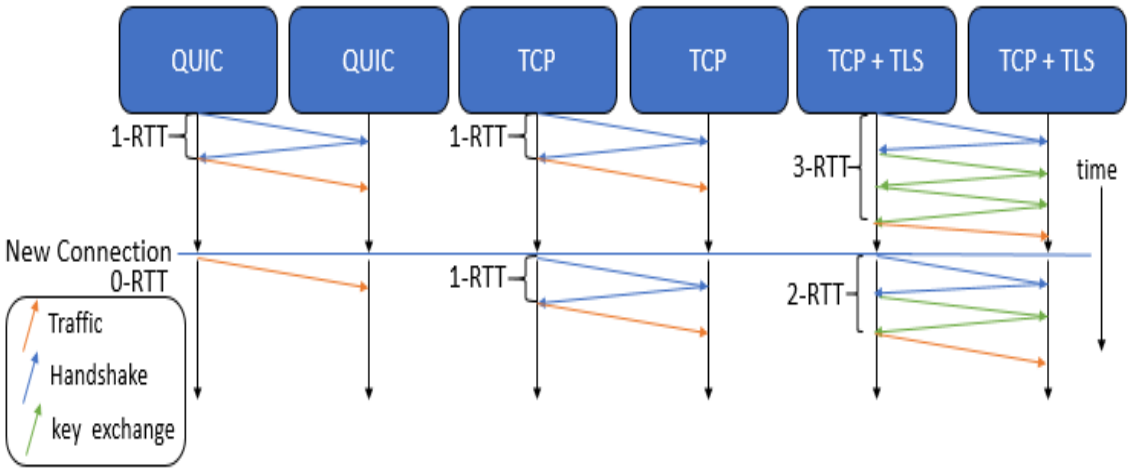


Figure 3.1: RTT for connection establishment of QUIC, TCP, TCP+TLS

However, QUIC provides features similar to TLS and TCP such as in-order reliable packet transmission, privacy, and congestion avoidance mechanisms, but it presents itself in the network as a UDP connection where it is a tunneling protocol running on top of UDP for multiplexing data streams together. One goal of QUIC is to improve the performance in case of packets loss. To achieve this goal QUIC employs packet pacing to estimate the available bandwidth by tracking the inter-arrivals of packets [35].

3.2.2 CUBIC

CUBIC [38] defines the increase rate of the congestion window as a cubic function of the elapsed time since the last congestion event and β which is a coefficient of multiplicative decrease. The dynamics of congestion window are controlled as shown in Eq. 3.1 [38].

$$W = C\left(\Delta - \sqrt{\frac{\beta \,.\, w_{max}}{C}}\right)^3 + w_{max} \tag{3.1}$$

Where C is a predefined constant for scaling, Δ is the elapsed time since the last congestion event. w_{max} is the congestion window size just before the last loss event. β is the multiplication decrease factor applied after a packet loss. When a packet loss event occurs the size of the congestion window will be reduced by β as in Eq. 3.2 [39]:

$$cwnd = cwnd \;.\; \beta \tag{3.2}$$

CUBIC sets β to 0.7 so that the reduction of the congestion window size after a loss event will be 30%. Setting β to a value bigger than 0.5 causes slower convergence which necessitated to add a new mechanism

to release a part of the bandwidth hold by existing flows for new flows. Therefore, when a loss event occurs, if the current size of w_{max} is less than w_{max} at the former loss event, CUBIC reduces w_{max} for the current congestion event further to give new flows more time to increase their windows [39].

3.2.3 Congestion Control of QUIC

At the time of writing, QUIC implements many TCP congestion control algorithms, but the default is CUBIC. However, QUIC employs packet pacing to estimate the available bandwidth by tracking the intervals between packets at the receiver and the sender [40]. Packet pacing can mitigate the congestion by decreasing the variation in packet flows [34] since a packet loss still represents a sign of congestion in the connection. QUIC reacts to loss events analogously to TCP, specifically, according to CUBIC algorithm. It reduces the sending rate according to the value of β where β equals to 0.7 with considering of emulating n connections, as shown in Eq. 3.3, which reduces the congestion window by 15% instead of 30% in the original CUBIC:

$$cwnd \quad = \quad cwnd \quad . \quad \frac{n-1+\beta}{n} \tag{3.3}$$

However, the intent is to emulate the impact of n connections so that when losing a packet, the streams over one connection have bandwidth equal to n times that of a one connection of traditional TCP to attain flow fairness, where $n = 2$. It is an algorithm known as (Multiple) MulTCP [41] [42].

3.2.4 Packet pacing

Packet pacing exists to mitigate the packet loss by sending below than full rate, but it decreases the overall throughput in case of high bandwidth. QUIC applies packet pacing to reduce the page load time where packet pacing tracks inter-packet spacing for estimating the available bandwidth such that a sender cannot send at maximum rate as well as it reduces the packet loss [35]. Experiments conducted by Google have shown that packet pacing has good impact on QUIC performance in the presence of packet loss. However, pacing rate is approximately equivalent to division of the congestion window size over an estimation of RTT [34].

To the best of our knowledge, at the time of the work, there is no publications regarding the experimental performance study of the impact of β values of QUIC's congestion control algorithm. However, Authors in [43] compared the congestion control dynamics of QUIC CUBIC and TCP CUBIC. They found that QUIC CUBIC has more stable congestion window dynamics than that of TCP CUIBIC because of the congestion window reduction in QUIC is smaller. Miller and Hsiao found that adjusting the value of β of TCP CUBIC resulted in less transfer time in case of packet loss [44].

3.3 Testbed and metrics

In this section we describe the testbed and the metrics applied to investigate the impact of two values of β on PLT of QUIC. We employed the testbed, presented in Figure 3.2, which consists of two computers. First one is employed as a client and it has Intel Core i5 2.4GHz, 4 GB RAM, Ubuntu 15.10, kernel 4.2.0-22. Second one is acting as a server and it has Intel Core i5 3.4GHz, 8 GB RAM, Ubuntu 14.04, kernel 3.16.0-38. On the server side we used the QUIC server available on Chromium project website [45] to carry out this evaluation, whereas we used Google Chrome v60 as a QUIC client which provides QUIC version 38. For emulating different network conditions, we applied traffic shaping between the client and the server. We used Netem of the TC (Traffic Control) to emulate different values of bandwidth, loss, queue length, and delay. For loss, we investigated many values of random loss: 0.2%, 0.5%, 1%, 2%, 5% to figure out the impact of β in low, medium and high packet loss conditions where the reduction of the congestion window size will occur after packet loss events. Also, for delay we measured the impact of low and high delay by setting RTT to 5, 10, 200 and 400ms. In all scenarios we set the buffer size equal to the bandwidth-delay product except one where we considered the impact of over-buffered and under-buffered network [46]. Table 3.1 presents the parameters applied in the experiment. We applied many combinations of previous parameters to download different web pages from the QUIC server to the QUIC client for both values of β.

We repeated each scenario ten times and computed the average. Furthermore, we investigated the impact of web page size so we conducted a scenario for two different size web pages, namely P = small, medium whereas we used the large size web page for the other scenarios [47]. However, all objects of web pages are jpg images only without Cascading Style Sheets (CSS) nor javascript files to eliminate the impact of processing as shown in the Table 3.2. Web pages have higher number of small size objects because QUIC loads small size objects fast [48]. Consequently, these parameters define 45 different scenarios. We used the developer tool of the Chrome browser which measure the time elapsed since requesting a web page until the page is fully loaded. We disabled the caching in the Chrome browser during our experiment to fetch data from the QUIC server all the time. We had a one connection between the server and the client. Table 3.3 summarizes the different values of congestion window reduction according to the investigated values of multiplication decrease factor β.

3.4 Results

In this section we present the results of comparing PLT of QUIC under the different scenarios. The goal of this study is to investigate the impact of applying $\beta = 0.6, 0.7$ on the performance of QUIC protocol. We mention that, other values of β resulted in an ordinary behavior such that PLT is better when β is

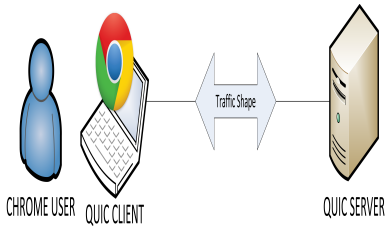


Figure 3.2: Testbed

Table 3.1: Parameters used in the evaluation

Parameter	Value
Bandwidth – B	100, 10 Mbps
RTT – D	5, 10, 200, 400 ms
Packet Loss – L	0.2%, 0.5%, 1%, 2%, 5%
Buffer length – Q	$RTT * BW$
multiplication decrease factor – β	0.6, 0.7

Table 3.2: Web page structure

Web page size - P	Number of objects		Size of an object in KB
Large: 3.3 MB	Small	153	15
	Large	8	135
Medium: 1 MB	Small	50	15
	Large	2	135
Small: 300 KB	Small	11	15
	Large	1	135

Table 3.3: Values of multiplication decrease factor

β	Congestion Window Reduction	Description
0.7	15%	QUIC default multiplication decrease factor
0.6	20%	5% larger than QUIC's default congestion window reduction

Table 3.4: Scenarios used in the evaluation

Scenario	Goal
Delay impact	Presents the impact of β with respect to different values of D and L
Buffer impact	Presents the impact of β with respect to different values of Q, $2 * Q$, $Q/2$ and L
Page size impact	Presents the impact of β with respect to different sizes of P and L

larger than 0.7 and it is worse when β is smaller than 0.7 as also found in previous publications [43] [44] except for $\beta = 0.6$. Therefore, we present only the interesting results yielded by applying $\beta = 0.6$ which represent a unique behavior. We compute the percentage of PLT change, as shown in Eq. 3.4, obtained by $\beta = 0.6$ with respect to $\beta = 0.7$ which is considered the reference value of the comparison due to the fact that it is the default value of β of QUIC. By tracking the percentage of the PLT change, we evaluate the impacts of different β values. All the following figures show percentages of the PLT change (PLTC) which have been computed by Eq. 3.4.

$$PLTC = \frac{PLT_{0.7} - PLT_{0.6}}{PLT_{0.7}} \times 100 \tag{3.4}$$

According to Eq. 3.4, the scenarios which have positive PLTC indicate that QUIC with $\beta = 0.6$ has better PLT by a value equal to the associated percentage in comparison to $\beta = 0.7$. On contrary, QUIC has better PLT with $\beta = 0.7$ in case of negative PLTC in comparison to $\beta = 0.6$. We considered three different types of scenarios as shown in Table 3.4 where we did not consider all possible combinations of parameters listed in Table 3.1. Each scenario has been applied five times to evaluate the impact of β under different rates of packet loss. However, we did not consider $L = 0\%$ due to the fact that the congestion control algorithm reduces the congestion window size when a loss event occurs only; furthermore, we introduced the packet loss by TC in the direction from the QUIC server to the QUIC client only to avoid the timeout behavior in case of acknowledgement packets are lost. Due to the fact that data flow moves from the server side to the client side only in our experiment, we applied values of β on the server side only.

3.4.1 Delay impact

In this section, we evaluate the impact of β on PLT of QUIC under different RTT values. We considered $B = 100$ Mbps, $D = 200$, 400, 10, 5ms, and $P = 3.3$ MB. Each combination of D, B, P, β has been applied five times to evaluate the PLT under all different values of L. Buffer size has been set to bandwidth-delay product in this scenario. Figure 3.3 presents PLTC when $\beta = 0.6$ in comparison to $\beta = 0.7$ under different loss rates. Figure 3.3 shows that all PLTC of QUIC with $\beta = 0.6$ are positive in case of RTT =

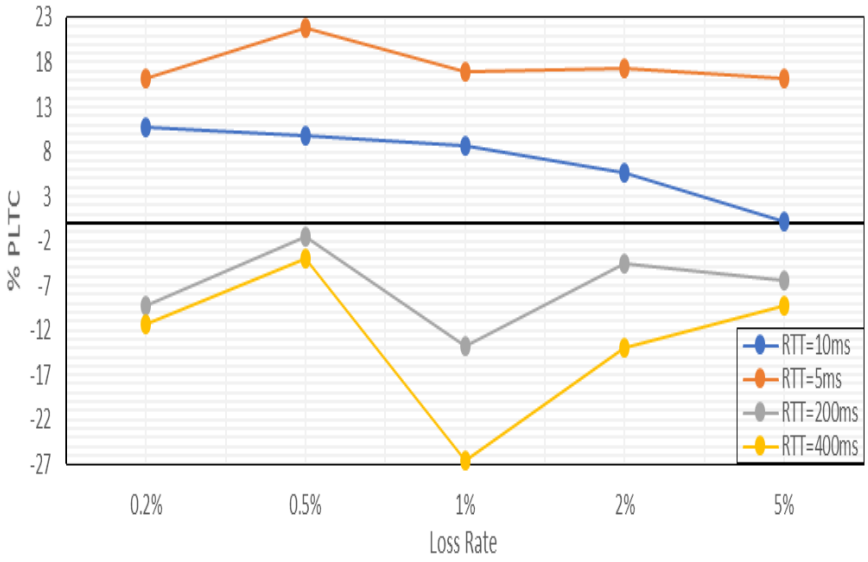


Figure 3.3: PLTC with respect to delay

10ms. This is due to the fact that when the network is congested, packet pacing estimates a larger data transmission rate in case of low RTT and the congestion window reduction is 20% than that in case of the congestion window reduction is 15%. This behavior has been approved by repeating same scenario but with smaller delay, as shown in Figure 3.3, where RTT has been set to 5ms resulting in PLT better than that when RTT = 10ms. To investigate the impact of high RTT, we performed the scenario with high RTT = 200 ,400ms as shown in Figure 3.3 which shows consistent results that PLT becomes higher than that with $\beta = 0.7$ when RTT is high. Consequently, in case of $\beta = 0.6$, the PLT will be better up to 21.9% when RTT is 5ms and up to 10.73% when RTT is 10ms than that in case of $\beta = 0.7$.

3.4.2 Page size impact

In this section we compute PLTC of $\beta = 0.6$ in comparison to that when $\beta = 0.7$ to investigate the impact of web page size in both cases under different loss rates. We considered $B = 10$ Mbps, $D = 10$ ms and P = Medium, Small. We did not consider the large web page for this scenario because it has been applied in the other scenarios. Figure 3.4 presents the PLTC in case of $\beta = 0.6$ under different loss rates in comparison to $\beta = 0.7$. It consists of two curves for the two different web page sizes. Figure 3.4 shows that $\beta = 0.6$ provides better PLT than the case when $\beta = 0.7$ under all loss rates for medium web page. This mainly due to the fact that when the network is lossy, smaller reduction in congestion window causes more packet loss when more data needs to be transferred and consequently the overall situation will be worse. Figure 3.4 shows that, when medium size web page needs to be transferred in a

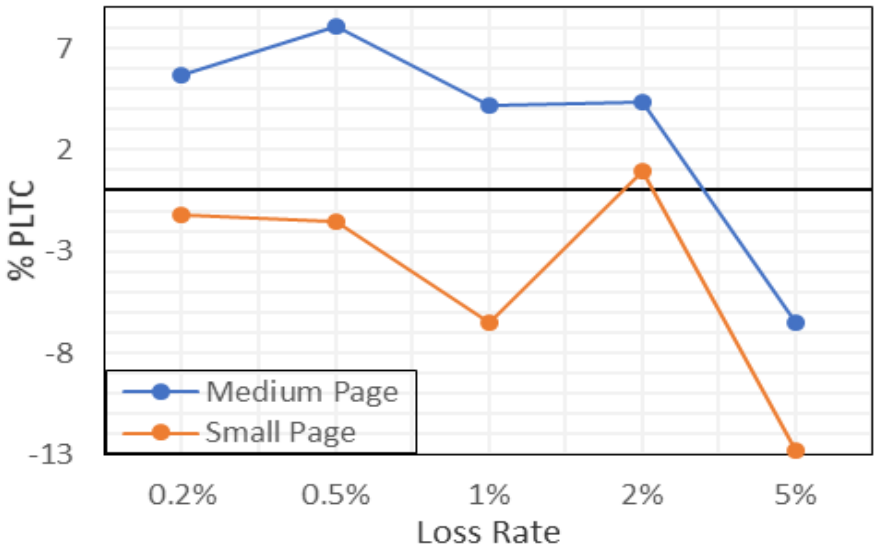


Figure 3.4: PLTC with respect to page size

lossy network, the PLT can be reduced to 8% by applying $\beta = 0.6$ in comparison to $\beta = 0.7$.

3.4.3 Buffer impact

In this section, we measure the impact of buffer size Q when applying two different values of β. We considered $B = 100$ Mbps, $D = 5$ ms and $P = 3.3$ MB for this scenario. Three different buffer sizes have been considered $Q = 64$, 128, 32KB with Token Pocket filter. In particular, for $Q = 64$KB the network is well-buffered, for $Q = 128$KB the network is over-buffered and for $Q = 32$KB the network can be considered under-buffered. We applied this scenario five times to consider the different values of loss rate, as shown in Figure 3.5 which presents the impact of different buffer sizes. The main observation is that when $\beta = 0.6$, QUIC needs less time to load the large web page than that in case of $\beta = 0.7$ regardless of the buffer size. It is due to the fact that we have a one connection between the QUIC client and the QUIC server so there are not a lot of data to be transferred; consequently, the receiver's buffer cannot be overwhelmed. The results of this scenario consistent with the results of previous scenarios where the delay is low and the size of web page is large (i.e. the amount of data needed to be transferred is not small) which yields better PLT by a value between 12% - 22% in case of $\beta = 0.6$ in comparison to $\beta = 0.7$ under different loss rates.

Furthermore, we summarize our measurement results in Table 3.5 where we present the count of the positive PLTC under the five loss rates of each scenario which implies that QUIC with $\beta = 0.6$ has better PLT in comparison to that one with $\beta = 0.7$. Table 3.5 shows that QUIC with $\beta = 0.6$ can provide

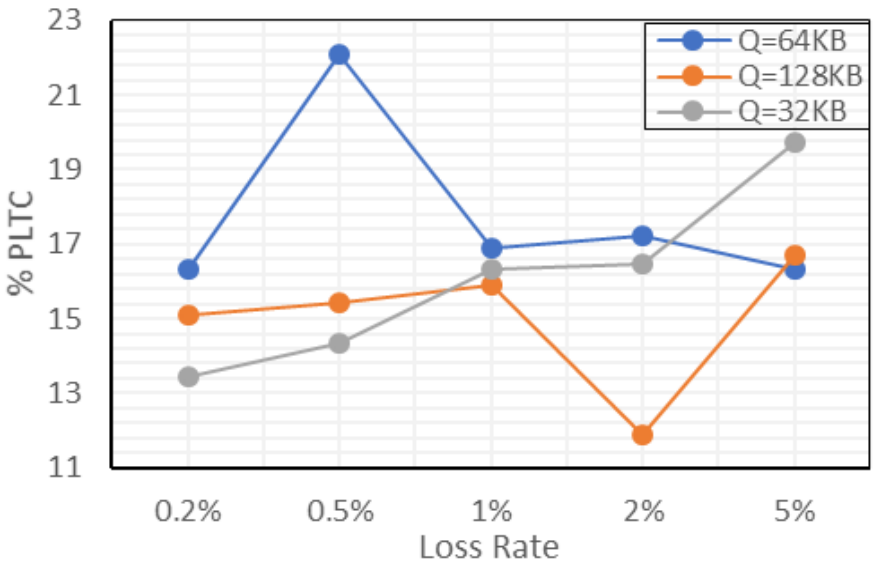


Figure 3.5: PLTC with respect to buffer size

better PLT under different loss rates in particular when RTT is low and the size of web pages need to be downloaded is not small. However, Table 3.5 shows that QUIC with $\beta = 0.6$ has better PLT in most of the cases even that it has larger window reduction.

Table 3.5: Number of positive changes in all scenarios

Scenario	Scenario parameter	Number of cases in which PLT is improved
Page Size	medium	4/5
	small	1/5
Delay	200ms	0/5
	10ms	5/5
	400 ms	0/5
	5 ms	5/5
Buffer Size	well	5/5
	over	5/5
	under	5/5
Total Number		30/45

3.5 Summary

In this chapter, we presented an experimental investigation of the impact of the multiplication decrease factor used in CUBIC congestion control algorithm applied by QUIC. Under different network conditions, we have measured the percentages of page load time changes. This study shows that page load time can be improved in case of using a reduction rate different from the default value upon congestion events.

Chapter 4

Flow scheduling frameworks in SDN based data center networks[1]

Data center topologies such as Fat-tree provide many redundant links between network switches in different layers what yields multi-paths between any source and destination. Flows through a bottleneck link towards a destination can yield congestion events. In this chapter, we present two "In-network" congestion control solutions. The first solution is scheduling QoS-based solution and the second is scheduling/rerouting solution. However, both solutions aim to improve FCT of mice flows and maintain the throughput of elephant flows in DCN as possible.

4.1 Introduction

Nowadays, many enterprises employ various services and applications by leveraging data center networks. As a result, the associated flow patterns impose new considerations. In particular, the majority of flows generated by applications like MapReduce send less than 10 KB of data and lasting for less than 10 seconds, and similar characteristics are obtained in case of web services [7]. In this context, the proliferation of IoT's prospective applications may boost these characteristics since the flow size of such applications are small [49].

Typically, applications in DCN generate two classes of flows which are mice and elephant flows. Mice flow is the smallest and shortest-lived flows, and it is more conservative to the communication delay. On the other hand, the largest and longest-lived flows (i.e., elephant flows) are more affected by the available bandwidth. The elephant flows are fewer than mice flows, but they carry most (e.g.,

[1]This chapter is based on published papers in [J2] [C2] [C3]

80%) of the transferred data [5] [6] in DCN. Therefore, these classes compete for network resources. In this context, SDN paradigm provides opportunities to improve network management and control. The separated control plane provides an effective resource handling in comparison with the traditional networks. Besides, OpenFlow protocol [18] is one of the primary protocols to provide communication between the SDN controller in the control plane and the switches in the data plane. In today's DCN, SDN plays a vital role in network resource allocation and traffic monitoring. Hence, the paradigm has been significantly applied by the research community for flow scheduling and traffic load balancing [20], [50]. The process of managing massive data transmissions in real-time requires an efficient resource and traffic management [7]. The network topology in DCN offers multi routes between every pair of end-hosts. Hence, identifying the best route avoiding the potential congestion events is challenging. In particular, such kind of congestions would profoundly degrade QoS of mice flows. Thus, deploying cloud-based applications in DCNs, which leverage the static hashing based scheduling such as ECMP, is not an efficient mechanism due to the probable packets collisions. Furthermore, employing network devices (e.g., hosts or switches) for flow scheduling is still challenging since it requires some modifications in the kernel or sometimes in the hardware. On the other hand, the fully central flow scheduling model yields overhead on the control plane.

In this chapter, we propose two frameworks to improve FCT of mice flows in DCN by leveraging of SDN paradigm without dramatically degrading the throughput of background flows. Furthermore, our frameworks can provide better service for mice flows by eliminating the delay resulted from contacting end-hosts upon congestion events. Besides, our frameworks do not require any modification in TCP stack nor switch hardware.

4.2 Related works

In this section, we present the literature about SDN-based solutions for flow scheduling. Most of the current flow scheduling solutions are either fully central like Hedera [4] or distributed like CONGA [51]. Furthermore, some works are intended to involve network devices (e.g., hosts or switches) in flow scheduling decision. However, these works are still challenging to be deployed in today's DCN. For example, Mahout [52] requires modifications in the kernel of servers to detect elephant flows, whereas some other works employ end-host like in [53] to calculate the transmission delay. The works in [51] and [54] add extra functions to the switches for sampling and delay calculations. Hedera [4] reschedules a flow when its consumption exceeds 10% of the link capacity. Besides, Hedera schedules mice flows by hashing-based ECMP only. Mahout [20] employs end-hosts in DCN to detect the elephant flows. However, deploying such a method yields overhead due to the contacting with the end-hosts. Yazidi et al. [55] propose hot and cold link detection mechanisms to reschedule elephant flows. This solution

requires to monitor demands of all flows. Tang et al. [56] propose a classification and a scheduling model for flows. The solution identifies the flows by tracking their inter-arrival times. The work in [57] presents a proactive method for flow scheduling by estimating an alternative path based on the port utilization, but monitoring every flow results in overhead in the control plane. The work in [58] proposes a flow scheduling algorithm in SDN-based DCN where it schedules aggregated elephant flows whose consumption is more than 10% of the link capacity. However, this study provides no investigation in terms of FCT of mice flows. Fu et al. [59] proposes a balancing flow table scheme to mitigate FCT of mice flows by computing the best path for each new mice flow, which incurs overhead since the number of mice flows is big in DCN.

A dynamic routing and rate limit system has been proposed in [60] to find non-congested routes and dynamically reroute flows upon detection the congestion on any switch port based on SDN paradigm as well as to define prioritized rate limits. Other SDN based solutions fulfill QoS requirements by setting data rates [61] [62]. However, no distinction between mice and background flows has been proposed in the previous solutions. The solution presented in [63] can avoid the congestion in private campus networks by creating many queues on switch ports and finding paths based on SDN paradigm such that satisfying the bandwidth requirements without exceeding the capacity of the links. Authors in [64] define QoS requirements for all applications in data center network to create prioritized queues on switch ports such that packets are dequeued out of ports based on their priority without exceeding the bandwidth of the ports to avoid the congestion.

4.3 First solution: QoS based flow scheduling framework in SDN based DCN

Our framework stems from the fact that end-to-end solutions that rely on window management can be far from effective, requiring many RTTs to react properly to congestion by end hosts. Also, the solutions rely on rerouting do not provide the satisfied guarantee for mice flows. Finally, some solutions require modifications to TCP stack and need time for a proper response more than the lifespan of mice flows in data center networks. Therefore, we introduce the architecture and functionality of our framework. The terminologies and variables used by the solution are presented and described in Table 4.1 and Table 4.2 respectively.

4.3.1 Framework architecture

Our mice flow framework aims to guarantee the required bandwidth for mice flows by reducing the data rate of elephant flows under high load epochs so that mice flows will be served with less delay. To control the data rate of elephant flows, queues have been created on each switch port, dedicated for elephant flows, as many as egress ports a downstream switch has as well as one more queue has been created on each port dedicated for mice flows. elephant flows in an upstream switch will be mapped to one of elephant queues based on their destination and all mice flows will be mapped to the mice queue. Figure 4.1 shows flows forwarded out of an egress port on the upstream switch to a port on the downstream switch. As a result, the data rate of the flows will have a direct impact on the queue length of egress ports on the downstream switch.

The solution checks the length of mice queues so that the data rate of background flows forwarded through the corresponded mapped queues on upstream switches will be adapted accordingly. The architecture of the framework is shown in Figure 4.2. We can see that the solution employs an SDN controller, where the control plane of SDN paradigm comprises the queue monitoring module, elephant flow selection module, and data rate control module. By using the SDN paradigm, the previous modules can control and monitor the data plane. The framework considers the main idea which is that the delay of mice flows in data center networks is yielded by the existence of elephant flows. Therefore, our framework proposes a new approach to deal with this problem by applying QoS based solution on upstream switches instead of notifying data sources due to the fact that such kind of notifications takes time longer than the lifespan of mice flows. For the sake of immediate reaction to any probable degradation in FCT of mice flows, our framework updates the data rate of the corresponding elephant queues on upstream switches commensurate with the length of mice queues on downstream switches.

4.3.2 Queue monitoring

Centrally in the control plane of SDN paradigm, Queue monitor module periodically probes the length of queues over a varied interval. Queue monitor module is based on the observation that more than 80% of flows in data center networks are mice flows, and their size less than 10KB [5]. According to this fact, we have defined the congestion threshold as a function of the number of egress ports. To mitigate the overhead, queue monitor module probes the length of the mice queue on switch ports over intervals proportionally vary to the length of mice queue such that the probing interval will be longer as the length of mice queue gets shorter. Algorithm 1 shows the steps of the queue monitor module.

Table 4.1: Terminologies

Terminology	Description
Upstream Switch	a switch from which a traffic flow is forwarded to its downstream switch
Downstream Switch	a switch receives a traffic flow from its upstream switch
Egress Port	any switch port out of which the arrived traffic on another port will be forwarded
mice queue	a queue dedicated for mice flows
elephant queue	a queue dedicated for elephant flows forwarded to a specific egress port on a downstream switch

4.3.3 elephant flow selection

Our elephant flow selection module leverages sFlow [65] to identify elephant flows. In addition, this module maintains an active elephant flows table that includes all active elephant flows forwarded out a specific port. Since our framework needs to maintain the required information to define elephant flows uniquely, each table entry comprises of *src_ip, dst_ip, src_port, dst_port, port.* elephant flow selection module adds a new entry to the table when the flow volume is larger than 10 KB. When the length of the mice queue is longer than the threshold, this module will retrieve all entries for background flows forwarded out a specific port. Whenever the length of the mice queue is less than the threshold, all entries associated with that port will be deleted. Furthermore, to maintain active elephant flows only in the table, all entries of idle flows will be deleted whenever their connections end. Algorithm 1 presents elephant flow entry insertion and deletion operations.

4.3.4 Data rate control

After adding entries into the active elephant flow table, data rate control module employs QoS capabilities of the SDN controller to regulate the data rate of elephant queues on the upstream switches in accordance with the length of elephant queues and that of the mice queue on an egress port in a downstream switch. More specifically, a switch port is overloaded if the received traffic rate is more than its link bandwidth. As a result, its queue will grow proportionally to the difference between the received traffic rate and link bandwidth. Therefore, mice flow framework mitigates the received traffic rate on a port by reducing the data rate of elephant queues on upstream switches to guarantee fast transmission for micro flows. Therefore, elephant flows will be forwarded out of an egress port on an upstream switch by a rate directly

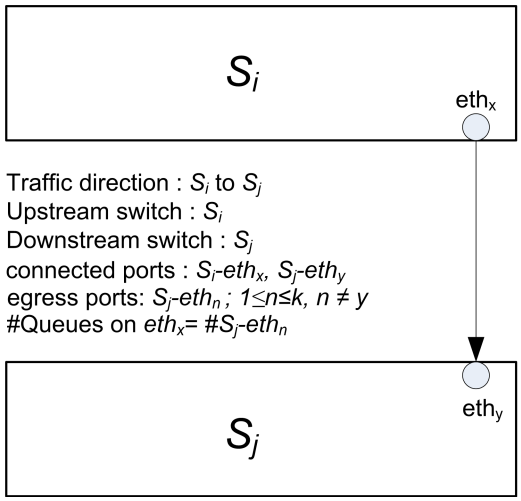


Figure 4.1: Parameters of two connected switches

proportional to the length of its elephant queue and inversely proportional to the length of the mice queue on the egress port of the downstream switch. The data rate of elephant queue on upstream switch port is computed as presented in Algorithm 1 where α and β are control parameters proportional to the length of the mice queue on the downstream switch and that of the elephant queue on the upstream switch port, respectively. Consequently, the data rate of elephant queues on ports of upstream switches will be mitigated up to 50% when the length of the mice queue on egress port of the downstream switch is about to have congestion event as well as no backlog in the elephant queue on upstream switches. In contrast, it cannot be mitigated when the length of a elephant queue is larger than or equal to the threshold. Figure 4.3 illustrates the work-flow of the framework. The framework was implemented in python to run along with any SDN controller can provide Rest API. We integrate our solution with Ryu controller [66].

4.3.5 Experimental results

We evaluated the performance of our framework via Mininet 2.3.0. In this section, we present the results of the experimental evaluation of mice framework where we compared the performance of TCP new Reno and TCP Cubic with drop tail Active Queue Management (AQM) enabled to that with mice flow framework. However, TCP Cubic and TCP new Reno use packet loss to detect network congestion, but TCP Cubic is less aggressive than new Reno and it is used by default in Linux kernels. For proper configuration, we enabled Selective Acknowledgment (SACK) for all scenarios, we set minimum Retransmission Timeout (RTO) to the default value of Linux which is 200ms, IP packet size to 1500 Bytes, and the buffer size to 36 packets (54 KB). We used fat-tree topology whose scale k=4 similar to the one proposed in [4]. Each core switch connects to four aggregation switches with 100 Mbps bandwidth and 250 μs one-way propagation delay links, 10 Mbps and 2 ms one-way propagation delay for links

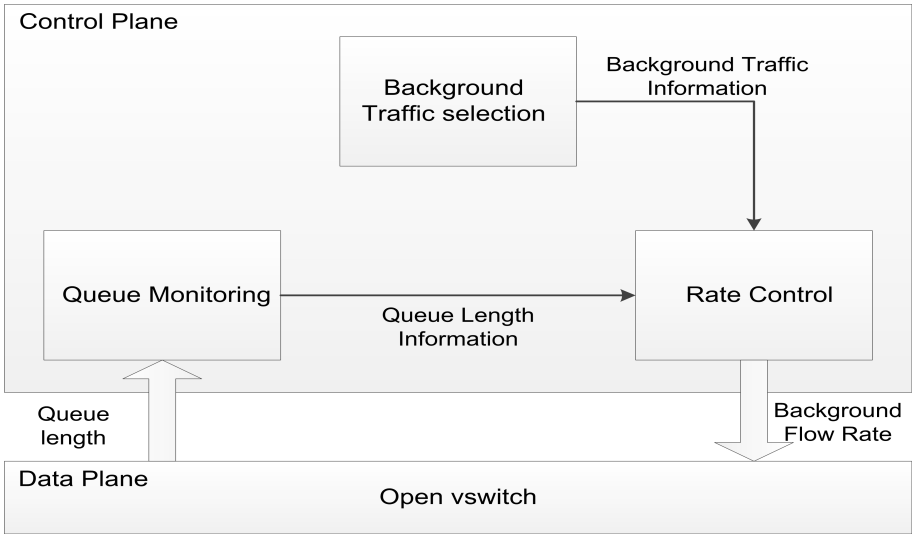


Figure 4.2: Architecture of mice flow framework

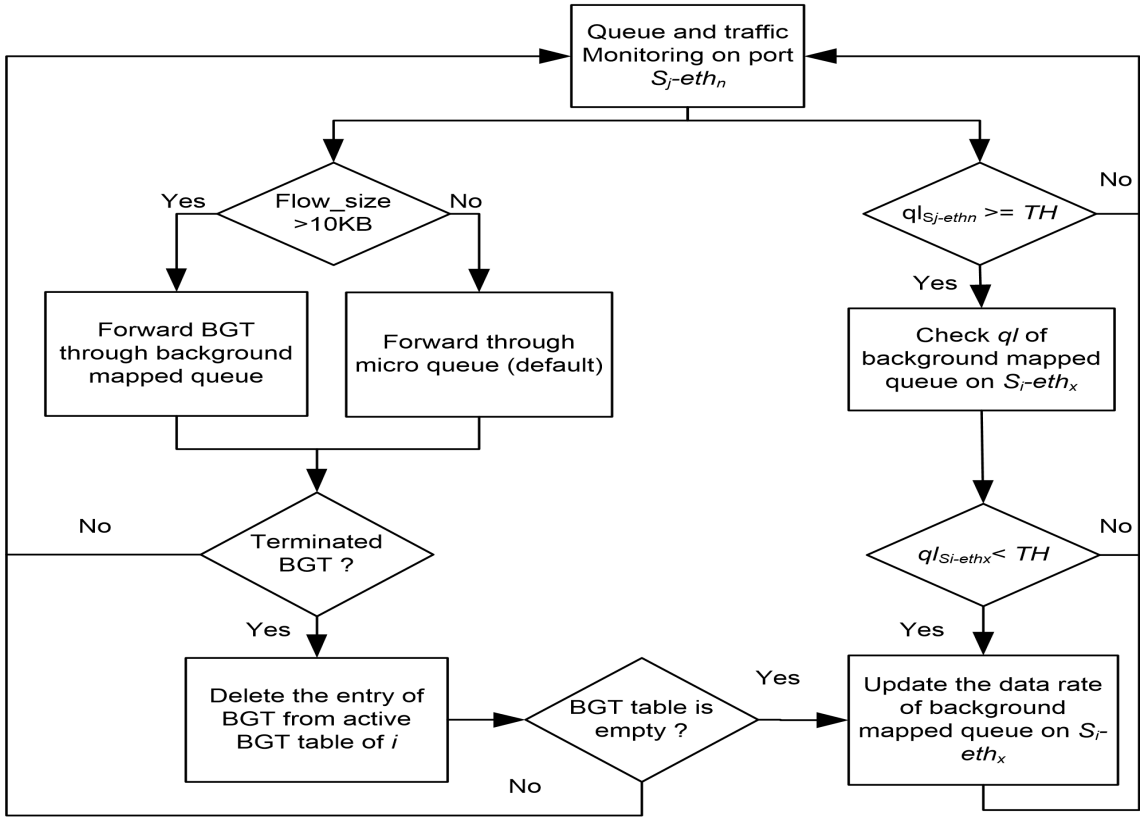


Figure 4.3: Framework workflow

Table 4.2: Used variables

Variable	Description
ql	Queue occupation
p	Number of switch ports
k	The scale of Fat-tree topology
qc	Queue capacity
BDP	Bandwidth delay product
$interval$	Probing queue length frequency
bgt_size	The threshold to identify elephant traffic
$timeout$	The maximum time of applying updated data rate
$s_a - eth_b$	The Ethernet port number b on switch a
$s\text{-}eth$	Any Ethernet port on any switch
pkt	The packet size
TH	The threshold of micro queue occupation
$flow\_size$	The size of a flow
bgt_flows	The set of all identified elephant flows
bgt_list	The active elephant flows table
α	the control parameter of mice queue length on downstream switch
β	the control parameter of elephant mapped queue length on upstream switch
TR	The data rate of elephant mapped queue on upstream switch
BW	Bandwidth of egress port on a downstream switch

connect aggregation and edge layers, and 5 Mbps and 5 ms one-way propagation delay for links connect edge switches and end-hosts where each edge switch has two servers. Therefore, the over-subscription ratio is 1:1 at edge level. We evaluate the performance of the framework by run 10 seconds simulations for three different scenarios contain a mix of 128 micro and elephant flows. In all scenarios, elephant and mice flows follow two patterns, the first for generating connections span all topology layers, and the second for connections span the edge and aggregation layers where all servers in all racks have been employed to generate the traffic volume for each scenario. We employed iperf for generating unlimited elephant traffic during the whole simulation period and Apache server [67] for generating mice flows by repeatedly requesting "index.html" webpage of size 10 KB as reported in [5] at the beginning of each two seconds during the simulation period. Therefore, the solution will be examined in a situation where mice flows are synchronized to generate burstiness, and elephant flows exist to evaluate the framework

under high load. We repeated the experiment 50 rounds for three different scenarios and the average throughput of elephant flows has been computed for each round while the average flow completion time of mice flows has been computed for each two seconds epoch. The details of three scenarios are as follows, in the first scenario, we simulated a elephant-to-mice ratio of 1:3 (i.e., 32 elephant and 96 mice flows) as the ratio reported in [5] [2]. In the second scenario, we employed higher elephant flows share of 3:1 to investigate the impact of our framework under a high volume of elephant traffic. In the third

---

Algorithm 1 Framework Functions

initial: $ql = 0$, $p = k$, $pkt = 1500$B, $qc =$ BDP, $interval = 0$, $bgt_size = 10$KB, timeout = 100ms

---

Function *Queue_Monitor*( )

repeat per *interval*:

$TH = (qc_{s_j-eth_n}$ - $(p$ - $1)$ * $pkt)$

  if $(ql_{s_j-eth_n}(t) \geqslant TH)$

    Datarate_Update (s_i- eth_x, $ql_{s_j-eth_n}$)

  endif

  $interval = \dfrac{TH}{ql_{s_j-eth_n}} * RTT$

Function *Find_BGT( )*

  if (*flow_size* > *bgt_size*):

    bgt_flows = requests.get(*s-eth*)

    for *m* in *bgt_flows*

      bgt_list[*m*] = (*ip_src, ip_dst, port_src, port_dst*)

      map *bgt_list[m]* to a queue

  for *m* in *bgt_flows*

    if *bgt_list*[*m*] **NOT IN** *bgt_flows*:

      del (*bgt_list*[*m*])

Function *Datarate_Update*(s_i-eth_x, $ql_{s_j-eth_n}$)

  $\alpha = \dfrac{ql_{s_j-eth_n}(t) - TH}{qc_{s_j-eth_n} - TH}$

  if $(ql_{s_i-eth_x} < TH)$:

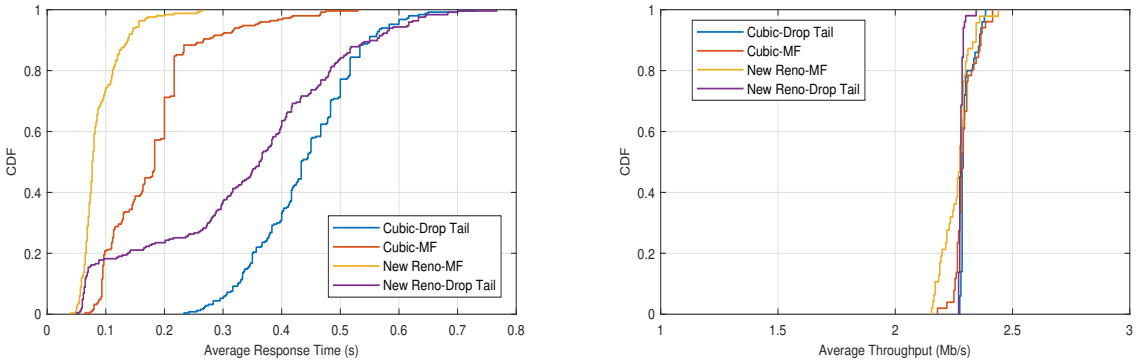
    $\beta = \dfrac{qc_{s_i-eth_x} - ql_{s_i-eth_x}}{qc_{s_i-eth_x}}$

  else:

    exit

  $TR = BW_{s_j-eth_n} \times (1 - \dfrac{\alpha \times \beta}{2})$

  change TR on s_i-eth_x for *timeout*



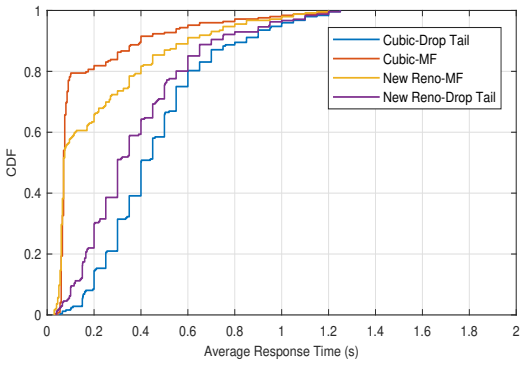
(a) Average FCT for mice flows

(b) Average throughput for elephant flows

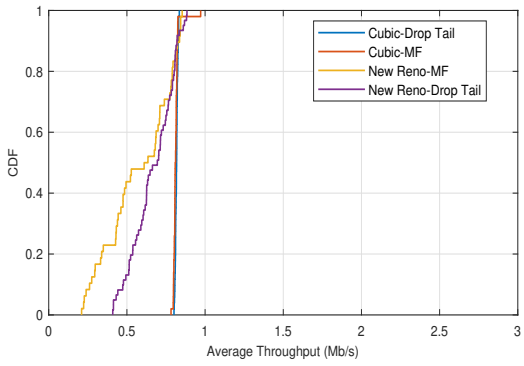
Figure 4.4: Performance of mice framework (MF) for TCP Cubic and TCP New Reno in 1:3 ratio scenario

scenario, we run a ratio of 1:1. For mice flows, we present Cumulative Distribution Function (CDF) of average FCT. These results are shown in Figure 4.4 for the 1:3 scenario, Figure 4.5 for the 3:1 scenario, and Figure 4.6 for the 1:1 scenario. For elephant flows, we essentially present CDF of the average throughput.

In 1:3 scenario, Figure 4.4a shows that mice flow framework (FM) can improve FCT of mice flows under high load of mice flows compared to both Cubic Drop Tail and New Reno Drop Tail. This mainly due to the fact that a high load of mice flows bloats the buffers which worsens their completion time. Figure 4.4b shows that our solution nearly does not aggressively impact on the throughput of background flows under the high ratio of mice flows due to the fact that when threshold hits, there is not a high load of elephant flows; therefore, the data rate of elephant flows will be proportionally reduced. In 3:1 scenario, Figure 4.5a shows similar results to the previous scenario under high ratio of background flows compared to both Cubic Drop Tail and New Reno Drop Tail because of buffers bloating which worsens the completion time of mice flows. Also, Figure 4.5b shows that the framework does not degrade the throughput of elephant flows due to that there are not so many threshold hits during this scenario. Finally, similarly Figure 4.6a shows that our framework improved FCT of mice flows under the balanced ratio. Furthermore, Figure 4.6b shows the same results as before. As a result, our solution maintains a low impact on the throughput of elephant flows because it temporarily mitigates the data rate of elephant flows which will be restored after a very short period equals at most to the transmission time of micro flow.

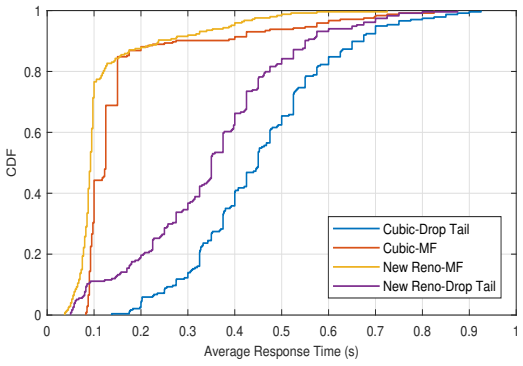


(a) Average FCT for mice flows

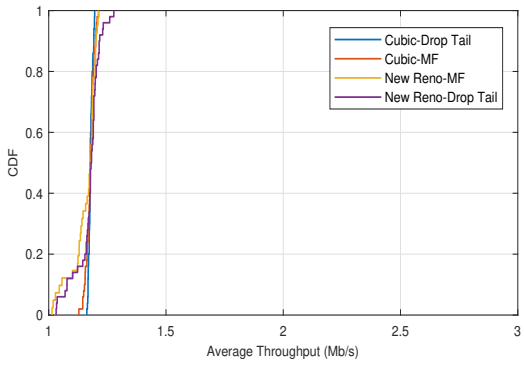


(b) Average throughput for elephant flows

Figure 4.5: Performance of mice framework (MF) for TCP Cubic and TCP New Reno in 3:1 ratio scenario



(a) Average FCT for mice flows



(b) Average throughput for elephant flows

Figure 4.6: Performance of mice framework (MF) for TCP Cubic and TCP New Reno in 1:1 ratio scenario

4.4 Second solution: scheduling/rerouting flow scheduling framework in SDN based DCN

4.4.1 Preliminary

Our solution (Sieve) is different from other solutions since it is distributed, but it does not yield the overhead resulted from end-hosts contacting. Furthermore, it does not require any modifications in kernel nor hardware. In addition, our solution mitigates the overhead due to its functions are applied on a portion of flows only.

The framework architecture is depicted in Figure 4.7. Basically, the proposed framework aims to guarantee QoS of mice flows by classifying network flows so that mice flows can be served with less delay. We propose three layers framework as depicted in Figure 4.7. The first layer resides in the data plane. It employs OpenFlow bucket group feature to provide packet sampling and ECMP-based scheduling. The second layer resides in the control plane. It contains the required functionalities to schedule the sampled flows using *shortest-path available-bandwidth* mechanism. The last layer resides in the control plane also. It contains the polling technique and elephant flows rescheduling functions.

In this subsection, we propose a flow scheduling algorithm that achieves the following objectives:

1. Propose a heuristic, adaptive, and scalable scheduling algorithm to schedule elephant and mice flows. The proposed algorithm is based on the available bandwidth to mitigate FCT of mice flows and to maintain throughput of elephant flows.

2. Propose a balanced scheduling scheme which divides the flow scheduling burden between our solution and ECMP.

3. Propose a flow detection technique to provide enough information about a portion of network flows.

Our solution attempts to resolve two dilemmas:

1. Polling information about every flow provides full flow visibility on the controller, but overwhelms the control plane.

2. Flow scheduling based on ECMP only provides rapid scheduling, but prevents any scheduling optimization by control plane.

In the following, we describe the proposed framework modules in more details.

4.4.2 Flow sampling

Our solution employs weighted bucket group feature of OpenFlow to sample flows as well as to mitigate the overhead on the control plane. In practice, the data plane does not send every packet to the controller,

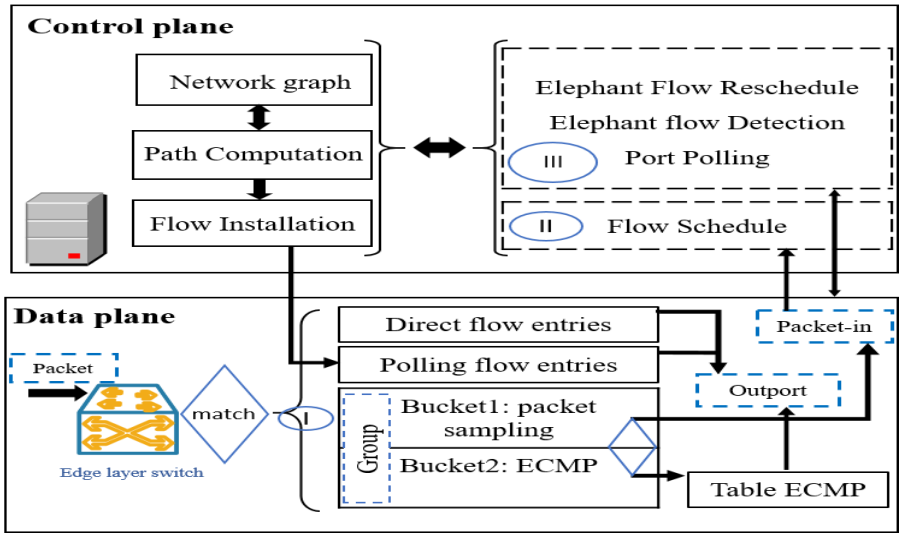


Figure 4.7: The proposed framework architecture

but rather the first packet of a flow (i.e., *packet-in*) is either scheduled based on ECMP or sent to the controller. We implement this mechanism using a group entry consists of four fields:

1. *Group identifier:* It is 32 bits integer value used as a unique identity.
2. *Group type:* We use "select" group type, so the switch executes one bucket's action based on the value of packet header hashing and bucket's weight value.
3. *Counter:* Represents the number of packets matched by the group.
4. *Action bucket:* An action associated with a specific bucket and applied to the matched packets.

Recall that the first packet of a flow did not match the direct nor polling flow entries on an edge switch, as shown in Figure 4.8, so it will be handled according to the sampling group entry presented in Table 4.3. Since the type of group is select, the switch chooses one bucket of actions based on its weight. In particular, the switch will hash the packet header information; then, based on the hash value and the values of buckets weights, one bucket will be selected to apply its associated action. Specifically, the switch will either forward the packet to the controller or to ECMP table with probabilities of 0.5 and 0.5, respectively. Our framework samples flows only from edge switches, so only edge switches contain the sampling group entry presented in Table 4.3. Specifically, edge layer switches contain two flow tables which are Table 0 shown in Figure 4.8 and Table ECMP contained ECMP-based scheduling group entries as shown in Figure 4.7. This partial sampling saves the controller from overwhelming samples load. Our framework treats the sampled packets equally; regardless they belong to elephant or mice flows.

Table 4.3: Sampling group entry

group_id	group_type	bucket_action
group_id=1	select	bucket=weight:50,actions=CONTROLLER, bucket=weight:50,actions=GOTO_TABLE:ECMP

Table 4.4: ECMP group entry

group_id	group_type	bucket_action
group_id=1	select	bucket=weight:50,actions=OUTPORT:1, bucket=weight:50,actions=OUTPORT:2

Flow Entry Type	Fields
Directly connected hosts	*ip_dst, action:outport*
Polling and Scheduling	*Ip_src, ip_dst, transport_src_port, transport_dst_port, action:outport*
Sampling	*bucket1, any, CONTROLLER, bucket2, any, GOTO_TABLE:ECMP*

Figure 4.8: Flow entry types in Table 0 on edge switch

4.4.3 ECMP-based scheduling

We utilize ECMP in scheduling a portion of flows since ECMP represents a fast scheduling technique. Therefore, we can avoid flow collisions which could happen in case of sole dependence on ECMP. In particular, we implement ECMP-based scheduling by proactively defining bucket group with equal weights into all switches of edge and aggregate layers. ECMP group entry is presented in Table 4.4. Specifically, our solution employs ECMP to schedule the flows forwarded out switches ports 1 or 2, on edge layer connected to the aggregate layer and on aggregate layer connected to the core layer only, based on packet header hash value and buckets weights values. On the other hand, our solution uses proactively defined flow entries for directly connected hosts and subnets for forwarding the flows from an upper layer toward a lower layer. Besides, polling flow entries are used for scheduling flows in either direction. Figure 4.9 presents the flow entries effective directions. However, different priority values are used to enable preferential scheduling. As a result, edge layer switches contain two flow entry tables, whereas one

Table 4.5: Flow tables in different switch layers and flow entry types with an indication whether they are proactively predefined or reactively defined

Flow Entry Type	Table_id	Switch_layer	Proactive/Reactive
Directly connected host	0	edge	proactive
Polling and Scheduling	0	edge/agg/core	reactive
Directly connected subnet	0	agg/core	proactive
ECMP-based scheduling	0	agg	proactive
Sampling	0	edge	proactive
ECMP-based scheduling	ECMP	edge	proactive

flow entry table exists into all other switches. The proactively and re-actively installed flow entries are depicted in Table 4.5 along with the table id, location and their types. Table 4.5 is descendingly sorted based on priority values of the flow entry types according to each table_id.

4.4.4 Flow schedule

Upon receiving *packet-in* at the control plane, our solution parses the packet header to retrieve the connection information. Then, *Path Computation* module will be invoked to find the four shortest paths between the source and destination, which has the best available bandwidth as well. As shown in Algorithm 1, *bottleneck_of_path* function will be invoked for each shortest path in *shortest_p*, which

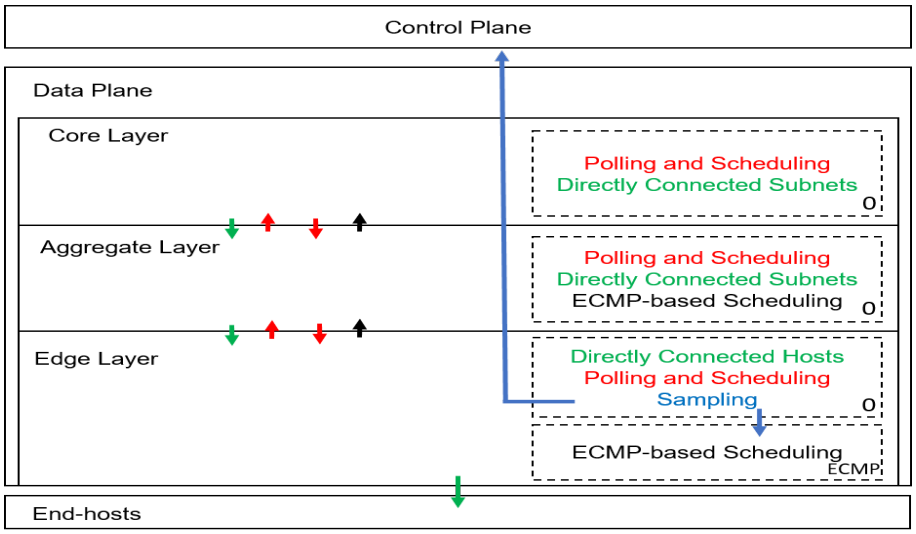


Figure 4.9: Flow entry effective forwarding directions where each flow entry type is created to forward traffic in dedicated directions across the Fat-tree DCN layers and end-hosts as well

contains information of the four shortest paths, to compute its bottleneck link, which is the highest loaded link. Then, the path with maximum bottleneck bandwidth will be the best one, *best_p*, to schedule the new flow onto. Finally, this module sends the connection information and the best path to *Flow Installation* module, which installs a new polling flow entry into all switches along the chosen path. In this layer, our solution serves new flows similarly regardless they are mice or elephant flows.

4.4.5 Port polling

Polling_stat function in Algorithm 2 periodically polls port statistics every PR from all switches in the network by sending OpenFlow *OFPPORTStats* message. Therefore, the framework has full available-bandwidth visibility. Our framework saves this information on a directed graph. Furthermore, port information used by it to reschedule elephant flows by comparing the occupied bandwidth of edge switch ports, connected to the aggregate layer, with the predefined threshold. Threshold refers to the bandwidth occupation on an edge switch port in this chapter unless otherwise indicated. Specifically, when the available bandwidth on any edge switch port, connected to the aggregate layer, passes the predefined threshold $U\_BW < Th$, this module will invoke *Elephant Flow Detection* module by invoking $reschedule(i, j, U\_BW)$ function, as shown in line 6 of Algorithm 2.

4.4.6 Elephant flow detection

Our framework employs polling flow entries on edge switches to find the sent byte counts. In particular, each polling flow entry is defined uniquely, and it consists of four-tuple fields which are source IP, destination IP, source transport port, and destination transport port. In particular, upon threshold hits, our solution fetches all elephant flows forwarding out a specific edge switch port (i.e., j) and whose accumulated sent bytes is more than 50KB (i.e., $F\_load > 50KB$) as well, as shown in lines 9-12 of Algorithm 2. In this context, we consider a flow whose size more than 50KB as an elephant flow. We adopt this classification principle according to the results in [7] [5].

4.4.7 Elephant flow reschedule

This module then receives the total number of elephant flows that are to be rescheduled (i.e., num_redir_EFlows) as shown in line 13 of Algorithm 2. In addition, the information of each elephant flow, the edge switch and the port, are sent from the previous module, as presented in lines 14 - 16. Furthermore, this module tries to find a new path that the original edge switch port is not a part of it as shown in lines 18-22. Then, It invokes function *get_best_path_by_bw* to compare the bottleneck available bandwidth of the possible new paths with the available bandwidth on the edge switch port. Then, if an alternative path has sufficiently higher available bandwidth than that on the original port as in line 30,

Algorithm 1 New flows scheduling

Input: *G=(V,E)* , *src_IP*, *dst_IP*, *src_port*, *dst_port*,
min_bw = capacity, *max_bw = 0* , *k*,
shortest_p= { }
Output: *best_p*= [ ]

```
1.for i in (0, k):
2.    if shortest_p [i]= shortest_paths (G, src_IP, dst_IP)
3.for j in shortest_p:
4.    min_bw = bottleneck_of_path (G, j, min_bw)
5.    if min_bw >  max_bw:
6.        max_bw =  min_bw:
7.        best_p = j
8.    return best_p
9.Function bottleneck_of_path (G, j, min_bw):
10.    min_bw_link = min_bw
11.    for i in (0, len(j)):
12.        bw = G[j[i]][j[i+1]]
13.        min_bw_link = min (bw, min_bw_link)
14.    return min_bw_link
```

it sends the information of the elephant flow and the new path to *Flow Installation* module; otherwise, a log message will be displayed that no path met the specified conditions.

In case only one elephant flow exists on an edge switch port upon threshold hits, our framework tries to reschedule it to another path with higher available bandwidth. On the other hand, if there are many elephant flows, the framework tries to reschedule as many as of them. It proportions the available bandwidth on the original edge switch port to the number of existed elephant flows as in line 13.

4.4.8 Path computation

This module receives requests from *Flow Schedule* and *Elephant Flow Reschedule* modules. In particular, the former module needs a path for a new sampled flow upon receiving *Packet-in* message from the data plane, and the later invokes this module upon threshold hits to reschedule as possible as of elephant flows on a specific edge switch port (i.e. function *bottleneck_of_path* of Algorithm 1 and function *get_best_path_by_bw* of Algorithm 2, respectively). This module tries to find the shortest paths between *src_ip* and *dst_ip* then selects the one whose bottleneck link has the maximum available bandwidth. Therefore, this module fetches the network graph, *G*, containing the available bandwidth. Finally, it

Algorithm 2 Detect and reroute elephant flows

Input: *G= (V, E), F_BW, Th, U_BW, min_bw= Capacity, dpid_list, max_bw= 0, k, shortest_p= { }, PR, EF_list= { }, Paths= { } , Portid_list*

Output: *best_p= [ ]*

1. **Repeat** *each PR*
2. **Function** *Polling_stat (dpid_list):*
3. *OFPPortStatsRequest (dpid_list)*
4. **If** $U\_BW_{ij} < Th$: ; $i \in$ *pdid_list,* $j \in$ *Portid_list*
5. *OFPFlowStatsRequest(i)*
6. Reschedule *(i,j,* U_BW_{ij}*)*
7. **Function** Reschedule *(i, j,* U_BW_{ij}*):*
8. *E_count = 0*
9. **For** *f* **in** *F_list:*
10. **If** *F_load > 50 KB & Output_port == j*:
11. *EF_list.append(f.info)*
12. *E_count++*
13. *num_redir_EFlows= E_count* $\times \frac{U\_BW(i,j)}{Capacity(i,j)}$
14. **If** *num_redir_EFlows > 0 :*
15. **For** *f* **in** *(0, num_redir_Eflows) :*
16. *get_best_Path (G, i, f.info,* U_BW_{ij}*, num_redir_EFlows)*
17. **Function** *get_best_Path (G, i, F.info,*U_BW_{ij}*, num_redir_Eflows)*
18. **For** *P* **in** *Shortest_P:*
19. **If** *(link (P[i] P [i+1]) != j):*
20. *Paths.append (P)*
21. **else:**
22. **Continue**
23. *best_p = get_best_Path_by_Bw(i, G, Paths,* U_BW_{ij}*)*
24. **return** *best_p*
25. **Function** *get_best_Path_by_Bw (i, G, Paths,* U_BW_{ij}*)*
26. min_bw= Capacity
27. max_bw= U_BW_{ij}
28. **For** *P* **in** *Paths :*
29. *min_bw = bottleneck_of_path (G, P, min_bw)*
30. **If** *(min_bw > max_bw and* $min\_bw - U\_BW_{ij} > 1Mbps$*) :*
31. *max_bw = min_bw*
32. *best_p = P*
33. **If** *(best_p):*
34. **return** *best_p*
35. **else:**
36. **print** *"No path met the conditions"*

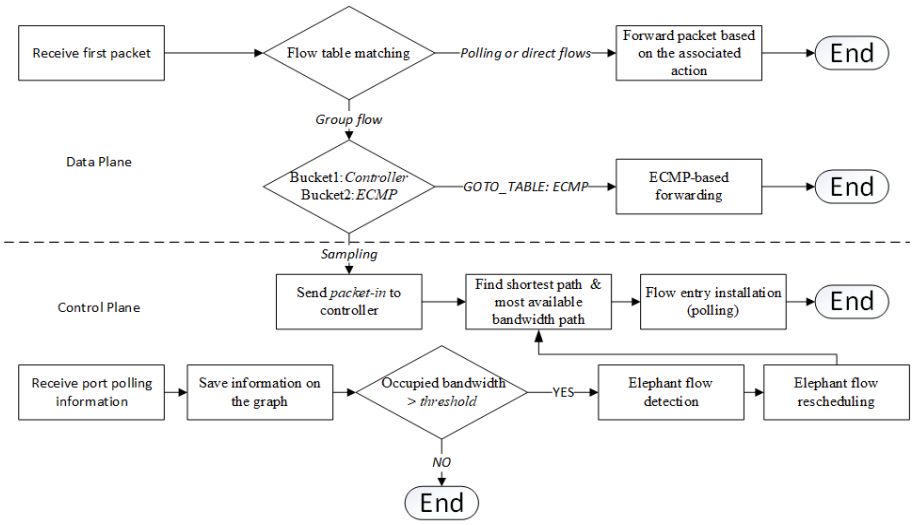


Figure 4.10: Flow chart of the framework

sends the chosen path to the *Flow Installation* module.

4.4.9 Flow installation

As depicted in Tabel 4.5, edge switches have proactive flow entries for directly connected hosts and buckets group entry. Specific priority values have been assigned in descending order, as shown in Tabel 4.5. However, this module installs new polling flow entries with incremental priority based on the last priority value provided that it can not be more than that of the directly connected hosts flow entries. Polling flow entries are installed into all switches along the path.

4.4.10 Network graph

The gathered network information is saved on a directed graph (G) to represent the network topology and situation. Hence, operations like finding the shortest path between any two hosts $<S, D>$ can be achieved by Dijkstra algorithm based on edges weights (G, S, D, W). The algorithm starts searching the graph G for the shortest and most-available-bandwidth path from the source node (S) to the destination node (D) by computing the edges' weights (W) which are the free available bandwidth. We present our proposed second layer functions in Algorithm 1, and the third layer functions in Algorithm 2. The variables used by our framework are presented and described in Table 4.6.
Figure 4.10 depicts the framework workflows.

Table 4.6: Used variables

Variable	Description
min_bw	Available bandwidth on the bottleneck link
max_bw	Maximum available bandwidth along a path
k	The scale of Fat-tree topology
shortest_p	List of the four shortest paths between src_ip and dst_ip
best_p	The best path between src_ip and dst_ip based on hop count and available bandwidth
F_bw	Available bandwidth on an edge switch port
Th	Threshold of the bandwidth occupation
U_bw	Utilized bandwidth on an edge switch port
dpid_list	List of switch IDs
PR	Polling rate
EF_list	List of active elephant flows on an edge switch port
Paths	List of the shortest paths excluding a specific edge switch port
port_list	List of port IDs

4.4.11 Framework design aspects

Algorithms 1 and 2 describe the functionalities of the framework. This section presents the design aspects and decisions of the framework.

4.4.11.1 Problem formulation

Network is modeled as a directed graph $G = (V, E)$, where V: set of the nodes, E: set of the directed edges, p: a path where $p = (v_0, v_1, ..., v_n)$, $\forall v_i \in V$, $\forall i \in [0, n]$, $n \in \mathbb{Z}$. However, the connection throughput is limited to the available bandwidth on the bottleneck link of a path as shown in Eq. 4.1. In particular, the load of any link e is $\frac{l_e}{C_e}$, where l_e is the currently used fraction of its capacity C_e as shown in Eq. 4.3 where s_i^e is the ith flow size. Therefore, the condition in Eq. 4.2 should be maintained to avoid congestion on path p, so that the utilization on any link along path p should be smaller than the bottleneck capacity link. Let us assume that flows arrive at a switch according to Poisson process with rate λ and size s, and the predefined threshold of flow size is T. Hence, the number of elephant flows until time t is as in Eq. 4.4 where F is CDF of flow sizes. In particular, elephant flows on path p_1 and path p_2 are $N_1(t)$ and $N_2(t)$, respectively. Portion of $N_1(t)$ should be redirected to p_2 if $N_1(t) > 0$ and $C_{p_1} > Th$ and $C_{p_2} < Th$, where Th is the threshold of triggering elephant flow rescheduling. Consequently, the

maximum number of elephant flows should be redirected to path p_2 is computed so that the condition in Eq. 4.2 is maintained.

$$C_{p_i} = min \quad C_e, \quad \forall e \in E \tag{4.1}$$

$$\frac{l_e}{C_e} < C_p \tag{4.2}$$

$$l_e = \sum_{i=1}^{N(t)} s_i^e \tag{4.3}$$

$$N(t) = \lambda(1 - F(T))t \tag{4.4}$$

4.4.11.2 Flow detection

There are many principles to classify flows in data center networks. In the case of consumption based classification [4], the flow throughput must be tracked periodically. In this context, the phase of flow detection starts by polling flow statistics from switches in terms of source IP, destination IP, source port, destination port and Byte count. Hence, any flow consumes bandwidth more than the predefined percentage of the link capacity is identified as an elephant flow. However, this methodology discards the large link capacity sizes in recent DCN. As a result, considering the percentage consumption as a classification criterion does not guarantee a rapid reaction for mice flows since they have a very short life span.

The other methodology employs flow size as a classification criterion [5]. In particular, whenever the cumulative flow size hits the predefined threshold T, it will be considered as an elephant flow. We adopt this methodology to distinguish between elephant and mice flows. Specifically, we follow the fact discovered in study [5], where more than 80% of the flows in DCN had less than 10KB.

4.4.11.3 Flow sampling

Our solution employs edge layer switches to sample a portion of flows by sending the first packet (i.e., *packet-in*). Our solution installs unique polling flow entries into the switches along the chosen path to forward packets belonging to the sampled flows. Subsequently, upon threshold hits (i.e., the occupied bandwidth on an edge switch port gets more than 25%) our framework will detect elephant flows forwarding out of the port based on the cumulative bytes count of the corresponding installed polling flow entries.

4.4.11.4 Mitigate obsolete information

As shown in Figure 4.11, the polling messages sent from the switches sequentially to the controller. Hence, there is a delay between the instant of sending the statistics at t_2, as a reply to the request at t_1, and

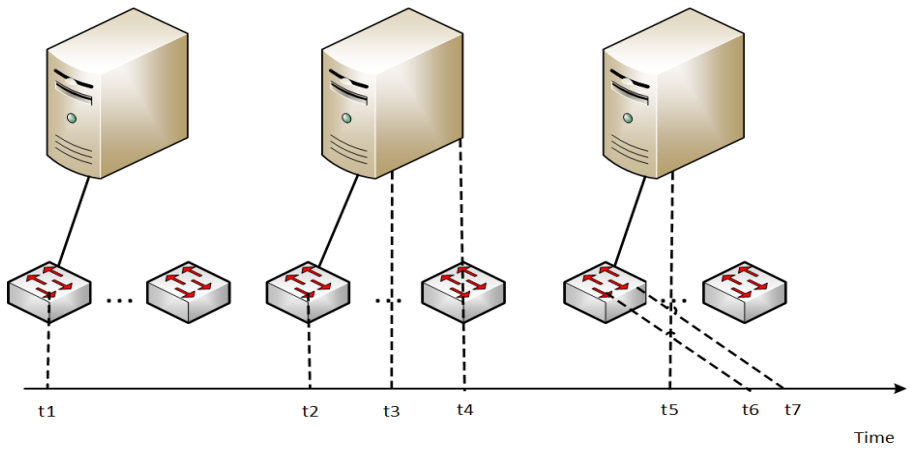


Figure 4.11: Estimate the error resulted from the delay between the time instants of exchanging control messages and reaction instant by the controller

the instant of taking a decision by the controller at t_4. The controller receives the statistical information message at t_3 then it sends the decision at t_5. Furthermore, the decision could lead to installing new flows in a switch at t_7 after receiving it by the switch at t_6. This delay is due to the number of switches, controller activities and network conditions. Consequently, some obsolete decisions could be taken, so the error estimation of the taken decisions can be computed as in Eq.4.5. Assuming τ is the delay between t_2 and t_4, P_{rate} is the polling rate, C is the capacity of the link connecting a switch with the controller, N is the switches number, M_{len} is the length of the reply message.

$$\tau = \frac{N \quad M_{len} \quad P_{rate}}{C} \tag{4.5}$$

$$C_l = N \quad P_{rate} \tag{4.6}$$

$$P_{rate} = \begin{cases} 10^{\frac{B_{edge} \quad - \quad \frac{\sum_{i=1}^{4k} V_i}{4k}}{B_{edge}}} & if \quad \frac{\sum_{i=1}^{4k} V_i}{4k} \leq B_{edge} \\ T_{base} = 2sec & otherwise \end{cases} \tag{4.7}$$

Based on Eqs. 4.5-4.6, the delay is related to the controller load C_l, where they are directly proportional to each other. For example, given the size of statistics message M_{len} = 112 Bytes[2], the link capacity connects the controller to the data plane is 1Gbps and there are 100 switches with P_{rate} = 2 seconds. Under these conditions, the delay can be maximum 179.2 μsec. Moreover, the polling period influences network stability; therefore, the polling period should be dynamic. Eq. 4.7 is used to compute the dynamic values of the polling rate. Since it is not necessary to probe the data plane when the average

[2]OpenFlow Switch Specification Version 1.3.1 Section A.3.5.6

utilization of the edge switch ports is far from the threshold value of the bandwidth occupation (i.e., B_{edge}) where V_i is the utilization of port i and k is the Fat-Tree scale (i.e. $k = 4$). T_{base} is $2sec$, and it is the default polling interval. Thus, the value of P_{rate} will not grow too much, and the controller can still probe the data plane when ports utilization under the predefined threshold. In particular, P_{rate} can extend from $1sec$ under high traffic until $10sec$ under light traffic to maintain the accuracy and to avoid the overhead, as shown in Eq. 4.7. Besides, the polling rate's default value can be used in the case of the average port utilization is more than the occupation threshold at any instant. As a result, based on Eqs. 4.5-4.7 and as the numerical example, the delay in our case (i.e., $N = 20$) will be 35.84 μsec which ensures delivering of up-to-date statistical information. Furthermore, based on the real traffic measurements in [7], P_{rate} value range is efficient since 25% of Web service, 85% of Cache and 25% of Hadoop flows are last for more than 1 sec. Therefore, P_{rate} value range is feasible to take rescheduling decisions for elephant flows where Our solution probes statistics at a rate whose value is within the elephant flows' life span.

4.4.11.5 Controller overhead

Let us assume that the sampling probability is p; hence, the controller receives a *packet-in* packet with a probability p. Likewise, n is the number of the sampled packets out of the total arrived packets, x.

Theorem 1: *The total number of packet-in, n, sampled to the controller is* $\ll \frac{x}{2}$ *given that x is the total number of the packets arrived to an edge switch*

Proof : Let us assume f is the number of flow entries on an edge switch, and c is the count of packets forwarded according to a flow entry. Consequently, $n(t)$ could be asymptotically computed as in Eq. 4.9.

$$x = n + cf \Rightarrow n = x - cf \tag{4.8}$$

$$\begin{aligned} n(t) &= \int_0^t x(t)dt - \int_0^t cf(t)dt \\ &= t^2 \left(\frac{x}{2} - \frac{cf}{2}\right) \end{aligned} \tag{4.9}$$

Accordingly, the maximum value of $n(t) \ll 50\%$ of the total number, since over the time $c\&f$ will get larger, and the load on the controller will be less consequently. For example, let us assume that no more new flows arrived at an edge switch after some time, so $cf \approx x$ which yields no more packets will be sent to the controller.

In the following, we analyze the expected overhead of processing new flows with Our solution. We set up an numerical study to inspect the number of the new flows, Our solution has to process in case of real DCN parameters. Our framework samples a portion of the new flows by employing two buckets.

Table 4.7: Parameters and values of the controller overhead evaluation

Parameter	Description	Value
H	Number of end-host	10000 [5]
R	Number of edge switches	578 [50]
S	Number of total switches	1445 [50]
F	Median inter-arrival time	2 ms [7]
B	Link bandwidth	10 Gbps [7]
P	Packet size	1500 Byte
P_{rate}	Default value of the polling rate	2 sec

Assuming that number of the sampled flows is half of the total number of flows. We consider a Fat-tree DCN with real parameters as shown in Table 4.7.

Our solution needs to handle half of $H \times 10^3/F$ flow set up per second, which is 2.5 million requests per second. Using specific hardware, a single controller can handle up to 12 million requests per seconds as in [68]. In this numerical study, we adopt a size of the commercial data centers as presented in [5]. On the other hand, Eq. 4.6 detects the rate at which our solution needs to process port statistics messages from switches. Consequently, our solution will handle $S \times P/P_{rate}$ (i.e., 723 packets per second). Assuming that the controller can handle these packets at the same rate of handling flow setup, as in [68], so it unlikely under the mentioned parameters that Sieve's performance will reduce severely.

4.4.11.6 Impact of threshold values

Our framework probes the occupied bandwidth on edge switch ports connected to the aggregate layer to figure out if it is below the predefined threshold. However, due to the overhead and rescheduling failure probability associated with different threshold values, we evaluate the occupied bandwidth threshold values. In particular, we investigate the effects of different threshold values on the number of $OFPFlowStats$ message replies to measure the yielded overhead in the controller, since upon threshold hits, our solution probes flow statistics by sending $OFPFlowStats$ message to detect elephant flows on a specific switch port. Besides, we measure the associated failure probability of rescheduling elephant flows to other paths in case of each threshold value by computing the number of successfully rescheduled elephant flows out of the total number of rescheduling requests. In particular, the frequency of rescheduling requests is inversely proportional to the threshold value. Table 4.8 presents the number of messages and the number of failures associated with the different values of the bandwidth occupation percentages under different traffic patterns, which are described in the next section. We aim to find the optimal value so that the probability of finding alternative paths for rescheduling elephant flows is reasonable. Besides, the threshold value should preserve the controller from an intensive message load as well. Therefore, we decided

Table 4.8: Effect of different values of occupied bandwidth threshold on the probability of finding alternative paths to reschedule elephant flows in case of CT and UT traffic patterns

Criterion	$OFPFlowStats$_number		Failure_Probability%	
Threshold	CT	UT	CT	UT
75%	0	25	0	8
50%	3	168	0	42.6
25%	296	1265	28.7	70.6
10%	1226	1686	5.9	60

to adopt 25% as a value for the predefined threshold of bandwidth occupation where it yields 28.7%, 70.6% as failure probabilities in CT and UT scenarios, respectively. In contrast, 10% threshold yields fewer failure probabilities, 5.9% and 60% for CT and UT, respectively, but with a much higher number of messages, which increases the load on the controller. On the other hand, such a small value as 10% of the bandwidth occupation threshold could result in network instability due to the elephant flows rescheduling fluctuations. However, 50% and 75% percentages are not desirable since we aim to improve the FCT of mice flows, and such values can slow down the framework reaction.

4.4.11.7 Number of flow table entries

We measure the number of flow entries generated by our framework and compare it to a fully reactive scheme. In particular, fully reactive scheme sends a *packet-in* packet upon receiving the first packet of a new flow to the controller. Subsequently, the controller tries to find a path, and then it installs a new flow entry into switches along the path. In this context, we aim to figure out the difference in flow numbers between fully reactive scheme and our scheme presented in Table 4.5 which is called proactive. Besides, we investigate if the controller can cap with the received requests and if the number of the flow entries can be absorbed based on the flow table size. Accordingly, we count the flow entries number generated by the second and the third layers in both cases under the uniform traffic pattern UT, which is described in the next section. The results shown in Figure 4.12 indicates that the proactive scheme (i.e., our scheme) has fewer flow entries up to 50% than that in case of the fully reactive one. In addition, since a controller can deal with about 10 millions of flows per second [68] [69] and the flow table can contain up to 5k flows [24], we conclude that our framework yields reasonable load and flow entry number. Furthermore, the numbers in Figure 4.12 are the cumulative numbers of the generated flow entries during the whole experiment whose length is 300 seconds. Therefore, this implies that the number of the simultaneous flow entries at one instant is so lower. Moreover, the presented numbers in Figure 4.12 in case of L2, coincides with Theorem 1 since the sampling process is held by L2 and the presented number in case of

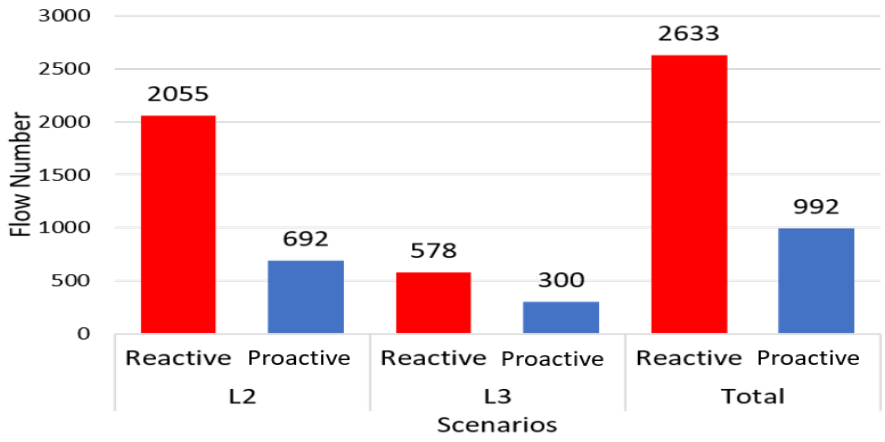


Figure 4.12: Flow entries numbers generated in case of proactive and reactive paradigm

the proactive scheme is less than that in the case of the reactive scheme by more than half.

4.4.11.8 Framework implementation

The framework modules in the control plane are implemented as Python modules and integrated with Ryu SDN controller [66]. We leverage OpenFlow 1.3.1, and the testbed environment has been implemented by Mininet 2.2.2d where we evaluate our framework in 4-ary FatTree data center network, as shown in [50]. We employed Intel Core i5-8400, 16 GB RAM, Ubuntu 16.04.

4.4.12 Experimental results

We compare our framework's performance to Hedera and ECMP [4] since Hedera is the mainstream scheduling and detection framework for DCN, and ECMP acts as a commonly used scheduler in academic and business sectors. Since Mininet runs in real-time and for the sake of precision, we did not use high values for link capacity. Therefore, each core switch connects to four aggregation switches with 100 Mbps bandwidth and 250 μs one-way propagation delay links, 20 Mbps and 1 ms one-way propagation delay for links connect aggregation and edge layers, and 10 Mbps and 2 ms one-way propagation delay for links connect edge switches and end-hosts where each edge switch connected with two end-hosts. Hence, the oversubscription ratio is 1:2 at the edge layer. We evaluate the framework performance by conducted three different 300 seconds scenarios containing a mix of mice and elephant flows for each scenario. In the first scenario, *Concentrated Traffic (CT)*, elephant and mice flows follow many-to-one patterns, in which twelve end-hosts send data to three end-hosts on different pods than the sources. The second one follows the uniform model, *Uniform Traffic (UT)*, where connections span all layers and all end-hosts have been employed to generate the traffic, and each source has a different destination. Finally, *Multi*

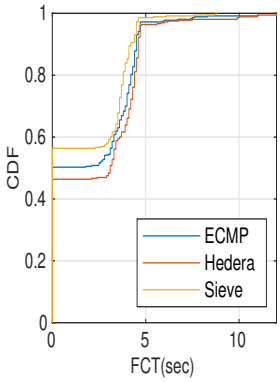
Destinations (MD), we generate traffic from two end-hosts connected to the same edge switch to ten different end-hosts on the other pods, five destinations for each source. We employ iperf for generating elephant flows and Apache server [67] for generating mice flows by repeatedly requesting a webpage of size 10 KB at the tenth second of the simulation lifespan and elephant flows last randomly between [20,60] seconds. In the case of this group of scenarios, our framework will be examined in a situation where mice flows are synchronized to generate burstiness and elephant flows exist to evaluate the framework under high load. Specifically, every 10 seconds, the previous pattern repeats to generate the burstiness during the experiment life span. We conduct the experiments for two different traffic classes. First, we employ high elephant flows share of 1:1 (i.e., mice:elephant ratio) to investigate the impact of the framework under a high volume of elephant traffic. Second, we simulate a mice:elephant ratio of 3:1 as the ratio reported in [5]. Moreover, Table 4.9 presents the details of the flow numbers generated in each scenario. We follow the traffic pattern in [25] to compare Sieve performance to Hedera and ECMP in terms of FCT of mice flows and goodput of elephant flows. We repeat each experiment 10 rounds for each different scenario. For mice flows, we present CDF of FCT. For elephant flows, we essentially present CDF of the goodput. These results are shown in Figure 4.13 for the CT scenario, Figure 4.14 for the UT scenario and Figure 4.15 for the MD scenario. Furthermore, we evaluate our framework under a second scenario group. Specifically, we compare our framework performance under the traffic pattern employed in [4] in terms of average goodput of elephant flows and the average aggregate throughput of all flows in the network. Finally, we conduct a third scenario group in which we employ real workloads to investigate Sieve performance in web services, cache [7] and data mining [16] applications scenarios.

4.4.12.1 FCT of mice flows

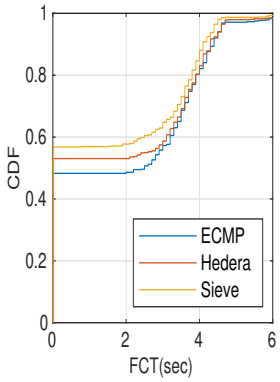
We compare FCT of each algorithm, as shown in Figure 4.13. Since mice flows are delay-sensitive flows, FCT is the most important metric to measure the algorithms performance. As shown in the Figure 4.14.a-4.14.b, our solution outperforms Hedera and ECMP in case of many-to-one traffic pattern. Consequently, our framework can mitigate the delay of mice flows by rescheduling elephant flows upon bandwidth occupation hits the threshold and efficiently utilises the other network links. On the other

Table 4.9: Number of mice to elephant flows in CT, MD and UT scenarios in case of equilibrium ratio which is 1:1 and when mice flows are three times more than elephant flows (i.e., 3:1)

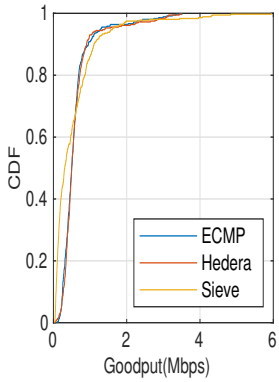
Scenario	1:1	3:1
CT	360:360	871:300
MD	300:300	871:300
UT	300:300	871:300



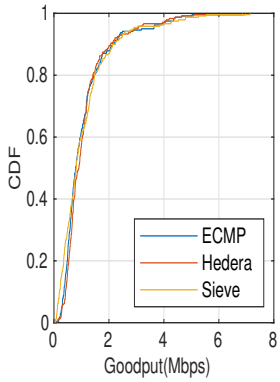
(a) FCT of mice flows under 1:1 pattern



(b) FCT of mice flows under 3:1 pattern

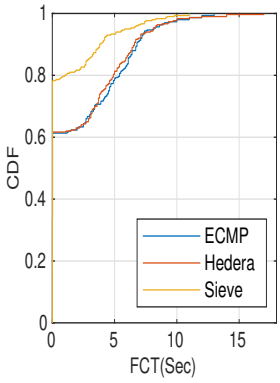


(c) Goodput of elephant flows under 1:1 pattern

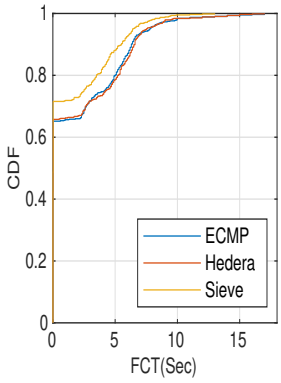


(d) Goodput of elephant flows under 3:1 pattern

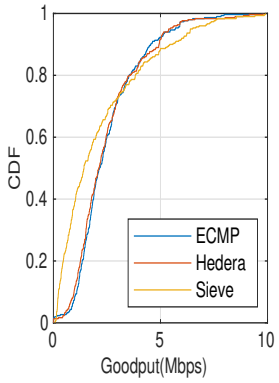
Figure 4.13: Performance of Sieve framework under CT scenario



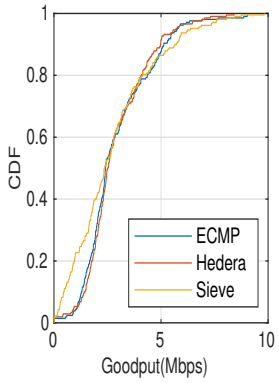
(a) FCT of mice flows under 1:1 pattern



(b) FCT of mice flows under 3:1 pattern

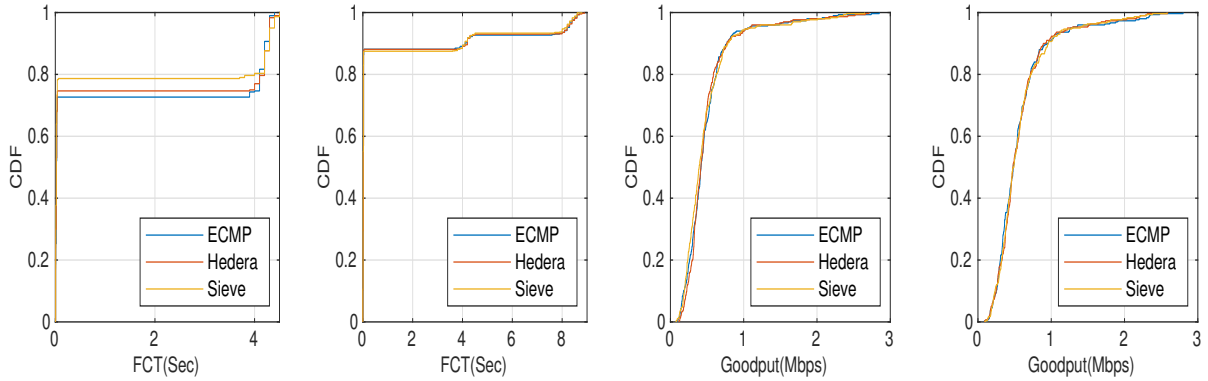


(c) Goodput of elephant flows under 1:1 pattern



(d) Goodput of elephant flows under 3:1 pattern

Figure 4.14: Performance of Sieve framework under UT scenario



(a) FCT of mice flows under 1:1 pattern

(b) FCT of mice flows under 3:1 pattern

(c) Goodput of elephant flows under 1:1 pattern

(d) Goodput of elephant flows under 3:1 pattern

Figure 4.15: Performance of Sieve framework under MD scenario

hand, ECMP provides no consideration of this problem, and Hedera does not invoke the first global fit algorithm based on link situation but based on elephant flows consumption. Similarly, FCT of mice flows provided by our solution is less compared to Hedera and ECMP in case of UT scenario, as shown in Figure 4.15.a-4.15.b. As the number of the sources and destinations are the same, the opportunity of finding other paths for elephant flows upon threshold hits is high. Finally, Figure 4.16.a-4.16.b show the results of MD scenario. The performance of all methods is the same because there are only two sources connected to the same edge switch. Therefore, the opportunity of finding other alternative paths for elephant flows is rare. Figure 4.16 presents the relative changes of average FCT of the mice flows under all scenarios. As shown, our solution outperforms Hedera and ECMP in all scenarios, but the greatest positive improvement is under UT 1-1 since the load in the network links is balanced. On the other hand, the lowest positive change value is in MD scenario where all links toward the sources are saturated, especially in the case of MD 3-1. Besides, our solution provides less FCT for mice flows under many-to-one traffic pattern, which is the common pattern in DCN.

Tables 4.10 4.11 present the confidence intervals of the average goodput and FCT of the elephant flows and mice flows, respectively, under the various scenarios, where the confidence interval is 95%. Based on the shown numbers, our solution has similar average goodput to Hedera and ECMP for all scenarios. As a result, our solution can provide almost equivalent average goodput to Hedera and ECMP for elephant flows but with better FCT for the mice flows.

4.4.12.2 Throughput of elephant flows

In this section, we compare the goodput of elephant flows of the first scenario group. our solution provides elephant flows with a goodput close to that of Hedera and ECMP under almost all cases. Furthermore, we

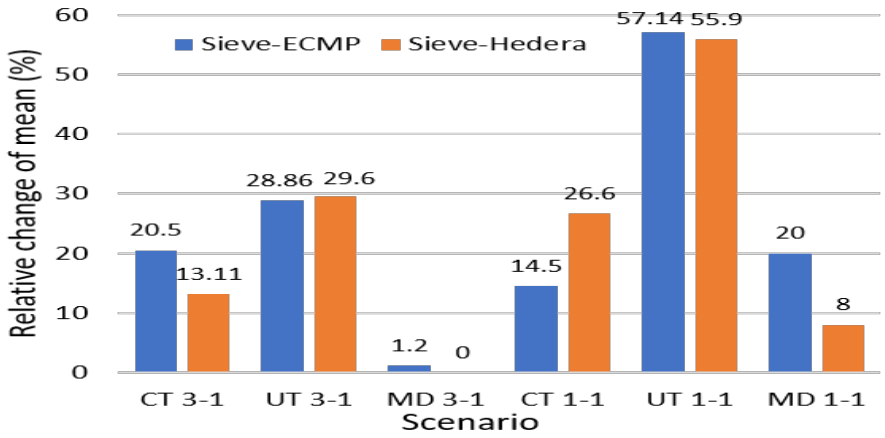


Figure 4.16: Relative changes of average mice flows FCT of Sieve in comparison to Hedera and ECMP in the first scenario group

Table 4.10: Confidence interval of goodput of all algorithms in different scenarios and flow ratios. CDF of algorithms' goodput is shown in Figures 4.13.c-4.13.d 4.14.c-4.14.d 4.15.c-4.15.d

Scenario	Hedera	ECMP	Sieve
CT 3-1	1.11 ± 0.11	1.09 ± 0.12	1.08 ± 0.12
UT 3-1	2.9 ± 0.2	2.92 ± 0.21	2.73 ± 0.26
MD 3-1	0.57 ± 0.04	0.57 ± 0.04	0.57 ± 0.04
CT 1-1	0.61 ± 0.05	0.59 ± 0.04	0.54 ± 0.07
UT 1-1	2.5 ± 0.18	2.53 ± 0.17	2.2 ± 0.23
MD 1-1	0.49 ± 0.04	0.49 ± 0.04	0.48 ± 0.04

Table 4.11: Confidence interval of FCT of all algorithms in different scenarios and flow ratios. CDF of algorithms' FCT is shown in Figures 4.13.a-4.13.b 4.14a-4.14.b 4.15.a-4.15.b

Scenario	Hedera	ECMP	Sieve
CT 3-1	1.83 ± 0.03	2 ± 0.03	1.59 ± 0.05
UT 3-1	1.96 ± 0.24	1.94 ± 0.23	1.38 ± 0.19
MD 3-1	0.8 ± 0.14	0.81 ± 0.15	0.8 ± 0.14
CT 1-1	2.33 ± 0.24	2 ± 0.23	1.71 ± 0.2
UT 1-1	2.11 ± 0.35	2.17 ± 0.35	0.93 ± 0.23
MD 1-1	1 ± 0.2	1.15 ± 0.21	0.92 ± 0.19

compare the goodput of elephant flows under the second scenario group. The generated traffic patterns consist of a random pattern and staggered probability pattern. In particular, the traffic patterns are

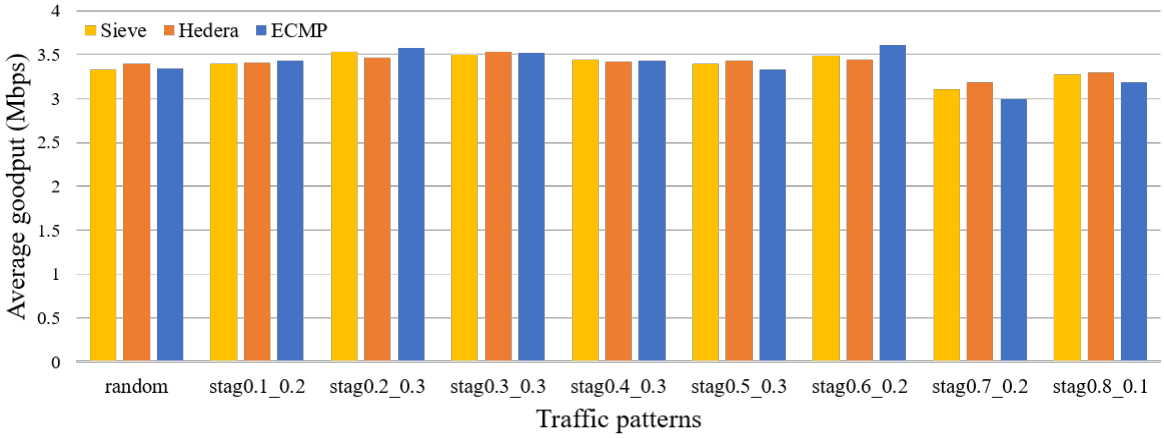


Figure 4.17: Average goodput of the elephant flows from H1 to H16 in case of all traffic patterns of the second scenario group, conducted by Aymen in [J2]

detailed as follows.

1. Random: each end-host sends traffic to any other end-host in the network with equal probability.
2. Staggered probability ($Edge\_p, Pod\_p$): each end-host sends traffic to another one connected to the same edge switch with probability ($Edge\_p$), to the same pod with probability (Pod_p) and to other pods in the network with probability ($1 - Edge\_p - Pod\_p$).

The generated flows differ in sizes and numbers. Each end-host generates one elephant flow lasts for 55 seconds as well as eight mice flows which have different and random sizes $\leq$ 50 KB. Furthermore, flows arrive in Poisson distribution with 10 ms inter-arrival time. Moreover, we repeat the experiment 10 rounds for each algorithm in case of each pattern. In addition, we generate three parallel elephant flows along with nine mice flows from H1 to H16 to compute the average goodput, where elephant flows lasts for 55 seconds.

Similar to the results we have in the first scenario group, our solution provides pretty close goodput in comparison to ECMP and Hedera for the flows initiated from H1 to H16, as shown in Figure 4.17. In this context, the works in [70] [71] [72] prove that Hedera and ECMP have so close throughput as well. On the other hand, Figure 4.18 depicts the average aggregated throughput of all elephant flows in the network. Basically, the throughput achieved by our solution confirms the approximate equivalence.

4.4.12.3 Real workload

In this scenario group, we employ real workloads from production datacenter networks. The flow distribution of web services is less than 10KB for 90% of the flows, and 90% of cache flows are less than

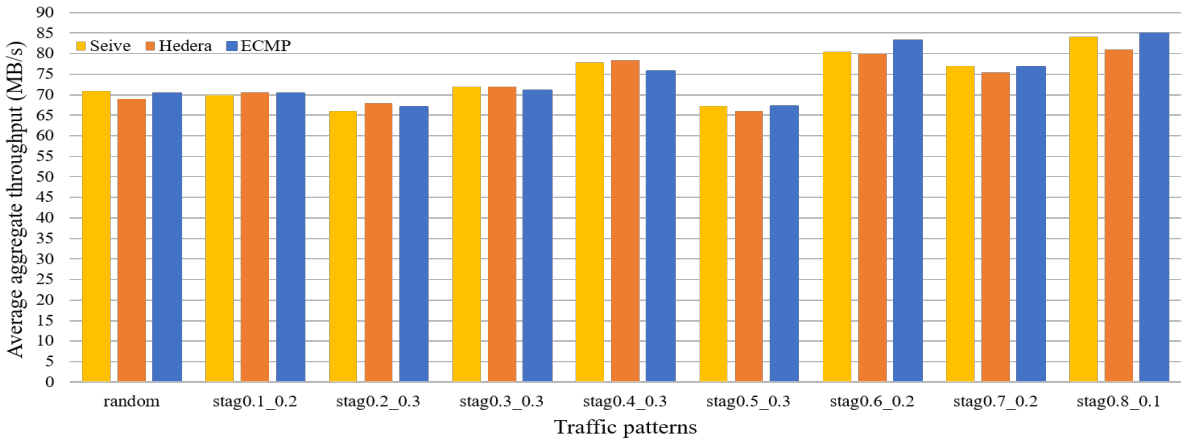


Figure 4.18: Average aggregate throughput of all elephant flows in the network in case of all traffic patterns of the second scenario group, conducted by Aymen in [J2]

500KB [7]. On the other hand, 90% of data mining flow size distribution is less 100KB [16]. Therefore, we generate flows based on the mentioned flow size distributions in the same topology, i.e., 4-ary Fat-tree, by employing uniform traffic pattern (UT) so that the load on all network segments are equal and the generated flow number of the three workloads as well. Specifically, we generate 1920 flows with inter arrival time of 10 ms as in [7], so 120 flows from each host. For this scenario group, we compare FCT of mice flows and goodput of elephant flows under our framework with ECMP and Hedera results.

Figure 4.19 depicts the relative change of mice flow FCT according to the workload types. As shown, our framework outperforms Hedera and ECMP consistent with the results presented in Figure 4.16. Similar to the performance in the first and second scenario groups, the goodput of elephant flows under our solution is equivalent to that in case of ECMP and Hedera, as shown in Figure 4.20. However, elephant flows in case of web services are higher than the other algorithms due to the fact the majority of the web flows are scheduled as mice flows, so the rescheduling of the other workload type flows yields more bandwidth for web flows.

So far, we presented our flow scheduling solution which outperformed Hedera and ECMP. Basically, Hedera detects elephant flows by monitoring all flows at edge layer switches. Since Hedera considers a flow as an elephant flow only if whose bandwidth consumption exceeds 10% of the total link capacity, the presence of multiple elephant flows might yields congestion in case their individual consumption is less than 10% of link capacity, before Hedera rescheduling algorithm triggered. As a result, mice flows might incur latency. On the other hand, mice flows scheduled by ECMP might be delayed or be lost in case of the elephant flows scheduled on the same egress switch port. In contrast, our framework reschedules elephant flows based on the link occupation threshold regardless of the flows types consume the link

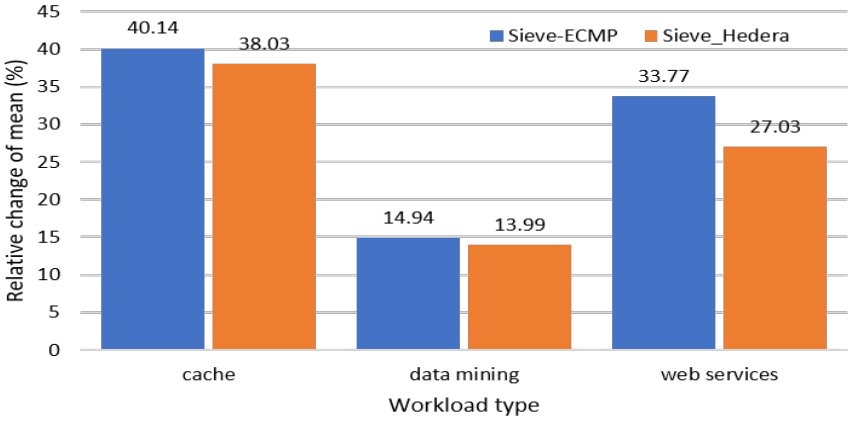
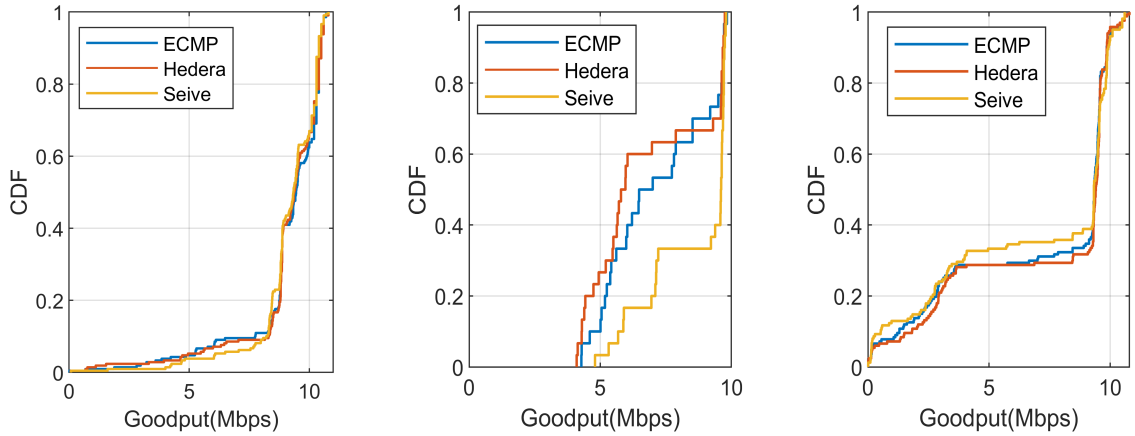


Figure 4.19: Relative changes of average mice flows FCT of Sieve in comparison to Hedera and ECMP resulted from employing realistic traffic loads in the third scenario group depicted according to traffic type



(a) cache (b) web services (c) data mining

Figure 4.20: CDF of the goodput of elephant flows resulted from employing realistic traffic loads in the third scenario group of all algorithms according to traffic type

bandwidth.

4.5 Summary

In this chapter, we present two congestion control solutions in DCN. Both solutions improve FCT of mice flows and maintain the throughput of elephant flows as much as possible. The first solution applies QoS-based technique by creating different queues for mice and elephant flows. In particular, the first framework decreases the transmission rate of elephant flows based on the length on mice flows queues. On the other hand, the second framework reroutes elephant flows along different paths based on bandwidth consumption which is employed as trigger to run elephant flows rerouting algorithm.

Chapter 5

Suitable SDN solution according to the requirements of cloud based data centers[1]

The growing deployment of SDN paradigm in the academic and commercial sectors resulted in many different NOS. As a result, adopting the right NOS requires a comparative study of the available alternatives according to the target use case. In this chapter, we evaluate the specifications of the most common open-source NOSs according to the requirements of Cloud Data Center (CDC). For this sake, we identify the requirements and characteristics of CDC then we present the specifications of six NOSs. As a result, we measure how much the studied NOSs suitable to be used in CDC.

5.1 Introduction

SDN paradigm introduces flexibility in data networks as network devices comply with the controller (i.e., NOS) instructions by separating the control and data planes. In particular, switches in the data plane perform packet forwarding related functions as determined by the control plane. The so-called NOS runs on computer hardware [73], and the control plane represents the business logic in SDN paradigm as it is the controller of network functions. Specifically, it determines how to handle incoming data packets using protocols such as OpenFlow [74]. On the other hand, SDN paradigm also provides a network abstraction for applications at the management plane so that the operator can efficiently perform different network tasks. Figure 5.1 shows the layers of the SDN paradigm [75]. The layers separation

[1] This chapter is the submitted article in [J3]

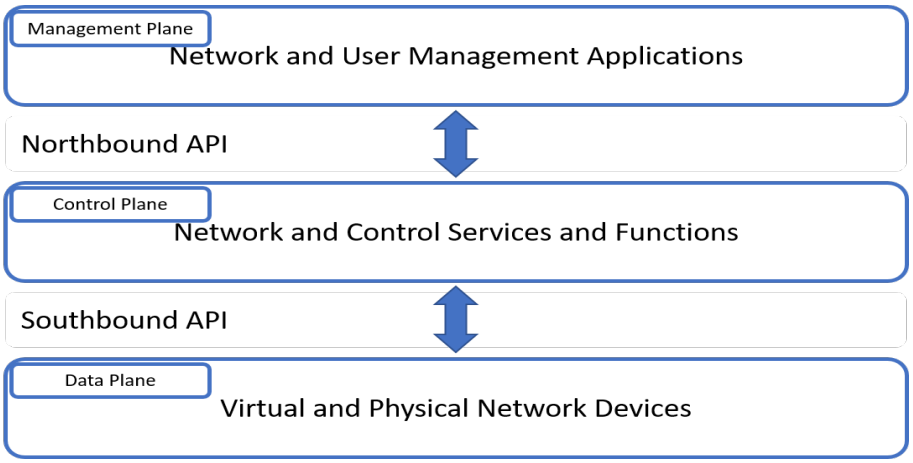


Figure 5.1: The layered architecture of SDN paradigm

provides flexibility for introducing new solutions for problems of the traditional network paradigm. The layered architecture provided by SDN simplifies the deployment of network services and functions such as virtualization, packet forwarding, troubleshooting, etc. SDN layers communicate with each other using dedicated Application Programming Interface (API) as depicted in Figure 5.1 Therefore, adopting the suitable NOS has significant impacts.

CDC motivated to develop SDN paradigm as it relies on virtualization technologies to provide services within the cloud environment. In this context, the traditional network paradigm does not provide the required flexibility to apply these technologies. In addition, data centers have unique traffic patterns, which are different from those in the traditional networks [76]. As a result, CDC becomes one of the most implemented use case of SDN paradigm [77]. Therefore, many commercial companies, organizations and research centers have developed NOSs, but no specific NOS that can be used for all use cases because there are several alternatives according to the requirements. Hence, the research problem was embodied in answering the following questions:

1. Which NOS is the best suited for CDC?
2. What are the functional and non-functional specifications to be verified for this use case?
3. How important is each of the specifications to answer the first question?

The importance of this research lies in the following aspects:

1. There are many NOSs, and their development frequency is rapid. Therefore, selection of the appropriate NOS for CDC requires an analytically comparative study.
2. The selection of the appropriate NOS for the aimed use case profoundly impacts the offered services as it is the control plane in SDN paradigm.

3. This study is expected to help researchers and operators in this field to transform from traditional network to SDN paradigm, identifying the most common open-source NOSs and determine the most appropriate NOS for CDC.

This chapter has the following contributions:

1. Framework to determine the best suited NOS to be used in CDC.
2. Provide an analytical study of the functional and non-functional specifications of six open-source NOSs.

5.2 Related works

Several research works compared NOSs. Khondoker et al. [78] conducted a comparison of specifications for each of the following NOSs, (ODL, Ryu, POX, Ryu, Floodlight, Trema). Ryu was found to be the best NOS according to the adopted criteria for the evaluation. Khondoker et al. did not consider so common NOSs such as ONOS and this study considered no specific use case. The study in [79] compared southbound API, Openflow version, programming language and round-trip time (RTT) delay of POX, ODL, ONOS, Ryu. They found that Ryu has the best score based on TOPSIS method. The work in [79] considered a small set of criteria which are OpenFlow version, NOS programming language, RTT, interfaces and documentation. However, we consider more criteria in this study to evaluate different NOS specifications. The work in [80] presented a feature AHP-based comparison of ODL, NOX, Beacon, Trema, POX, OpenMUL, Ryu, Floodlight, OpenContrail and ONOS in terms of traffic classification as the targeted use case. They found that ODL provides the best specifications for the use case. However, the study did not consider criteria related to cloud based data center similar to what we consider in Table 5.2. The work in [81] presented an ANP-based comparative study of features and performance of Floodlight, ONOS, ODL, POX, Ryu and Trema. The authors found ODL has the best score. However, this study considered many quantitative measurements such delay, throughput and CPU utilization to find the optimal NOS whereas we consider only the qualitative criteria in this study. The comparison conducted in [82] presented the optimum NOS for deploying SDN-WAN. The authors compared Ryu, ODL, POX, Trema, Floodlight. Using AHP, the authors tackled the problem of finding the best NOS might be used for real use case which is connecting university campuses using SDN instead of Multiprotocol Label Switching (MPLS) Border Gateway Protocol (BGP) technology. Amiri et al. [83] compared features and performance of many open-source NOSs by using Best-Worst Method (BWM) decision-making method and Cbench to assess the qualitative and quantitative criteria of NOSs. The work in [83] employs Cbench to measure throughput and latency as quantitative criteria and ODL has the highest rank in both. However, the authors did not aim a specific use case. Ali at el. [84] proposed a prioritized features

method used for selecting the best NOS. The authors employed ANP to implement the proposal. The investigated NOSs are ODL, ONOS, POX, RYU and Trema. The aim of this study was reducing the computational complexity of the selection process by selecting high weight features and ignoring low weight features. However, the study did not consider a wide spectrum of criteria and it did not aim CDC as a use case. The work in [85] investigated many features of ODL, ONOS, Floodlight, POX, RYU and Trema to find the NOS which has the best feature set. In particular, the authors' final target was creating a hierarchical cluster of the best NOS to improve performance such as delay, throughput, fault tolerance and scalability. This work is not tailored for specific use case but it aimed to improve the cluster performance, and it did not consider CDC related features.

On the other hand, many studies reviewed SDN state of the art to investigate different aspects of SDN solutions and NOSs. In this context, our study introduces different aspects of NOSs as well but for identifying the best NOS for CDC. In this context, Abuarqoub [86] reviewed SDN NOSs in terms of scalability. On the other hand, Sarmiento et al. [87] surveyed SDN NOSs in terms of the challenges yielded from inter-site networking services. Studies in [88] [89] observed SDN based load balancing techniques and [90] reviewed SDN NOSs according to controller placement problem.

However, to the best of our knowledge, we consider a use case that has not been considered in the literature. In this study, we aimed to find the best suited NOS for CDC. In addition, our study covers the support of different features such as fault verification and troubleshooting, traffic security and packet forwarding techniques. We conducted a qualitative comparison only since there are many previously published papers on performance comparisons, such as in [79–81, 91]

5.3 Research methodology

We experimentally verified the NOS specifications to decide on criteria and alternatives weights. We installed NOSs, and many of their features and services were investigated. Furthermore, we investigated information provided on their official websites. Several criteria were imposed according to the aimed use case. Initially, the specifications required to be provided by the suitable NOS were identified. Then AHP [92] was applied by which the studied specifications of each NOS were evaluated.

5.3.1 The Studied NOS

The most common open-source NOSs presented at the time of this study were studied, which are ONOS, POX, RYU, ODL, Floodlight and Tungsten. The reason for choosing these NOSs is that they are considered the most widespread open-source NOSs. Most of the other open-source NOSs are either designed for a specific purpose or no longer being developed. However, all unconsidered NOSs in this study are used less according to the frequency of their appearance in research works [93] [94].

ONOS is the most recent NOS produced. It is developed by ON-Lab, which is a non-profit organization. One of its main aspects is that it supports legacy networks. The inspected version in this study is Sparrow [95]. POX was developed to be the successor to NOX. an open research community developed POX. It is suitable for conducting SDN-based researches. The studied version is dart [96]. According to the component-based model, Ryu has been developed by NTT company to facilitate the process of adding and modifying its components in any programming language. The studied version is 4.23 [97]. Big Switch developed Floodlight, and its main goal was to provide a NOS capable of dealing with open network hardware (i.e., hardware that does not contain software and is called "white box"). The involved version is v1.2 [98]. As a result of the effort of a consortium of several international companies, under Linux Foundation, ODL was introduced to obtain the acceptance of companies and users and to receive improvements continuously. The inspected version is Sodium [99]. Finally, some NOSs are considered as cloud oriented SDN NOSs where they aim at managing the network infrastructure of CDC. In this context, we consider Tungsten [100] which is the open source version of Juniper Contrail and now it is a project of Linux Foundation. Tungsten 5.1 is involved in this study.

5.3.2 Characteristics and requirements of CDC

Nowadays, cloud computing drives most businesses and shapes a new era of technology delivery. Cloud computing provides different services such as software, storage, and virtual resources, so it abstracts the technical complexities, and it eliminates the cost and risk associated with hardware maintaining and acquisition [77]. In recent years, SDN has been employed in CDCs since SDN provides central control of the network. SDN-based Data Center Network (DCN) is preferred to the traditional DCN since it can improve DCN efficiency [101]. Furthermore, hypervisors are used in CDC to enable the deployment of virtual resources, so functions such as traffic control and isolation are crucial for providing efficient services. In this context, SDN and Network Function Virtualization (NFV) are seen as enablers of network slicing to create multiple virtual networks over a shared infrastructure. Each virtual network can be logically isolated from other virtual networks and assigned to serve specific requirements. Since SDN controller has a global view of the network, traffic routing decisions can satisfy service demands, and each network slice can be provisioned with network and cloud resources based on QoS and Service Level Agreement (SLA). Due to the programmable model of SDN and the separation of the control plane and data plane, SDN facilitates the deployment of network functions that are essential for managing virtual resources where Virtualized Network Function (VNF) can be deployed in networks dynamically by SDN's configuration and automation functionalities.

In this context, many protocols are used as a southbound interface so that NOS can programmatically contact the switches to exchange information and send instructions. Besides, many NOSs expose

Table 5.1: Non-functional requirements of CDC.

Feature			Description
Easy to use	Usability	graphical user interface (GUI)	Display information about network components and metrics
		Documentation	Information required for utilizing and developing NOS
	Development	North API	The standard used for communications between the control and management planes
		Modularity	NOS architecture based which new modifications may be introduced
		Community	The contributions by developers and operators to improve open source NOSs
Maturity	Update Frequency		The rate of developments the NOS receives
	Application Availability		Applications provided to be used based on NOS
Interoperability	Control Protocols		Communication protocols used to control network hardware by NOS
	Management Protocols		The protocols used to manage the settings of network hardware by NOS
Security			The security features provided by NOS to perform its functions safely
Scalability			The ability of running multiple instances of NOS within a distributed and consistent cluster
Availability			The ability of providing services by NOS in case of failures

Table 5.2: Functional requirements of CDC

Feature		Description
Virtualization	Overlay Networks	Creation of virtual networks based on physical network hardware
	Isolation	Maintaining the isolation of the shared virtual resources
Packet Forwarding Techniques	Load Balancing	Improve network utilization according to the network situation
	Quality of Service	Mechanisms used to ensure that the required level of service is achieved
Traffic Protection Solutions		Detect security threats by monitoring network situation and applying pre-defined security policies
Fault Verification and Troubleshooting		Mechanisms used for faults recovery and avoidance

northbound interfaces that can be used by the management plane and cloud orchestrator to implement different solutions (e.g., Quality of Service (QoS) management, traffic engineering, fault recovery, network statistics, topology discovery, etc.). Furthermore, the dynamically changing status of the network should be considered by the cloud systems in terms of the management of computing and storage resources, Virtual Machine (VM) provisioning, and addressing virtual resources requirements [102, 103]. The massive scale of a CDC, which consists of thousands of compute nodes and network devices, imposes the necessity for network management, performance improvement and dynamic provisioning of computing and network resources [104]. Therefore, the architecture of the control plane should be taken into account whether it should be centralized or distributed. The centralized control plane represents a single point of failure. Although centralized architecture might be an efficient solution for small scale CDCs, it may not be efficient for large scale CDCs. Therefore, a distributed control plane consisted of many NOSs should be deployed. However, such architecture yields other challenges such as synchronization and heterogeneity of the underlying data plane [105].

On the other hand, SDN paradigm triggers additional motivations for the attackers. SDN NOSs and OpenFlow are considered potential security vulnerabilities which require efficient security solutions to improve the safety [106]. In particular, cloud orchestrator and SDN NOS integrate each other. Cloud orchestrator manages virtual resources like VMs provisioning, while SDN NOS manages physical and

virtual network resources by southbound API (e.g., Openflow) and VM traffic. Besides, they communicate by north-bound API (e.g., Rest API) [107]. Hence, since SDN NOS and CDC orchestrator are integrated, maintaining the security is vital. Therefore, security solutions should utilize SDN capabilities and protect SDN components as well. In this context, SDN can be employed to prevent a distributed denial-of-service (DDoS) attack by utilizing SDN functions such as traffic analysis. On the other hand, the SDN NOS must be secured itself as it the control plane [108].

As Big Data and Internet of Things (IoT) applications exploit CDC, utilizing SDN-based CDC can enhance the performance of these applications by applying load balancing and QoS solutions [109]. For this sake, SDN can provide different QoS mechanisms for different applications, such as bandwidth slicing. In addition, scalability is an essential concern [110]. Since SDN NOS gathers information from data plane devices, it is crucial to avoid deploying a single SDN NOS in large scale CDC. Single SDN NOS represents a single point of failure, which could be problematic because there is a big number of devices in the data plane of CDC. Therefore, the distributed architecture of many SDN NOSs, to improve the availability, can be employed using westbound and eastbound APIs to maintain the synchronization [111] [112].

Furthermore, the faults represent a critical challenge to meet SLAs. In CDC, fault tolerance techniques provide the ability to maintain QoS. In this context, improving the reliability in CDC can be proactively achieved to avoid fault occurrence. For example, system load can be identified as an indication of potential future failures so that new resources might be added to avoid failures proactively [113] [114].

Although there are many open-source NOSs, and they are differently matured in terms of CDC requirements. It is challenging to find NOS that can provide CDC requirements, and it can integrate with the CDC orchestrator. As a result, based on the former characteristics of CDC, we summarize in Tables 5.1 and 5.2 the functional and non-functional requirements, respectively, which are required to be provided by the best suited NOS.

5.3.3 Decision support system

In this study, AHP [92] is used to make a decision according to multiple criteria. It is the most appropriate method according to the nature of the problem since it is a mathematical method for decision-making, which generates criteria weights through pair-wise comparisons. This comparison is also used to evaluate alternatives against each criterion. According to AHP, criteria and alternatives are treated separately. In addition, AHP presents problems in a hierarchical structure, which helps classify the criteria into levels according to the nature of the studied problem. Therefore, AHP reduces the size of the comparison matrices, thus maintaining the accuracy and consistency of results. Furthermore, AHP checks the consistency of the evaluation as long as the dimensions of the comparison matrices are under ten. On the other hand, other decision support systems like Technique for Order Preference by Similarity to Ideal

Table 5.3: Priority scale

Priority Value	Indication
1	equally important
3	moderately more important
5	strongly more important
7	very strongly more important
9	extremely more important

Solution (TOPSIS) [115] is suitable for problems with negative criteria, while all criteria in this study have positive values only. Analytical Network Process (ANP) [116] is appropriate in case of correlated criteria and alternatives, while all the criteria and alternatives in this study are independent. On the other hand, Best-Worst Method (BWM) [117] identifies the worst and the best criteria in advanced so that the rest criteria are compared with them. However, in our case, we have many criteria, in different levels of the hierarchy, with similar importance what makes AHP the prefer method for this problem as well [118] [119].

5.4 The problem statement and AHP analysis

This section presents the problem as a decision making problem applying AHP. The studied alternatives are A_n where $n = 6$, and the adopted criteria are C_m where $m \in Z^+$. We aim to find the best suited NOS among the investigated NOSs using AHP according to the features listed in Tables 5.1 and 5.2. Therefore, the hierarchical presentation of the studied criteria is depicted in Figures 5.2,5.3 and 5.4. AHP requires to assign weights for the evaluation criteria. For this sake, we created $m \times m$ matrices for the criteria on the same levels of the hierarchy, and their elements are the priorities for each pair of the criteria to signify the importance of a single criterion to another, as shown in Eq. 5.1 where C_{ij} represents the importance of criterion C_i in comparison to C_j. In order to assign priority, AHP defines a scale between 1 and 9 to present the prioritization, as shown in Table 5.3.

$$\begin{bmatrix} 1 & \cdots & C_{1m} \\ \vdots & \ddots & \vdots \\ C_{m1} & \cdots & 1 \end{bmatrix}, C_{ij} > 0, C_{ij} = \frac{1}{C_{ji}} \tag{5.1}$$

Then, the columns are normalized to find the relative weights by applying Eq. 5.2:

$$\overline{C}_{ij} = \frac{C_{ij}}{\sum_{i=1}^{m} C_{ij}} where \sum_{i=1}^{m} \overline{C}_{ij} = 1 \tag{5.2}$$

Next, vectors of adding the elements of each row are created as shown in Eq. 5.3:

$$v_i = \sum_{i=1}^{m} \overline{C}_{ij} \tag{5.3}$$

By calculating the average of the previous values, we obtain a vector of weights $W_{m\times i}$, which shows the weights of the criteria according to their pair-wise priorities, as shown in Eqs. 5.4,5.5

$$w_i = \frac{v_i}{m} \quad where \sum_{i=1}^{m} w_i = 1 \tag{5.4}$$

$$w_{i\times j} = \begin{bmatrix} w_{1\times j} \\ \vdots \\ w_{m\times j} \end{bmatrix} \tag{5.5}$$

The final weights of the leaf criteria are calculated by finding the resulted values in Eq. 5.5 multiplied with their parent criteria. Figures 5-2,5-3 and 5-4 present the final global weights, which indicate the significance of each criterion to achieve the goal. Then, the priority vector consistency is verified by employing the notion of Eigen-value to compute Consistency Index (CI) as in Eq. 5.6:

$$CI = \frac{\lambda_{max} - m}{m - 1} \tag{5.6}$$

where λ_{max} is the summation of multiply weight vectors with the summation vectors of columns of the pair wise comparison matrix. Consistency Ratio (CR) can be computed as in Eq. 5.7 using Random Index (RI):

$$CR = \frac{CI}{RI} \tag{5.7}$$

The values of RI are presented in Table 5.4. According to [92], RI is the pre-calculated average consistency index computed by [92] of randomly generated comparison matrices with different scales, denoted as *scale* in Table 5.4. RI can be used as a reference value to check how CI, which is computed in Eq. 5.6, of our comparison matrices is far or close to the matched-scale RI. In particular, if CR, in Eq. 5.7, is 1, this means that we have completely random priorities. Therefore, we have no meaningful comparisons, and we have to revise our comparisons. The opposite is true in case Eq. 5.7 (i.e., CR) equals zero. In case CR is below 0.1 (i.e., 10%), the priority values are supposed to be consistent; otherwise, pair-wise priorities

Table 5.4: Values of Random Index

Scale	1	2	3	4	5	6	7	8	9
RI	0	0	0.58	0.9	1.12	1.24	1.32	1.41	1.45

should be modified, and the pair-wise comparisons should be repeated. Similarly, the alternatives are pair-wise compared against each criterion as in Eq. 5.8 based on the convention in Table 5.3, and weight vectors of the alternatives are computed, and their consistency is inspected as in Eq. 5.7, as well. Finally, the final value of each alternative is computed as in Eq. 5.9 where $W_{m\times 1}$ is the weight vector of child criteria, $\acute{W}_{n\times m}$ is the alternative's weights against all criteria, and $X_{n\times 1}$ is a vector contains the final result of each alternative.

$$\begin{bmatrix} 1 & \cdots & A^m_{1n} \\ \vdots & \ddots & \vdots \\ A^m_{n1} & \cdots & 1 \end{bmatrix}, A_{ij} > 0, A_{ij} = \frac{1}{A_{ji}} \tag{5.8}$$

$$X_{n\times 1} \quad = \quad \acute{W}_{n\times m} \quad \times \quad W_{m\times 1} \tag{5.9}$$

Tables 5.5-5.14 present the matrices resulted from Eq. 5.1. The pair-wise comparisons of criteria present that functional and non-functional criteria have equal importance. In relation to the non-functional requirements, we set weights of easy of use and maturity less than that of the remaining criteria since all considered NOSs are open-source, and they can be further developed according to requirements. On the other hand, development has a higher weight than usability as NOSs might be modified to propose new functions and services. As a result, modularity has a higher weight in comparison to development community and north API as it simplifies NOSs modification. As shown in Table 5.9, documentation has a higher weight than GUI, where it is essential to adopt, use and develop NOSs. Finally, all remaining criteria have similar weights except virtualization since we aim to find the best suited NOS for CDC. We present pair-wise comparison matrices of the alternatives resulted from Eq. 5.8 in Tables 5.15-5.32. Table 5.34 presents $\acute{W}_{n\times m}$ of all alternatives according to all leaf criteria. Final values of all alternatives, $X_{n\times 1}$, are presented in Figure 5.5.

5.5 Results and discussion

Table 5.33 summarizes the detailed specifications of the studied alternatives based on the different criteria.

Table 5.5: Top level criteria

Criteria	non-Functional Requirements	Functional Requirements
non-Functional Requirements	1	1
Functional Requirements		1

Consistency Ratio: 0.00%

Table 5.6: Non-Functional Requirements

Criteria	Scalability	Security	Easy to Use	Maturity	South API	Availability
Scalability	1	1	3	3	1	1
Security		1	3	3	1	1
Easy to Use			1	1/2	1/3	1/3
Maturity				1	1/3	1/3
South API					1	1
Availability						1

Consistency Ratio: 0.22%

Table 5.7: Easy to Use

Criteria	Development	Usability
Development	1	3
Usability		1

Consistency Ratio: 0.00%

Table 5.8: Development

Criteria	Modularity	Development Community	North API
Modularity	1	2	1
Development Community		1	1/2
North API			1

Consistency Ratio: 0.00%

Table 5.9: Usability

Criteria	GUI	Documentation
GUI	1	1/3
Documentation		1

Consistency Ratio: 0.00%

Table 5.10: Maturity

Criteria	Update Frequency	Application Availability
Update Frequency	1	3
Application Availability		1

Consistency Ratio: 0.00%

Table 5.11: Interoperability

Criteria	Management Protocols	Control Protocols
Management Protocols	1	1
Control Protocols		1

Consistency Ratio: 0.00%

Table 5.12: Functional Requirements

Criteria	Faults Verification and Troubleshooting	Packet Forwarding Techniques	Virtualization	Traffic Protection Solutions
Faults Verification and Troubleshooting	1	1	1/3	1
Packet Forwarding Techniques		1	1/3	1
Virtualization			1	3
Traffic Protection Solutions				1

Consistency Ratio: 0.00%

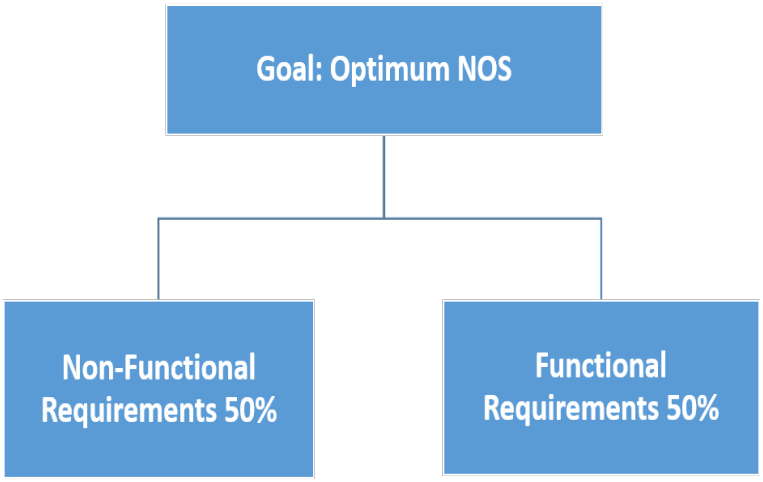


Figure 5.2: Top level of the criteria hierarchy

Table 5.13: Packet Forwarding Techniques

Criteria	Load Balancing	Quality of Service
Load Balancing	1	1
Quality of Service		1

Consistency Ratio: 0.00%

5.5.1 Non-functional features

As shown in Figure 5.5, ODL is the best suited NOS for CDC according to the specified criteria. The final values of ONOS and ODL are close since they are being developed continuously by a large and active community, and they are supported by many industrial and research groups in comparison to the other NOSs. This presents the importance of adopting open-source software projects by relevant companies and institutions.

Table 5.14: Virtualization

Criteria	Overlay Networks	Isolation
Overlay Networks	1	1
Isolation		1

Consistency Ratio: 0.00%

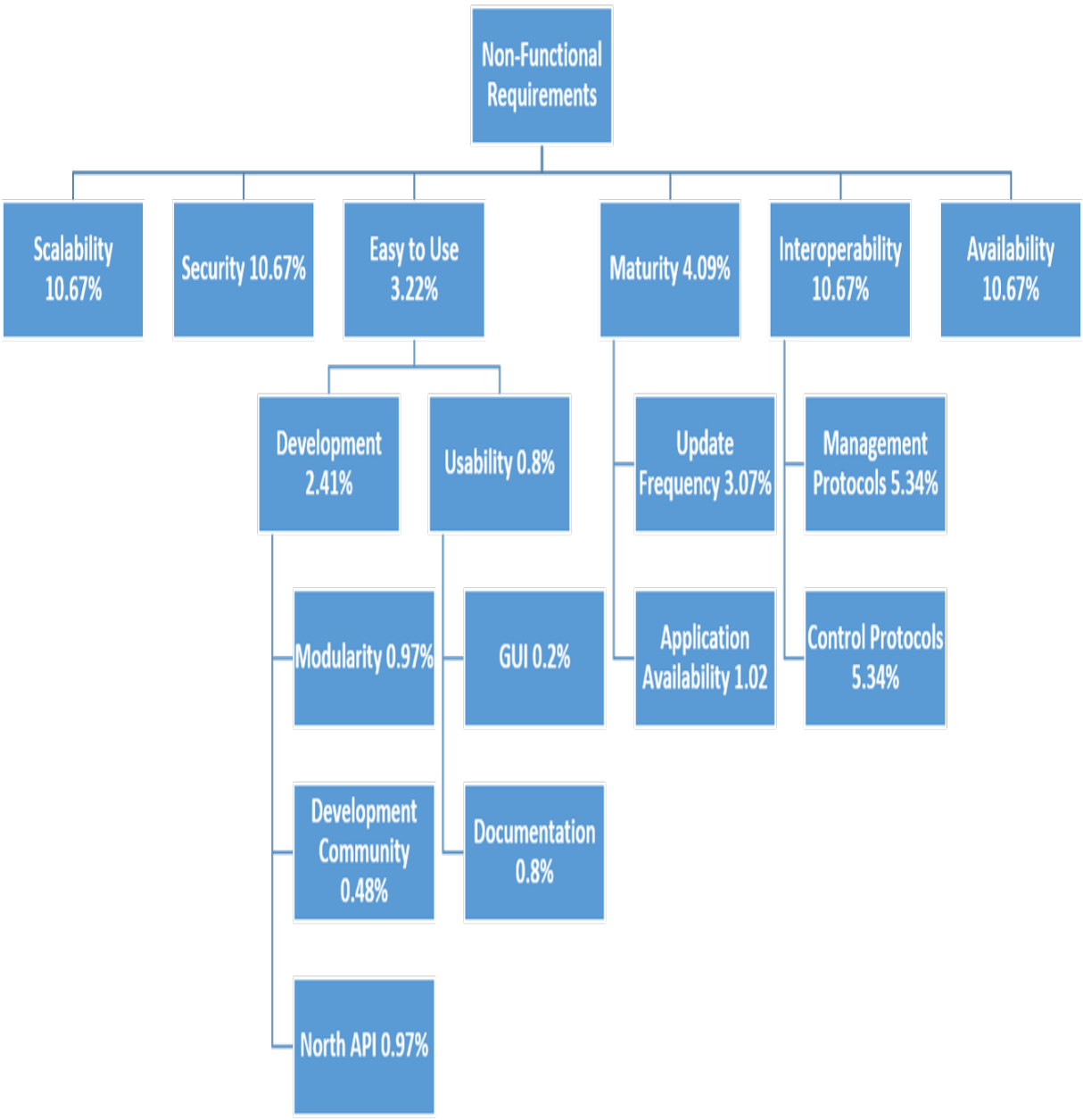


Figure 5.3: Hierarchy of the non-functional Requirements

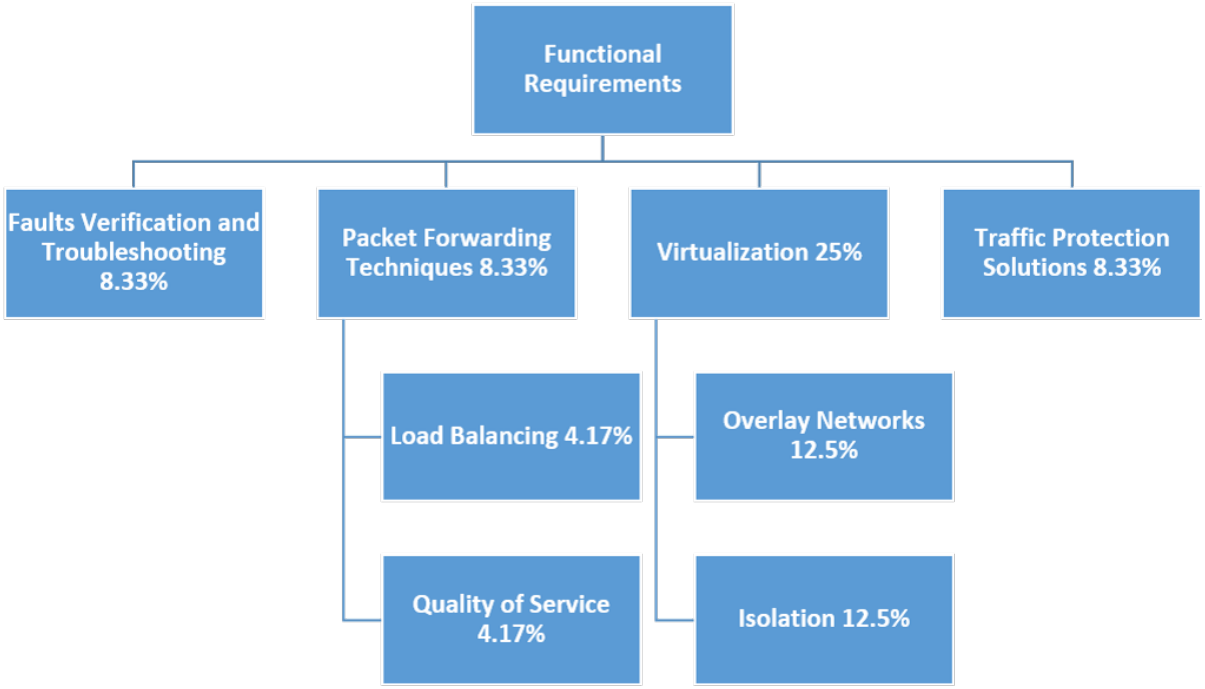


Figure 5.4: Hierarchy of the functional Requirements

5.5.1.1 Scalability

ONOS, ODL and Tungsten are more scalable than the other NOSs, since they support distributed model in which multiple NOS instances can inter-operate simultaneously. However, ONOS can assign segments of Wide Area Network (WAN) or DCN to a specific instance to mitigate the instance load. ONOS employs Atomix database to store network information, which is also used for coordinating between the instances. In particular, ONOS represents a logically centralized control plane since the infrastructure might consist of multiple domains with one ONOS instance for each domain. On the other hand, ODL employs Advanced Message Queuing Protocol (AMQP) to exchange information between ODL instances where each instance maintains information of its own domain. ODL leverages Federation and NetVirt projects to create virtual networks across many OpenStack instances, but these inter-site communications are managed at ODL levels, so overlapping is likely at Neutron level, e.g., duplicated IPs in different OpenStack instances is probable. Tungsten controller has control nodes run in a cluster to maintain scalability.

5.5.1.2 Security

All NOSs provide secure connections with the devices in the data plane by applying Transport Layer Security/Secure Sockets Layer (TLS)/(SSL). Moreover, ODL has Unified Secure Channel (USC), which

consists of an agent, plugin and manager to initiate and maintain the connection with ODL, authenticates ODL and presents the state of USC in ODL DLUX user interface. Besides, ODL provides authentication, authorization, and accounting (AAA) framework controlling access to ODL features, applying policies to use those features and auditing the usage.

5.5.1.3 Easy to use & Maturity

In terms of Modularity, new modules can be added to all NOSs feasibly either as Python component or Apache Karaf. However, ONOS provides Yang management system that abstracts data modelling details of applications and device drivers by generating YANG modelled JAVA objects corresponding to device and application schema so that the business logic will be the sole concern. Hence, it can seamlessly support any interface like Representational state transfer (REST), Network Configuration Protocol (NETCONF), Extensible Markup Language (XML), etc. ONOS and ODL have the richest documentations. In addition, ODL and ONOS have the fastest update frequency and the most varied and numerous applications, but Tungsten provides richer GUI than other NOSs as configurations in terms of QoS, IP addressing, Ports, policies, etc can be depicted and defined using the web-based GUI.

5.5.1.4 Interoperability

Tungsten, ONOS and ODL provide Simple Mail Transfer Protocol (SNMP) as a management protocol which provides so many facilities in an environment such as CDC. In relation to south API control protocols, ODL provides Model-Driven Service Adaptation Layer (MD-SAL), which simplifies the communications between ODL modules and data plane through the available southbound APIs by supporting user-defined payload formats, including payload serialization and adaptation. ODL provides OpFlex, which is an extensible policy protocol designed to exchange abstract policy between ODL and the devices support policy rendering. On the other hand, Tungsten provides Extensible Messaging and Presence Protocol (XMPP) as a southbound API to communicate with vRouter, which can be deployed either as a kernel module, Data Plane Development Kit (DPDK) in user space, SmartNIC programmable network interface, or by using Single Root I/O Virtualization (SR-IOV) for accessing Network Interface Card (NIC) from VMs directly.

5.5.1.5 Avialability

Availability has been boosted in all NOSs except POX by applying Master/Slave model. The continuous mirroring operates between master and slave instances so that the slave takes responsibility upon master failure. However, since ODL maintains the virtual network information on ODL level in case of multiple CDC sites, non-disconnected ODL instances can provide inter-site services during network disconnections

Table 5.15: Scalability

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/3	1/4	1	1
RYU		1	1/3	1/4	1	1
ODL			1	1/3	3	3
ONOS				1	5	3
POX					1	1
Tungsten						1

Consistency Ratio: 1.55%

where ODL instances employ AMQP to communicate with each other so that no need to share information between instances. On the contrary, ONOS instances present a logically distributed control plane by sharing information using Atomix database.

5.5.2 Functional features

5.5.2.1 Fault verification and troubleshooting

Most outstanding troubleshooting capabilities have been found in ODL and ONOS as well. ODL provides a diagnostics framework to report the status of ODL itself by performing continuous monitoring of registered modules and built-in services to maintain the overall health of the system, and it additionally triggers alarms. On the other hand, ONOS provides a framework to break the routing loop, routing black holes and applications conflicts. As well, ONOS integrates with services like CPMan and Ganglia to maintain the information of the ONOS state. In addition, ONOS provides fault management by polling SNMP capable network devices to track events like device adding, updating, link down, etc. Users can access and manipulate alarms by CLI, Rest API or GUI. FlowScale provided by Floodlight diverts all traffic directed to a down port to other up ports. Tungsten's analytic nodes collect metrics from compute, storage and network nodes as well as their workloads to facilitate troubleshooting and monitoring. In particular, Zookeeper is used to distribute the responsibility of collecting data among analytic nodes to avoid nodes overwhelming.

5.5.2.2 Packet forwarding technique

For the sake of improving the availability and the performance, ONOS balances the load between the instances in a cluster. Besides, ONOS improves QoS by providing SDNi, which manages flow setup by considering information such as path requirement, QoS and Service Level Agreement (SLA). Tungsten balances the load among the virtual resources by providing load balancing as a service employing different

Table 5.16: Security

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	2	1/3	1/2	2	1
RYU		1	1/4	1/3	1	1/2
ODL			1	2	4	3
ONOS				1	2	3
POX					1	1/2
Tungsten						1

Consistency Ratio: 0.66%

Table 5.17: Modularity

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/3	1/3	1	3
RYU		1	1/3	1/3	1	3
ODL			1	1	3	6
ONOS				1	3	6
POX					1	3
Tungsten						1

Consistency Ratio: 0.44%

Table 5.18: Development Community

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/6	1/6	3	1/4
RYU		1	1/6	1/6	3	1/4
ODL			1	1	6	2
ONOS				1	6	2
POX					1	1/5
Tungsten						1

Consistency Ratio: 3.29%

Table 5.19: North API

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	3	1/3	2	5	2
RYU		1	1/5	1	3	1
ODL			1	3	7	3
ONOS				1	3	1
POX					1	1/2
Tungsten						1

Consistency Ratio: 1.87%

Table 5.20: GUI

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	2	1/3	1/3	3	1/6
RYU		1	1/3	1/3	1	1/6
ODL			1	1	5	1/3
ONOS				1	5	1/3
POX					1	1/9
Tungsten						1

Consistency Ratio: 2.08%

Table 5.21: Documentation

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/3	1/3	3	1
RYU		1	1/3	1/3	3	1
ODL			1	1	5	3
ONOS				1	5	3
POX					1	1/3
Tungsten						1

Consistency Ratio: 0.93%

Table 5.22: Update Frequency

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1/3	1/6	1/6	3	1/5
RYU		1	1/3	1/3	7	1/2
ODL			1	1	9	2
ONOS				1	9	2
POX					1	1/7
Tungsten						1

Consistency Ratio: 2.22%

Table 5.23: Application Availability

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	2	1/3	1/3	4	1
RYU		1	1/6	1/6	3	1/2
ODL			1	1	9	3
ONOS				1	9	3
POX					1	1/4
Tungsten						1

Consistency Ratio: 0.73%

Table 5.24: Management Protocols

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1/2	1/6	1/5	3	1
RYU		1	1/6	1/3	4	2
ODL			1	2	6	6
ONOS				1	6	5
POX					1	1/3
Tungsten						1

Consistency Ratio: 4.06%

Table 5.25: Control Protocols

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/2	1	4	2
RYU		1	1/2	1	4	2
ODL			1	2	6	3
ONOS				1	4	2
POX					1	1/2
Tungsten						1

Consistency Ratio: 0.22%

Table 5.26: Availability

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	2	1/5	1/5	5	1
RYU		1	1/7	1/7	4	1/2
ODL			1	1	7	5
ONOS				1	7	5
POX					1	1/4
Tungsten						1

Consistency Ratio: 4.87%

Table 5.27: Faults Verification and Troubleshooting

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/3	1/3	3	1/2
RYU		1	1/3	1/3	3	1/2
ODL			1	1	5	2
ONOS				1	5	2
POX					1	1/4
Tungsten						1

Consistency Ratio: 1.11%

Table 5.28: Load Balancing

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	5	1	1/7	1	1
RYU		1	1/5	1/9	1/5	1/5
ODL			1	1/7	1	1
ONOS				1	7	7
POX					1	1
Tungsten						1

Consistency Ratio: 3.54%

Table 5.29: Quality of Service

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/3	1/5	3	1
RYU		1	1/3	1/5	3	1
ODL			1	1/3	5	3
ONOS				1	7	5
POX					1	1/3
Tungsten						1

Consistency Ratio: 1.99%

Table 5.30: Overlay Networks

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/5	1/5	1	1/5
RYU		1	1/5	1/5	1	1/5
ODL			1	1	5	1
ONOS				1	5	1
POX					1	1/5
Tungsten						1

Consistency Ratio: 0.00%

Table 5.31: Isolation

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	1	1/5	1/5	1	1/5
RYU		1	1/5	1/5	1	1/5
ODL			1	1	5	1
ONOS				1	5	1
POX					1	1/5
Tungsten						1

Consistency Ratio: 0.00%

Table 5.32: Traffic Protection Solutions

NOS	Floodlight	RYU	ODL	ONOS	POX	Tungsten
Floodlight	1	3	1/7	1/5	3	1/3
RYU		1	1/9	1/7	1	1/5
ODL			1	3	7	5
ONOS				1	5	3
POX					1	1/5
Tungsten						1

Consistency Ratio: 5.51%

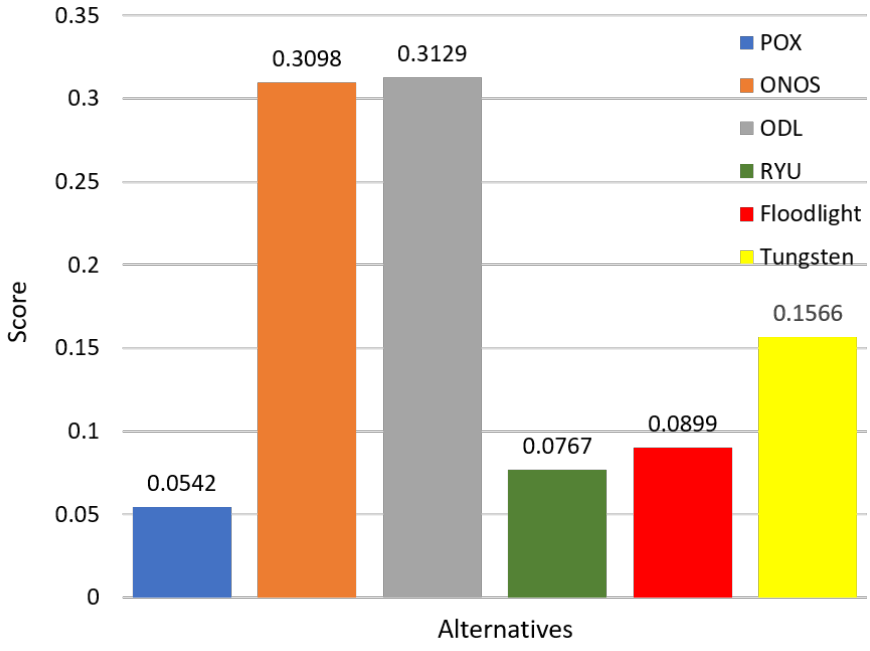


Figure 5.5: Total final scores that represent the NOSs competence

drivers such as HAproxy. ONOS provides intent framework by which applications can specify their network control desires as a policy (e.g., tunnel provisioning, flow rule installation). Hence, different packet forwarding techniques can be applied at run-time. However, the other NOSs support QoS by creating queues, Openflow meters, Differentiated Services (DiffServ) or creating Label-Switched Path (LSP) by Path Computation Element Configuration Protocol (PCEP).

5.5.2.3 Virtualization

In relation to virtualization, all NOSs except POX integrate with Openstack orchestrator, support Virtual Local Area Network (VLAN), moreover, Tungsten, ODL and ONOS support Virtual Extensible LAN (VxLAN) and L3 tunnelling. In this context, ODL manages overlay tunnels established within a transport zone using VxLAN, Generic Routing Encapsulation (GRE), Generic Protocol Extension for VxLAN (VxLAN-GPE) and MPLS over GRE (MPLSoGRE). Besides, ODL provides L2, L3 tunneling, network address translation (NAT) and access control list (ACL) services for the virtualized resources using Neutron northbound API, which communicates with ODL driver in Openstack. On the other hand, ONOS provides a tenant network virtualization service for CDC provisioning using L2 VxLAN or L3 GRE. Furthermore, ONOS provides L2, L3 tunneling and NAT services using Open Virtual Switch (OVS) in the compute nodes and provides horizontal scalability of the gateway node, which connects the virtual networks with the outside by BGP. ONOS isolates the virtual network traffic by Ethernet Virtual Connections (EVC), which uses VLAN ID to tag traffic of different EVCs uniquely and define the associated components like User Network Interface (UNI) and bandwidth profile. Similarly, ODL provides UNI manager, which provisions EVC in physical and virtual network elements of multi-vendor using YANG based APIs and drivers. In addition, ONOS Simplified Overlay Network Architecture (SONA) and ODL NetVirt integrate with Kubernetes as well. On the other hand, Tungsten employs BGP Ethernet Virtual Private Network (EVPN) and VxLAN to connect VMs in different orchestrator domains like Kubernetes, OpenStack and VMWare vCenter. Furthermore, Tungsten enables virtual networks to connect with external network by BGP peering with gateway routers. Tungsten employs vRouter instead of Linux bridge or OVS in hosts where Tungsten controller nodes configure vRouter to implement the network and security policies. Tungsten maintains the isolation using MPLS over User Datagram Protocol MPLSoUDP or VxLAN encapsulation tunneling between fabric Virtual Routing and Forwarding (VRF) and VM VRFs.

5.5.2.4 Traffic protection solutions

Finally, Floodlight provides proactive ACL and basic firewall by installing Openflow flow entries reactively in terms of secure traffic. Besides, POX can block connections based on Media Access Control (MAC) address. RYU can apply firewall principles based on VLAN and Switch id. However, Tungsten,

Table 5.33: Functional and Non-functional features comparison between NOSs

Criterion	**Alternatives**					
	FloodLight	**RYU**	**ODL**	**ONOS**	**POX**	**Tungsten**
Scalability	Centralized	Centralized	Distributed	Distributed (Network Segmentation)	Centralized	Distributed
Security	TLS/SSL	TLS/SSL	USC	TLS/SSL, HTTPs	TLS/SSL	TLS/SSL
Modularity	Maven	Python Components	Apache Karaf	Apache Karaf	Python Components	Unknown
Development Community	Good	Good	Excellent	Excellent	Poor	Very Good
North API	REST API	REST API	REST/RESTCONF API	REST API	REST API	Rest API
GUI	Yes	Yes	Yes	Yes	No	Yes
Documentation	Good	Good	Excellent	Excellent	Poor	Excellent
Update Frequency	Poor	Good	Excellent	Excellent	Poor	Very Good
Applications Availability	Good	Good	Excellent	Excellent	Poor	Very Good
South API – Management Protocols	N/A	OF-config, NETCONF, OVSDB, Nicira ext	NETCONF, OVSDB, SNMP	NETCONF, OVSDB, SNMP	N/A	NETCONF, SNMP
South API – Control Protocols	OF 1.0, 1.3, OpenFlowJ-Loxigen	OF 1.0-1.5, BGP	OF 1.1, 1.3, OpFlex, PCEP, BGP, LISP, MD-SAL	OF 1.1, 1.3, PCEP, BGP	OF 1.0	XMPP, BGP
Availability	Master/Slave	Master/Slave	Active/Backup	Master/Standby	N/A	Active/Backup
Faults Verification & Troubleshooting	Oftest, Flowscale	ofctl, of13, ofctl_test	Status And Diagnostics Framework, networking-odl ML2	CPMan, Ganglia Troubleshooting module	N/A	Analytic Nodes
Load Balancing	Basic	N/A	Basic	Mastership rebalancing	Basic	Third party
Quality of Service	OF meter & queue	DiffServ	PCEP	SDNi	N/A	DiffServ
Overlay Networks	Virtual Network	VLAN	VLAN, VXLAN, L3 GRE	VLAN, VXLAN, L3VPN	VLAN	BGP EVP, VxLAN
Isolation	L2 based	L2 based	L2 & L3 Tunneling, EVC	L2 & L3 Tunneling, EVC	N/A	MPLSoUDPGRE, VxLAN
Traffic Protection Solutions	Forwarding rules, ACL	VLAN-id & SW-id based	SFC	SFC	MAC blocker	SFC, Applications tag

Table 5.34: Final numbers of the pair wise comparison of all alternatives regarding all leaf criteria

Criterion	**Alternatives**					
	FloodLight	**RYU**	**ODL**	**ONOS**	**POX**	**Tungsten**
Scalability	0.0876	0.0876	0.2345	0.4129	0.0842	0.0932
Security	0.131	0.0736	0.3631	0.2276	0.0736	0.131
Modularity	0.1111	0.1111	0.312	0.312	0.1111	0.0428
Development Community	0.0628	0.0628	0.3226	0.3226	0.0361	0.1932
North API	0.2164	0.1	0.4162	0.115	0.0446	0.1077
GUI	0.0837	0.0548	0.1857	0.1857	0.0403	0.4498
Documentation	0.1128	0.1128	0.3075	0.3075	0.0466	0.1128
Update Frequency	0.0499	0.1216	0.3068	0.3068	0.0252	0.1898
Applications Availability	0.117	0.0636	0.3359	0.3359	0.0306	0.117
South API – Management Protocols	0.0692	0.1121	0.434	0.2789	0.0367	0.0692
South API – Control Protocol	0.1782	0.1782	0.3252	0.1782	0.0468	0.0936
Availability	0.0953	0.0598	0.3621	0.3621	0.0302	0.0905
Faults Verification & Troubleshooting	0.1016	0.1016	0.2897	0.2897	0.0447	0.1726
Load Balancing	0.0988	0.0274	0.0988	0.5772	0.0988	0.0988
Quality of Service	0.0916	0.0916	0.2292	0.4564	0.0395	0.0916
Overlay Networks	0.0556	0.0556	0.2778	0.2778	0.0556	0.2778
Isolation	0.0556	0.0556	0.2778	0.2778	0.0556	0.2778
Traffic Protection Solutions	0.0703	0.0331	0.4684	0.2516	0.0377	0.1389

ONOS and ODL provide Service Function Chaining (SFC) by which an ordered list of network services (e.g. firewalls, intrusion detection system, etc.) can be defined, consequently deploying softwarized functions in the CDC becomes more flexible. In this context, OpenStack provides integrating SFC drivers to implement SFC using ODL and ONOS. Moreover, ODL provides a logical service function forwarder that facilitates service function mobility, load balancing and failover. In addition, ONOS provides a policy framework as an abstraction layer that hides the details of the control and data planes. This framework simplifies the enforcement of actions to the network by installing OpenFlow rules to provide functionalities like firewall and NAT. ODL provides NetVirt to apply ACL in ingress and egress modes to VMs created by Neutron northbound API. Tungsten provides security policies based on tags applied to projects, networks, vRouters, VMs and interfaces. Consequently, more granular policies can be created as well as security policy administration and troubleshooting are more feasible.

5.6 Summary

In this chapter, we presented the best suited NOS to be used in CDC by assessing the specifications of six NOSs, according to the criteria imposed by cloud data center requirements. Besides, we presented to which extend each NOS meets these criteria. To the best of our knowledge, this the first study in the literature tackling such a problem. A continuous follow-up of NOSs should be conducted since Software Defined Cloud Computing (SDCC) is still in the stages of maturity and adoption. Moreover, the related technologies are developing rapidly because of the tendency to convert CDC into Software Defined Environment (SDE). We also identified the essential requirements for SDN based CDC. However, several requirements need more investigation under other use cases. In this context, it is essential to extend the current NOSs to provide network services in case of inter-CDCs (e.g., deploying SFC whose VMs belong to multiple CDCs managed by different cloud orchestrating instances). Furthermore, maintaining the synchronization between multiple NOSs belong to different CDCs managed by different cloud orchestrating instances is still problematic, and it needs more investigation to resolve potential synchronization conflicts.

Chapter 6

Summary of results and future work

6.1 Summary of results

This dissertation focused on the congestion control in SDN based data center networks by providing solutions belong to "In-network" solution class. My proposed solutions include algorithms and numerical methods, whose efficiency is evaluated by simulation. The new results can be summarized as follows:

1. Evaluate the performance of currently used algorithms in data center networks in related to loss rate and throughput of elephant flows [J1].

2. Propose different reduction rate upon congestion events of QUIC protocol. We validated the proposed value using simulation results which showed decrease in page load time up to 22% [C1].

3. Design and create a QoS based framework which schedules mice and elephant flows using dedicated queues of connected switches. The main target was improving FCT of mice flows. The results show improvement in FCT of mice flows up to 400ms [C2].

4. Design and create a framework which schedules portion of flows in the data center networks. On the other hand, it reschedules elephant flows. The main target was improving FCT of mice flows. The framework provides lower FCT of mice up to 58% compared to Hedera and ECMP [C3] [J2].

5. Design and create a sampling mechanism to enable the previous framework identifying elephant flows. The proposed mechanism employs the flow size as a parameter to distinguish.

6. Investigate and identify of requirements and characteristics of CDC as well as specifications of six NOSs using mathematical modeling of AHP [J3].

6.2 Future work

Cloud data centers are rapidly developing to meet requirements of new applications. Many open areas should be considered. We can briefly describe the future works of my research as follows:

1. Distributed control plane should be considered to improve availability and reliability of proposed frameworks. Employing features of new OpenFlow versions such as flow monitoring feature introduced in OpenFlow 1.4 by which different subsets of flow tables can be associated with specific controller instances and egress processing feature introduced in OpenFlow 1.5 to enable packet processing in the context of outbound traffic.

2. Expand our solutions to provide different services based on differentiating between UDP and TCP traffic.

3. Improve the applicability of flow detection mechanism by considering the detection time based on the required time to capture the sufficient number of flow packets.

4. Scheduling and rerouting of flows traversing between different DC should be considered as well.

5. A continuous follow-up of the NOSs should be conducted since Software Defined Cloud Computing (SDCC) is still in the stages of maturity and adoption since SDN related technologies are developing rapidly because of the tendency to convert CDC into Software Defined Environment (SDE).

6. It is essential to extend the current NOSs to provide network services in case of inter-CDCs (e.g., deploying SFC whose VMs belong to multiple CDCs managed by different cloud orchestrating instances).

7. Maintaining the synchronization between multiple NOSs belong to different CDCs managed by different cloud orchestrating instances is still problematic, and it needs more investigation to resolve potential synchronization conflicts.

8. Asymmetric DCN is common in today data centers, so our future work will cover evaluating our scheduling frameworks in such an environment.

6.3 Publications

6.3.1 International Journals

J1. Alawadi, Aymen Hasan and **Zaher, Maiass** and Molnár, Sándor, "Methods for Predicting Behavior of Elephant Flows in Data Center Networks", Infocommunications Journal, Vol. XI, No 3, September 2019, pp. 34-41. DOI: 10.36244/ICJ.2019.3.6. (WOS, Scopus, CS=0.8)

J2. **Zaher, Maiass** and Alawadi, Aymen Hasan and Molnár, Sándor, "Sieve: A flow scheduling framework in SDN based data center networks", Computer Communications, 171, 2021, pp. 99-111. DOI: 10.1016/j.comcom.2021.02.013. (WOS, IF=3.16, Scopus, CS=5.8)

J3. **Zaher, Maiass** and Molnár, Sándor, "A Comparative and Analytical Study for Choosing the Best Suited SDN Controller for Cloud Data Center". (under review, submitted to Annals of Emerging Technologies in Computing, Journal)

6.3.2 International Conferences

C1. **Zaher, Maiass** and Molnár, Sándor, "On the Impact of the Multiplication Decrease Factor of QUIC", In 2018 International Symposium on Networks, Computers and Communications (ISNCC), pp. 1-5. IEEE, 2018.

C2. **Zaher, Maiass** and Molnár, Sándor, "Enhancing of micro flow transfer in SDN-based data center networks", In ICC 2019-2019 IEEE International Conference on Communications (ICC), pp. 1-6. IEEE, 2019.

C3. **Zaher, Maiass** and Alawadi, Aymen Hasan and Molnár, Sándor, "Class-based Flow Scheduling Framework in SDN-based Data Center Networks", In 2020 International Conference on Computing, Electronics Communications Engineering (iCCECE), pp. 51-56. IEEE, 2020.

Bibliography

[1] Guck, J.W., Van Bemten, A., Kellerer, W.: Detserv: Network models for real-time qos provisioning in sdn-based industrial environments. IEEE Transactions on Network and Service Management **14**(4), 1003–1017 (2017)

[2] Alizadeh, M., Greenberg, A., Maltz, D.A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., Sridharan, M.: Data center tcp (dctcp). In: Proceedings of the ACM SIGCOMM 2010 Conference, pp. 63–74 (2010)

[3] Hafeez, T., Ahmed, N., Ahmed, B., Malik, A.W.: Detection and mitigation of congestion in sdn enabled data center networks: A survey. IEEE Access **6**, 1730–1740 (2017)

[4] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., Vahdat, A.: Hedera: dynamic flow scheduling for data center networks

[5] Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement, pp. 267–280 (2010)

[6] Mori, T., Kawahara, R., Naito, S., Goto, S.: On the characteristics of internet traffic variability: spikes and elephants. IEICE TRANSACTIONS on Information and Systems **87**(12), 2644–2653 (2004)

[7] Roy, A., Zeng, H., Bagga, J., Porter, G., Snoeren, A.C.: Inside the social network's (datacenter) network. In: Proceedings of the ACM Conference on Special Interest Group on Data Communication, pp. 123–137

[8] Foundation, T.A.S.: Apache Hadoop. Available from: http://hadoop.apache.org/ [last accessed April 2019] (2019)

[9] Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)

[10] Ahn, J.H., Binkert, N., Davis, A., McLaren, M., Schreiber, R.S.: Hyperx: topology, routing, and packaging of efficient large-scale networks. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11 (2009)

[11] Kim, J., Dally, W.J., Abts, D.: Flattened butterfly: a cost-efficient topology for high-radix networks. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, pp. 126–137 (2007)

[12] Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. ACM SIGCOMM computer communication review **38**(4), 63–74 (2008)

[13] Rashid, J.A.: Sorted-gff: An efficient large flows placing mechanism in software defined network datacenter. Karbala International Journal of Modern Science **4**(3), 313–331 (2018)

[14] Yu, S., Lin, X., Misic, J.: Networking for big data: part 2 [guest editorial]. IEEE network **29**(5), 4–5 (2015)

[15] Hopps, C., et al.: Analysis of an equal-cost multi-path algorithm. Technical report, RFC 2992, November (2000)

[16] Greenberg, A., Hamilton, J.R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D.A., Patel, P., Sengupta, S.: Vl2: a scalable and flexible data center network. ACM SIGCOMM computer communication review **39**(4), 51–62

[17] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., Vahdat, A., *et al.*: Hedera: dynamic flow scheduling for data center networks. In: Nsdi, vol. 10, pp. 89–92 (2010). San Jose, USA

[18] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. ACM SIGCOMM computer communication review **38**(2), 69–74 (2008)

[19] Chhabra, A., Kiran, M.: Classifying elephant and mice flows in high-speed scientific networks. Proc. INDIS, 1–8 (2017)

[20] Curtis, A.R., Kim, W., Yalagandula, P.: Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In: 2011 Proceedings IEEE INFOCOM, pp. 1629–1637 (2011). IEEE

[21] Wang, C., Hu, B., Chen, S., Li, D., Liu, B.: A switch migration-based decision-making scheme for balancing load in sdn. IEEE Access **5**, 4537–4544 (2017)

[22] Tang, F., Zhang, H., Yang, L.T., Chen, L.: Elephant flow detection and differentiated scheduling with efficient sampling and classification. IEEE Transactions on Cloud Computing (2019)

[23] Liu, J., Li, J., Shou, G., Hu, Y., Guo, Z., Dai, W.: Sdn based load balancing mechanism for elephant flow in data center networks. In: 2014 International Symposium on Wireless Personal Multimedia Communications (WPMC), pp. 486–490 (2014). IEEE

[24] Curtis, A.R., Mogul, J.C., Tourrilhes, J., Yalagandula, P., Sharma, P., Banerjee, S.: Devoflow: Scaling flow management for high-performance networks. In: Proceedings of the ACM SIGCOMM 2011 Conference, pp. 254–265 (2011)

[25] Wang, Y.-C., You, S.-Y.: An efficient route management framework for load balance and overhead reduction in sdn-based data center networks. IEEE Transactions on Network and Service Management **15**(4), 1422–1434 (2018)

[26] Wang, C., Zhang, G., Chen, H., Xu, H.: An aco-based elephant and mice flow scheduling system in sdn. In: 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA), pp. 859–863 (2017). IEEE

[27] Dimitrova, D.S., Kaishev, V.K., Tan, S.: Computing the kolmogorov-smirnov distribution when the underlying cdf is purely discrete, mixed, or continuous. Journal of Statistical Software **95**(1), 1–42 (2020)

[28] Walter, J.-C., Barkema, G.: An introduction to monte carlo methods. Physica A: Statistical Mechanics and its Applications **418**, 78–87 (2015)

[29] Alexander, C.: Market Risk Analysis, Value at Risk Models vol. 4. John Wiley & Sons, ??? (2009)

[30] Hamming, R.W.: The Art of Probability: for Scientists and Engineers. CRC Press, ??? (2018)

[31] Hamilton, R., Iyengar, J., Swett, I., Wilk, A.: QUIC: A UDP-based secure and reliable transport for HTTP/2. Available from: https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02 [last accessed December 2017]

[32] Jeong, E., Wood, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., Park, K.: mtcp: a highly scalable user-level {TCP} stack for multicore systems. In: 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pp. 489–502 (2014)

[33] Marinos, I., Watson, R.N., Handley, M.: Network stack specialization for performance. ACM SIGCOMM Computer Communication Review **44**(4), 175–186 (2014)

[34] Roskind, J.: Quick udp internet connections: Multiplexed stream transport over udp. Adresse: https://docs. google. com/document/d/1RNHkx_ VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit (besucht am 05. 07. 2017) (2012)

[35] Roskind, J.: QUIC design document and specification rationale. Available from: https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/mobilebasic [last accessed December 2017]

[36] Mike, B., Roberto, P.: SPDY Protocol. Available from: https://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3 [last accessed December 2017]

[37] Belshe, M., Peon, R., Thomson, M.: Hypertext transfer protocol version 2 (HTTP/2). RFC 7540 (2015)

[38] Ha, S., Rhee, I., Xu, L.: Cubic: a new tcp-friendly high-speed tcp variant. ACM SIGOPS operating systems review **42**(5), 64–74 (2008)

[39] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., Scheffenegger, R.: Cubic for fast long-distance networks. Technical report, Internet-draft draft-rhee-tcpm-cubic-02 (2009)

[40] De Cicco, L., Carlucci, G., Mascolo, S.: Understanding the dynamic behaviour of the google congestion control for rtcweb. In: 2013 20th International Packet Video Workshop, pp. 1–8 (2013). IEEE

[41] Kuo, F.-C., Fu, X.: Probe-aided multcp: an aggregate congestion control mechanism. ACM SIGCOMM Computer Communication Review **38**(1), 17–28 (2008)

[42] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., *et al.*: The quic transport protocol: Design and internet-scale deployment. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp. 183–196 (2017)

[43] Carlucci, G., De Cicco, L., Mascolo, S.: Http over udp: an experimental investigation of quic. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 609–614 (2015)

[44] Miller, K., Hsiao, L.W.: Tcptuner: congestion control your way. arXiv preprint arXiv:1605.01987 (2016)

[45] inc., G.: Chromium. Available from: https://cs.chromium.org/chromium/src/net/quic/ [last accessed December 2017]

[46] Appenzeller, G., Keslassy, I., McKeown, N.: Sizing router buffers. ACM SIGCOMM Computer Communication Review **34**(4), 281–292 (2004)

[47] Web-Archive: Report: State of the Web. Available from: https://httparchive.org/reports/state-of-the-web?start=latest [last accessed December 2017]

[48] Megyesi, P., Krämer, Z., Molnár, S.: How quick is quic? In: 2016 IEEE International Conference on Communications (ICC), pp. 1–6 (2016). IEEE

[49] Sivanathan, A., Gharakheili, H.H., Loi, F., Radford, A., Wijenayake, C., Vishwanath, A., Sivaraman, V.: Classifying iot devices in smart environments using network traffic characteristics. IEEE Transactions on Mobile Computing **18**(8), 1745–1759

[50] Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable commodity data center network architecture. ACM SIGCOMM Computer Communication Review **38**(4), 63–74

[51] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V., Matus, F., Pan, R., Yadav, N., Varghese, G.: Conga: Distributed congestion-aware load balancing for datacenters. In: Proceedings of the 2014 ACM Conference on SIGCOMM, pp. 503–514

[52] Curtis, A.R., Kim, W., Yalagandula, P.: Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. Infocom **11**, 1629–1637

[53] Liu, L., Jiang, Y., Shen, G., Li, Q., Lin, D., Li, L., Wang, Y.: An sdn-based hybrid strategy for load balancing in data center networks, 2019. IEEE Symposium on Computers and Communications, 1–6

[54] Wang, C.A., Hu, B., Chen, S., Li, D., Liu, B.: A switch migration-based decision-making scheme for balancing load in sdn. IEEE Access **5**, 4537–4544

[55] Yazidi, A., Abdi, H., Feng, B.: Data center traffic scheduling with hot-cold link detection capabilities. In: Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems, pp. 268–275

[56] Tang, F., Zhang, H., Yang, L.T., Chen, L.: Elephant flow detection and differentiated scheduling with efficient sampling and classification. IEEE Transactions on Cloud Computing

[57] Sminesh, C., Kanaga, E.G.M., Ranjitha, K.: Flow monitoring scheme for reducing congestion and packet loss in software defined networks. In: 2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS), pp. 1–5 (2017). IEEE

[58] Jingwen, X., Muqing, W., Xiaolan, H.: A traffic scheduling scheme for data center networks based on sdn. In: 2019 IEEE 5th International Conference on Computer and Communications (ICCC), pp. 1417–1422 (2019). IEEE

[59] Fu, Q., Sun, E., Wang, Q., Wang, Z., Zhang, Y.: A joint balancing flow table and reducing delay scheme for mice-flows in data center networks. In: 2019 IEEE Global Communications Conference (GLOBECOM), pp. 1–6 (2019). IEEE

[60] Huang, N.F., Liao, I.J., Liu, H.W., Wu, S.J., Chou, C.S.: A dynamic qos management system with flow classification platform for software-defined networks. In: 8th International Conference on Ubi-Media Computing (UMEDIA), pp. 72–77

[61] Sharma, S.: Implementing quality of service for the software defined networking enabled future internet. In: Third European Workshop on Software Defined Networks (EWSDN), pp. 49–54

[62] Tomovic, S., Prasad, N., Radusinovic, I.: "sdn control framework for qos provisioning,". In: 22nd Telecommunications Forum Telfor (TELFOR), pp. 111–114

[63] Xiao, J., Chen, S., Sui, M.: The strategy of path determination and traffic scheduling in private campus networks based on sdn. Peer Peer Netw. and Appl., 1–10

[64] Li, F., Cao, J., Wang, X., Sun, Y.: "a qos guaranteed technique for cloud applications based on software defined networking,". IEEE Access **5**, 21229–21241

[65] sflow.org: sflow. Accessed 2 August 2017. https://sflow.org/.

[66] Ryu Accessed 12 Mar. 2019). http://osrg.github.io/ryu/

[67] Apache.org: Apache HTTP server benchmarking tool. Accessed 12 Mar. 2019). http://httpd.apache.org/docs/current/install.html

[68] Erickson, D.: The beacon openflow controller, proceeding of 2nd acm sigcomm workshop hot topics softw. Defined Netw, 13–18

[69] Zhao, Y., Iannone, L., Riguidel, M.: On the performance of sdn controllers: A reality check, proceeding ieee conf. Netw. Funct. Virtualization Softw. Defined Netw, 79–85

[70] Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., McKeown, N.: Reproducible network experiments using container-based emulation. In: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, pp. 253–264

[71] A. Alawadi, S.M. M. Zaher: Methods for predicting behavior of elephant flows in data center networks. Infocommunications Journal **XI**(3), 34–41

[72] Zhang, H., Tang, F., Barolli, L.: Efficient flow detection and scheduling for sdn-based big data centers. Journal of Ambient Intelligence and Humanized Computing **10**(5), 1915–1926

[73] Fang, S., Yu, Y., Foh, C.H., Aung, K.M.M.: A loss-free multipathing solution for data center network using software-defined networking approach. IEEE transactions on magnetics **49**(6), 2723–2730 (2013)

[74] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review **38**(2), 69–74 (2008)

[75] ONF: Software-Defined Networking (SDN) Definition. Available from: https://www.opennetworking.org/sdn-definition/ [last accessed February 2021] (2019)

[76] Delimitrou, C., Sankar, S., Kansal, A., Kozyrakis, C.: Echo: Recreating network traffic maps for datacenters with tens of thousands of servers. In: 2012 IEEE International Symposium on Workload Characterization (IISWC), pp. 14–24 (2012). IEEE

[77] Wang, Y., Wang, X., Li, H., Dong, Y., Liu, Q., Shi, X.: A multi-service differentiation traffic management strategy in sdn cloud data center. Computer Networks **171**, 107143 (2020)

[78] Khondoker, R., Zaalouk, A., Marx, R., Bayarou, K.: Feature-based comparison and selection of software defined networking (sdn) controllers. In: 2014 World Congress on Computer Applications and Information Systems (WCCAIS), pp. 1–7 (2014). IEEE

[79] Zobary, F.F.: Applying topsis method for software defined networking (sdn) controllers comparison and selection. In: Proceedings of the International Conference on Communicatins and Networking in China, pp. 132–141 (2016). Springer

[80] Belkadi, O., Laaziz, Y.: A systematic and generic method for choosing a sdn controller. International Journal of Computer Networks and Communications Security **5**(11), 239–247 (2017)

[81] Ali, J., Roh, B., Lee, S.: Qos improvement with an optimum controller selection for software-defined networks. PloS One **14**(5) (2019)

[82] AlShehri, M.A.R., Mishra, S.: Feature based comparison and selection of sdn controller. International Journal of Innovation and Technology Management **16**(05), 19500291–23 (2019)

[83] Amiri, E., Alizadeh, E., Rezvani, M.H.: Controller selection in software defined networks using best-worst multi-criteria decision-making. Bulletin of Electrical Engineering and Informatics **9**(4) (2020)

[84] Ali, J., Lee, B., Oh, J., Lee, J., Roh, B.-h.: A novel features prioritization mechanism for controllers in software-defined networking. Computers, Materials & Continua **69**(1), 267–282 (2021)

[85] Ali, J., Roh, B.-h.: Quality of service improvement with optimal software-defined networking controller and control plane clustering. CMC-COMPUT MATER CONTINUA **67**(1), 849–875 (2021)

[86] Abuarqoub, A.: A review of the control plane scalability approaches in software defined networking. Future Internet **12**(3), 49 (2020)

[87] Sarmiento, D.E., Lebre, A., Nussbaum, L., Chari, A.: Decentralized sdn control plane for a distributed cloud-edge infrastructure: A survey. IEEE Communications Surveys & Tutorials (2021)

[88] Belgaum, M.R., Musa, S., Alam, M.M., Su'ud, M.M.: A systematic review of load balancing techniques in software-defined networking. IEEE Access **8**, 98612–98636 (2020)

[89] Hamdan, M., Hassan, E., Abdelaziz, A., Elhigazi, A., Mohammed, B., Khan, S., Vasilakos, A.V., Marsono, M.: A comprehensive survey of load balancing techniques in software-defined network. Journal of Network and Computer Applications, 102856 (2020)

[90] Isong, B., Molose, R.R.S., Abu-Mahfouz, A.M., Dladlu, N.: Comprehensive review of sdn controller placement strategies. IEEE Access **8**, 170070–170092 (2020)

[91] Zhu, L., Karim, M., Sharif, K., Xu, C., Li, F., Du, X., Guizani, M.: Sdn controllers: A comprehensive analysis and performance evaluation study. ACM Computing Surveys (CSUR) **53**(6), 1–40 (2020)

[92] Saaty, T.L.: The Hierarchy Analytical Process. McGraw-Hill, New York (1980)

[93] Kreutz, D., Fernando, R., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: A comprehensive survey. Proceedings of the IEEE **103**(1), 14–76 (2014)

[94] Saraswat, S., Agarwal, V., Gupta, H.P., Mishra, R., Gupta, A., Dutta, T.: Challenges and solutions in software defined networking: A survey. Journal of Network and Computer Applications **141**, 23–58 (2019)

[95] ONOSProject: ONOS. Available from: https://wiki.onosproject.org/display/ONOS/ONOS [last accessed February 2021] (2019)

[96] McCauley: POX-doc. Available from: https://noxrepo.github.io/pox-doc/html/ [last accessed February 2021] (2019)

[97] Community, R.: Welcome to RYU the Network Operating System(NOS). Available from: https://ryu.readthedocs.io/en/latest/index.html [last accessed February 2021] (2019)

[98] Community, F.: Floodlight Controller. Available from: https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/overview [last accessed February 2021] (2019)

[99] Project, O.: Getting Started Guide. Available from: https://docs.opendaylight.org/en/stable-sodium/getting-started-guide/index.html [last accessed February 2021] (2019)

[100] Foundation, L.: Tungsten Fabric Architecture. Available from: https://tungstenfabric.github.io/website/Tungsten-Fabric-Architecture.html [last accessed February 2021] (2019)

[101] Cao, J., Ma, Z., Xie, J., Zhu, X., Dong, F., Liu, B.: Towards tenant demand-aware bandwidth allocation strategy in cloud datacenter. Future Generation Computer Systems **105**, 904–915 (2020)

[102] Adami, D., Martini, B., Sgambelluri, A., Donatini, L., Gharbaoui, M., Castoldi, P., Giordano, S.: An sdn orchestrator for cloud data center: System design and experimental evaluation. Transactions on Emerging Telecommunications Technologies **28**(11), 3172 (2017)

[103] Cziva, R., Jouët, S., Stapleton, D., Tso, F.P., Pezaros, D.: Sdn-based virtual machine management for cloud data centers. IEEE Transactions on Network and Service Management **13**(2), 212–225 (2016)

[104] Son, J., Buyya, R.: A taxonomy of software-defined networking (sdn)-enabled cloud computing. ACM Computing Surveys (CSUR) **51**(3), 1–36 (2018)

[105] Jararweh, Y., Al-Ayyoub, M., Benkhelifa, E., Vouk, M., Rindos, A., *et al.*: Software defined cloud: Survey, system and evaluation. Future Generation Computer Systems **58**, 56–74 (2016)

[106] Shin, S.W., Porras, P., Yegneswara, V., Fong, M., Gu, G., Tyson, M., *et al.*: Fresco: Modular composable security services for software-defined networks. In: 20th Annual Network & Distributed System Security Symposium (2013). Ndss

[107] Son, J., Buyya, R.: Sdcon: Integrated control platform for software-defined clouds. IEEE Transactions on Parallel and Distributed Systems **30**(1), 230–244 (2018)

[108] Yan, Q., Yu, F.R., Gong, Q., Li, J.: Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges. IEEE communications surveys & tutorials **18**(1), 602–622 (2015)

[109] Qin, P., Dai, B., Huang, B., Xu, G.: Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data. IEEE Systems Journal **11**(4), 2337–2344 (2015)

[110] Karakus, M., Durresi, A.: A survey: Control plane scalability issues and approaches in software-defined networking (sdn). Computer Networks **112**, 279–293 (2017)

[111] Benamrane, F., Benaini, R., *et al.*: An east-west interface for distributed sdn control plane: Implementation and evaluation. Computers & Electrical Engineering **57**, 162–175 (2017)

[112] Almadani, B., Beg, A., Mahmoud, A.: Dsf: A distributed sdn control plane framework for the east/west interface. IEEE Access **9**, 26735–26754 (2021)

[113] Al-Sharif, Z.A., Jararweh, Y., Al-Dahoud, A., Alawneh, L.M.: Accrs: autonomic based cloud computing resource scaling. Cluster Computing **20**(3), 2479–2488 (2017)

[114] Abbasi, A.A., Abbasi, A., Shamshirband, S., Chronopoulos, A.T., Persico, V., Pescapè, A.: Software-defined cloud computing: A systematic review on latest trends and developments. IEEE Access **7**, 93294–93314 (2019)

[115] Tzeng, G., Huang, J.: Multiple Attribute Decision Making: Methods and Applications. CRC press, NW USA (2011)

[116] Saaty, T.L.: Decision Making with Dependence and Feedback: The Analytic Network Process vol. 4922. RWS publications Pittsburgh, PA USA (1996)

[117] Rezaei, J.: Best-worst multi-criteria decision-making method. Omega **53**, 49–57 (2015)

[118] Şahin, M.: A comprehensive analysis of weighting and multicriteria methods in the context of sustainable energy. International Journal of Environmental Science and Technology **18**(6), 1591–1616 (2021)

[119] Singh, M., Pant, M.: A review of selected weighing methods in mcdm with a case study. International Journal of System Assurance Engineering and Management, 1–19 (2020)